LLMs WITH MATLAB

V1.0 – July 2025

github.com/yanndebray/LLMs-with-MATLAB-book

To Zoé

TABLE OF CONTENTS

PREFACE

"The proper study of man is anything but himself, and it is through the study of language that we learn about our own nature." – J.R.R. Tolkien

Language is a foundation for humanity. It is through language that we have elevated ourselves, from the silence of nature. First, we existed, then our essence was granted by a Voice. We shared with one another. We started telling each other stories, sometimes simply relaying information, sometime making things up. We elaborated ideas, started reasoning about them, turning them into plans. A sound became words – letters, semantics, lexicons – words transformed into writings, writing into books, books into cultures, cultures into civilizations. Leading us all the way to the current digital age, with the inter-connected-net of articles, blogs, images, videos. All this content that started to feed machines, eager to learn.

Ever since the release of an apparently harmless chatbot application, we have observed an acceleration in the development of language modeling. Artificial intelligence has marked a new epoch in our relationship with language. These models, built on the crunching of numbers by the largest supercomputers in the world, have enabled computers to understand and generate human language with unprecedented accuracy. The next wave that is gradually taking the human away from the conversational loop is AI agents. Agents build on large language models, but they can act autonomously, by using tools and carrying out entire workflows without necessarily needing tight supervision from end-users, such as browsing the web to perform research and analysis, to return detailed reports.

This book is designed to equip you with the knowledge and skills needed to harness the power of agents, opening new horizons in the field of artificial intelligence. Whether you are a seasoned developer or a curious enthusiast, this practical guide will help you navigate the complexities of building AI agents, transforming your ideas into reality.

WHO IS THIS BOOK FOR

Ever heard of FOMO (Fear Of Missing Out)? This is probably what you are feeling with everything happening in the field of AI. Every month comes with something new, and it might feel overwhelming. You might not know where to start, as you probably

wonder if and how it might affect your career. Then you are in the right place; I'll make sure you find just what you need in this book.

I am assuming that you are *NOT an AI expert* (quite the opposite actually). If you already have some notions of what machine learning means, those might prove useful as we go deeper into describing what Generative AI is and how it differs from the previous waves of AI.

You are probably *tech savvy*, with likely some background in scientific or technical studies. Don't worry, I won't use gory mathematical equations to explain the concepts manipulated in the book. Instead, I'll use concrete examples of simple applications you can develop and tailor to your needs.

I'll assume some *basic programming skills*, but no necessary experience with AI toolboxes like stats & machine learning, deep learning or text analytics. It would be good for you to have some basics of numeric, with core MATLAB. Those should be easy to acquire. Overall, you should have some appetite for coding, as it will make this experience much more enjoyable and give you a deeper understanding of the concepts.

ABOUT THE AUTHOR

I graduated from college with a degree in mechanical engineering. Some of the courses that I really got into involved numerical analysis with Maple and Mathematica. But it's only after my studies that I started to learn about data science and AI. This was 2013 and the rise of online courses, so like many, I followed the machine learning course from Stanford professor Andrew Ng on Coursera.

I discovered myself a passion for technical computing, and joined a French start-up called Scilab. They were spinning off from the French research to build a consulting business around open-source software. It ended up being an exciting but challenging journey, with new cloud products rollouts and an acquisition a few years later.

Then in the beginning of 2020, as the world was starting to lock down, I got a call from a company in Boston to work on the famous software MATLAB, the leader in technical computing. A few years later, here I am, more passionate than ever about numeric and eager to learn about its new forms, mostly called AI now. And the best way to learn is to teach. So, buckle up, and let's go for the ride!

WHAT YOU WILL LEARN

The book is divided into 10 chapters, each covering a different topic and a different aspect of calling LLMs from MATLAB. The chapters are:

- *Chapter 1: Introducing LLMs***.** How they work and their evolution.
- *Chapter 2: OpenAI APIs.* In this chapter, you will learn how to use the OpenAI APIs, a simple way to create AI apps. You will start by learning how to create your own chatbot.
- *Chapter 3: Prompting, Chaining & Summarization.* In this chapter, you will learn how to engineer prompts, chain calls to a Large Language Model and use it to summarize texts, such as articles, books, or transcripts.
- *Chapter 4: Vector search & Question Answering.* In this chapter, you will learn how to use embeddings and vector search as a way to retrieve informative answers to answer questions while quoting sources.
- *Chapter 5: Agents & Tools.* In this chapter, you will learn to build an agent, called Smith, that has access to tools, such as compute, search & weather.
- *Chapter 6: Speech-to-Text & Text-to-Speech.* In this chapter, you will learn how to use LLMs to transcript text from speech (such as Youtube videos) and synthetize speech from text (to create your own tech podcast).
- *Chapter 7: Vision.* In this chapter, you will learn to process and analyze images, such as mock-ups or drawings. You will learn how to use Vision APIs, to perform various tasks such as text recognition, or video captioning.
- *Chapter 8: Image generation.* In this chapter, you will learn how to create stunning and creative images from any text input.
- *Chapter 9: Appendix.* In this chapter, you will find additional resources to keep up with the latest developments and innovations in the field of AI.

By the end of this book, you will have a solid understanding of how to program LLM applications mostly leveraging the OpenAI services with MATLAB. You will also have a deeper appreciation of the power and possibilities of AI, and how it goes beyond processing natural language, speech and images.

Disclaimer: I have used ChatGPT to help me write this book. But I would argue that the person who does not leverage AI to do her work is going to be left behind. Or said in a more politically correct tone: “You won’t be replaced by AI, but you’ll be replaced by someone using AI”.

Setting up the Programming Environment

This section will guide you through the necessary tools and libraries, as well as a step-by-step setup process.

I'll take as assumption that you are running on a Windows machine. Wherever there is a major difference in OS, I'll try to make sure that I give explanations for the different platforms.

Run the following line of code to automate the next two steps:

```
addpath("preface"); setup
```

Connect LLMs with MATLAB

Large Language Models (LLMs) with MATLAB[1] (a.k.a. "LLMs with MATLAB") is the official library provided by MathWorks for interacting with the OpenAI APIs as well as other LLMs. There are two ways you can install this library manually or programmatically:

1. Go to "Add-Ons" in the Home tab of MATLAB interface to open the Add-On Explorer.
2. Search "Large Language Models (LLMs) with MATLAB".
3. Click "Add" to install the package.
4. Add it manually to your MATLAB path:

Or

```
folderName = 'llms-with-matlab';
if ~isfolder(folderName)
    gitclone("github.com/matlab-deep-learning/llms-with-matlab");
else
    disp('The folder already exists.');
end
addpath(genpath("llms-with-matlab")
```

[1] https://github.com/matlab-deep-learning/llms-with-matlab

API Keys and Authentication (For OpenAI)

Obtain an API key from OpenAI by registering on their platform. Set up the authentication to OpenAI by adding your API key to your environment variables with an .env file.

1. Create a new file in the editor:
   ```
   >> edit .env
   ```
2. Type *OPENAI_API_KEY=<your key>* and save it as an .env file (the file won't appear by default in the file browser in MATLAB Online, but you can change this setting)
3. Load your API key every time you start a new MATLAB session:
   ```
   >> load(".env")
   ```
4. Retrieve your API key:
   ```
   >> getenv("OPENAI_API_KEY")
   ```

Test Installation

Verify your setup by running a small script to interact with the OpenAI API.

```
addpath("path/to/llms-with-matlab");
loadenv("path/to/.env");
client = openAIChat( ...
         ApiKey=getenv("OPENAI_API_KEY"), ...
         ModelName="gpt-4o-mini");
res = generate(client,"Say this is a test")
```

Use MATLAB Online

Throughout this book, we will use MATLAB Online[2], which provides MATLAB, Simulink, and other commonly used toolboxes in the web browser. MATLAB Online is free for up to 20 hours a month without a license, hence it is one of the most accessible options. You can use it without the monthly cap if you have an existing commercial or academic license. You can also use the desktop version of MATLAB. If you are new to MATLAB, create a MathWorks account to get access to MATLAB Online.

MATLAB Online always comes with the latest version of the MATLAB language and development environment. At the time of this writing, I am using 24b. MATLAB Online also provides a pre-installed version of Python (3.10 compatible with MATLAB

[2] https://www.mathworks.com/products/matlab-online.html

24b). Here is a table of the versions of Python compatible with MATLAB, should you choose to install everything locally[3]:

Table 0-1 MATLAB and Python versions compatibility

Release	Python versions supported
R2024b	3.9, 3.10, 3.11, 3.12
R2024a	3.9, 3.10, 3.11
R2023b	3.9, 3.10, 3.11
R2023a	3.8, 3.9, 3.10

You can download Python from the official Python website[4].

Install Python packages in MATLAB Online

In my book about MATLAB with Python, I go over a simple way to install Python packages into MATLAB Online[5].

First you need to retrieve pip and save it in a temporary folder

```
>> websave("/tmp/get-pip.py","https://bootstrap.pypa.io/get-pip.py");
```

```
>> !python /tmp/get-pip.py
```

```
>> !python -m pip --version
```

```
pip 24.2 from /home/matlab/.local/lib/python3.10/site-packages/pip (python 3.10)
```

You can now simply install a package like langchain and transformers as such:

```
pip install langchain transformers
```

LangChain [6] is useful for chaining language model calls and building complex applications. *Transformers* [7] is a popular library for working with open-source AI models. You will learn about those packages later in the book, and how to invoke them from MATLAB.

Bear in mind that the MATLAB Online environment is ephemeral, and that you will have to repeat this process each time you start a new session.

[3] https://www.mathworks.com/support/requirements/python-compatibility.html
[4] www.python.org
[5] https://github.com/yanndebray/matlab-with-python-book/blob/main/8_Resources.md
[6] https://www.langchain.com/
[7] https://huggingface.co/docs/transformers

1. Introducing LLMs

Let's start with a bit of theory. If you're reading this book, I will assume you have a basic math and science education, and some programming experience in MATLAB. I don't want to go too deeply into the different dimensions of large language models (LLMs), generative AI (genAI), and artificial intelligence (AI) either. My goal is to provide enough necessary background so that you can apply AI to your concrete applications. Figure 1-1 is a simple illustration of the relationships among these three concepts.

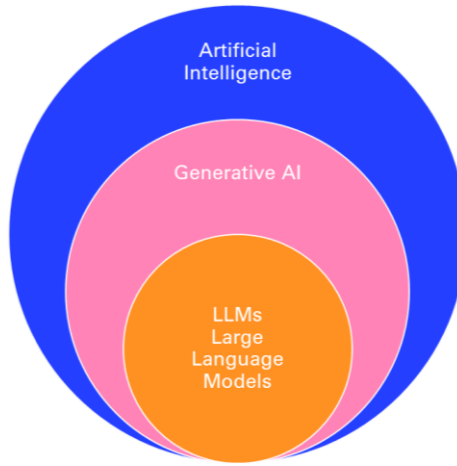


Figure 1-1 Levels within AI

Large Language Models Are Next-Word Predictors

Large language models (LLM) are neural networks - mimicking the structure of the human brain - that can process and generate natural language texts based on a given input, such as a word, a phrase, or a prompt. LLMs operate word by word, based on the probability of each word being the most likely next word, kind of like the autocomplete function on your phone. Figure 1-2 shows an example of how an LLM might predict (output) the next word given an input of words.

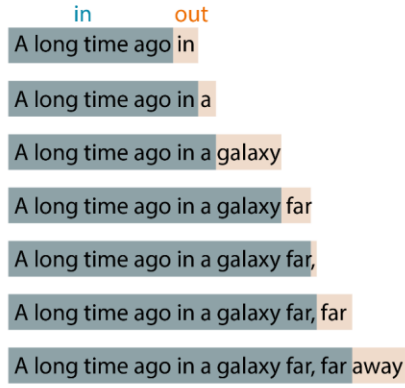


Figure 1-2 Next word prediction

Given the input *A long time ago*, an LLM might generate the output *in a galaxy far, far away*. It works like this:

1. To generate the first word *in*, it uses only the input *A long time ago*.
2. To generate the second word *a*, it uses the input plus the first word it just generated: *A long time ago + in*.
3. To generate the third word, it uses the input plus the first word and the second word, and so on.

By working this way, after being trained on many texts and learning from the patterns and structures of natural language, an LLM can generate coherent and fluent texts of its own. This process is based on statistics and probabilities, as were former generations of AI models.

This kind of model is sometimes referred to as autoregressive. *Autoregressive* is a term that describes a model that predicts the next output based on the previous inputs as well as subsequent outputs from each iteration. Such a model assumes that the current value of a variable depends on its past values. It's a little bit like how you might be able to guess the next word in a sentence if someone paused while speaking. Autoregressive models are widely used for applications other than text generation, such as time series forecasting and speech synthesis.

An LLM can also perform various natural language understanding tasks, such as answering questions, summarizing texts, or extracting information. LLMs can be seen as general-purpose language engines that can handle multiple tasks and applications with minimal supervision. As you'll learn later in this chapter, GPT models are only one kind of LLM, developed by OpenAI in 2018. Since then, other prominent commercial and open-source alternatives have been released, such as the Llama models from Meta, Gemini from Google, Claude from Anthropic, the Mistral models, and others.

/! LLMs are not by nature assistants. If you prompt them with the beginning of a sentence, they will by default try to complete the sentence. They won't try to make sense of the sentence or answer it as if it were a question. This is a fundamental shift that ChatGPT introduced by fine-tuning OpenAI's GPT 3 model to perform question answering in the form of a chatbot.

LLMs are trained on massive amounts of text data, usually *trillions* of words from the internet (hence the term *large*), and from that training they learn to capture the statistical patterns and structures of natural languages in their parameters (generally in *billions*). You will discover in the next sections how this training process works. One way to think about LLMs is to consider them as a compression of internet knowledge. If you look at a model like one of the first Llama from Meta with 70 billion parameters (each represented as two bytes) and download it to your machine, you would have a 140 GB file (including the parameters). This represents a compression ratio of roughly 100:1. For state-of-the-art models like GPT-4, those numbers are off by a factor of 10 at least. Figure 1-3 tries to capture this idea.

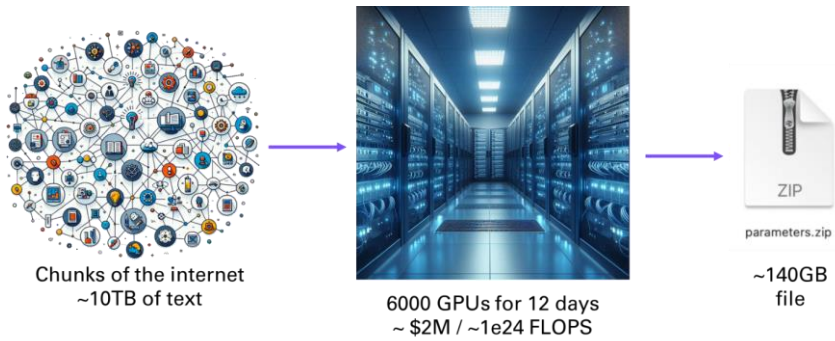


Figure 1-3 LLMs can be said to compress the Internet

LLMs kept growing larger and larger. Something quite remarkable and somewhat unexpected happened when LLMs reached a certain scale: some new properties started to emerge. I'll talk more about the characteristics of these emergent properties when we look at the architecture of LLMs.

Table 1-1 should give you a sense of how LLMs have grown larger over time, using the GPT family of models to illustrate.

Table 1-1 Size of the different generations of GPT models

Model	Dataset size (billions of tokens)	Model size (billions of parameters)
GPT 1	1-2	0.11
GPT 2	10-20	1.4
GPT 3	300	175
GPT 4	10,000	??? (est. 1,800)

TRANSFORMERS: BUILDING BLOCKS OF LANGUAGE INTELLIGENCE

GPT stands for generative pre-trained transformers. The term refers to a family of LLMs created by the company OpenAI. Like all LLMS, GPTs are trained on massive amounts of text data from the web, such as Wikipedia articles, news stories, blog posts, books, and more. They learn to capture the patterns and structures of natural language, such as grammar, syntax, semantics, and style. Let's look a little closer at the three words contained in the term *GPT*.

GENERATIVE

Generative models can create new data from existing data. For example, given an image, a generative model can produce another image that is similar but not identical to the original one. Similarly, given a text, a language model can produce another text that is related but not identical to the original one. Since GPT-4, the line between language models and other kinds of generative models is blurring, as GPT-4 supports text as well as image and sound (both as input and output).

Going back to pure language modeling, here is an example of summarizing the introduction of this book preface with GPT-2[8]:

```
mdl = gpt2();
text = ["Language is a foundation for humanity...";
"Ever since the release of an apparently harmless chatbot...";
"This book is designed to equip you with the knowledge and skills needed to harness the power of agents, opening new horizons in the field of artificial intelligence..."];
text = strjoin(txt,newline);
generateSummary(mdl, text)
```

```
The book is designed to equip you with the knowledge and skills needed to harness the power of agents, opening new horizons in the field of artificial intelligence.
```

To get a sense of how the first GPT model works, you can use the Write With Transformer app available at *transformer.huggingface.co*. Simply select the model you would like to take for a spin, and you will be able to type in a prompt and see the model generate text based on it, as shown in Figure 1-4.

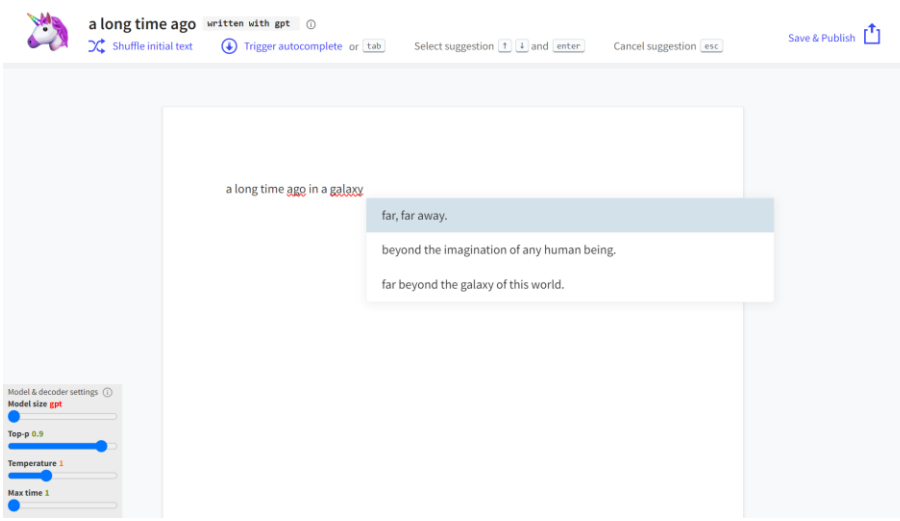


Figure 1-4 Write With Transformer example

[8] https://github.com/matlab-deep-learning/transformer-models

PRE-TRAINED

LLMs have been *pre-trained* on massive amounts of text from the internet and can produce coherent and diverse responses on almost any topic. They can also perform various tasks, such as answering questions, summarizing texts, generating images, and more.

This process of pre-training involves two main stages:

1. Pre-training a base model
2. Fine-tuning an assistant model

Table 1-2 summarizes the LLM training process.

Table 1-2 LLM training pipeline

	Pre-training	**Fine-tuning**
Dataset	Raw internet: Trillions of words. Low quality, large quantity	Demonstrations: ~10-100K Ideal assistant responses (prompt, response) written by contractors. High quality, low quantity
Model	Base model	Assistant model
Method	Self-supervised next word prediction	Supervised on manually labeled data
Compute	1000s of GPUs, months of training	1-100 GPUs, days of training

Language contains impressive inherent properties, such as structure and syntax, context and semantics, intrinsic hierarchical information, predictability and sheer richness in the data. The big breakthrough enabled by generative models was due to the fact that they could learn language constructs by *self-supervision*. Traditional supervised learning requires a curated set of inputs and outputs. Generative models on the other end simply train their parameters by predicting the last word of a sequence with a sliding window, as shown in Figure 1-5.

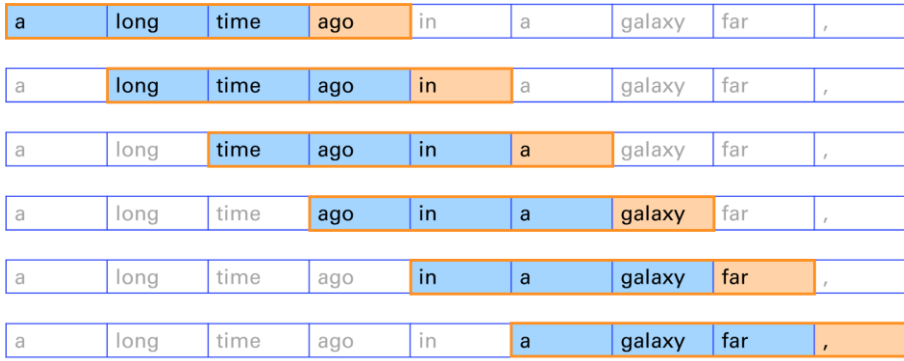


Figure 1-5 Self-supervision with a sliding window

The emergence of properties that I mentioned appeared with the scaling up of the size of the neural network and the dataset on which it was trained. As Aristotle said, "The whole is greater than the sum of its parts." This topic raises the question of *artificial general intelligence* (AGI), a hypothetical type of AI that would possess the ability to perform any intellectual task that a human can. Such a concept is well beyond the scope of this book. Here we will focus on more practical aspirations: to teach you how to leverage LLMs to assist you in your daily tasks.

TRANSFORMERS

LLMs use a special kind of neural network architecture called a *transformer*, introduced in 2017 in a paper by Ashish Vaswani et al called "Attention Is All You Need"[9]. Similarly to previous generations of neural networks, like convolutional or recurrent neural networks, transformers are composed of many *layers* of artificial neurons that are activated in a specific order during the training and prediction phases depending on their arrangement. For instance, recurrent neural networks (RNNs) maintain a memory of the previous state of information processed by the network, which is useful to predict future elements of a sequence. But as the name of the paper hints, the new network architecture introduced with the transformers departs from previous approaches to convert sequences to sequences, by focusing on a key ingredient, called the *attention mechanism*.

Attention provides contextual understanding, which enables the model to consider the meaning of each word in a sentence, regardless of its position. This means each word's encoding in the neural network is influenced by every other word in the sentence, allowing for a more nuanced understanding of language. The attention

[9] Attention is all you need: https://arxiv.org/abs/1706.03762

mechanism equips the model with an efficient method for processing text. Unlike RNNs where the words are processed sequentially, transformers process all words in a sentence simultaneously, leading to significant gains in computational efficiency. For example, given a sentence in English, transformers can learn to translate it to French. In Figure 1-6, you can observe the more verbose translation in French. The lighter yellow positions in this attention matrix capture the translations of the words (attention is the same word in French).

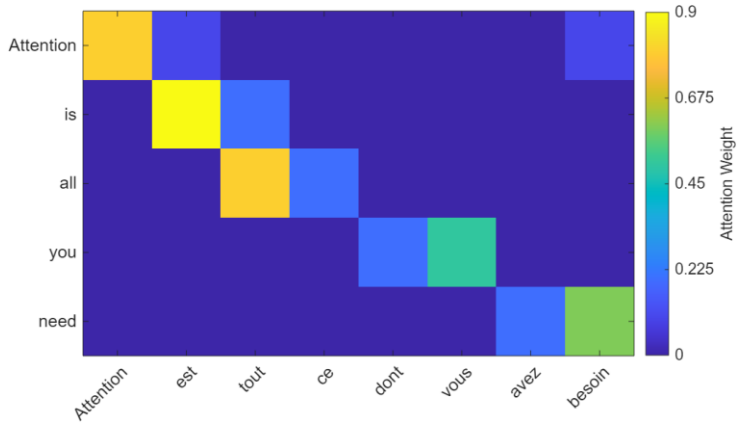


Figure 1-6 Simple translation from English to French

The original transformer from the paper consists of two parts: an encoder and a decoder[10]. The *encoder* takes the input text and transforms it into a numerical representation, i.e. a series of numbers mapping each part of the sentence, capturing the meaning and context of the input. The *decoder* takes the numerical representation and generates the output text. This is particularly useful for translation.

GPT models, like most LLMs nowadays, are decoders-only because this type of network is particularly well-suited for text completion. By focusing solely on the decoder, GPT models can streamline their architecture to better specialize in understanding and generating language. Decoders-only architectures reduce complexity because they focus on one task (next-word prediction) and don't require the additional overhead of encoders. This makes them easier to scale and optimize for large datasets and extensive training without needing to balance encoder-decoder interactions. Since decoders-only models are trained autoregressively (predicting the next token based on previous ones), this setup naturally lends itself to highly parallelized and efficient training.

[10] Encoder-decoder architecture: Overview https://www.youtube.com/watch?v=zbdong_h-x4

Transformers have become the dominant architecture for LLMs, and can be described as general-purpose machines, being expressive, optimizable, and efficient:

- *Expressive:* They can model complex relationships and patterns in data
- *Optimizable*: Their ability to handle complexity is not compromised by the fact that transformers can be trained on large amount of data
- *Efficient*: The training and prediction can be parallelized on graphics cards.

You can learn more about the transformer anatomy in the appendix of this book.

Evolution of LLMs: Genealogy of the GPT family

The development of LLMs has been a revolutionary journey in the field of artificial intelligence and natural language processing. This section delves into the fascinating progression of GPT models, from their humble beginnings to the powerful language models we see today. We'll explore the key milestones, technological advancements, and the increasing capabilities that each iteration brought to the table. By understanding this evolution, we can better appreciate the current state of AI and glimpse into its potential future applications.

Life BC (Before ChatGPT)

As I've mentioned, the first GPT model (later called GPT-1) was released in 2018 by OpenAI, along with a paper called "Improving Language Understanding by Generative Pre-training."[11] This first model had 117 million parameters, trained on a large corpus of text from the web, called WebText, which contained about 40 GB of data. As its parameters and architecture were released with an open-source license, it enabled researchers to fine-tune it to specific tasks (such as text classification and sentiment analysis) with relatively little new data, leveraging its pre-trained knowledge.

However, GPT-1 had some limitations. For example, it could not handle long-term dependencies, because of the small context window at the core of its relatively shallow attention layer. That meant it could not remember or use information that had appeared earlier in the text. It also struggled with factual consistency, which means that it could not verify or correct the information that it generated. Moreover, it sometimes produced offensive or biased texts, which reflected the quality and diversity of the data that it was trained on.

[11] https://openai.com/index/language-unsupervised/

To address these issues, OpenAI released GPT-2 in 2019. The accompanying paper was titled "Language Models are Unsupervised Multitask Learners"[12] highlighting the emerging ability of this model to perform tasks on natural language such as question answering, without the need for supervised training on task-specific datasets. This new set of capabilities came from its 1.5 billion parameters, which was more than 10 times the size of GPT-1. It was also trained on a much larger corpus of text, called WebText2, which contained about 570 GB of data.

GPT-2 improved significantly on the performance and quality of GPT-1 and achieved state-of-the-art results on many natural language benchmarks. It also demonstrated a remarkable ability to generate coherent and engaging texts on a variety of topics and styles, such as news articles, essays, stories, and reviews. However, GPT-2 also raised some ethical and social concerns. Due to its high level of realism and versatility, GPT-2 could potentially be used for malicious purposes, such as spreading misinformation, impersonating others, or generating fake content. Therefore, OpenAI decided to release GPT-2 gradually, starting with a smaller version of 124 million parameters and then releasing larger versions over time, along with some tools and guidelines to help researchers and developers use GPT-2 responsibly and safely.

The big breakthrough came with GPT-3 and the appropriately titled paper "Language Models are Few-Shot Learners"[13], which was released in 2020 by OpenAI. With a staggering 175 billion parameters, more than 100 times the size of GPT-2, it was trained on an enormous corpus of text, called WebText3, which contained about 45 TB of data. One of the key features of GPT-3 was its ability to perform tasks with little to no task-specific training, known as *zero-shot* or *few-shot learning*. This means GPT-3 could understand and respond to prompts in a meaningful way, even if it hadn't been explicitly trained on that task.

GPT-3 was not only a powerful language generator, it was a general-purpose artificial intelligence system that could learn to perform any task that can be described in natural language. For example, given a prompt like *Write a summary of this article*, or *Create a slogan for this company*, or *Solve this math problem*, GPT-3 could generate appropriate and accurate responses by using its vast knowledge and understanding of language and the world. And to guide the model in the type of results you were expecting, you could provide one or several examples as part of the prompt.

[12] https://openai.com/index/better-language-models/
[13] Language Models are Few-Shot Learners: https://arxiv.org/abs/2005.14165

ChatGPT and Alignment

GPT-3.5, also known as ChatGPT is an improved version of GPT-3 that focuses on enhancing its conversational skills. ChatGPT is designed to be a friendly and engaging chatbot that can chat with humans about any topic and provide relevant and interesting information as well as tell jokes. It closely follows a paper from early 2022 called "Training language models to follow instructions with human feedback"[14], introducing instructGPT and paving the way for the revolution that was to come…

ChatGPT is based on the same architecture and data as GPT-3, but with some key differences. ChatGPT aimed to fix what is called alignment. *Alignment* is the process of encoding human values and goals into large language models to make them as helpful, safe, and reliable as possible. Essentially, it builds in guardrails so that the AI doesn't embed the biases that can be found on the internet.

How did OpenAI achieve this? ChatGPT was initially meant only as a research project to demonstrate the potential of *reinforcement learning from human feedback* (RLHF). This involved leveraging some of the early work of OpenAI in the field of reinforcement learning that is often associated with robotics or games like chess, go, or Mario.

In traditional reinforcement learning, an agent learns to make decisions by interacting with an environment to maximize cumulative reward. The agent explores actions, observes results, and receives rewards or penalties, which guide future actions. In RLHF, human feedback supplements or replaces the predefined reward signals. Human evaluators observe the agent's actions and provide feedback on their quality or appropriateness. This feedback helps shape the reward function. Figure 1-7 illustrates the three steps of the RLHF process as implemented by OpenAI.

[14] InstructGPT paper: https://openai.com/index/instruction-following/

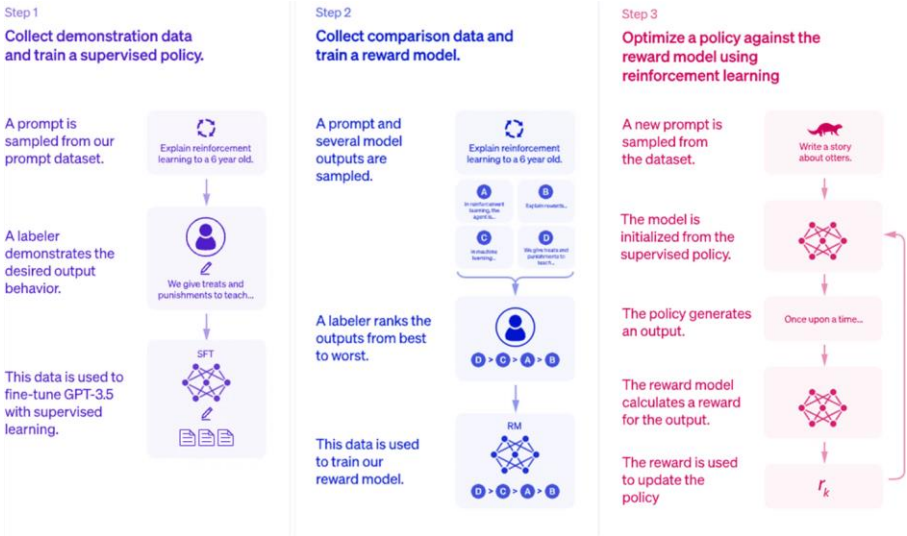


Figure 1-7 Humans are participating at different steps of the fine-tuning of ChatGPT

Here are the steps explained for the alignment of ChatGPT through reinforcement:

1. *Collect demonstration data and train a supervised policy:* Human feedback is used to label the training data. This is used to fine-tune the model.
2. *Collect comparison data and train a reward model:* Humans rate the output of the model from the previous step. This is captured in a reward function.
3. *Optimize a policy against the reward model:* This is now an iterative process without manual human input where the two previous steps are brought together to improve alignment.

GPT-4 AND BEYOND

In March 2023, OpenAI released GPT-4, marking another significant leap in the capabilities of large language models. GPT-4 introduced several key advancements.

GPT-4 demonstrates enhanced logical reasoning and problem-solving skills, performing better on complex tasks that require multi-step thinking. GPT-4 outperforms GPT-3.5 by scoring in higher approximate percentiles among test-takers as displayed in table 1.3 where we see the comparison of the performance of the two models.

Table 1-3 GPT-4 surpasses GPT-3.5 in its advanced reasoning capabilities

	GPT-3.5	GPT-4
Uniform Bar Exam	10th	90th
Biology Olympiad	31st	99th

Unlike its predecessors, GPT-4 can process both text and images as input. This allows it to analyze, describe, and reason about visual information in addition to text. This ability is called *multimodality*.

On top of this new modality, audio support came in May of 2024 and earned the model a new nickname, GPT-4o where "o" stands for "omni"

Finally, on a more practical note, GPT-4 can handle much longer inputs, with versions gradually able to process up first 8k tokens, then 32k tokens, to finally reach an astonishing 128k tokens in a single prompt. Later in the book, you will learn more about the capabilities of GPT-4.

In September 2024, a new series of models called o1 was introduced. Those models spend time *thinking* before answering to a question, making them more efficient in complex reasoning tasks, like science and programming. A good use case for this generation of models is to engineer a comprehensive prompt to prime the pump on the development of a new complete app.

Building Your Own LLM applications

This book describes different examples of LLM applications offering rich options and details. Some very clever programmers like Andrej Karpathy[15] have even gone so far as to build LLMs from scratch, or based on open-source implementations such as the Llama family of models (from Meta). This isn't the approach I take in this book, as it is way more involved and requires very deep AI and programming skills.

Instead, I will adopt a practical approach to building your own AI-powered applications. For this we will attempt to just get the right level of understanding of how LLMs are operating primarily by becoming familiar with the OpenAI application programming interface (API), and a few open-source alternatives.

[15] Let's build GPT: from scratch, in code, spelled out. https://www.youtube.com/watch?v=kCc8FmEb1nY

To explain the need for the API, you can consider the cases of using vs building AI-powered products. In the next chapter, you will learn how to use LLMs via API calls.

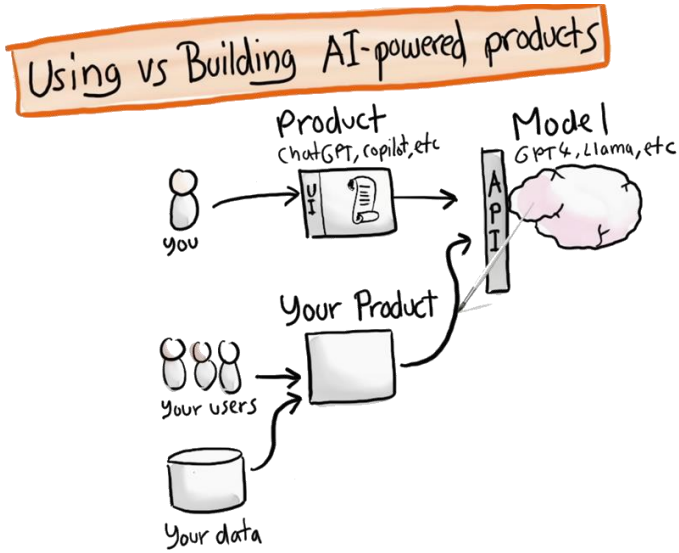


Figure 1-8 using vs building AI-powered products

2. OpenAI APIs

In this chapter you will discover the main Application Programming Interfaces (APIs) that OpenAI provides to interact with their GPT models and the MATLAB functions that enable to call those interfaces. As an application of your learning, you will build your very first chatbot with MATLAB. It will look like this:

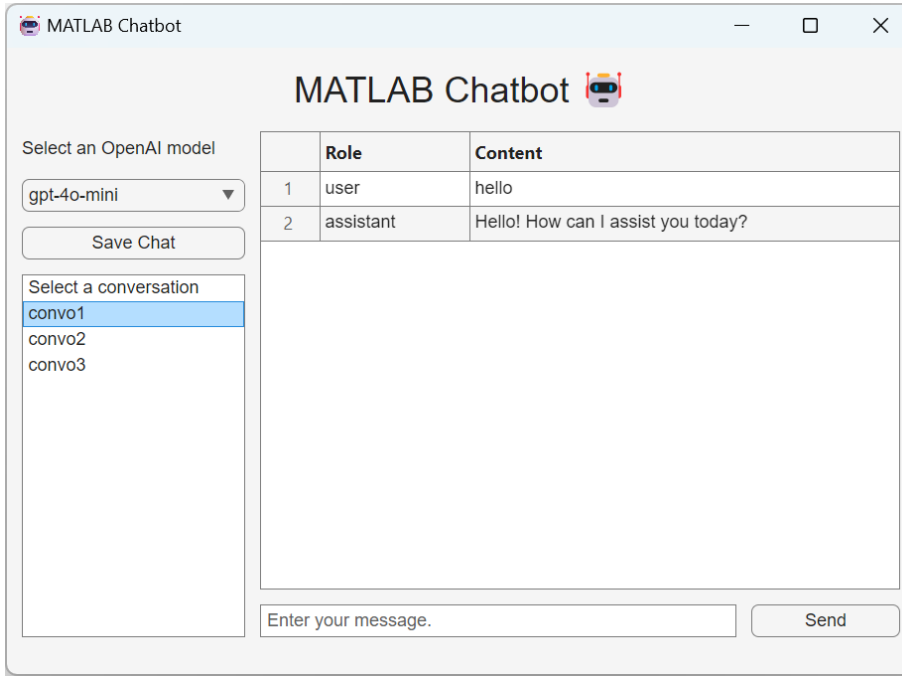


Figure 2-1 Your first chatbot app

Quick tour of the OpenAI developer platform

First you will need to create an OpenAI account on their developer platform (*platform.openai.com*). Once you are logged in to your OpenAI account, the landing page of the developer platform will take you to a playground that enables you to get access to different kinds of models. The main kind of models are chat models on which I will focus for this chapter. But you will see other kinds of models in the rest of the book, like the Assistant and TTS (Text-To-Speech) models. Completion models are now considered legacy.

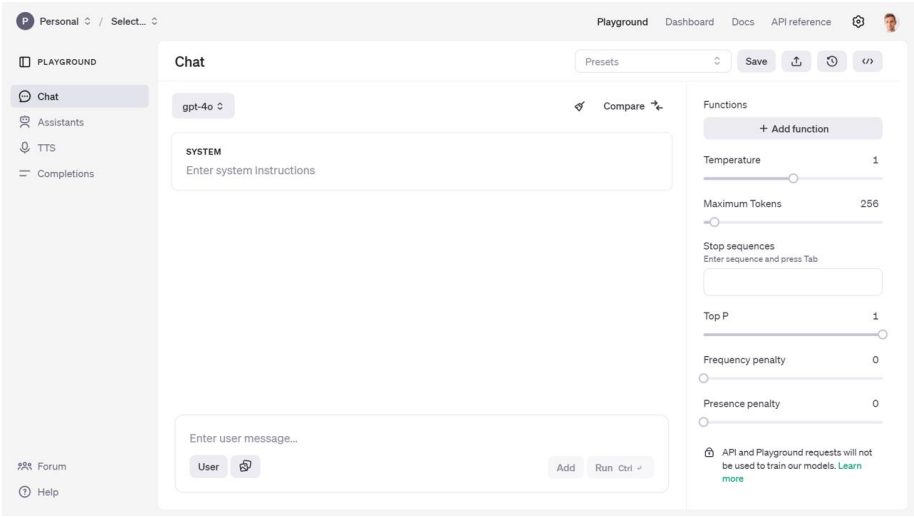


Figure 2-2 Play around in the playground and test the model APIs

This will give you an experience close to the one you have with the ChatGPT web app. But from there you will be able to access more advanced parameters of the model and view the code necessary to replicate the call to the API from your own program. If you are lacking inspiration, and you don't know where to start, you will find some prompt examples from the presets in the documentation[16].

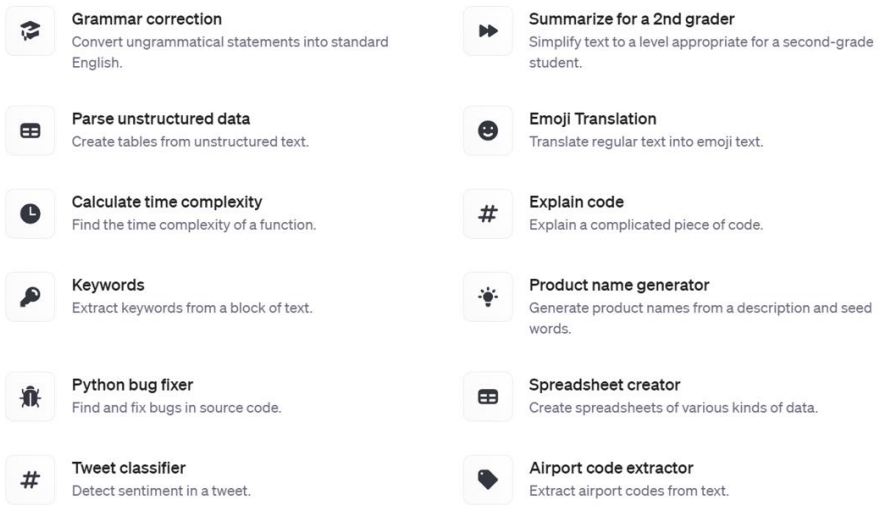


Figure 2-3 Prompt examples

[16] Prompt examples: https://platform.openai.com/docs/examples

If you want to take a guided look at the documentation[17] before trying anything or go lower level to the definition of the functions in the API reference[18].

Finally you can also access a dashboard to access management services:

- *Assistants*: to create and manage your own assistant models
- *Fine-tuning*: to fine-tune your own models
- *Batches*: to manage your batch jobs
- *Storage*: to manage files and vector stores
- *Usage*: to give you a sense of your consumption of the OpenAI web services.
- *API keys*: to manage your API keys

/! As an important data privacy disclaimer, API and Playground requests will not be used to train OpenAI models. This isn't the case of the public ChatGPT App, which by default can learn from users' conversations.

GETTING STARTED WITH THE CHAT COMPLETION API

A message to the chat completion API is made up of `role` and `content`. There are three distinct roles: system, user, and assistant.

- *System*: This is the initial instruction for the LLM that guides its subsequent responses and actions, known as "system prompt". It serves as context for the rest of the conversation, forcing the chat to behave in a certain way. Some developers have found success in continually moving the system message near the end of the conversation to keep the model's attention from drifting away as conversations get longer. You can set the system prompt when you initialize `openAIChat`.

```
 % Load environment variables and initialize the chat client with a
system prompt
 loadenv("path/to/.env");
 chat = openAIChat("If I say hello, say world", ...

      ApiKey=getenv("OPENAI_API_KEY"), ...
      ModelName="gpt-4o-mini");
```

[17] Doc: platform.openai.com/docs
[18] API Reference: platform.openai.com/api-reference

- *User*: This is the prompt a user provides to the LLM.
- *Assistant*: This is the response from the LLM.

To add your prompt, initialize `OpenAIMessages` and use `addUserMessage` to add your message.

```
% Initialize message history and add a user query
messages = messageHistory;
messages = addUserMessage(messages,"hello");
[txt,msgStruct,response] = generate(chat,messages);
disp(txt) % world
```

We can add the response to the messages using `addResponseMessage` to continue the conversation without losing the previous exchanges.

```
messages = addResponseMessage(messages, msgStruct);
```

Here is the content of `messages`:

```
>> messages.Messages{1}


ans =

    struct with fields:

       role: "user"
    content: "hello"


>> messages.Messages{2}


ans =

    struct with fields:

       role: "assistant"
    content: "world"
```

As you can see from the messages object passed to the OpenAI chat client, it consists of a cell array of struct, each entry containing a role (either *user* or *assistant*) and a content entry.

Let's save the messages in a .mat file in a "chat" folder for later use:

```
save(fullfile("chat","convo1.mat"),"messages");
```

You can convert the messages to a table for better visualization:

```
% Display the message history as a table
msgCells = messages.Messages;
msgStructArray = [msgCells{:}];
T = struct2table(msgStructArray);
disp(T)
```

The table output will look like:

Table 2-1 Chat History

	role	content
1	"user"	"hello"
2	"assistant"	"world"

The `messages` can contain as many *user* and *assistant* exchanges as you want, as long as you do not exceed the model's context window. For GPT-4o[19], the number of tokens accepted is 128k. Inputs and outputs tokens are summed, but outputs cannot exceed 16,385 tokens.

Additional parameters can be passed when creating the OpenAI Chat client or generating a response (like max number of tokens, number of responses to generate, and streaming option).

Prices of the API

The prices[20] have evolved quite a bit since the introduction of the ChatGPT API. As of the time of this writing (gpt-4o-2024-11-20):

[19] https://platform.openai.com/docs/models/gpt-4o
[20] https://openai.com/api/pricing

Table 2-2 Prices of the API

model	input price ($ per 1M tokens)	output price ($ per 1M tokens)
gpt-4o-mini	$0.150	$0.600
gpt-4o	$2.5	$10

/! Make sure to check the latest prices of the API, and consider how you could be submitting your request as a *batch*[21]. Responses will be returned within 24 hours for a 50% discount. If some of your input tokens are repeated across requests, they will be automatically *cached*[22] giving you 50% discount compared to uncached prompts.

To give you a sense of the cost of using the API on a daily basis, here is a view over the month of May, where I spend the most of my time writing this book and developing the associated GPTs.

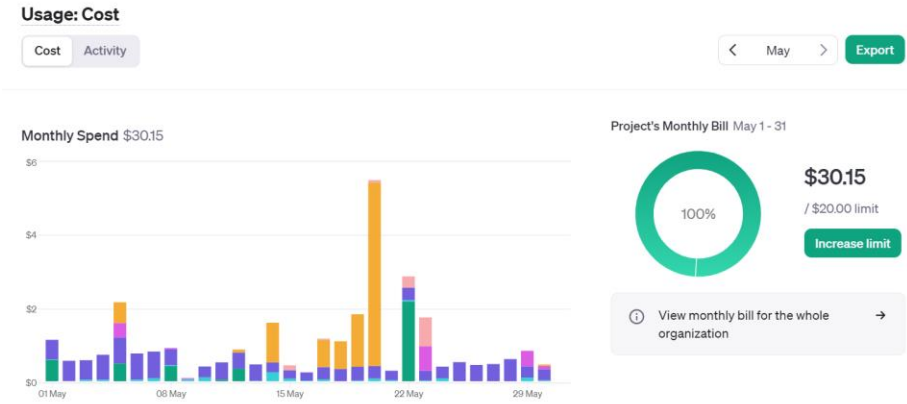


Figure 2-4 Cost of OpenAI services over the period of a busy month

If you hover over the graphic, you can see the breakdown by service (image, audio, embeddings, …) or by model (3.5, 4o, …). The big spike mid-month was due to the Dall-E 3 service usage for over $5 on 1 day.

[21] https://platform.openai.com/docs/guides/batch
[22] https://platform.openai.com/docs/guides/prompt-caching

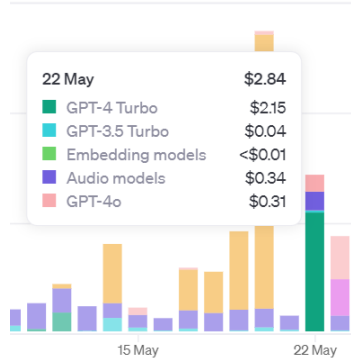


Figure 2-5 Breakdown of the cost per service

You can add email alerts and set budget limits to control your spending. As I started integrating more AI into my apps over the year, I ended up creating new keys for each project, and even distributing keys to friends and colleagues:

NAME	SECRET KEY	TRACKING ⓘ	CREATED	LAST USED ⓘ	PERMISSIO
Streamlit	sk-...hq8w	Enabled	Dec 5, 2022	May 30, 2024	All
Adam	sk-...5Mub	Enabled	Mar 22, 2023	May 30, 2024	All

Figure 2-6 Table of active OpenAI keys

This enabled me to have a finer grain control over the different projects including AI. You now have the ability to actually create a “project” that can contain members and have dedicated limits attached to it

BUILD YOUR FIRST CHATBOT APP

In this section, you will learn how to build your very first chatbot application, leveraging the OpenAI APIs. It will serve as a foundation for the subsequent apps that will be built throughout the book. You will start by experiencing the basics of building applications in MATLAB. The resulting app is stored on GitHub in the file *chatbot.m,* and I will walk you step by step through the development of such an app.

I will break down the development of the chatbot into the following steps:

1. *Graphical components*: Define the basic user interactions with the app
2. *Chat elements*: Display the chat conversation as a table in the main area, and implement convenience functions to save and load the chat history

3. *User data*: Store the state of your app in a user variable to simplify the interaction schema
4. *Response streaming*: Display the response as it is being generated

Graphical Components

This part is going to focus more on the MATLAB app-building framework. Typically, we use App Designer[23] to design MATLAB apps interactively. You can also create simple apps programmatically. Let's start with that approach. Figure 2-7 represents a simple app built in a MATLAB UI figure window.

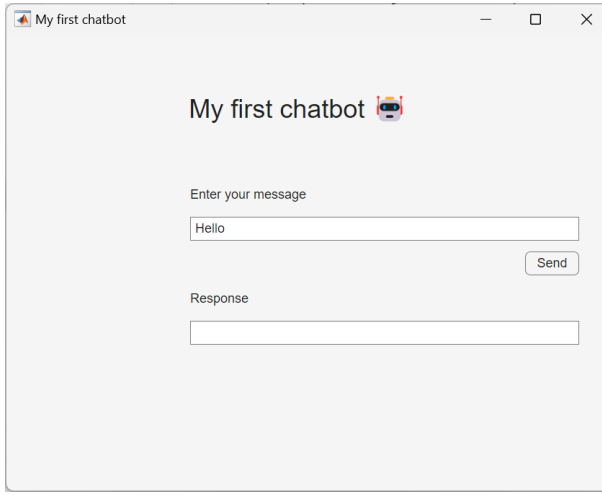


Figure 2-7 Simple chat user interface

This is the code for the first chat Graphical User Interface in MATLAB:

```
% create a UI figure window
fig = uifigure(Name="My first chatbot");
% add a 7x4 grid
g = uigridlayout(fig);
g.RowHeight = {'1x',22,22,22,22,22,'1x'};
g.ColumnWidth = {150,300,50,'1x'};
% add a title
ttl = uilabel(g,Text="My first chatbot 🤖");
```

[23] https://www.mathworks.com/help/matlab/app-designer.html

```
ttl.HorizontalAlignment = "center";
ttl.FontSize = 24;
ttl.Layout.Row = 1;
ttl.Layout.Column = [1,3];
% add an input field
eflabel = uilabel(g,Text="Enter your message");
eflabel.Layout.Row = 2;
eflabel.Layout.Column = 2;
ef = uieditfield(g);
ef.Layout.Row = 3;
ef.Layout.Column = [2,3];
ef.Value = "Hello";
% add an output field
oflabel = uilabel(g,Text="Response");
oflabel.Layout.Row = 5;
oflabel.Layout.Column = 2;
of = uieditfield(g);
of.Layout.Row = 6;
of.Layout.Column = [2,3];
% add a button
btn = uibutton(g,Text="Send") ;
btn.ButtonPushedFcn=@(src,event) chat(ef,of);
btn.Layout.Row = 4;
btn.Layout.Column = 3;

% this function runs when the button is clicked
function chat(inputField,outputField)
    systemPrompt = "If I say hello, say world";
    % modify this depending on which release you use
    client = openAIChat(systemPrompt, ...
            ModelName="gpt-4o-mini");
            % ApiKey=getenv("OPENAI_API_KEY")
    prompt = string(inputField.Value);
    [txt,msgStruct,response] = generate(client,prompt);
    if isfield(response.Body.Data,"error")
        error(response.Body.Data.error)
    else
        outputField.Value = txt;
    end
end
```

You will need to define all the local functions at the end of the script file unless you use R2024a or later (In the latest versions of MATLAB, you can define local functions anywhere in a script).

MATLAB apps are built on a UI figure[24] window. It is a good practice to add a UI grid layout[25] to the UI figure to organize UI components within the UI figure.

The basic elements that are useful for any basic app are the UI edit field[26] and the UI button[27], which lets you define the app behavior in a callback function.

As mentioned in the preface section about authentication, the API key is stored locally in a file *.env*. and loaded via the *setup.mlx* file in *preface/*. You also need to make sure that the LLMs-with-MATLAB repo is on the MATLAB path, as well as its subfolder, as part of the setup.

We can gradually increase the complexity of the app, by adding a UI dropdown[28] to choose the model *before* the UI button definition:

```
% add a dropdown before the button
items = ["gpt-4o-mini","gpt-4o"];
dd = uidropdown(g,Items=items);
dd.Layout.Row = 2;
dd.Layout.Column = 1;
```

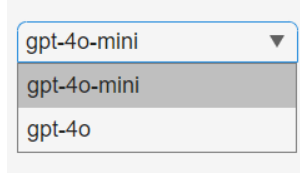


Figure 2-8 Model selection with drop-down menu

You also need to update the callback function in the UI button to include the UI dropdown to the input arguments.

```
% update the callback input arguments
btn.ButtonPushedFcn=@(src,event) chat(dd,ef,of));
```

Here is the updated function definition:

```
function chat(selection,inputField,outputField)
    systemPrompt = "If I say hello, say world";
```

[24] https://www.mathworks.com/help/matlab/ref/uifigure.html
[25] https://www.mathworks.com/help/matlab/ref/uigridlayout.html
[26] https://www.mathworks.com/help/matlab/ref/uieditfield.html
[27] https://www.mathworks.com/help/matlab/ref/uibutton.html
[28] https://www.mathworks.com/help/matlab/ref/uidropdown.html

```
    % modify this depending on which release you use
    client = openAIChat(systemPrompt, ...
         ApiKey=getSecret("OPENAI_API_KEY"), ...
        ModelName=selection.Value);
    prompt = string(inputField.Value);
    [txt,msgStruct,response] = generate(client,prompt);
    if isfield(response.Body.Data,"error")
        error(response.Body.Data.error)
    else
        outputField.Value = txt;
    end
end
```

A drop-down element hides the selection to only reveal the selected option. An alternative way to display multiple choices is to use radio buttons. This other type of graphical component enables a finite number of choice options to be displayed for an exclusive selection of one only (unlike checkboxes that enable multiple choices).

Chat Elements

The simple app you just built can only handle a single-turn chat. To support a multi-turn chat, you need to use a UI component that can show multiple lines of text. One such component is the UI table[29].

```
% create a ui figure window
fig = uifigure(Name="My conversation table");
% set the figure size
fig.Position(end) = 150;
% add a 3x1 grid
g = uigridlayout(fig,[3,1]);
g.RowHeight = {80,22,'1x'};
% add a table
uit = uitable(g);
uit.Data = ["user","hello";"assistant","world"];
uit.ColumnName = ["Role","Content"];
uit.Layout.Row = 1;
% add an input field
ef = uieditfield(g,Placeholder="Enter your message");
ef.Layout.Row = 2;
```

[29] https://www.mathworks.com/help/matlab/ref/uitable.html

As you can see from the structure of the code, the rendering of the messages will be done after a new message has been added to the Data property of the UI table. Please note that the input field is placed below the table with a placeholder text.

Figure 2-9 A basic chat dialog

In order to start building a chat that has more than one question and one answer (not much of a chat), you will save the conversation and load it back (making the app stateless and relying on storing the history to disk).

First let's start by loading an existing conversation, with a function called *load_chat* that loads a chat history for debugging:

```
function load_chat(listbox, outputField)
    historyfile = fullfile("chat", listbox.Value + ".mat");
    if isfile(historyfile)
        load(historyfile, "convo");
        roles = cellfun(@(x) string(x.role), convo.Messages');
        contents = cellfun(@(x) string(x.content), convo.Messages');
        outputField.Data = [roles, contents];
    else
        outputField.Data = [];
    end
end
```

The chat history is stored in a .mat file in the `chat` folder. The chat history is loaded from the file and then displayed in the output field.

You also need a function to populate a listbox with available filenames.

```
function list_history(inputField)
```

```
    if isfolder("chat")
        s = dir("chat");
        isMat = arrayfun(@(x) endsWith(x.name, ".mat"), s);
        filenames = arrayfun(@(x) string(x.name), s(isMat));
    else
        mkdir("chat");
        filenames = [];
    end
    items = "Select a conversation";
    if ~isempty(filenames)
        filenames = extractBefore(filenames, ".mat");
        items = [items, filenames'];
    end
    inputField.Items = items;
end
```

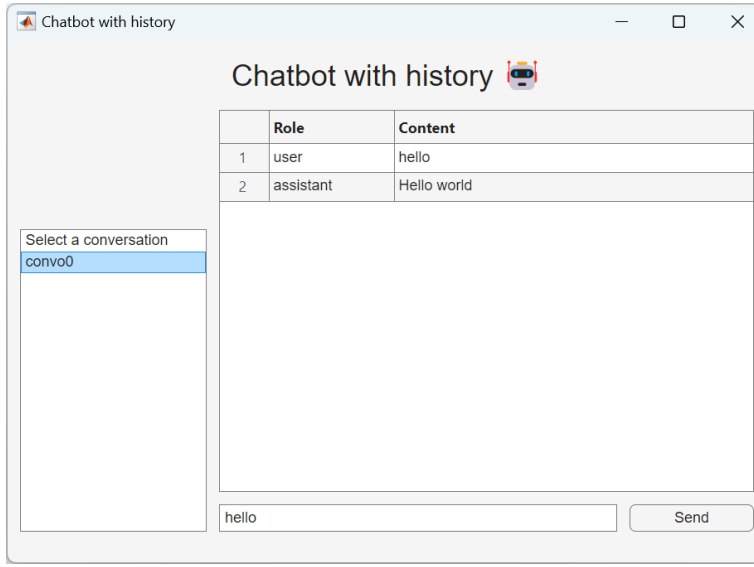


Figure 2-10 Conversation history stored as mat file

You can call those functions as follows.

```
% Chatbot with history

% Create the uifigure with minimal width.
```

```
initialHeight = 420; initialWidth = 610;
fig = uifigure("Name", "ChatGPT-like bot");
fig.Position = [100 100 initialWidth initialHeight];

% Add an 8x5 grid layout
g = uigridlayout(fig);
g.RowHeight = {'1x',22,22,22,'5x',22,22,5};
g.ColumnWidth = {150,100,200,'1x',100};

% Title label
ttl = uilabel(g, "Text", "ChatGPT-like bot 🤖");
ttl.FontSize = 24;
ttl.Layout.Row = 1;
ttl.Layout.Column = [1,5];
ttl.HorizontalAlignment = "center";

% Table to show the conversation
uit = uitable(g);
uit.ColumnWidth = {100, '1x'};
uit.ColumnName = ["Role", "Content"];
uit.Layout.Row = [2,6];
uit.Layout.Column = [2,5];

% Dropdown for conversation history
dd = uidropdown(g);
list_history(dd);
dd.ValueChangedFcn = @(src,event) load_chat(dd, uit);
dd.Layout.Row = 3;
dd.Layout.Column = 1;
```

Second let's define the chat function. While designing the behavior of the app, I'd recommend avoid calling OpenAI each time by creating a dummy chat function:

```
function dumb_chat(inputField, dropdown, convo)
    prompt = string(inputField.Value);
    convo = addUserMessage(convo, prompt);
    txt = "Hello world";
    msgStruct = struct("role", "assistant", "content", txt);
    convo = addResponseMessage(convo, msgStruct);
    save_chat(convo, "convo0");
    list_history(dropdown);
end
```

You need to output the conversation outside of the function. However, this function is called from a button click, so there is no direct way to output the conversation to use elsewhere. To solve this problem, you save the conversation to a local file using the `save_chat` function:

```
function save_chat(convo, filename)
    historyfile = fullfile("chat", filename + ".mat");
    save(historyfile, "convo");
end
```

You can implement additional UI components and logic to accept user input and save the conversation to a file.

```
% Input field for messages
ef = uieditfield(g, Placeholder="Enter your message.");
ef.Layout.Row = 7;
ef.Layout.Column = [2,4];

% Send button for the chat (run button)
sendBtn = uibutton(g, Text="Send");
sendBtn.ButtonPushedFcn    =    @(src,event)    dumb_chat(ef,    ld,
openAIMessages);
sendBtn.Layout.Row = 7;
sendBtn.Layout.Column = 5;
```

When you send your message by clicking on the button, the conversation is saved in the “convo1.mat” file and you can display it by selecting “convo1” in the dropdown.

User Data

In the previous section, you have seen how to save and load a conversation. But you don’t really want to be writing to disk every element of our app that needs to store a state. When you design an app, you need to store app data to be shared across various callback functions. The best way to store data is to use the `UserData` property of the top-level UI figure, because it is accessible from all other UI components. In

our example, you need to manage the state of the conversation as it is loaded, updated and saved.

By storing the conversation in the `UserData` property of the UI figure, you can access it from any child component using ancestor[30] function:

```
fig = ancestor(uicomponent,"figure","toplevel");
fig.UserData
```

Let's see how this works in the updated `dumb_chat` function:

```
function dumb_chat(inputField,outputField)
    % use ancestor to get the fig
    fig = ancestor(inputField,"figure","toplevel");
    % get the messages from the UserData property of the fig
    convo = fig.UserData;
    prompt = string(inputField.Value);
    if isempty(outputField.Data)
        outputField.Data = ["user",prompt];
    else
        outputField.Data = [outputField.Data;"user",prompt];
    end
    convo = addUserMessage(convo,prompt);
    txt = "Hello world";
    msgStruct = struct("role","assistant","content",txt);
    convo = addResponseMessage(convo,msgStruct);
    % update the UserData property of the fig
    fig.UserData = convo;
    outputField.Data = [outputField.Data;"assistant",txt];
    inputField.Value = "";
end
```

The `save_chat` function is no longer used in the `dumb_chat` function. Let's repurpose it so that it can be called from a button.

```
function save_chat(inputField,outputField)
    fig = ancestor(inputField,"figure","toplevel");
    convo = fig.UserData;
    if ~isempty(convo.Messages)
        % create a new filename based on the folder content
        s = dir("chat");
        isMat = arrayfun(@(x) endsWith(x.name,".mat"), s);
        filenames = arrayfun(@(x) string(x.name), s(isMat));
        if isempty(filenames)
```

[30] https://www.mathworks.com/help/matlab/ref/ancestor.html

```
            filename = "convo1";
        else
            filenames = extractBefore(filenames,".mat");
            suffix = str2double(extractAfter(filenames,"convo"));
            filename = "convo" + (max(suffix) + 1);
        end
        historyfile = fullfile("chat", filename + ".mat");
        save(historyfile,"convo");
        list_history(inputField);
    end
    % reset the UserData and output field
    fig.UserData = openAIMessages;
    outputField.Data = [];
end
```

Then you need to initialize the `UserData` property of the UI figure when you build the app:

```
% initialize the UserData property in the UI figure
fig = uifigure(Name="ChatGPT-like bot");
fig.UserData = openAIMessages;
```

Let's add a new button that called the `save_chat` function before the "Send" button.

```
% add a save chat button
saveBtn = uibutton(g,Text="Save Chat");
saveBtn.ButtonPushedFcn=@(src,events) save_chat(uit,dd);
saveBtn.Layout.Row = 4;
saveBtn.Layout.Column = 1;
```

You also need to update the input arguments for the callback function for the input field:

```
% modify the function input arguments
sendBtn.ButtonPushedFcn=@(src,event) dumb_chat(ef,uit);
```

Response Streaming

Finally, one small thing is missing to make the app look and feel just like ChatGPT. You will modify the call to the OpenAI chat completion API, by adding the parameter `StreamFun`.

Setting `StreamFun` in a client makes the model return tokens as soon as they are available, instead of waiting for the full sequence of tokens to be generated. It does not change the time to get all the tokens, but it reduces the time for the first token for an application where you want to show partial progress or are going to stop generations. This can be a better user experience and a UX improvement so it's worth experimenting with streaming.

Here is the code that replaces the `dumb_chat` function:

```
function chat_stream(dropdown,inputField,outputField)
    % modify this depending on which release you use
    client = openAIChat(ModelName=dropdown.Value, ...

        ApiKey=getSecret("OPENAI_API_KEY"), ...
        StreamFun=@(x) printStream(outputField,x));
    fig = ancestor(inputField,"figure","toplevel");
    convo = fig.UserData;
    prompt = string(inputField.Value);
    if isempty(outputField.Data)
        outputField.Data = ["user",prompt];
    else
        outputField.Data = [outputField.Data; "user",prompt];
    end
    convo = addUserMessage(convo,prompt);
    [txt,msgStruct,response] = generate(client,convo);
    convo = addResponseMessage(convo,msgStruct);

    if isfield(response.Body.Data,"error")
        error(response.Body.Data.error)
    else
        fig.UserData = convo;
        outputField.Data(end) = txt;
        inputField.Value = "";
    end
end
```

Additionally, you need to define the function for the `StreamFun` parameter:

```
function printStream(h,x)
    data = string(h.Data);
    if strlength(x) == 0
        data = [data; "assistant",string(x)];
    else
        data(end) = data(end) + string(x);
    end
```

```
    h.Data = data;
    pause(0.1)
end
```

Let's update the dropdown so that you can use it to choose a model:

```
% replace the dropdown
ddlabel = uilabel(g,Text="Select an OpenAI model");
ddlabel.Layout.Row = 2;
ddlabel.Layout.Column = 1;
dd = uidropdown(g,Items=["gpt-4o-mini","gpt-4o"]);
dd.Layout.Row = 3;
dd.Layout.Column = 1;
```

```
% replace dumb_chat with chat_stream
sendBtn.ButtonPushedFcn=@(src,events) chat_stream(dd,ef,uit);
```

Now you should be able to stream the response from the API.

Before you finish, you need to make one more change. Since you updated the dropdown to select a model, you lost the ability to load a chat history. Let's add a few more things to complete the app.

```
% add a listbox
lb = uilistbox(g);
list_history(lb);
lb.ValueChangedFcn = @(src,events) load_chat(lb,uit);
lb.Layout.Row = [5,7];
lb.Layout.Column = 1;
```

You also need to update some functions.

```
function load_chat(listbox,outputField)
    historyfile = fullfile("chat", listbox.Value + ".mat");
    if isfile(historyfile)
        load(historyfile,"convo");
        fig = ancestor(outputField,"figure","toplevel");
        fig.UserData = convo;
        roles = cellfun(@(x) string(x.role), convo.Messages');
        contents = cellfun(@(x) string(x.content), convo.Messages');
        outputField.Data = [roles,contents];
    else
        outputField.Data = [];
    end
```

```
end
```

Congratulations 🎉 You have created your first chatbot. If you assemble the learnings from the previous sections, you will be able to replicate the code in this example in *chatbot.m*.

3. PROMPTING, CHAINING AND SUMMARIZATION

Now that you have seen the basics of how to develop a chatbot application, let's discover how to go from a generic AI agent into more specific use cases. In this chapter, you will learn how to craft efficient prompts, to leverage advanced chaining methods, and to apply those methods to a concrete summarization application on texts, such as articles, books, or transcripts.

THE ART OF PROMPTING

This chapter is meant to give you an intuitive understanding of how to interact with GPT models by careful writing prompts, in such a way that the model can understand. This simplistic explanation is what is currently known as the art and science of *prompt engineering*.

Or as Monsieur Jourdain, the main character of Molière's *The Bourgeois Gentleman*, would describe it: speaking in prose without knowing it. Shortly after the release of ChatGPT, you may have seen a lot of buzz in articles online, as well as the new profession of "prompt engineer" that supposedly is the future of many jobs such as programmer. I won't engage much further on this debate, but I will instead attempt to list down the recipes that make a good prompt.

Here is a simple example of what could be described as a bad prompt, inherited from the age of googling and simple keyword search, with one quick fix that precise a little more the intent behind the request:

- *Ineffective Prompt*: "Quantum mechanics."
- *Effective Prompt*: "In simple terms, explain the basic principles of quantum mechanics to a high school student."

Creating an effective prompt involves clarity, context, and specificity.

- *Clarity*: Be explicit about the task.
- *Context*: Provide necessary background information.
- *Specificity*: Define the format and style of the desired response.

There are some additional techniques for improving responses:

- *Role Assignment*: Assign a role to guide perspective.

```
prompt = "You are a movie critic. Analyze the Star Wars saga."
```

- *Formatting Instructions*: Guide the structure of the output (you can specify the desired length of the response).

```
prompt = "List the top 3 best Star Wars movies in bullet points."
```

- *Style Guidelines*: Specify tone and style.

```
prompt = "Write a friendly email from Chewbacca to Han Solo."
```

An important part of effective prompting is providing the necessary context. There are different approaches to this that will be covered throughout the book. The simplest method that was introduced in the GPT-3 paper is few-shot prompting. Essentially, the idea is to provide examples of inputs and outputs from which the language model can generalize the pattern. This is an illustration for a translation task:

```
prompt=["Translate the following sentences from English to French:";
         "1. 'Hello, how are you?'";
         "-> 'Bonjour, comment ça va?'";
         "2. 'Good morning.'";
         "->"];
prompt = strjoin(prompt,newline)
```

Different variations of this method will be leveraged, and more generally you will see a mix of each of the techniques above used in some of the code snippets of the book that perform calls to GPT models.

PROMPT ENGINEERING TECHNIQUES IN APP DEVELOPMENT

You should not only think about agents as chatbots. Here is an example that illustrates an application requiring the use of several AI agents to manage and curate a database of bookmarks. The pitch is the following: As a tech professional, I often accumulate a vast collection of bookmarks for articles, tutorials, documentation, and blog posts. Managing this extensive library presents several challenges:

- *Insights*: Identifying trends or areas of focus within the collection.
- *Organization*: Keeping bookmarks categorized and easy to navigate.

- *Summarization*: Obtaining concise overviews without re-reading entire articles.
- *Retrieval*: Quickly finding relevant articles when needed (it will be covered in the next chapter).

Throughout this chapter, we will develop an app that:

1. *Extracts and format* content from each bookmarked article.
2. *Translates* all articles in a target language (here mostly from French to English).
3. *Classifies* articles into relevant tech categories (e.g., Machine Learning, Web Development).
4. *Summarizes* concise abstracts for quick reference, after identifying some key points.
5. *Scales* to thousands of bookmarks efficiently with prompt templates.

Extract and Format content

The first step is to fetch the content from each URL and extract the main text using web scraping tools.

```
%% Extracting Content from a URL
addpath("utils");

url = "https://blogs.mathworks.com/matlab/2025/02/04/how-to-run-local-deepseek-models-and-use-them-with-matlab";
% This function extracts the title and content from a given URL.
[title, content] = extractContent(url);
% Format the content using a custom function
formatted_content = formatContent(title, content)

% Extract the last part of the URL as filename
urlParts = strsplit(url, '/');
filename = urlParts{end};
outputFile = filename + ".md";

% Write title and content to the file
fid = fopen(outputFile, 'w');
fprintf(fid, '%s', formatted_content);
fclose(fid);
```

For this we will use the following function, that makes use of the extractHTMLText[31] function from the text analytics toolbox:

```
function [title, content] = extractContent(url)
    try
        % Make a web call with a custom timeout of 30 seconds
        options = weboptions('Timeout', 30);
        html = webread(url,options);
        tree = htmlTree(html);

        % Get the title element text
        t = findElement(tree, 'title');
        if ~isempty(t)
            title = extractHTMLText(t.Children);
        else
            title = "No Title Found";
        end
        % Extract body from the HTML tree
        body = findElement(tree, 'body');
        if ~isempty(body)
            content = extractHTMLText(body);
        else
            error("No body found in the HTML.");
        end

    catch ME
        fprintf("Error fetching %s: %s\n", url, ME.message);
        title = "";
        content = "";
    end
end
```

Based on this extracted content, you can format it properly and store it in a text file. Here is a function that uses OpenAI to format the content:

```
function formatted_content = formatContent(title, content)
        prompt = [
        "You are an expert text formatter and summarizer." ;
        "Here is an article:" ;
        "Title: " + title ;
        "Content:" ;
        content ;
```

[31] https://www.mathworks.com/help/textanalytics/ref/htmltree.extracthtmltext.html

```
        "Please format this scraped text into a clean, readable
content."];
        prompt = strjoin(prompt, newline);
    % Call the bot function with the constructed prompt
    formatted_content = bot(prompt);
 end
```

For any of the operations requiring AI in this chapter, let's implement a generic function using the OpenAI API:

```
function [text, response] = bot(prompt, temperature)
    % Set defaults if not provided
    if nargin < 3, temperature = 0; end

    % Load environment settings (if needed)
    loadenv("../.env");

    % Define the model name (as in our chap1 example)
    modelName = "gpt-4o-mini";
    chat = openAIChat("You are a MATLAB expert.", ...
            "ModelName", modelName, ...
            "Temperature", temperature, ...
            "TimeOut", 30 ...
    );

    % Initialize message history and add the user prompt
    messages = messageHistory;
    messages = addUserMessage(messages, prompt);

    % Generate the response
    [text, response] = generate(chat, messages);
end
```

/! The temperature is set to 0 by default as most of the operations you will be performing in this example do not require much creativity.

Translate articles: Zero-shot Prompting

An important step to standardize all the content for the knowledge base is to translate all the articles in English, or in another language like French. Luckily for us, translation is one of the natural language processing tasks in which language models excel, especially as they are getting quite large and capable in multiple languages.

Here is a short function that uses a simple prompt, without the need to give any illustrative example:

```
%% Translate Articles
addpath("utils");
content = readlines("how-to-run-local-deepseek-models-and-use-them-with-matlab.md");
content = strjoin(content, newline);
translation = translateArticle(content, "French")
fid = fopen('traduction.md', 'w');
fprintf(fid, '%s', translation);
fclose(fid);
```

/! You actually do not need to translate in order to perform further processing like classification from different languages. The GPT models are able to reason around different language inputs. But this is much more convenient to start with this if there is a need to involve a human in the loop to verify the results of each step of the pipeline.

Classify articles: Few-Shot Prompting

Classification is a perfect illustration of few-shots prompting. This concept introduced in chapter 1 with the GPT-3 paper demonstrates how giving one or several examples of the expected result can greatly increase the performance of the prediction.

In this case, you can provide examples to help the model classify articles into predefined categories.

```
%% Classifying Articles
addpath("utils");
options = weboptions('Timeout', 30);
html = webread("https://blogs.mathworks.com/matlab/2024/10/02/4-ways-of-using-matlab-with-large-language-models-llms-such-as-chatgpt-and-ollama", options);
tree = htmlTree(html);
```

```
 body = findElement(tree, 'body');
 content = extractHTMLText(body);
 category = classifyArticle(content)
```

```
 function category = classifyArticle(content)
     prompt = ["You are an AI assistant that classifies technical
articles into categories." ;
         "The categories are: " ;
         "- artificial intelligence (AI)" ;
         "- Data Science" ;
         "- High Performance Computing" ;
         "- Python" ;
         "- Quantum Computing" ;
         "- MATLAB Online" ;
         "- New Releases"
         "- Open Source" ;
         "- Others" ;
         "For example:" ;
         "Article: How to run local DeepSeek models and use them with
MATLAB" ;
         "Category: artificial intelligence (AI)" ;
         "Article: We've been listening: MATLAB R2025a Prerelease
update 5 now available" ;
         "Category: New Releases" ;
         "Now classify this new content" ;
         content;
         "Category: "];
     prompt = strjoin(prompt, newline);
     category = bot(prompt);
 end
```

/! One way to make the result of this function easier to process would be to leverage the JSON mode[32].

[32]Structure outputs in JSON: https://github.com/matlab-deep-learning/llms-with-matlab/blob/main/doc/OpenAI.md#json-formatted-and-structured-output

SUMMARIZE ABSTRACTS: CHAIN-OF-THOUGHT PROMPTING

As we compare GPT models to the human brain, one concept that enables us to draw this parallel is *Chain-of-Thoughts*[33]. It is typically expressed with a prompt that contains an expression such as:

"Think step by step before you answer"

This way the model is using additional tokens in his answer to "spell out" some of its thinking and can reflect on it in a later stage. In our case, you can encourage the model to think step by step to extract relevant key points from the article.

```
%% Summarizing Articles
addpath("utils");
content = readlines("how-to-run-local-deepseek-models-and-use-them-with-matlab.md");
content = strjoin(content, newline);
summary = summarizeArticle(content)
```

```
function summary = summarizeArticle(content)
    keyPointsPrompt = [
        "You are an AI assistant that summarizes technical articles. Read the article below and think through " ;
        "the main points step by step before writing the final summary.";
        "Article:";
        content;
        "First, outline the key points and main ideas of the article.";
        "Then, write a concise summary incorporating these points."];
    keyPointsPrompt = strjoin(keyPointsPrompt, newline);
    keyPoints = bot(keyPointsPrompt)

    summaryPrompt = [
        "Extract the summary only.";
        "Key Points + Summary:";
        keyPoints;
        "Summary:"];
    summaryPrompt = strjoin(summaryPrompt, newline);
    summary = bot(summaryPrompt);
end
```

[33] Chain-of-Thought:https://blog.research.google/2022/05/language-models-perform-reasoning-via.html

If you uncomment the part of the function outputting key points, you will see how those are taken into account in crafting the summary.

```
### Key Points and Main Ideas

     1. **Introduction to DeepSeek Models**: The article discusses
the use of the DeepSeek-R1 AI models in MATLAB, particularly focusing
on the smaller model, deepseek-r1:1.5b.

     2. **Installation Steps**:
        - **Ollama Installation**: Users need to download and install
Ollama from its official website.
        - **Running the Model**: After installation, the model can be
run using a command in the command line.

     3. **Setting Up MATLAB**:
        - Users should install the "Large Language Models (LLMS) with
MATLAB" add-on through the MATLAB environment.

     4. **Creating an `ollamaChat` Object**:
        - The article provides a MATLAB command to create a chat
object that interacts with the DeepSeek model, detailing its
properties.

     5. **Generating Responses**:
        - The author demonstrates how to generate responses from the
model using a sample question about the speed of light.
        - The model's responses are shown to vary with each query,
highlighting its ability to provide detailed and complex answers.

     6. **Conclusion**: The author encourages readers to experiment
with the model, emphasizing its capabilities and the engaging nature
of LLM-based AI technology.

     ### Summary

     The article by Mike Croucher provides a guide on how to run the
DeepSeek-R1 AI models, specifically the smaller version deepseek-
r1:1.5b, using MATLAB. It outlines the installation process for
Ollama and the necessary MATLAB add-on for large language models.
After setting up, users can create an `ollamaChat` object to interact
with the model and generate responses to queries. The author
illustrates this with an example question about the speed of light,
showcasing the model's ability to produce varied and detailed
```

```
answers. The article concludes by encouraging readers to explore the
model's capabilities, highlighting its potential in the realm of AI
technology."
```

Summarization is one of the most useful applications of language models. And you can get very varied results based on how you approach this task. You can consider using the o1 family of models for this type of task as they excel at reasoning, and stuff everything into one single prompt. But on the other hand of the spectrum, it doesn't necessarily take a very advanced or large model to only perform summarization, and you can consider farming this out to dedicated services with small language models.

Whenever you involve humans in a language processing task, summarization is a key ingredient, so make sure that you validate the results to make sure that the quality of the summary aligns with the level of information that you want to present to your users.

/!\ You might hit up against the context length. You will learn in the following sections of this chapter how to work around this.

Scale processing: Prompt templates

You can process bookmarks in batches to handle large volumes efficiently. Remember that in such cases you can use the batch API from ChatGPT to pay 50% less, if you don't need the result right away. You can also consider parallelizing this processing loop, as each call can be performed independently.

```
%% Scale Processing of bookmarks

bookmarks = ["https://blogs.mathworks.com/matlab/2025/03/20/weve-
been-listening-matlab-r2025a-prerelease-update-5-now-available";
        "https://blogs.mathworks.com/matlab/2025/02/04/how-to-run-
local-deepseek-models-and-use-them-with-matlab";
        "https://blogs.mathworks.com/matlab/2024/10/02/4-ways-of-
using-matlab-with-large-language-models-llms-such-as-chatgpt-and-
ollama";
        "https://blogs.mathworks.com/matlab/2024/09/26/matlab-now-
has-over-1000-functions-that-just-work-on-nvidia-gpus"];
```

```
 processedArticles = struct('title', {}, 'category', {}, 'summary',
{});
 for i = 1:numel(bookmarks)
     % Extract current bookmark info
     bookmark = bookmarks(i);

     % Replace these "extract", "format", etc. with real
implementations
     [title, content] = extractContent(bookmark);
     formattedContent = formatContent(title, content);
     category         = classifyArticle(formattedContent);
     summary          = summarizeArticle(formattedContent);

     % Populate a struct with the processed results
     articleData.title    = title
     articleData.category = category
     articleData.summary  = summary

     % Append to the results array
     processedArticles(end+1) = articleData;
 end
 % Save the processed articles to a JSON file
 jason = jsonencode(processedArticles, PrettyPrint=true);
 writematrix(jason, 'processed_articles.json', 'FileType', 'text');
```

Chaining & Summarization

Why Chaining

Chaining is a technique that allows you to send multiple requests to an LLM in a sequence, using the output of one request as the input of another. This way, you can leverage the power of an LLM to perform complex tasks that require multiple steps, such as summarization, text simplification, or question answering.

The benefits of chaining are the following:

- *Enhanced contextual understanding*: By chaining calls, the LLM can maintain context over a longer interaction, leading to more coherent and relevant responses.
- *Complex task handling*: Chaining allows for the breakdown of complex tasks into simpler sub-tasks, which the model can handle more effectively.

Most of the chaining frameworks that appeared in 2023 came to complement LLMs like ChatGPT or open-source alternatives such as Llama. But it is good to note that with new features enabled by OpenAI and other LLM builders throughout the years, the need for frameworks became less and less relevant as OpenAI and others have increasingly been providing end-to-end integrated solutions. I'll cover one framework in particular in this section: LangChain 🦜🔗.

LangChain is a framework that enables developers to chain calls to Large Language Models (such as GPT models). It offers implementations both in Python and Typescript (to migrate easily your experimentations into a JavaScript web app).

Many of the concepts introduced in the early days of the LangChain framework were meant to complement the Large Language Models such as GPT 3.5:

- Extend the length of the context
- Augment generation by retrieving elements not in the context
- Add tools such as loading documents, searching the internet or doing basic math (more about tools in chap 5 about agents)

Here is an example of Python script leveraging LangChain for summarization[34]:

```
# langchain_summarization.py

from langchain_community.document_loaders import WebBaseLoader
from langchain.chat_models import init_chat_model
from langchain.chains.combine_documents import
create_stuff_documents_chain
from langchain.chains.llm import LLMChain
from langchain_core.prompts import ChatPromptTemplate
# from dotenv import load_dotenv

# Set env var OPENAI_API_KEY or load from a .env file
# load_dotenv()

# Load documents from the web
loader = WebBaseLoader("https://lilianweng.github.io/posts/2023-06-
23-agent/")
docs = loader.load()

llm = init_chat_model("gpt-4o-mini", model_provider="openai")

# Define prompt
prompt = ChatPromptTemplate.from_messages(
```

[34] Summarize Text with LangChain: https://python.langchain.com/docs/tutorials/summarization/

```
    [("system", "Write a concise summary of the
following:\\n\\n{context}")]
)

# Instantiate chain
chain = create_stuff_documents_chain(llm, prompt)

# Invoke chain
result = chain.invoke({"context": docs})
print(result)
```

In order to make this code run on your machine (or on MATLAB Online), you will need the following dependencies:

```
pip install tiktoken bs4 langchain-community langchain-openai
```

This Python script can be called from MATLAB:

```
pyrunfile("langchain_summarization.py")
```

You will see in the next section how to implement those functionalities directly in MATLAB. To learn more about LangChain, I would recommend the Youtube channel of Greg Kamradt[35].

We will break down the analysis of frameworks over the course of the next chapters:

- First, we will look at the basics of chaining to go around the limitations of the LLM context
- Second, we will look at methods such as vector search to extend the LLM context
- Third, we will look at properties that appear when we extend the reasoning of LLMs

Working around the context length

Summarization is one of the main use cases of Natural Language Processing that is used productively in a professional setting. The first application that came to my mind to leverage ChatGPT was summarizing meetings.

[35] Langchain tutorials:
https://youtube.com/playlist?list=PLqZXAkvF1bPNQER9mLmDbntNfSpzdDIU5
https://github.com/gkamradt/langchain-tutorials

Now there is one problem with this: a typical one-hour meeting is around 8k to 10k tokens. But initially the GPT-3.5 model powering ChatGPT could only handle 4096 tokens.

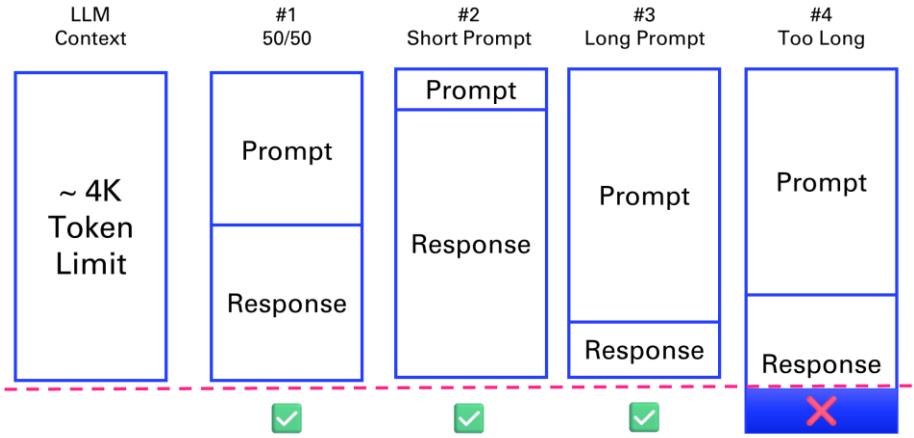


Figure 3-1 scenarios where the prompt and response fit the context length and 1 where it doesn't

What are *tokens* anyway? A helpful rule of thumb is that one token generally corresponds to ~4 characters of text for common English text. This translates to roughly ¾ of a word (so 100 tokens ~= 75 words). In what follows, we will use the tiktoken package from OpenAI anytime we need to compute precisely the number of tokens within our applications.

What does this mean in terms of pages? One page in average is 500 words[36] (like the first page of the forewords in Letter format 8.5" x 11"). Which means that 4k tokens can fit 6 pages of text. As the previous diagram suggests, this context length is shared between the prompt and the response.

Let's illustrate this with a basic example of a YouTube transcript. For this, we will need the following Python package:

```
pip install youtube_transcript_api
```

```
% What Is Sentiment Analysis?
video_id = "hkwuaRdwEeI"
% Retrieve the transcript for the specified video ID using Python
tr2 = py.utils.youtube.get_transcript(video_id);
```

[36] Words per page: https://wordcounter.net/words-per-page

You can call this function from Python first to check that it is outputting the right result before integrating it with MATLAB.

We can save the transcript as a plain text file.

```
% Save the transcript string into a text file
fileID = fopen('transcript.txt', 'w');
fprintf(fileID, '%s\n', tr);
fclose(fileID);
```

Let's count the numbers of token in the conversation, first by a simple approximation, second by using the tiktoken Python package.

```
% Reload the transcript for the first video
tr = fileread('transcript.txt');
tr = string(tr);
simpleTokenCounter(tr)
```

```
ans = 1177
```

```
function n= simpleTokenCounter(str)
    % Approximate token count: 1 token per 4 characters
    n = ceil(length(char(str)) / 4);
end
```

```
tiktoken = py.importlib.import_module('tiktoken');
encoding = tiktoken.encoding_for_model('gpt-4o-mini');
tok = encoding.encode(tr)
```

```
tok =
  Python list with values:

    [61335, 616, 679, 22171, 9247, 2360, 6247, 6439, 395, 13369, ...

    Use string, double or cell function to convert to a MATLAB array.
```

```
length(tok)
```

```
ans = 865
```

In the next parts, we will go over different chain types to address the limitation of the context length[37].

STUFFING

The first solution presented is "stuffing". If a document is within the model's context limit (4k tokens in figure 3-6), it can be directly fed into the API for summarization. This method is straightforward but limited by the maximum context length.

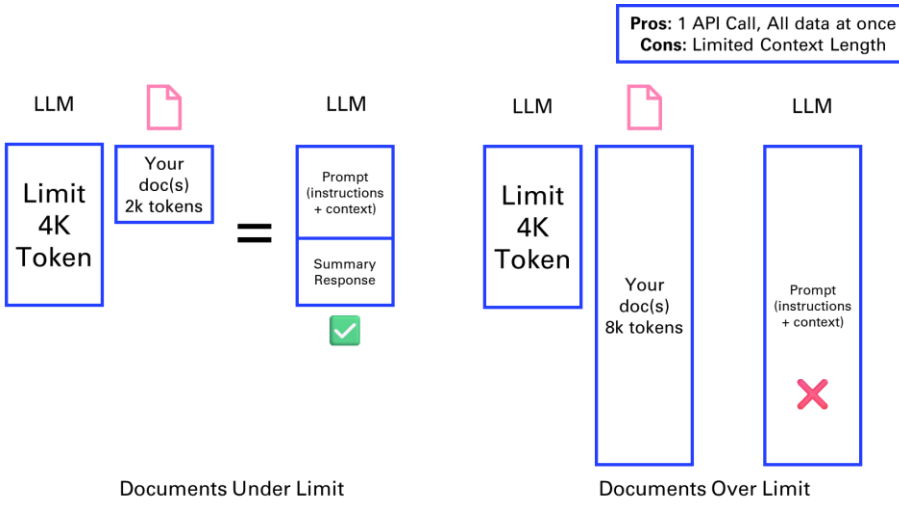


Figure 3-2 Stuffing a document into the context

You have seen in the previous section how the code would look like with LangChain.

MAP REDUCE

The MapReduce method involves splitting a large document (e.g., 8k tokens) into smaller chunks, summarizing each separately, and then combining these summaries into a final summary. This approach allows for processing larger documents in parallel API calls but may lose some information.

[37] Workaround Token Limits With Chain Types: https://www.youtube.com/watch?v=f9_BWhCI4Zo

This is a term that some might be familiar with, from the space of big data analysis, where it represents a distributed execution framework that parallelized algorithms over data that might not fit on a single core.

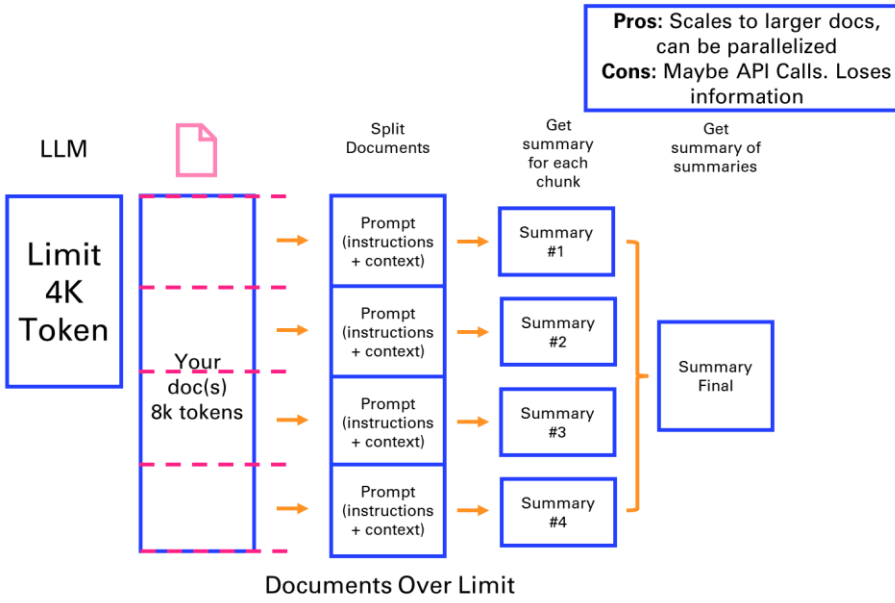


Figure 3-3 Map reduce approach to breaking down a summary into parallel steps

Let's reuse the previous example to analyze what happens at each steps if we break down the text into chunks of 100 tokens.

```
[finalSummary, chunkSummaries] = mapReduceSummarize(tr, 100)
```

```
chunkSummaries = 1x9 string

"Humans have communicated through natural language for m
illennia, docum…  " …
```

```
function [finalSummary, chunkSummaries] = mapReduceSummarize(text,
chunkSize)
    if nargin < 2, chunkSize = 1000; end

    % Split text into words and group them into chunks based on token
count.
    words = split(string(text));
```

```
    currentChunk = "";
    chunks = strings(0,1);  % Initialize as an empty string array

    for i = 1:length(words)
        numTokens = py.utils.tokenization.num_tokens(currentChunk +
" " + words(i));
        % Estimate tokens when appending the next word.
        if double(numTokens) < chunkSize
            currentChunk = currentChunk + " " + words(i);
        else
            chunks(end+1) = strtrim(currentChunk);   % Append to
string array
            currentChunk = words(i);
        end
    end
    if currentChunk ~= ""
        chunks(end+1) = strtrim(currentChunk);
    end

    % Summarize each chunk using a string array for summaries.
    chunkSummaries = strings(size(chunks));   % Preallocate as a
string array
    for i = 1:length(chunks)
        prompt = ["Summarize the following text:", ...
                  chunks(i), ...
                  "Summary:"];
        prompt = strjoin(prompt, newline);
        [chunkSummaries(i), ~] = bot(prompt);
        % Display each step of the Mapping process
        disp(chunkSummaries(i))
    end

    % Combine the chunk summaries into a final summary.
    combinedText = strjoin(chunkSummaries, newline);
    finalPrompt = ["Combine the following summaries into a coherent
final summary:", ...
                   combinedText, ...
                   "Final Summary:"];
    finalPrompt = strjoin(finalPrompt, newline);
    [finalSummary, ~] = bot(finalPrompt);
end
```

REFINE

In the Refine method, the document is split into chunks, and each chunk is summarized sequentially, with each summary informed by the previous one. This method provides relevant context but is time-consuming due to its sequential nature. This is one of my favorite methods for summarizing.

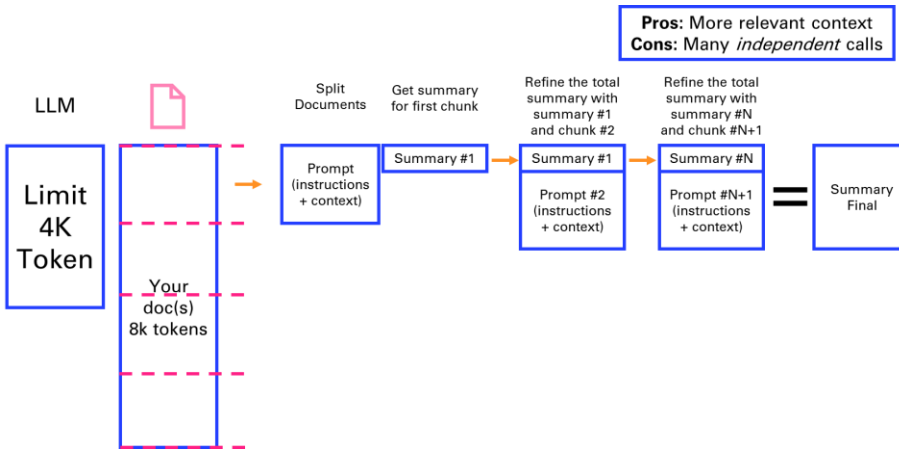


Figure 3-4 Refining a summary in sequential steps

We can directly try this method on the same content as the previous section and see how it compares.

```
finalSummary= refineSummarize(tr, 300)
```

```
finalSummary =

    "Humans have long used natural language to express thoughts and
feelings, and recent technological advancements enable the automation
of analyzing written language through sentiment analysis. This process
classifies the emotional tone of text as positive, negative, or
neutral, which can be challenging due to factors like slang, sarcasm,
and context. Sentiment analysis is widely applied in various fields,
such as trading strategies and market research, helping businesses
make informed decisions by tracking user sentiment from product
launches.


     To automate sentiment analysis effectively, it is crucial to
prepare the data and select an appropriate classifier. This involves
structuring intricate language data for processing, with a key step
being tokenization, which converts text into tokens—sequences of
characters that serve as features for the classifier. Depending on the
```

type and predictive power of the chosen classifier, additional preprocessing may be necessary.

There are two primary approaches to sentiment analysis: rule-based systems and AI models. Rule-based sentiment analysis relies on predefined rules, such as compiling lists of positive and negative words and counting their occurrences in the text. While these systems are straightforward to implement, they lack the flexibility and accuracy of AI models. In contrast, AI models, particularly those built using supervised machine learning algorithms, can learn explicit and implicit patterns in the data. These models are trained on large datasets with corresponding labels and can utilize classical algorithms like decision trees or advanced deep learning architectures, such as convolutional neural networks.

A notable advancement in this field is the use of large language models (LLMs), such as BERT and GPT. These models leverage transformer architectures to capture complex relationships between words and the nuances of human language, making them particularly effective for sentiment analysis. However, LLMs are large and complex, requiring substantial training data and high-end hardware, which can be a barrier for some applications. Fine-tuning a pretrained model, such as adding additional layers to BERT and retraining it with a smaller dataset, can help overcome these challenges while still achieving high accuracy. Despite the benefits of fine-tuning, the inherent complexity of large models can make their decision-making processes difficult to interpret.

Ultimately, the choice of classifier depends on the specific requirements of your application. Depending on your needs, you may opt for a simpler model or employ techniques that enhance interpretability. By leveraging AI, sentiment analysis can automate the classification of emotions in large text datasets, accurately addressing diverse and challenging scenarios. For further insights into natural language processing, machine learning, deep learning, and the application of AI in MATLAB for engineering tasks, check out our AI video playlist."

```
function refinedSummary = refineSummarize(text, chunkSize)
    if nargin < 2, chunkSize = 1000; end

    % Split the text into chunks (using a simple word-based splitter)
```

```
    words = split(text);
    currentChunk = "";
    chunks = strings(0,1);  % Initialize as an empty string array
    for i = 1:length(words)
        numTokens = py.utils.tokenization.num_tokens(currentChunk +
" " + words(i));
        if double(numTokens) < chunkSize
            currentChunk = currentChunk + " " + words(i);
        else
            chunks(end+1) = strtrim(currentChunk);
            currentChunk = words(i);
        end
    end
    if currentChunk ~= ""
        chunks(end+1) = strtrim(currentChunk);
    end

    % Start with a summary of the first chunk
    prompt = ["Summarize the following text:";
              chunks(1);
             "Summary:"];
    prompt = strjoin(prompt, newline);
    [refinedSummary, ~] = bot(prompt);

    % Refine the summary with each subsequent chunk
    for i = 2:length(chunks)
        refinedPrompt = ["Refine the following summary with this
additional context:";
            "Context:";
            chunks(i);
            "Existing Summary:";
            refinedSummary;
            "Refined Summary:"];
        refinedPrompt = strjoin(refinedPrompt, newline);
        [refinedSummary, ~] = bot(refinedPrompt);
    end
 end
```

EVOLUTIONS OF THE CONTEXT WINDOW

This early view of limitations of Large Language Models has to be updated over time as the context length of new generations of LLMs are growing:

- GPT 4[38] (introduced on March 14, 2023) originally had an 8k tokens context window. A 32k tokens version was made available gradually in parallel. In November 2023, an 128k token version was introduced.
- GPT 3.5 got a 16k version in September 2023.
- Anthropic Claude[39] extended its context window to 100k tokens in May 2023, enabling to process an entire book like the Great Gatsby of around 200 pages (standard paperback edition).
- Claude 2.1[40] (Nov 2023) and Claude 3 (Mar 2024) propose an impressive 200k tokens context, as well as Claude 3.5 and 3.7.
- LLaMa[41] from Meta initially came with a 2k tokens context window in February 2023.
- Llama2[42] released in July 2023 has a 4k tokens context window. Llama3 released in April 2024 doubles it again to 8k tokens. This is considered one of the leading open-source LLM.
- GPT 4o[43] (introduced on May 13, 2024) has a 128k tokens context window.
- GPT 4.1[44] (introduced on April 14, 2025) can process up to 1 million tokens.

There are still advantages to apply the previous methods, either to mitigate the cost of long context windows and larger models, or to use smaller more dedicated models that can be open-source for certain specific tasks like summarizing.

[38] GPT 4 model: https://platform.openai.com/docs/models/gpt-4
[39] Anthropic Claude 100k context window: https://www.anthropic.com/index/100k-context-windows
[40] Claude 2 & 3: https://www.anthropic.com/news/claude-2-1 - https://www.anthropic.com/news/claude-3-family
[41] LLaMa: https://ai.meta.com/blog/large-language-model-llama-meta-ai/
[42] Llama 2 & 3: https://llama.meta.com/llama2/ - https://llama.meta.com/llama3/
[43] https://openai.com/index/hello-gpt-4o/
[44] https://openai.com/index/gpt-4-1/

4. VECTOR SEARCH & QUESTION ANSWERING

Another very popular application of Large Language Models is question answering. Unlike summarization that can be solved with a fixed context, answering questions can be a moving target.

You probably remember ChatGPT saying that he did not have access to information after a certain date (like January 2022 in Figure 4-1). This date is gradually updated by OpenAI but there is always a lag.

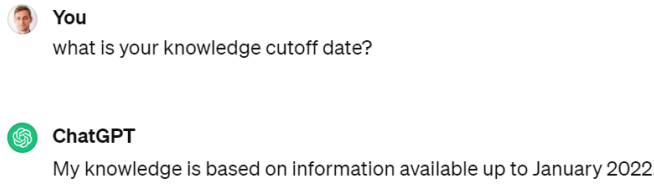


Figure 4-1 A typical response from ChatGPT mentioning is knowledge limitations

This is why you will investigate methods to extend the context with additional tools such as search, and introduce a special kind of search called similarity search over a vector database of embeddings.

RETRIEVAL-AUGMENTED GENERATION

Instead of directly getting an answer from an LLM, another approach consists first in retrieving the meaningful context for the request to the LLM. This is called *Retrieval-Augmented Generation*.

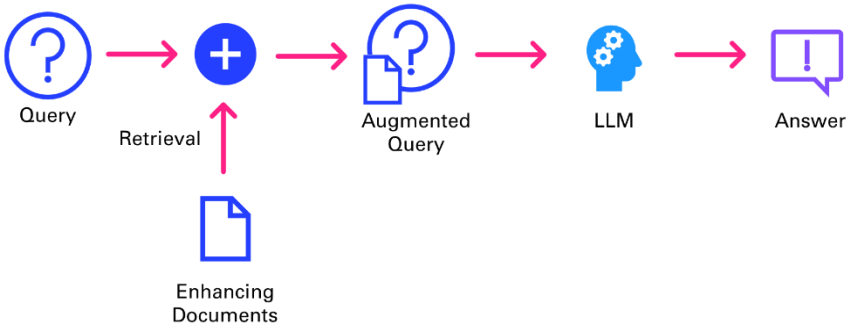


Figure 4-2 Retrieval Augmented Generation workflow

As described in Figure 4-2, the query isn't directly answered be the LLM. It first goes through an information retrieval system to pull the relevant context that will be fed to the LLM alongside the original query, to deliver an answer grounded in an extended knowledge base.

TRADITIONAL SEARCH

One of the first applications of ChatGPT was to answer questions that you previously were searching on the internet, via a search engine like Google. After all, a large language model is nothing more than a (very) compressed version of the internet, as it is the corpus on which they are primarily trained.

In this chapter, you will implement a very simple search tool that will input information from a search engine into our AI request. This way, the chatbot will be able to answer news related questions such as "Who is the CEO of Twitter?".

```
prompt = "Who is the CEO of Twitter?";
model = openAIChat(ModelName="gpt-4o-mini");
response = generate(model,prompt)
```

```
response = "As of my last update in October 2023, Elon Musk was the
CEO of Twitter. However, please verify with up-to-date sources, as
leadership positions can change frequently."
```

ChatGPT is at least taking the precaution of stating the date of its recollection. But if we check on DuckDuckGo, we can see that things have changed quite a bit at Twitter since 2023 (even the name).

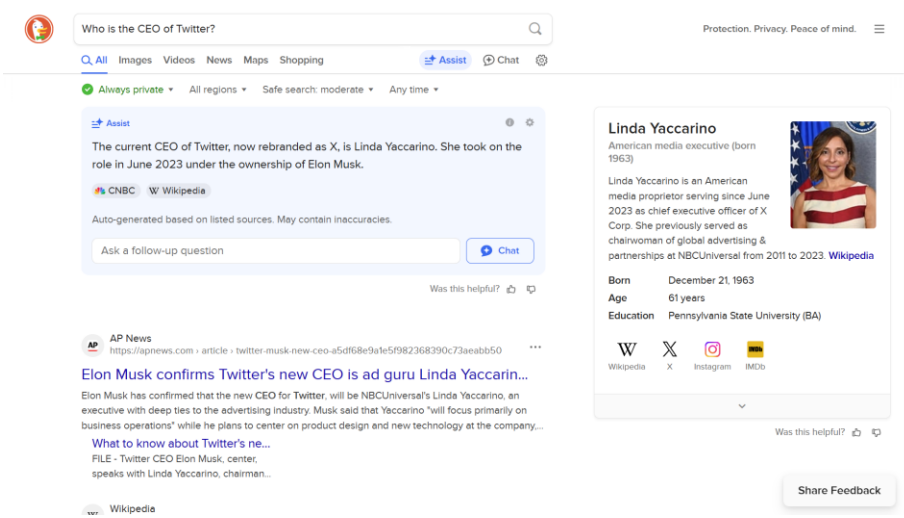


Figure 4-3 The good old days of searching the internet for information

Let's implement a simple search engine that will be able to answer questions about an evolving topic. In the following example, you will use the DuckDuckGo search engine via a Python package[45]:

```
pip install duckduckgo-search
```

```
function T = search(query, maxResults)
    if nargin < 2
        maxResults = 10; % Default value
    end
    ddgs = py.duckduckgo_search.DDGS();
    res = ddgs.text(query, max_results=maxResults);
    df = py.pandas.DataFrame(res);
    T = table(df);
end
```

It provides by default 10 results, but this can be modified with the named argument *maxResults*.

```
query = "Who's the last CEO of Twitter?";
T = search(query)
```

```
T = 10×3 table
```

[45] Search for text, news, images and videos using the DuckDuckGo.com search engine Python Package: https://pypi.org/project/duckduckgo-search/

	title	...
1	"Parag Agrawal - Wikipedia"	
2	"Jack Dorsey - Wikipedia"	
3	"Here are the executives that have exited Twitter - The Hill"	
4	"Twitter has had 5 CEOs in 17 years, Linda Yaccarino may be company's ..."	
5	"Who is Parag Agarwal? - The new CEO of Twitter - Business Standard"	
6	"Elon Musk names Linda Yaccarino new Twitter boss - BBC"	
7	"New Twitter CEO steps from behind the scenes to high profile"	
8	"Twitter CEO Jack Dorsey steps down, replaced by CTO Parag Agrawal"	
9	"Jack Dorsey is stepping down as CEO of Twitter - CNN"	
10	"Twitter has a new mystery CEO | The Verge"	

The search engine result pages contain small excerpt of each page indexed (stored in the table columns named *body*). You might simply rely on those excerpts that are typically dense enough to stuff into a model and get a good answer. Sometimes, you might want to retrieve the top 3 or 5 pages from the search to get a more comprehensive answer.

```
writetable(T,"search_results.csv")
T = search(query,3);
for i = 1:3 %height(T)
    c = extractHTMLText(webread(T.href(i)));
    T.content(i) = string(c);
end
```

Now that the context is gathered, you can reiterate your request to OpenAI:

```
context = join(T.content,newline);
prompt = ["Based on the following context, answer the query:";
          "-------";
          "Context:";
           context;
          "-------";
          "Query:";
           query];
```

```
prompt = strjoin(prompt,newline);
model = openAIChat();
response = generate(model,prompt)
```

```
response = "The last CEO of Twitter is Linda Yaccarino, who was appointed to the position after Elon Musk."
```

The heavy lifting in this example is performed by the search engine, with algorithms like PageRank[46] from Google, that ranks web pages in search results by evaluating the number and quality of links to a page. The LLM is relegated to the second role by "simply" ingesting the raw unprocessed text of the result page to make sense of the information and present it in a human friendly way.

Traditional search can also be called *keyword search*, in that it is leveraging inverted indexes, a data structure that allows a very quick lookup of documents containing certain words. This enables fast and accurate retrieval of documents based on keyword matches. The distributed nature of this architecture allows search engines to scale seamlessly, making it possible for platforms like Google to index the vast expanse of the early internet in the late 1990s.

Fetching documents is just one part of delivering a performant search experience; ranking is equally crucial. The introduction of TF-IDF[47] (term-frequency, inverse-document-frequency) marked a significant breakthrough in ranking methodologies. TF-IDF assigns a score to each document based on the frequency of query terms within it (TF) and across the entire corpus (IDF). This scoring mechanism ensures that documents matching the query well and containing rare, specific terms are ranked higher, enhancing relevance.

While traditional search engines have laid the groundwork for modern information retrieval systems, they come with several limitations[48] that impact their effectiveness and user experience.

- *Text Processing Challenges*

[46] Google Still Uses PageRank. Here's What It Is & How They Use It https://ahrefs.com/blog/google-pagerank/
[47] Understanding TF-ID: A Simple Introduction https://monkeylearn.com/blog/what-is-tf-idf/
[48] Vector vs Keyword Search https://www.algolia.com/blog/ai/vector-vs-keyword-search-why-you-should-care/

Search engines rely on keyword-based indexing, which poses challenges in text processing. Issues such as hyphenation, language differences, and case sensitivity can lead to inconsistencies in indexing and retrieval, affecting the search quality.

- *Exact Matches and Stemming*

Exact matching poses challenges, as queries for “cats” might not retrieve documents containing “cat.” Stemming, the process of reducing words to their root form, is a common solution. However, it introduces edge cases and exceptions, such as “universal” and “university” being stemmed to the same token.

- *Word Ambiguity and Synonyms*

Ambiguous words like “jaguar” present challenges in understanding user intent, as the context is often required to disambiguate meanings. Additionally, synonyms and related terms need to be mapped to ensure comprehensive retrieval, adding complexity to the system.

- *Misspelling and Autocorrection*

Spelling accuracy is crucial in keyword-based search engines. Misspelled queries can lead to irrelevant or no results. Implementing an effective autocorrect feature requires specific development tailored to the platform's domain and user base.

- *Language Support*

Supporting multiple languages introduces additional complexity, requiring the resolution of each aforementioned challenge for each supported language. This multiplies the development effort and can result in varying search quality across languages.

All those limitations motivate the need for another class of search, called vector search.

VECTOR SEARCH

Now, let's look into the concept of vector search, a technique for information retrieval that leverages a numeric representation of text (as vectors) to find semantically similar documents or passages.

First, we need to understand how language models convert prompts into chunks of text, that can be mapped to numbers. This is referred to as *tokens*.

Figure 4-4 gives you a sense of how words are broken into tokens, with a simple example of the GPT-4 tokenizer[49] at play.

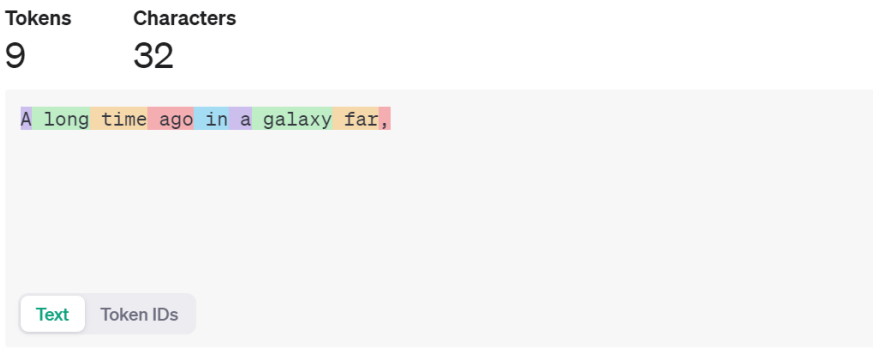


Figure 4-4 Tokenization performed by GPT-4 under the hood

To replicate those results locally, you can install the Python package *tiktoken*[50] and change the input text to see how it is tokenized:

```
text = 'A long time ago in a galaxy far,'
tiktoken = py.importlib.import_module('tiktoken');
encoding = tiktoken.encoding_for_model('gpt-4o-mini');
tokens = encoding.encode(text)
disp(tokens) % [32, 1317, 892, 4227, 304, 264, 34261, 3117, 11]
```

But this step isn't sufficient to capture the meaning of words when they are simply mapped to a number. You need to turn those numbers into vectors that make sense.

Vector embeddings[51] capture the semantic meaning of text by mapping words, sentences, or documents into high-dimensional vector spaces. Similar items in this space are close to each other, while dissimilar items are far apart. This property makes vector embeddings ideal for tasks like search and recommendation.

[49] OpenAI tokenizer page: https://platform.openai.com/tokenizer
[50] Tiktoken package: https://github.com/openai/tiktoken
[51] Vector Embeddings: https://www.pinecone.io/learn/series/nlp/dense-vector-embeddings-nlp/

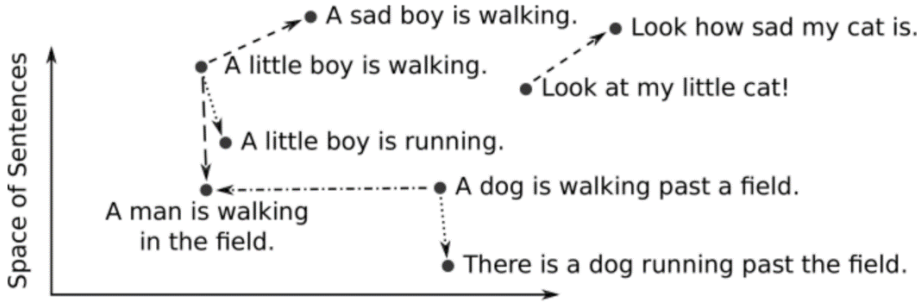


Figure 4-5 Embedding representation in a simplified 2-D space of a few sentences

This conversion from text to vector can be processed in several ways. A simple approach is to look at each letter as a number. Another approach is words-level embeddings, like Word2Vec[52], developed by Google. It uses shallow neural networks to produce word embeddings.

Let's look at an example with the OpenAI's text embeddings[53], on the first paragraph of chapter 1 of A Tale of Two Cities by Charles Dickens[54]

```
 % Define the paragraph as a string literal using concatenation for
readability
 paragraph = "It was the best of times, it was the worst of times, "
+ ...
     "it was the age of wisdom, it was the age of foolishness, " +
...
     "it was the epoch of belief, it was the epoch of incredulity, "
+ ...
     "it was the season of Light, it was the season of Darkness, " +
...
     "it was the spring of hope, it was the winter of despair, " +
...
     "we had everything before us, we had nothing before us, " + ...
     "we were all going direct to Heaven, we were all going direct
the other way " + ...
     "- in short, the period was so far like the present period, " +
...
     "that some of its noisiest authorities insisted on its being
received, " + ...
     "for good or for evil, in the superlative degree of comparison
only.";
```

[52] Word2Vec: https://www.tensorflow.org/text/tutorials/word2vec
[53] OpenAI's text embeddings: https://platform.openai.com/docs/guides/embeddings
[54] A Tale of Two Cities by Charles Dickens: https://www.gutenberg.org/ebooks/98

```
% Split the paragraph into sentences using ", " as the delimiter
sentences = split(paragraph, ", ");

% Select the first sentence
sentence1 = sentences(1)
```

```
sentence1 = "It was the best of times"
```

```
emb = extractOpenAIEmbeddings(sentence1)
```

```
emb = 1×1536

    0.0100   -0.0019    0.0083   -0.0402   -0.0179    0.0026
-0.0008 ⋯
```

```
size(emb)
```

```
ans = 1×2

           1         1536
```

We can batch things up by sending the whole list of sentences to the OpenAI embeddings service, and use numpy as a way to store and index the vectors:

```
embs = extractOpenAIEmbeddings(sentences)
```

```
embs = 18×1536

    0.0100   -0.0019    0.0083   -0.0402   -0.0179
0.0026   -0.0008 ⋯
   -0.0169    0.0028    0.0003   -0.0268   -0.0102
0.0076   -0.0040
    0.0282   -0.0129   -0.0060   -0.0285   -0.0036   -
0.0019   -0.0106
      ⋮
```

Different factors that come into consideration when choosing an embedding model, such as the size of the dictionary, the performance of the calculation, and the price of the service (if hosted).

Using larger embeddings, for example storing them in a vector store for retrieval, generally costs more and consumes more compute, memory and storage than using smaller embeddings. By default, the length of the embedding vector will be 1536 for `text-embedding-3-small` or 3072 for `text-embedding-3-large`.

The Massive Text Embedding Benchmark (MTEB)[55] from Hugging Face helps in assessing the performance of different embedding models, representing here models by average English MTEB score (y) vs speed (x) vs embedding size (circle size):

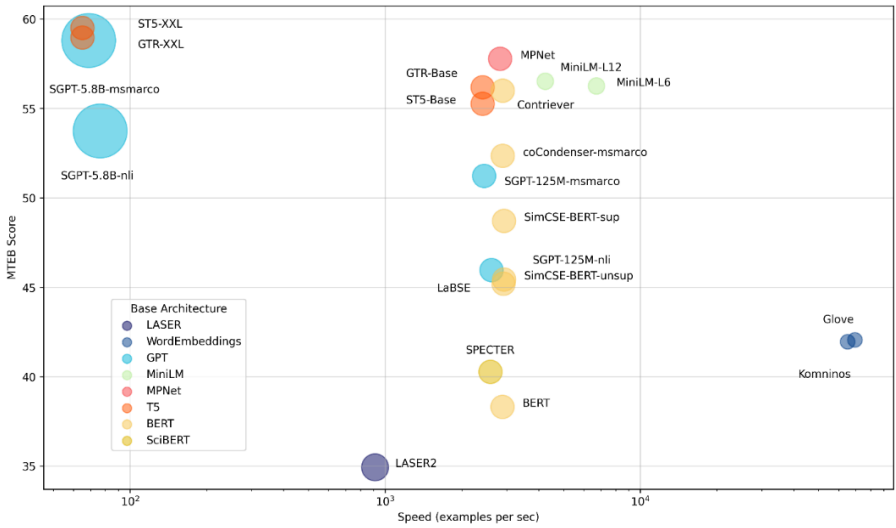


Figure 4-6 Massive Text Embedding Benchmark

Some of those top performing embeddings on the top right of the benchmark can be accessed directly from MATLAB, such as MiniLM-L6[56] and L12. Those models are open-source and can be executed efficiently locally.

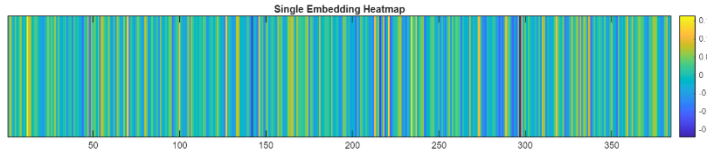
```
model = documentEmbedding(Model='all-MiniLM-L6-v2'); % or 'all-MiniLM-L12-v2'
embedding1 = model.embed(sentence1)
```

```
embedding1 = 1×384

   -0.0452    0.0716    0.0172   -0.0316    0.0476   -0.0080    0.0121 ⋯
```

A nice way to represent a vector embedding resembles a DNA sequence:

```
plotSingleEmbedding(embedding1)
```

[55] Massive Text Embedding Benchmark: https://huggingface.co/blog/mteb
[56] Support package for the all-MiniLM-L6-v2 sentence embedding model in MATLAB: www.mathworks.com/matlabcentral/fileexchange/156399-text-analytics-toolbox-model-for-all-minilm-l6-v2-network



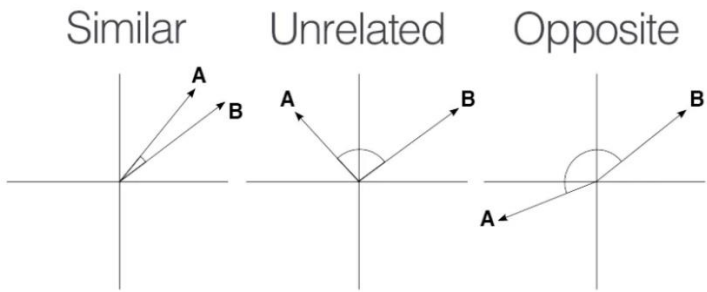
As an alternative to this native model import in MATLAB, you can use Sentence transformers [57], a python framework that provides access to state-of-the-art embeddings models, including the one referred above.

```
pip install -U sentence-transformers
```

```
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2') # or 'all-mpnet-base-v2'
# Sentences are encoded by calling model.encode()
embedding = model.encode(sentence)
embedding.shape
```

```
(384,)
```

There are different ways to measure semantic similarity[58], by calculating the distance in a high dimensional vector embedding space. One approach is to use trigonometry, with cosine similarity described with the following 3 cases in 2 dimensions:



[57] Sentence Transformers: https://sbert.net/
[58] Sentence Similarity With Sentence-Transformers in Python: https://www.youtube.com/watch?v=Ey81KfQ3PQU

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

Figure 4-7 Similarity search explained in a 2-D space

Since the norm of the vector embeddings is 1, cosine similarity between the vectors results in a simple dot product.

```
norm(emb)
```

```
ans = 1.0000
```

```
dot(embs(1,:),embs(2,:))
```

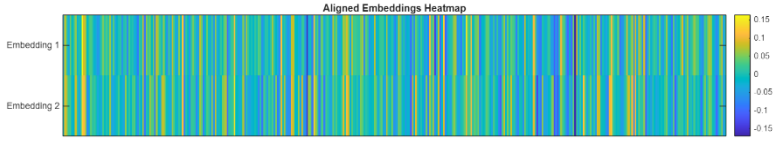
```
ans = 0.9032
```

```
sentences(1),sentences(2)
```

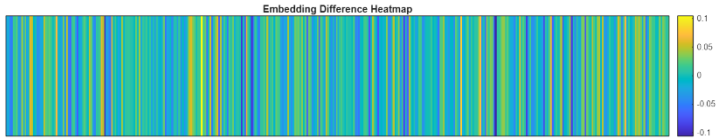
```
ans = "It was the best of times"

ans = "it was the worst of times"
```

You can see that those first two sentences of the paragraph are close semantically.

```
plotAlignedEmbeddings(embedding1, embedding2)
```




```
plotDifferenceEmbeddings(embedding1,embedding2)
```



Now let's look at a new sentence, and how it compares to the rest of the paragraph:

```
% Find the closest to a new sentence
query_sentence = "good times";
```

```
query_embedding = model.embed(query_sentence);
distance = cosineSimilarity(query_embedding,embeddings)
```

```
distance = 1×18

    0.6778    0.5655    0.1804    0.1735    0.1918    0.1941
0.2153 ⋯
```

```
[B,I] = sort(distance,"descend")
```

```
B = 1×18

    0.6778    0.5655    0.2953    0.2941    0.2857    0.2735
0.2245 ⋯

I = 1×18

     1     2    11     9    17    12    15     7    13
6     5    10 ⋯
```

```
s = sentences(I); s(B>0.5)
```

```
ans = 2×1 string

"It was the best of ti…
"it was the worst of t…
```

The previous approach of vector search by computing on the fly the distance with every vector of the database isn't scalable. That is why you need to index your database[59].

LLAMAINDEX: BUILDING AN INDEX

One of the early frameworks that competed with LangChain nicely to enable question answering was LlamaIndex [60] (initially known as GPTindex). It stood out by the simplicity of its implementation.

The concept is quite straightforward:

- You store your documents in tidy location
- You build an index on your data

[59] Vector Databases simply explained! (Embeddings & Indexes)
https://www.youtube.com/watch?v=dN0lsF2cvm4
[60] Llama-index: https://docs.llamaindex.ai/en/stable/

- You define a query engine/retriever based on this index and ask questions against this query engine (in the first versions, you would ask questions directly against the index)

Let's illustrate this by a simple demo based on their starter tutorial[61]:

- *Step 0*: Install the necessary Python packages

```
pip install llama-index pypdf requests tiktoken
```

- *Step 1*: Load documents (PDF reader)

With a little bit of practice, you will realize that the performance and robustness of your LLM application relies a lot on the preprocessing pipeline that feeds chunks of text for retrieval-augmented generation.

As such, it is interesting to spend some time on the inputs of your knowledge retrieval engine. Document loaders are an important part of the equation, as they structure the information that is ingested.

In this chapter, we will use the book Impromptu as context to answer questions with ChatGPT. Let's start by extracting the content of this pdf with pypdf[62]:

```
# process_pdf.py

import requests, io, pypdf, os
# get the impromptu book
url = 'https://www.impromptubook.com/wp-
content/uploads/2023/03/impromptu-rh.pdf'

def pdf_to_pages(file):
    "extract text (pages) from pdf file"
    pages = []
    pdf = pypdf.PdfReader(file)
    for p in range(len(pdf.pages)):
        page = pdf.pages[p]
        text = page.extract_text()
        pages += [text]
    return pages

r = requests.get(url)
f = io.BytesIO(r.content)
pages = pdf_to_pages(f)
# print(pages[1])
```

[61] Llama-index starter tutorial:
https://docs.llamaindex.ai/en/stable/getting_started/starter_example.html
[62] https://pypdf.readthedocs.io/en/stable/

```
if not os.path.exists("impromptu"):
    os.mkdir("impromptu")
for i, page in enumerate(pages):
    with open(f"impromptu/{i}.txt","w", encoding='utf-8') as f:
        f.write(page)

sep = '\n'
book = sep.join(pages)
# print(book[0:35])
```

You can call this script from MATLAB to retrieve the content that will be indexed:

```
[pages,book] = pyrunfile("process_pdf.py",["pages","book"]);
string(pages(2))
```

```
ans =
    "Impromptu
     Amplifying Our Humanity
     Through AI
     By Reid Hoffman
     with GPT-4"
```

```
tiktoken = py.importlib.import_module('tiktoken');
encoding = tiktoken.encoding_for_model('gpt-4o-mini');
tokens = encoding.encode(book);
length(tokens)
```

```
ans = 82542
```

- *Step 2:* Build an index

```
## Build index
from llama_index.core import SimpleDirectoryReader, VectorStoreIndex
documents = SimpleDirectoryReader("impromptu").load_data()
index = VectorStoreIndex.from_documents(documents)
# save to disk
index.storage_context.persist()
```

- *Step 3:* Query the index

```
## Query index
query_engine = index.as_query_engine()
response = query_engine.query('what is the potential of AI for
Education?')
print(response)
```

```
"The potential of AI for education lies in its ability to transform
the way we learn and deliver instruction. AI-driven tools can automate
and streamline tasks like grading and content creation, allowing
teachers to focus more on engaging and inspiring students.
Additionally, AI can provide personalized and individualized learning
experiences tailored to each student's needs and interests, ultimately
enhancing the educational process."
```

You can even retrieve the sources that have been selected to produce this answer:

```
# Cite the sources
sources = [s.node.get_text() for s in response.source_nodes]
print(sources[0][0:11])
```

```
47
    Education
    the technology will also create an educational system
    that is less equitable and accessible. ...
```

The beauty of this approach is that is simply stores the embeddings into json files. You can take a look at the storage folder created that maps documents hash to an embedding.

But this simple text file approach doesn't scale so well when it comes to storing large document bases. For this, let's look into vector databases.

VECTOR DATABASES

A vector database is a data store that stores data as high-dimensional vectors, which are mathematical representations of attributes or features. Some examples of vector databases include: Weaviate, Qdrant, Pinecone, Chroma, Faiss, MongoDB

Let's try Facebook AI Similarity Search (faiss)[63], which is known to be insanely performant.

```
pip install faiss-cpu
```

[63] FAISS: https://github.com/facebookresearch/faiss

For this last example of the chapter, you will use a very serious text: *The You You Are by Dr. Ricken Lazlo Hale, PhD on Apple Books*[64]

You will index embeddings from the same MiniLM model as in the previous section.

```
 model = documentEmbedding(Model='all-MiniLM-L6-v2');
 paragraph = "It's said that as a child, Wolfgang Mozart killed another
boy by slamming his head in a piano." + newline + ...
     "Don't worry, my research for this book has proven the claim
untrue." + newline + ...
     "As your heart rate settles though, consider the power an author,
heretofore referred to as Me, can hold over a reader, heretofore called
You." + newline + ...
     "But what, indeed, is… You? All creatures from the leaping cat
to the cowering shrew think of themselves as You, a logical center
for the universe." + newline + ...
     "Yet the cat eats the shrew, and we, like Schrödinger, live on
to wonder what it means." + newline + ...
     "I'm not asking out of mere politeness. For You, my friend, are
no mere consumer of this book, sucking down nouns and adverbs like a
plump babe to the teat." + newline + ...
     "In fact, your relationship to this work is far more intimate
and profound." + newline + ...
     "You are its subject.";

 % Split the paragraph into sentences
 sentences = splitlines(paragraph);
```

First you instanciate an index of the right size (dimension of the embeddings):

```
d = py.int(384); % Dimension of the embeddings
index = py.faiss.IndexFlatL2(d);
index.add(embeddings)

% Save the index to a file
py.faiss.write_index(index, "you.index");

% Later, load the index from the file
index = py.faiss.read_index("you.index");
```

[64] The You You Are: https://books.apple.com/us/book/the-you-you-are/id6738364141

Then you can query the index:

```
 % Encode the sentence "who are you"
 query_sentence = "who are you";
 query_embedding = model.embed(query_sentence);

 % Convert query_embedding to a numpy array and reshape to [1, 384]
 query_embedding = py.numpy.reshape(query_embedding,
py.tuple({py.int(1), d}));

 k = py.int(3);  % Number of nearest neighbors to retrieve
 res = index.search(query_embedding, k);
 distance = double(res{1})
```

```
distance = 1×3

    1.3337    1.4270    1.4818
```

```
 indices = int64(res{2})
```

```
indices = 1×3 int64 row vector

   7   2   3
```

Finally, let's look at the nearest neighbors (smaller distance is better):

```
 % Loop over the number of neighbors returned.
 % Note: FAISS returns 0-indexed indices, so add 1 for MATLAB
indexing.
 for i = 1:size(indices, 2)
     idx = indices(1, i) + 1;
     % Use either sentences{idx} (if sentences is a cell array)
     % or sentences(idx) (if sentences is a string array)
     fprintf("Sentence: %s, Distance: %f\n", sentences{idx},
distance(1, i));
 end
```

```
Sentence: You are its subject., Distance: 1.333668

Sentence: As your heart rate settles though, consider the power an
author, heretofore referred to as Me, can hold over a reader,
heretofore called You., Distance: 1.427042
```

```
Sentence: But what, indeed, is… You? All creatures from the leaping
cat to the cowering shrew think of themselves as You, a logical center
for the universe., Distance: 1.481814
```

As an opening thought, after standing up a vector database, you can build a RAG production system:

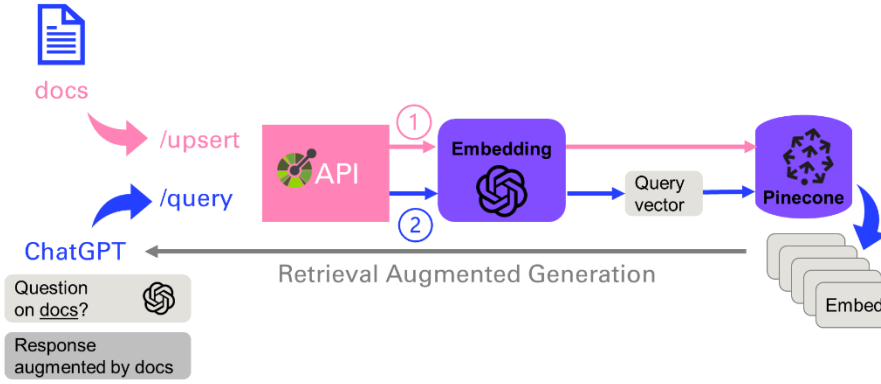


Figure 4-8 Architecture of a production system leveraging RAG

This application[65] can be nicely architectured with plugins, that clearly define the API with two main endpoints: upsert (to update or insert the vector database), or query that will convert the prompt into an embedding and perform a vector search to find the N (= 5) closest ones.

[65] ChatGPT Plugins: https://www.youtube.com/watch?v=hpePPqKxNq8

5. AGENTS & TOOLS

Large Language Models are one (big) step in the direction of what is called Artificial General Intelligence (AGI)[66]. But a further shift was operated from 2023 to 2025 from conversation chatbots to systems that can increasingly operate with minimal human supervision. This chapter will break down the steps of this transformation.

AGENTS SELECT TOOLS

It was a little difficult at first for me to understand the notion of an agent. But it became much easier after attending a presentation at PyCon called *Building LLM-based Agents*[67] from Haystack. Here is the example of a prompt that illustrates how agents operate:

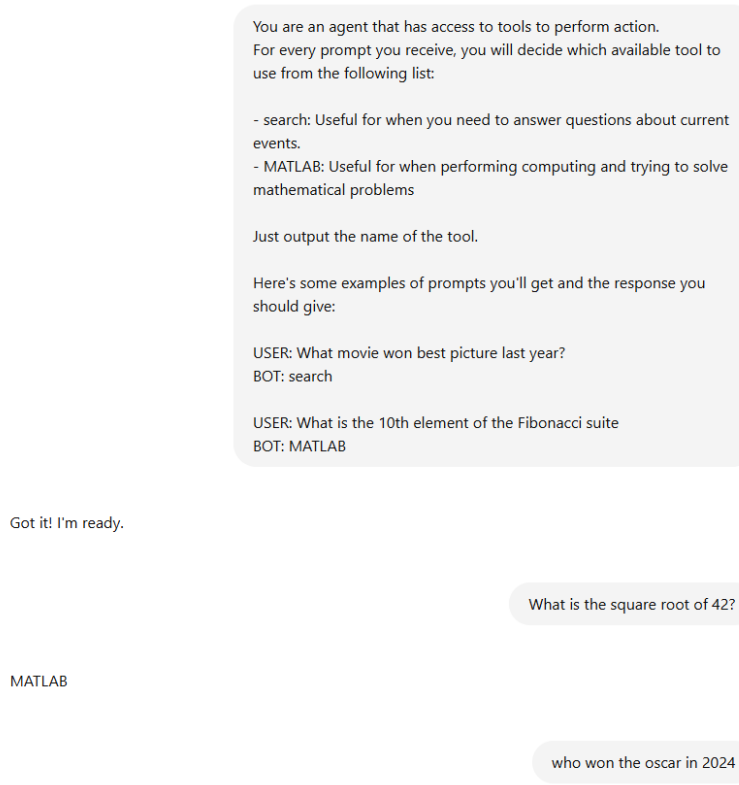


Figure 5-1 Prompting ChatGPT to act as an agent with few shot examples of available tools

[66] Sparks of Artificial General Intelligence: Early experiments with GPT-4: https://arxiv.org/abs/2303.12712
[67] Tuana Celik: Building LLM-based Agents: https://www.youtube.com/watch?v=LP8c9Vu9mOQ

Here is a simple representation of what is happening here:

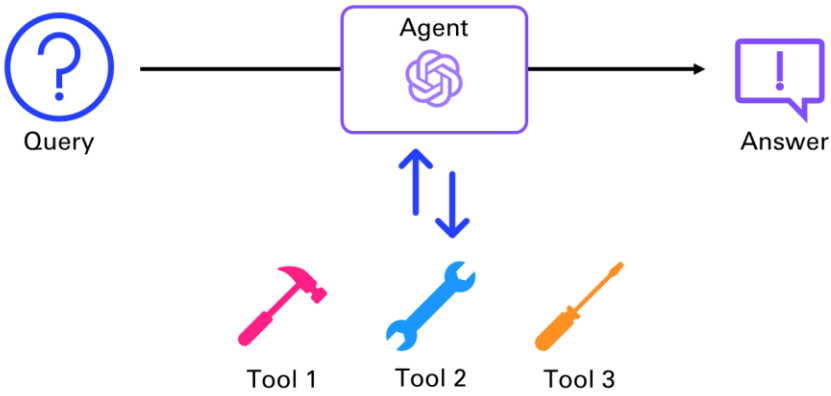


Figure 5-2 Agent making a choice between several tools

SMITH: MY PEDAGOGICAL AGENT FRAMEWORK

I implemented a simple MATLAB live script that goes through the basic principle of agentic workflows. In this single turn implementation, I'm passing the tools context as a system prompt to the LLM.

You can verify that the agent behaves like any LLM without tools:

```
prompt ="Who is the CEO of Twitter?";
% Calling agent without passing tools
agent(prompt)
```

```
ans = "As of my last update in October 2023, Elon Musk was the CEO of
Twitter, Inc. However, leadership positions can change frequently, so
it's a good idea to verify this information from a current news source
or Twitter's official announcements for the most accurate and up-to-
date information."
```

Let's load the list of tools, containing their name, a short description and an example.

```
tools = loadTools()
```

```
tools = 2×1 struct
```

Fields	name	...
1	'search'	
2	'MATLAB'	

To start resembling standards in tool definition, I have stored this information in *tools.jsonl*:

```
[
    {"name" : "search",
    "desc" : "Useful for when you need to answer questions about
current events.",
    "example" : "What movie won best picture last year?"
    }
    ,
    {"name" : "MATLAB",
    "desc" : "Useful for when performing computing and trying to
solve mathematical problems",
    "example" : "What is the 10th element of the Fibonacci suite"
    }
]
```

Now let's call the agent with the context provided by the tools by a single call, no loops, and no reasoning:

```
tools = loadTools();
agent(prompt, tools=tools)
```

```
tool: search

------------------

ans = "The CEO of Twitter, now called X Corp, is Linda Yaccarino."
```

Let's take a look at the two steps taken by the agent, when a list of tools is passed:

- Pick a tool
- Use the tool

```
query = "What movie won best picture in 2024?";
selection = pickTool(query, tools)
```

```
selection = "search"
```

```
agent(query, tools=tools)
```

```
tool: search

------------------

  ans = "The movie that won Best Picture in 2024 is "Oppenheimer.""
```

The system prompt is used to pass the tool chosen as context into the second request to the agent (Upon selection of the tool, creating the context necessary to answer the initial query, No looping or reasoning, single tool call in this implementation.)

```
agent("what is Fibonacci of 42", tools=tools)
```

```
tool: MATLAB
------------------

ans =
    "fibonacci = @(n) (n <= 1) * n + (n > 1) * (fibonacci(n - 1) +
fibonacci(n - 2));
     result = fibonacci(42);"
```

For this example, we would need to iterate by first generating the code, executing it, and passing the answer once again to the LLM to finally answer the question. This more evolved loop is typically handled with reasoning.

Reasoning and Acting

Agents are more than just LLMs, and they are complementing inherent AI limitations:

- *Reasoning*: LLMs are not proceeding like the human brain – remember, they simply infer the next logical word in a sentence. As such they are not good at solving basic math problems.

- *Access to Data and Tools*: LLMs do not have access to your data, only the ones they have seen during the training phase. By default, they do not have access to search over the internet. As such, it would not make sense to compare ChatGPT with Google search, even if their use cases overlap significantly.

A major paper that has been at the origin of many of the improvements of the usage of LLMs is *React: Synergizing Reasoning and Acting in Language Models*[68].

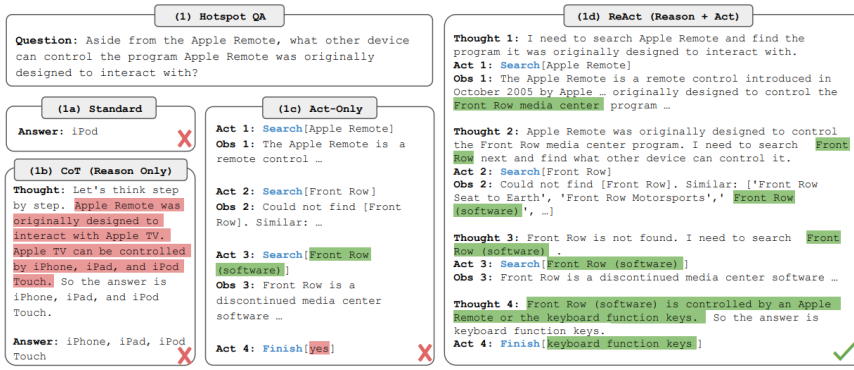


Figure 5-3 Example from the ReAct paper combining Reasoning and Acting

Instead of directly answering a question, this approach breaks down the process into Reasoning + Acting. A number of agent frameworks emerged to provide developers with the ability to integrate AI into their applications, by implementing the ReAct concepts, like LangChain and LlamaIndex.

Figure 5-5 illustrate this concept, where an agent is iterating to decide the next step dynamically based on the query and the available Tools.

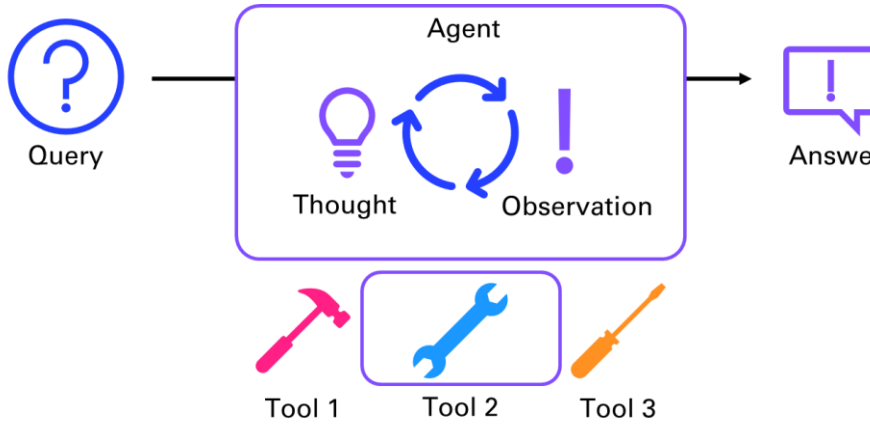


Figure 5-4 Chain of Thoughts for tool selection

[68] ReAct paper: https://arxiv.org/abs/2210.03629

You can provide the agent with a set of tools that it can choose to use to reach a result. At each iteration, the agent will pick a tool from the ones available to it. Based on the result, the agent has two options: It will either decide to select a tool again and do another iteration, or it will decide that it has reached a conclusion and return the final answer. Now, one crucial part of the puzzle hasn't been addressed: how to nicely integrate code logic in an LLM flow. This is what you will see next with function calling.

OpenAI Functions

In July 2023, OpenAI announced Function calling capabilities[69]. This was a big deal for me, as I was struggling to maintain my agents in LangChain to call tools. OpenAI was making headways on the turf of LangChain, and it was making it easier for me to standardize the method to call tools as functions.

Setup functions list

Let's look at a very simple example illustrated in the blog post: a weather agent.First you will need to specify the functions that the AI can call, with the attributes:

- Name
- Description (giving indications to the AI on what it does and when to use it)
- Parameters (including with ones are required)

You will equip the weather agent with the right function to call:

```
 % Step 0: Setup the agent equipped with his one tool, the weather function
 f = openAIFunction("getCurrentWeather","Get the current weather in a given location");
 f = addParameter(f,"location", type="string", description="Name of a city");
 f    =    addParameter(f,"unit",    type="string",    enum=["celsius" "fahrenheit"], description="Unit for the temperature");
 agent = openAIChat("You are a helpful weather agent", Tools=f)
```

```
agent =
  openAIChat with properties:
```

[69] https://openai.com/blog/function-calling-and-other-api-updates

```
          ModelName: "gpt-4o-mini"
        Temperature: 1
               TopP: 1
      StopSequences: [0×0 string]
            TimeOut: 10
       SystemPrompt: {[1×1 struct]}
     ResponseFormat: "text"
   PresencePenalty: 0
  FrequencyPenalty: 0
     FunctionNames: "getCurrentWeather"
```

STEPS TO FUNCTION CALLING

Once the functions are specified (only one here), we will follow the following steps:

Figure 5-5 Function calling workflow

```
% Step 1: send the conversation to the agent
prompt = "What is the weather in Boston?";
messages = messageHistory;
messages = addUserMessage(messages, prompt);
[~,completeOutput] = agent.generate(messages)
```

```
completeOutput = struct with fields:
          role: 'assistant'
       content: []
    tool_calls: [1×1 struct]
       refusal: []
   annotations: []
```

```
messages = addResponseMessage(messages,completeOutput);
% Step 2: Check if the agent wants to call a function
```

```
 if         isfield(completeOutput,         'tool_calls')         &&
isstruct(completeOutput.tool_calls)
     % Extract the tool call details and execute the function call
     toolCallID = string(completeOutput.tool_calls.id);
     functionCalled                                                =
string(completeOutput.tool_calls.function.name);
     args = jsondecode(toolCall.arguments);
     location = args.location;
     unit = args.unit;
     if functionCalled == "getCurrentWeather"
         weatherData = getCurrentWeather(location, unit)
         messages                                                  =
addToolMessage(messages,toolCallID,functionCalled,weatherData);
     end
 end
```

```
weatherData =
'{"location":"Boston","temperature":"72","unit":"celsius","forecast":
["sunny","windy"]}'
```

```
 % Step 3: Format the response
 generatedText = agent.generate(messages)
```

```
generatedText = "The current weather in Boston is 72°C, which seems
unusually high. The forecast indicates it is sunny and windy. Would
you like me to check something else?"
```

6. SPEECH-TO-TEXT AND TEXT-TO-SPEECH

In this chapter, you will learn how to transcribe text from speech (such as Youtube videos) and synthetize speech from text (such as articles). This flavor of AI involving speech was popularized with personal & home assistants such as Alexa from Amazon. But up until the new wave of Generative AI coming with ChatGPT, interacting with voice was primarily reduced to *speech command recognition*. The assistant awakens when you call her name – "Alexa" or "Ok Google" – awaits instructions and synthesizes the answer. Another pop reference to this kind of AI in science fiction is J.A.R.V.I.S[70] the engineering assistant of iron-man, that appeared in the first movie from 2008.

TRANSCRIPTION

Transcribing spoken language into text has traditionally been a complex and resource-intensive process. Like with text, the signal carried by the sound of a voice can be processed by a neural network that has been trained to convert it into text.

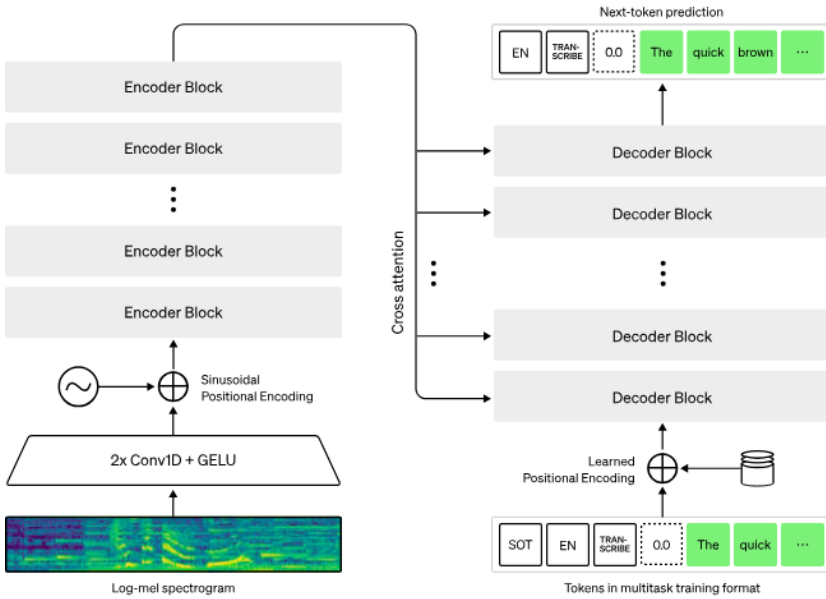


Figure 6-1 Ingesting audio signal in a Transformer architecture to predict the next token

[70]Just a Rather Very Intelligent System https://en.wikipedia.org/wiki/J.A.R.V.I.S.

With the recent boom of Generative AI[71], speech-to-text (STT) has become more accurate, efficient, and accessible than ever before. Architectures very similar to Large Language Models excel in understanding context, semantics, and nuances in language, making them ideal candidates for speech transcription tasks. As you can see in figure 6-1, the input audio signal is fed as a spectrogram, that is then encoded into an intermediate representation, further decoded into text, through the same next-word prediction mechanism as ChatGPT.

A few pure players of Speech AI have specialized in this task, like AssemblyAI[72] and Gladia[73]. But here again, a large chunk of this market is driven by OpenAI, with the release of Whisper[74] as open-source in September 2022 (just a few months before ChatGPT). Whisper can be downloaded locally to perform transcription without having to send your audio files over the internet. This is useful for privacy reasons, or when you have a slow or no internet connection.

There are five model sizes, four with English-only versions, offering speed and accuracy tradeoffs. Below are the names of the available models and their approximate memory requirements and inference speed relative to the large model; actual speed may vary depending on many factors including the available hardware.

Table 6-1 Whisper models family

Size	Parameters	English-only model	Multilingual model	Required VRAM	Relative speed
tiny	39 M	tiny.en	tiny	~1 GB	~32x
base	74 M	base.en	base	~1 GB	~16x
small	244 M	small.en	small	~2 GB	~6x
medium	769 M	medium.en	medium	~5 GB	~2x
large	1550 M	N/A	large	~10 GB	1x

[71] Recent developments in Generative AI for Audio https://www.assemblyai.com/blog/recent-developments-in-generative-ai-for-audio/
[72] https://www.assemblyai.com/ - https://www.youtube.com/watch?v=r8KTOBOMm0A
[73] https://www.gladia.io/ - https://techcrunch.com/2023/06/19/gladia-turns-any-audio-into-text-in-near-real-time/
[74] https://openai.com/research/whisper

Whisper is available as a support package in MATLAB and requires the Audio toolbox.

First, you need to download the Whisper model size that you want, and unzip it to a location on the path. A doc example provides a script that can be adapted for different sizes of the model. For a simple example, we will use the base multilingual model:

```
% Download the base Whisper model
% openExample('audio/DownloadWhisperSpeechtoTextModelExample')
modelSize = "base";
downloadLink="https://ssd.mathworks.com/supportfiles/audio/whisper-
"+modelSize+".zip";
downloadFolder = fullfile(tempdir,"whisperDownload");
loc = websave(downloadFolder,downloadLink);
modelsLocation = "models";
% modelsLocation = tempdir;
unzip(loc,modelsLocation)
addpath(fullfile(modelsLocation,"whisper-base"))
```

Let's load an example to transcribe.

```
% Download the MP3 from GitHub
audioURL="https://github.com/yanndebray/programming-
GPTs/raw/refs/heads/main/data/audio/enYann-tale_of_two_cities.mp3";
fileName  = "enYann-tale_of_two_cities.mp3";
websave(fileName, audioURL);
[audio, fs] = audioread(fileName);
sound(audio,fs)
```

Finally you can run the speech2text command:

```
client = speechClient("whisper","ModelSize","base");
transcript = speech2text(audio, fs, Client=client)
```

```
transcript = "It was the best of times. It was the worst of times. It
was the age of wisdom. It was the age of foolishness. ..."
```

As an alternative to this programmatic approach, you can also use the Signal Labeler[75] in order to perform this transcription process semi-interactively. This enables you to correct what might have been miss-transcribed.

[75] www.mathworks.com/help/signal/ug/label-spoken-words-in-audio-signals-using-external-api.html

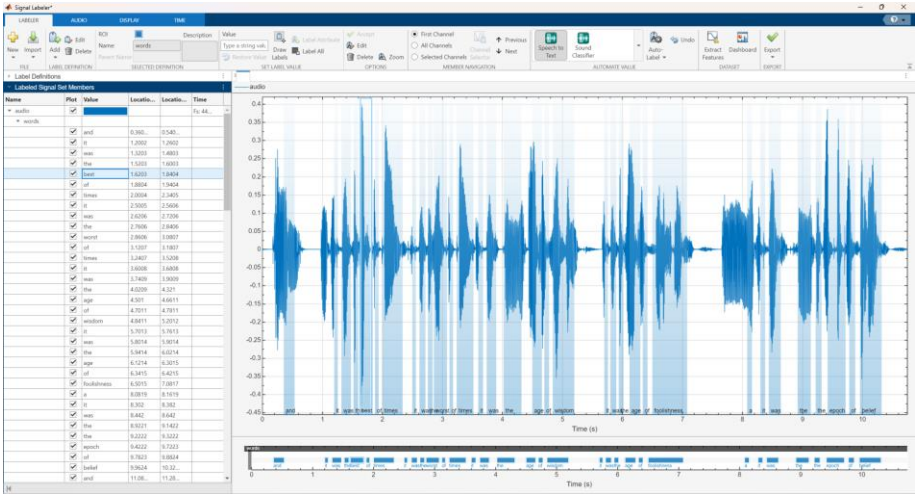


Figure 6-2 Signal Labeler to edit the transcription interactively

You can also use Whisper through the transcription API[76] of openAI. Currently, there is no difference between the open-source version of Whisper and the version available through the API. However, through the API, OpenAI offers an optimized inference process which makes running Whisper through the API much faster than doing it through other means. For this let's use the Python OpenAI package.

```
# chap6_stt.py

import openai
from pathlib import Path
fileName = "enYann-tale_of_two_cities.mp3"
file_path = Path(fileName)
transcription = openai.audio.transcriptions.create(model="whisper-1",
file=file_path)
print(transcription.text)
```

[76] https://platform.openai.com/docs/guides/speech-to-text

VOICE SYNTHESIS

If you listened to the audio recording of the previous example, you might have been thinking that I sounded weird with a German accent at time, and a British pronunciation here and there. What I did not tell before is that this short narration was generated using a clone of my voice.

To perform this magic trick, I am using a service called Elevenlabs[77]. It offers a free tier with 10,000 Characters per month (~10 min audio). But you will need to upgrade to the starter tier for $5/month in order to clone your voice with as little as 1 minute of audio. I simply extracted 5 samples of 1 min from a meeting where I was presenting, but the quality could be improved, especially if I avoid saying "hum".

OpenAI released a Text-to-Speech[78] API that you can conveniently use since you already have an OpenAI account and API key. They are two versions of the model (`tts-1` optimized for speed, `tts-1-hd` optimized for quality), and 6 voices to choose from (alloy, echo, fable, onyx, nova, and shimmer). Here is a way to call it from Python

```
# chap6_tts.py

import openai
speech_file_path = "speech.mp3"
response = openai.audio.speech.create(
  model="tts-1", voice="alloy",
  input="The quick brown fox jumped over the lazy dog.")
response.stream_to_file(speech_file_path)
```

APPLICATION: DAILY TECH PODCAST

Let's apply text-to-speech to create a daily tech podcast:

- Parse Techcrunch RSS feed
- Synthetize the last 5 news articles into separate audio files
- Schedule a GitHub Action to run daily

[77] https://elevenlabs.io/
[78] https://platform.openai.com/docs/guides/text-to-speech

Figure 6-3 Workflow to produce a daily tech podcast

PARSE THE TECHCRUNCH RSS FEED

RSS[79] stands for Really Simple Syndication, and an RSS feed is a file that automatically updates information and stores it in reverse chronological order. RSS feeds can contain headlines, summaries, update notices, and links back to articles on a website's page. They are a staple of digital communications and are used on many digital platforms, including blogs and websites. RSS feeds can be used to:

- Get the latest news and events from websites, blogs, or podcasts
- Use content for inspiration for social media posts and newsletters
- Allow creators to reach audiences reliably

This might sound to you like pre-2000 internet stuff, but it is actually quite handy to build applications like our daily podcast. In 2018, Wired published an article named "It's Time for an RSS Revival"[80], citing that RSS gives more control over content compared to algorithms and trackers from social media sites. At that time, Feedly[81] was the most popular RSS reader. Chrome on Android has added the ability to follow RSS feeds as of 2021.

In this example, you will use the xml parsing capabilities in Text Analytics Toolbox:

```
function dailytechtask
    % Parse Techcrunch RSS Feed
    url = "https://techcrunch.com/feed";
    dt = datetime('now','Format','yyyy-MM-dd');
    dateStr = string(dt);
    episode = "tech_" + dateStr;

    % Get the RSS channel
    channel = rssFeed(url, episode, dateStr);
```

[79] https://en.wikipedia.org/wiki/RSS
[80] https://www.wired.com/story/rss-readers-feedly-inoreader-old-reader/
[81] https://feedly.com/news-reader

```
    % Extract all <item> nodes in the channel
    items = channel.getElementsByTagName('item');
    numItems = items.getLength();

    % Loop through each item, scrape the article
    for i = 0:(numItems-1)
        itemNode = items.item(i);
        [title, link, textData] = scrapeArticle(itemNode, episode);
        % Optionally display progress
        fprintf('Scraped: %s\n', title);
        fprintf('Link: %s\n',link);
    end
end
```

This task calls two custom functions:

1. Saving the rss feed,
2. Scraping the articles it links to

```
function channel = rssFeed(url, episode, dateStr)
    % Get the RSS data from the feed URL as text
    response = webread(url);  % returns a character vector

    % Save the entire RSS feed to a file
    rssFolder = fullfile("podcast", episode, "rss");
    if ~exist(rssFolder, 'dir')
        mkdir(rssFolder);
    end

    rssFile = fullfile(rssFolder, "techcrunch_" + dateStr + ".xml");
    fid = fopen(rssFile, 'w');
    fwrite(fid, response);
    fclose(fid);

    % Get the <channel> element
    doc = xmlread(rssFile);  % Parse the file as XML Document Object
    channel = doc.getElementsByTagName('channel').item(0);
end
```

/!\ Article titles can contain characters that are not valid to serve as file names.

For this, we will use the regular expression function *regexprep*[82] to locate and replace a pattern in a string (don't run away).

```
title = regexprep(title, '[<>:"/\\|?*]', '-');
```

This expression means match any character that is in this set, and replace them with the character **-** . The characters in this set are `<`, `>`, `:`, `"`, `/`, `\`, `|`, `?`, and `*`. These are characters that are not allowed in filenames in many file systems.

```
 function [title, link, textData] = scrapeArticle(itemNode, episode)
    % Extract title text from <title> node

title=char(itemNode.getElementsByTagName('title').item(0).getFirstCh
ild.getData());
    % Replace forbidden characters with '-'
    title = regexprep(title, '[<>:"/\\|?*]', '-');

    % Extract link text from <link> node
    link                                                              =
char(itemNode.getElementsByTagName('link').item(0).getFirstChild.get
Data());

    % Fetch HTML of the article
    html = webread(link);

    % In MATLAB R2021b or later, you can parse HTML with htmlTree
    tree = htmlTree(html);

    % Find elements whose class="entry-content"
    entryContent = findElement(tree, '.entry-content');

    if ~isempty(entryContent)
        % Extract readable text
        textData = extractHTMLText(entryContent(1));
    else
        textData = ''; % fallback if no match
    end

    % Save text content
    textFolder = fullfile("podcast", episode, "text");
    if ~exist(textFolder, 'dir')
        mkdir(textFolder);
```

[82] https://www.mathworks.com/help/matlab/ref/regexprep.html

```
    end
    textFile = fullfile(textFolder, title + ".txt");

    fid = fopen(textFile, 'w', 'n', 'UTF-8');
    fwrite(fid, textData, 'char');
    fclose(fid);
end
```

SYNTHETIZE THE LAST 3 NEWS ARTICLES INTO SEPARATE AUDIO FILES

We can start with just the last article for simplicity. But it's more convenient to batch it up and save several at a time. Between 3 and 5 seem like a good number for your daily commute.

```
function podcastgentask
    dt = datetime('now','Format','yyyy-MM-dd');
    dateStr = string(dt);
    episode = "tech_" + dateStr;
    listing = dir(fullfile("podcast",episode,"text","*.txt"));
    tbl = struct2table(listing);
    articles = string(tbl.name);
    audioFolder = fullfile("podcast", episode, "audio");
    if ~exist(audioFolder, 'dir')
        mkdir(audioFolder);
    end
    for article = articles(1:3)'
        disp(article)
        filePath = fullfile("podcast",episode,"text",article);
        txt =  fileread(filePath);
        numChar = length(txt);
        disp("Number of characters:")
        disp(numChar)
        if numChar > 4000
            disp("Summarizing...")
            txt = summarizeArticleForPodcast(txt);
            disp("New character count:")
            disp(length(txt))
        end
        response = pyrun( ...
            "import openai                          " + ...
            "; response = openai.audio.speech.create(" + ...
```

```
            "    model='tts-1',                          " + ...
            "    voice='alloy',                          " + ...
            "    input=text                              " + ...
            ")                                           " , ...
            "response", ...   % Python variable name you want back
            text=txt ... % inject MATLAB var `txt` as Py var `text`
            );

response.stream_to_file(fullfile("podcast",episode,"audio",replace(a
rticle,".txt",".mp3")))
     end
 end
```

⚠ TTS services have *characters limits* (4000 for OpenAI, 5000 for ElevenLabs)

If the length of the article exceeds the transcription limit, you have two options:

- split the article into several parts and then concatenate the transcriptions.
- summarize the article (by setting a character limit) and then transcribe it.

You will focus on the second approach, similar to what you learned in chapter 3:

```
function summary = summarizeArticleForPodcast(content)
     summaryPrompt = [
            "Summarize the article in less than 4000 characters";
            content;
            ];
    summaryPrompt = strjoin(summaryPrompt, newline);
    [summary,~] = bot(summaryPrompt);
    summary = char(summary);
end
```

This is what appears in the console when the article is too long for tts:

```
A comprehensive list of 2025 tech layoffs.txt

Number of characters:

        18837
```

```
Summarizing...
New character count:
        2254
```

SCHEDULE A GITHUB ACTION TO RUN DAILY

GitHub Actions[83] is a beautiful feature that enables you to automate workflows around your code. You can use different triggers to run those actions. In this example, you will trigger on a schedule (every morning).

In a repository of your choice, simply navigate to the Actions tab, and select set up a workflow yourself.

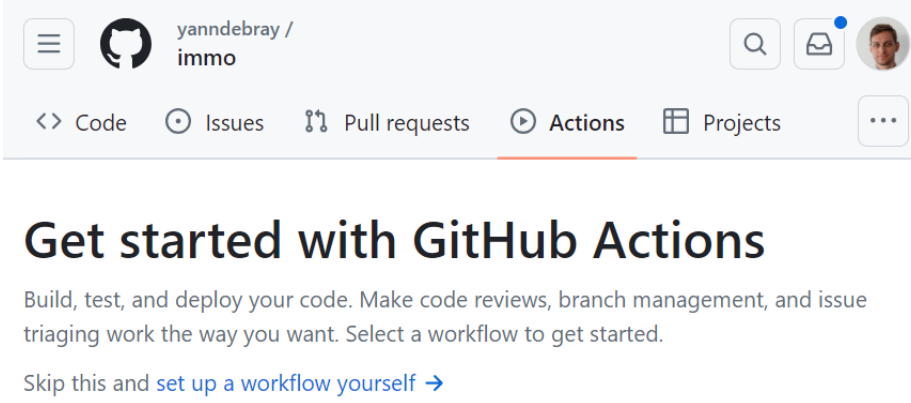


Figure 6-4 Tab for GitHub Actions in a repo

This will create the file: `<repo_name>/.github/workflows/podcast.yml` - Enter the following code:

```yaml
name: Daily Tech Podcast
run-name: ${{ github.actor }} is scheduling a MATLAB task
on:
  # schedule:
  #   - cron: "0 7 * * *"
  workflow_dispatch: {}
```

[83] https://github.com/features/actions

```
jobs:
  task:
    runs-on: ubuntu-latest
    steps:
      # Checks-out your repository under $GITHUB_WORKSPACE
      - name: Check out repo
        uses: actions/checkout@v4
      # Sets up MATLAB on a GitHub-hosted runner
      - name: Set up MATLAB
        uses: matlab-actions/setup-matlab@v2
        with:
          products: >
            Text_Analytics_Toolbox
      - name: Install Python dependencies
        run: python3 -m pip install openai
      # You can use "run-command" to execute custom MATLAB scripts
      - name: Run MATLAB script
        uses: matlab-actions/run-command@v2
        with:
                  command:  disp('Running  my  task!');  setup;
pyenv(Version="/usr/bin/python3"); dailytechtask; podcastgentask;
      # Save the result as artifact
      - name: Archive output data
        uses: actions/upload-artifact@v4
        with:
          name: podcast
             path:  podcast/tech_[0-9][0-9][0-9][0-9]-[0-9][0-9]-[0-
9][0-9]/

  env:
    OPENAI_API_KEY: ${{secrets.OPENAI_API_KEY}}
```

Some explanations of what this is doing:

- Actions use a YAML format[84] to specify the elements of the job to run.
- Actions use the cron syntax[85] to specify the schedule at which to run.

[84] https://docs.github.com/en/actions/writing-workflows/workflow-syntax-for-github-actions
[85] https://crontab.guru/every-day - https://docs.github.com/en/actions/writing-workflows/workflow-syntax-for-github-actions#onschedule

```
name: Daily Tech Podcast
run-name: ${{ github.actor }} is scheduling a MATLAB task
on:
  schedule:
    - cron: "0 7 * * *"
  workflow_di
jobs:
```

Runs at 07:00.
Actions schedules run at most every 5 minutes using UTC time. Learn more

Figure 6-5 Contextual help when editing workflow syntax in GitHub

- You can also specify workflow_dispatch to get the following manual trigger in the actions tab:

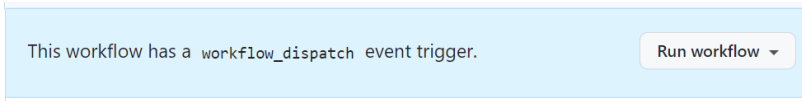


Figure 6-6 Manual trigger for an Action

- Setup MATLAB and install the necessary dependencies in the runner[86]. This should take about 1min30s / 2 min (longer with more toolboxes enabled).
- If the run fails, you will get an email. You can analyze the log:

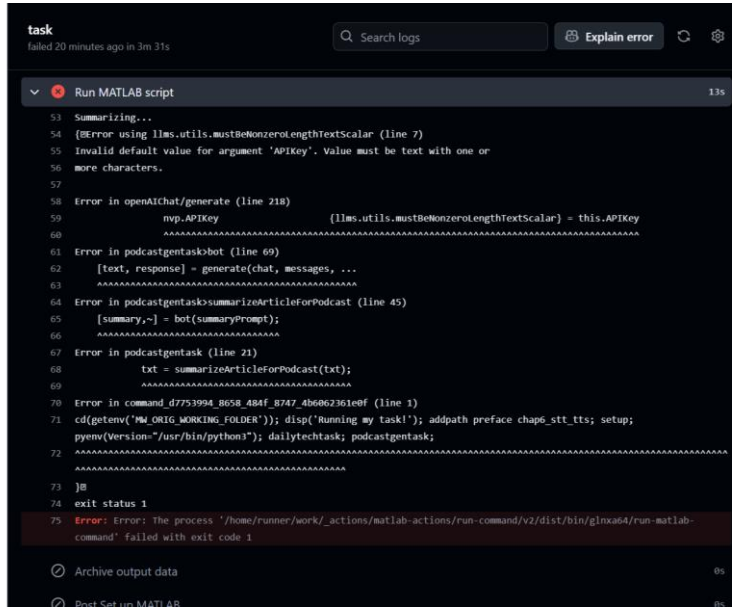


Figure 6-7 Logs for a job called "task" in a GitHub Action

[86] https://github.com/matlab-actions/setup-matlab

- As you can see from the previous error, we forgot to provide the OPENAI_API_KEY. You can set it up as a repository secret, under the repo settings/secrets/actions:

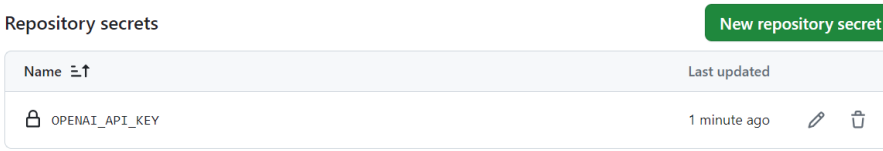


Figure 6-8 Managing secrets as part of a GitHub Action

- Licensing MATLAB for private repos leverages the same mechanism, passing a batch token[87] as a secret. Public repos don't require a license, except for transformation products, such as MATLAB Coder and MATLAB Compiler.

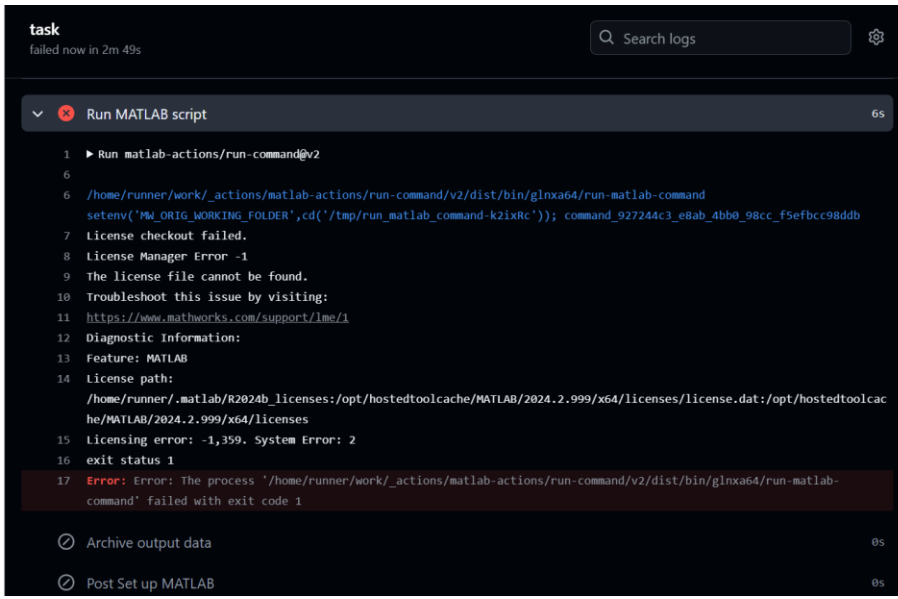


Figure 6-9 License checkout fails if you run in a private repo without a MATLAB batch token

- Once the jobs ran according to plan (which happens), you can upload the resulting artifact[88]:

  Point to the file or the entire directory:

[87] https://github.com/matlab-actions/setup-matlab#licensing
[88] https://github.com/actions/upload-artifact

```
path: path/to/artifact/result.txt
```

```
path: path/to/artifact/
```

Insert a wild card in the path, in order to take variations into account:

```
path: path/**/[abc]rtifac?/*
```

This wildcard pattern matches paths that start with "path/", followed by any number of directories (including none) due to the "**" wildcard, then a directory with any single character, followed by "rtifac", then a single character, and finally, anything (file or directory) due to the last "*". The square brackets "[abc]" indicate that the character at that position can be either 'a', 'b', or 'c'. The question mark "?" indicates that there can be zero or one occurrence of any character at that position.

The artifact will be available in the detail of the run, as a zip file:

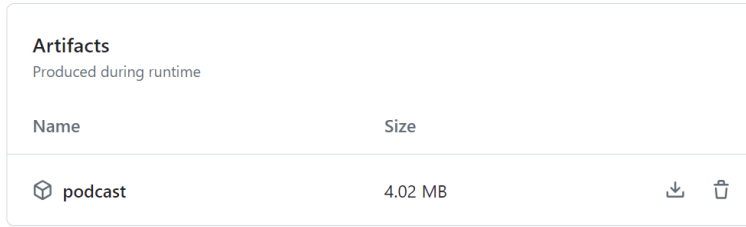


Figure 6-10 Artifacts produced by a GitHub Action

A quick note on the use of GitHub actions for this repo. If you navigate to this tab, you will see that actions are also triggered every time a new change has been pushed. It is building a website that renders all the markdown files of the repo as html on: yanndebray.github.io/programming-GPTs/

This other amazing capability is called GitHub Pages[89].

[89] https://pages.github.com/

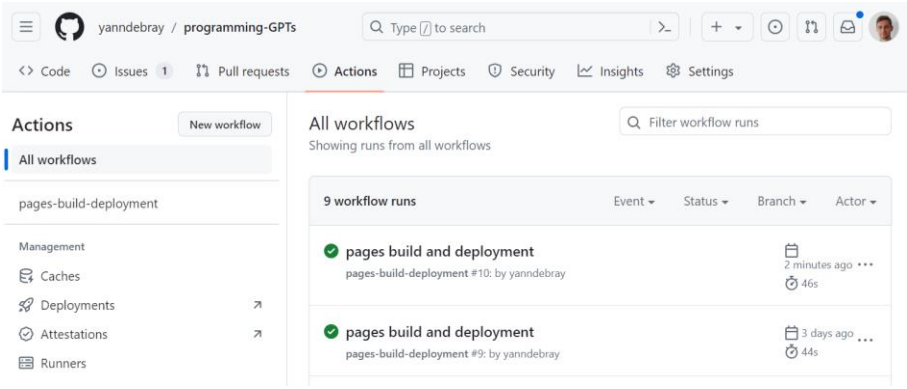


Figure 6-11 Dashboard of all the workflows that ran for a set of GitHub Actions in a repo

7. VISION

FROM TRADITIONAL COMPUTER VISION TO MULTI-MODAL LANGUAGE MODELS

Computer vision is the field of computer science that deals with enabling machines to understand and process visual information, such as images and videos. Computer vision has many applications, such as autonomous driving, medical imaging, objects or humans detection and augmented reality. However, computer vision alone cannot capture the full meaning and context of visual information, especially when it comes to natural language tasks, such as captioning, summarizing, or answering questions about images. For example, a computer vision system may be able to identify objects and faces in an image, but it may not be able to explain their relationships, emotions, or intentions.

This is where "multi-modality" comes in. With regard to LLMs, modality refers to data types. Multimodal language models are systems that can process and generate both text and images and learn from their interactions. By combining computer vision and natural language processing, multimodal language models can achieve a deeper and richer understanding of visual information and produce more natural and coherent outputs. Multimodal language models can also leverage the large amount of available text and image data on the web and learn from their correlations and alignments.

In March 2023 during a GPT-4 developer demo livestream[90], we got to witness the new vision skills. GPT-4 can process images and answer questions about them. Language model systems used to only take one type of input, text. But what if you could provide the model with the ability to "see"? This has been made possible by GPT-4. And in May 2024, OpenAI brought significant upgrades with GPT-4o ("o" for "omni").

Let's start with a basic example of the capabilities of GPT-4o. You can pass local images along with the request, or pass an url link to the image online.

```
prompt = "What is in the image?";
image = "img/parrot8bits_400-400.png";
chat = openAIChat(ModelName="gpt-4o-mini");
messages = messageHistory;
```

[90] GPT-4 Developer Livestream https://www.youtube.com/watch?v=outcGtbnMuQ

```
messages = addUserMessageWithImages(messages,prompt, image_url);
generate(chat, messages)
```

```
ans = "The image features a colorful, pixel art representation of a parrot, prominently displaying vibrant shades of blue and orange. The bird's feathers are detailed, showcasing a stylized appearance typical of retro video game art."
```

```
image_url="https://raw.githubusercontent.com/yanndebray/programming-GPTs/main/img/dwight-internet.jpg";
im = imread(image_url);
imshow(im)
```

Figure 7-1 Image from the TV show The Office

```
chatVision(prompt, image)
```

```
ans = "This image features two characters from a television show. They appear to be talking while walking outside past a parked car. The character on the left is wearing a mustard-colored shirt with a patterned tie and glasses, and the character on the right is wearing a dark suit with a blue tie. There is also a subtitle overlay that reads, "They're gonna be screwed once this whole internet fad is over." This subtitle suggests that the scene might be humorous or ironic, especially since the "internet fad" has proven to be a fundamental part of modern society."
```

OBJECT DETECTION

Object detection[91] is the task of identifying and locating objects of interest in an image or a video. Object detection can be useful for various applications, such as recognizing road signs to assist in driving cars. There are different AI approaches to object detection, such as:

- *Template matching*: This approach involves comparing a template image of an object with the input image and finding the best match. This method is simple and fast, but it can be sensitive to variations in scale, orientation, illumination, or occlusion.

- *Feature-based*: This approach involves extracting distinctive features from the input image and the template image, such as edges, corners, or keypoints, and matching them based on their descriptors. This method can handle some variations in scale and orientation, but it may fail if the features are not distinctive enough or if there are too many background features.

- *Region-based*: This approach involves dividing the input image into regions and classifying each region as an object or a background. This method can handle complex scenes with multiple objects and backgrounds, but it may require a large amount of training data and computational resources.

- *Deep learning-based*: This approach involves using neural networks to learn high-level features and representations from the input image and output bounding boxes and labels for the detected objects. This method can achieve state-of-the-art performance on various object detection benchmarks, but it may require a lot of data and compute, and it may be difficult to interpret or explain.

- *LLM-based*: This approach involves using a pretrained language and vision model, such as GPT-4o, to input the image and the text as queries and generate an answer based on both. This method can leverage the large-scale knowledge and generalization ability of the LLM, but it may require fine-tuning or adaptation for specific domains or tasks.

[91] https://www.mathworks.com/discovery/object-detection.html

APPLICATION: DETECTING CARS

Let's take an example from a self-driving dataset from Udacity[92]. This project starts with 223GB of open-source Driving Data. The repo[93] has been archived, but you can still access it. Streamlit developed an associated app[94] and hosted data on AWS[95].

The AWS S3 bucket is publicly available, so we can access the content list as XML:

```
<ListBucketResult xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
    <Name>streamlit-self-driving</Name>
    <MaxKeys>1000</MaxKeys>
    <IsTruncated>true</IsTruncated>
    <Contents>
        <Key>1478019952686311006.jpg</Key>
        <LastModified>2019-09-03T22:56:13.000Z</LastModified>
        <ETag>"17593334a87be9a26a6caa1080d32137"</ETag>
        <Size>27406</Size>
        <StorageClass>STANDARD</StorageClass>
    </Contents>
```

Let's navigate the list:

```
 % URL of the (public) S3 bucket
 bucketURL="https://streamlit-self-driving.s3-us-west-
2.amazonaws.com/";
 % Save the XML locally
 xmlFileName = "bucket_list.xml";
 websave(xmlFileName, bucketURL);
 % Read and parse the XML directly from the URL as a struct
 S = readstruct(xmlFileName)
```

```
S = struct with fields:
    xmlnsAttribute: "http://s3.amazonaws.com/doc/2006-03-01/"
              Name: "streamlit-self-driving"
            Prefix: ""
            Marker: ""
```

[92] https://medium.com/udacity/open-sourcing-223gb-of-mountain-view-driving-data-f6b5593fbfa5
[93] Udacity repo: https://github.com/udacity/self-driving-car
[94] Streamlit app: https://github.com/streamlit/demo-self-driving/blob/master/streamlit_app.py
[95] AWS S3 bucket: https://streamlit-self-driving.s3-us-west-2.amazonaws.com/

```
          MaxKeys: 1000
      IsTruncated: "true"
         Contents: [1×1000 struct]
```

```
T = struct2table(S.Contents)
```

T = 1000×5 table

	Key	LastModified	…
1	"1478019952686311006.jpg"	"2019-09-03T22:56:13.000Z"	
2	"1478019953180167674.jpg"	"2019-09-04T00:28:39.000Z"	
3	"1478019953689774621.jpg"	"2019-09-03T22:57:41.000Z"	
4	"1478019954186238236.jpg"	"2019-09-03T23:04:32.000Z"	
5	"1478019954685370994.jpg"	"2019-09-03T23:01:42.000Z"	

```
image = T.Key(48)
```

```
image = "1478019976185898081.jpg"
```

```
websave(T.Key(48),bucketURL+T.Key(48));
im = imread(bucketURL+T.Key(48));
imshow(im)
```

Figure 7-2 Image from the Udacity self driving car dataset

LLM-BASED OBJECT DETECTION

As mentioned in the beginning of this section, GPT-4 is quite flexible in the type of request you can formulate through natural language, however it will not be as efficient as traditional computer vision methods to detect objects.

```
 prompt = " Detect a car in the image. Provide x_min, y_min, x_max, ymax coordinates as json";
 response = chatVision(prompt, image)
```

```
response =
    "{
       "car": {
         "x_min": 150,
         "y_min": 210,
         "x_max": 220,
         "y_max": 250
       }
     }"
```

```
 % Draw a bounding box around the car
 coords = jsondecode(response).car;
 imshow(im, 'Parent', axesHandle);
 hold on; set(axesHandle, 'Color', 'none');
 rectangle('Position', [coords.x_min, coords.y_min, coords.x_max - coords.x_min, coords.y_max - coords.y_min], 'EdgeColor', 'r', 'LineWidth', 2); hold off
```

Figure 7-3 Bounding box attempting to catch a car

As you can see, this clearly isn't a success yet for object detection and localization.

Traditional Computer Vision

YOLO (You Look Only Once)[96] is a state-of-the-art, real-time object detection system. It is fast and accurate. You can use a pretrained YOLO model to detect cars[97] with the Automated Driving Toolbox.

```
% Load the pre-trained vehicle detector
% (requires Automated Driving Toolbox)
detector = vehicleDetectorYOLOv2();

% Perform vehicle detection on the image
[bboxes, scores, labels] = detect(detector, im);

% Draw bounding boxes around detected vehicles
hold on;
imshow(im)
for i = 1:size(bboxes, 1)
    rectangle('Position', bboxes(i, :), 'EdgeColor', 'g',
'LineWidth', 2);
    text(bboxes(i, 1), bboxes(i, 2), string(labels(i)), 'Color',
'g', 'FontSize', 12);
end
hold off;
```

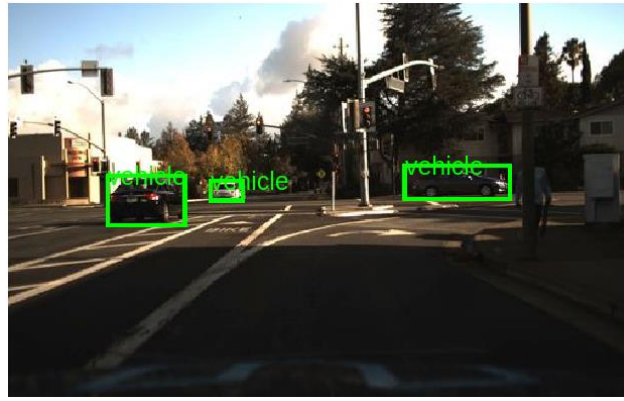


Figure 7-4 Better bounding boxes for cars

Another project to add to my TODO list is setting up my Raspberry Pi with a camera pointed at a parking spot to notify me when the spot is available.

[96] YOLO website: https://pjreddie.com/darknet/yolo/
[97] https://www.mathworks.com/help/driving/ref/vehicledetectoryolov2.html

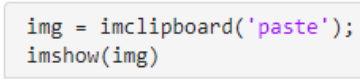
Optical Character Recognition

Optical character recognition (OCR) is a computer vision task that involves extracting text from images, such as scanned documents, receipts, or signs. OCR can enable various applications, such as digitizing books, processing invoices, or translating text in images. However, traditional OCR methods have some limitations, such as:

- They may not handle noisy, distorted, or handwritten text well, especially if the text is in different fonts, sizes, or orientations.
- They may not capture the semantic and contextual information of the text, such as the meaning, tone, or intent of the words or sentences.
- They may not integrate visual and textual information of the image, such as layout, colors, or symbols that may affect the interpretation of the text.

Let's try out GPT-4o with an easy example of code copied with a simple printscreen (`win+shift+s`). You can also use the Windows snipping tool to grab images on your screen. This requires a function from File Exchange[98] (it doesn't support MATLAB Online):

```
img = imclipboard('paste');
imshow(img)
```




```
imwrite(img,"printscreen.png")
code = chatVision("Extract the text from the image, return only the code, without markdown formatting.","printscreen.png")
```

```
code =
    "img = imclipboard('paste');
     imshow(img)"
```

Let's implement a simple app that will provide us with an OCR assistant, with a paste button getting images from the clipboard. This is what the ocrAssistant.mlapp looks like:

[98] https://www.mathworks.com/matlabcentral/fileexchange/28708-imclipboard

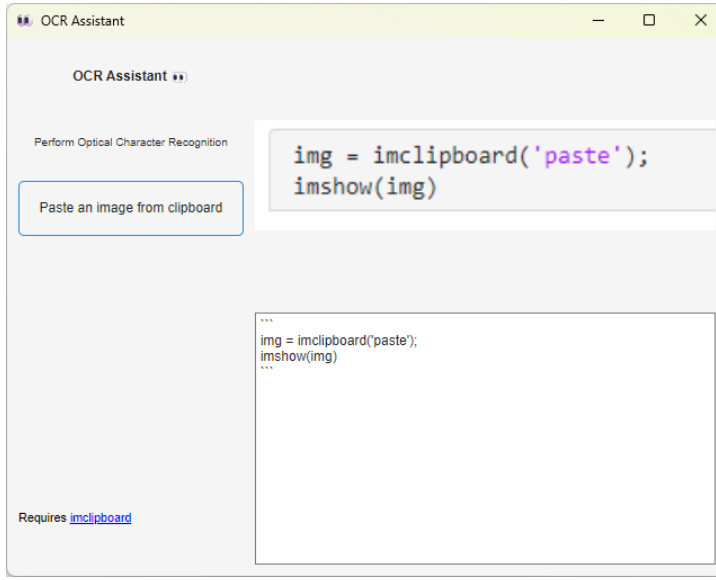


Figure 7-5 Optical Character Recognition app

FROM MOCK TO WEB UI

One exciting application of GPT-4o is to generate code from a hand-drawn mock-up of a webpage. This was part of the GPT-4 unveiling demo[99]. You can take a printscreen from the YouTube video containing the frame of the Mock-up at 18:27.

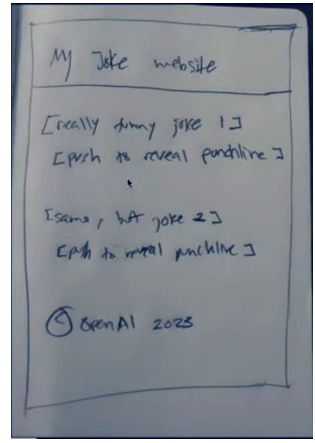


Figure 7-6 Picture of a notebook displaying the mock of a basic website

[99] https://www.youtube.com/watch?v=outcGtbnMuQ&ab_channel=OpenAI

In order to obtain this cropped view of the notebook, open the frame in the *image viewer app*[100] in MATLAB (requires image processing toolbox), to crop only the part that you want to pass to GPT-4o.

```
imageViewer("frame.jpg")
```

Let's reuse the prompt from the demo:

```
imagePath = "cropped_image.jpg";
originalPrompt = "Write brief HTML/JS to turn this mock-up into a colorful website, where the jokes are replaced by two real jokes. (Only the code, no markdown, no explanation)";
code = chatVision(originalPrompt, imagePath)
% % Save the code to a file
fileID = fopen('jokewebsite.html', 'w');
fprintf(fileID, '%s', code);
fclose(fileID);
```

And Voila! We have our website coded for us (the resulted html is served up as a GitHub page): yanndebray.github.io/programming-GPTs/chap7/joke_website

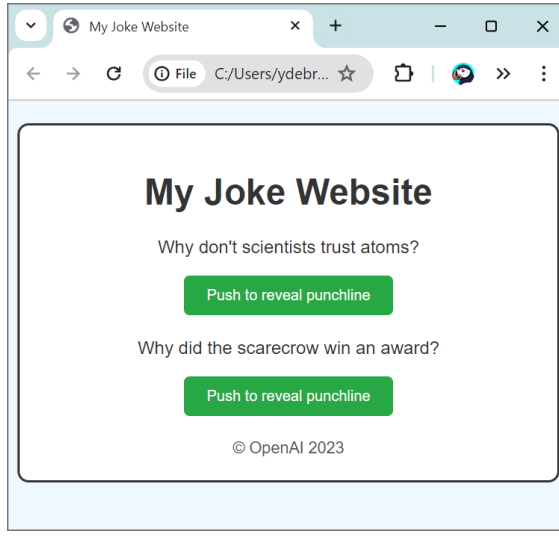


Figure 7-7 Generated website based on the mock

[100] https://www.mathworks.com/help/images/crop-image-using-image-viewer-app.html

VIDEO UNDERSTANDING

Let's look at another use case: Processing and narrating a video with GPT's visual capabilities[101]. For this example, I retrieved a video from Youtube that I created, and you will add a different voiceover to it.

```
if ~exist("video_img\","dir")
    mkdir("video_img")
end
videoPath = "MATLAB and Simulink - The Symphony of Imagination.mp4";
v = VideoReader(videoPath);
imageFiles = strings(0);
frameIdx = 1;
while hasFrame(v)
    frame = readFrame(v);
    if mod(frameIdx, 60) == 1
        % Construct filename and save frame as JPEG
        filename = fullfile("video_img", sprintf("frame_%04d.jpg",
frameIdx));
        imwrite(frame, filename);
        imageFiles(end+1) = filename;
    end
    frameIdx = frameIdx + 1;
end

imshow(frame) % show the last frame
```

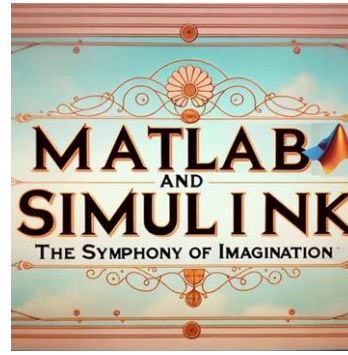


Figure 7-8 Last frame from the video displayed at the end of the loop

[101] https://cookbook.openai.com/examples/gpt_with_vision_for_video_understanding

```
 prompt = "These are frames of a movie trailer. Create a short voiceover script in the style of Wes Anderson. Only include the narration.";
 response = chatVision(prompt, imageFiles,60) % Add timeOut to 60
```

```
response =
    "**[Voiceover Script]**

     In a world where equations dance and dreams intertwine,
     A young visionary named Max embarks on an extraordinary
adventure.

     Amidst charming chaos and whimsical whimsy,
     He discovers that equations come to life,
     Transforming the mundane into the marvelous —
     A symphony of imagination unfolds.

     Guided by the eccentric Prof. Anderson,
     Together, they navigate a vibrant tapestry of numbers,
     Where even the smallest details spark brilliance.

     Join us as we explore this meticulously crafted universe,
     Painted in pastel hues and peculiar patterns,
     Where innovation meets artistry,
     In a delightful collision of heart and mind.

     Prepare for an unforgettable journey,
     In a new film by the inimitable Wes Anderson —
     "MATLAB & Simulink: The Symphony of Imagination.""
```

8. IMAGE GENERATION

This whole journey started for me in the first days of December 2022, right after the release of ChatGPT. But since the API to call the GPT-3.5 model did not become available until March of 2023, I played around with the other services provided by OpenAI. Back then, image generation [102] was one of the APIs that could call programmatically a model named DALL-E[103]. In parallel other Generative AI models, like the open-source Stable Diffusion[104] or the proprietary Midjourney[105] enabled very realistic and artistic images creation. Since then, image generation is integrated in the GPT APIs enabling use cases like generating images with text represented in it.

In the previous chapter, the video in the style of Wes Anderson was completely generated with AI. I even created a mini-series of short videos on TikTok called `Biomachines`.

Figure 8-1 Image generated by Midjourney in a Cyberpunk style

The story was derived from the following prompt:

```
Write a story about humans being biological machines trained by the
experience of life. The story takes place in 2025, after the advent of
large language models
```

[102] https://platform.openai.com/docs/guides/images
[103] https://openai.com/index/dall-e-2/
[104] https://stability.ai/news/stable-diffusion-public-release
[105] https://www.midjourney.com/

In this chapter we will see how to create a graphical novel on the same topic.

The methods investigated are:

- Generation
- Edits (In- & Outpainting)
- Variation

The capabilities accessible through the ChatGPT app are different from what can be done programmatically. For clarity, I'll be explicit in this chapter about the use of the ChatGPT App, and the use of the OpenAI API endpoints. This logo here represents the "GPT" in the ChatGPT app.

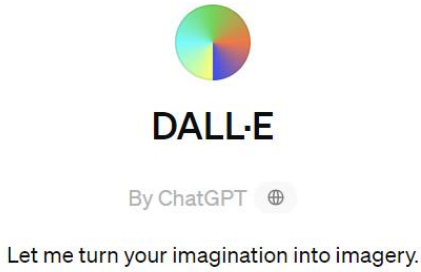


Figure 8-2 Dall-E in the GPT Store

GENERATION

The first step in a story is a character. The difficulty with AI generated image is to maintain consistency in the characters of your plot. With DALL-E in the ChatGPT app, you can iterate and refine your prompt in a conversation with @DALL-E. If you are not happy with a result, you can ask to regenerate the image.

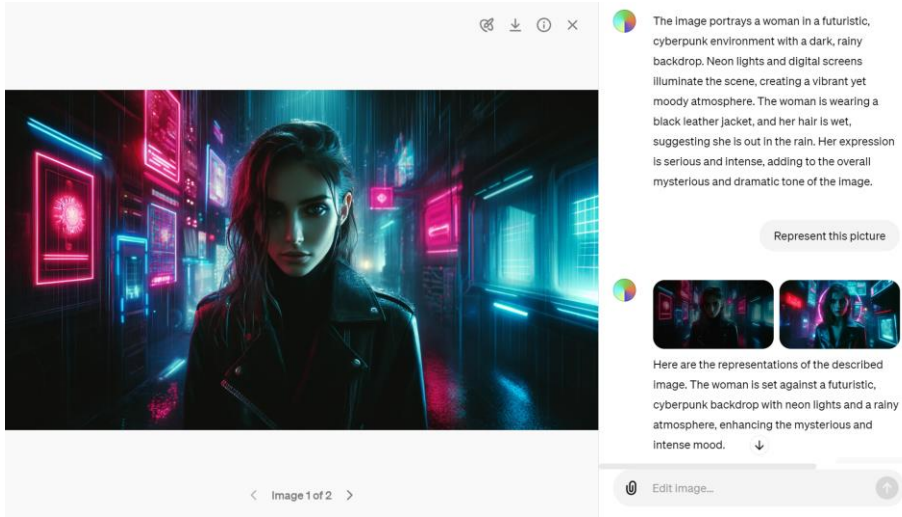


Figure 8-3 Generating images with Dall-E in ChatGPT

With the app you can also upload an existing image to ask ChatGPT to describe it for you. This iterative process is helpful to capture the essence of the scene that you are trying to generate.

The image generations API endpoint allows you to create an original image given a text prompt. With DALL-E 3, images can have a size of 1024x1024, 1024x1792 or 1792x1024 pixels. With DALL-E 2, sizes are 1024x1024, 512x512 or 256x256.

By default, images are generated at standard quality, but when using DALL-E 3 you can set quality: "hd" for enhanced detail. Square, standard quality images are the fastest to generate.

```
 model = openAIImages(ModelName="dall-e-3"); % By default Dall-E 2
 prompt = "An 8-bit pixelated game side scroller called biomachines."
+ ...
     "cyberpunk story about humans being biological machines." + ...
     "Pixelated like NES. Bright colors. Huge glowing pixels.
Pixelated.";
 Im = generate(model,prompt,"Size","1024x1024");
 imshow(Im{1})
```

Figure 8-4 An 8-bit pixelated image generated by Dall-E 3

To perform edits and variations, you will need to generate square images (ratio 1:1).

EDITS

Edits can be performed interactively with the ChatGPT app by selecting a region in the image:

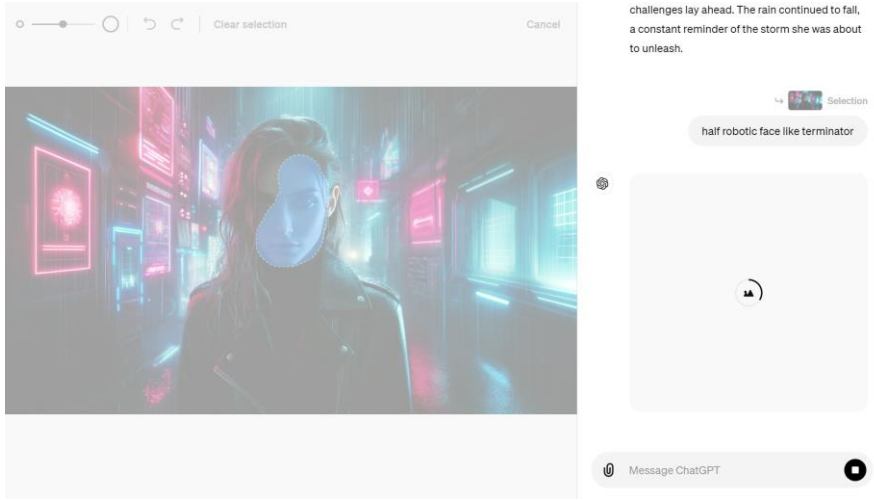


Figure 8-5 Edit images interactively in ChatGPT

Figure 8-6 Modified image of a cyberpunk character

"Inpainting" via the API are only available with DALL-E 2. The image edits endpoint allows you to edit or extend an image by uploading an image and mask indicating which areas should be replaced. The transparent areas of the mask indicate where the image should be edited, and the prompt should describe the full new image, *not just the erased area*.

To perform this operation manually, you can use Paint.net[106] to erase the masked area in the picture.

```
 mdl = openAIImages(ModelName="dall-e-2");
 imagePath = "biomachines-8bits_1024x1024.png";
 editPrompt = "Terminator giant skull face in a cyberpunk 8 bits
pixelated game";
 [images,resp]=edit(mdl, imagePath, editPrompt,
MaskImagePath=maskImagePath);
 if isfield(resp.Body.Data,'data')
     figure
     imshow(images{1});
 else
     disp(resp.Body.Data.error)
 end
```

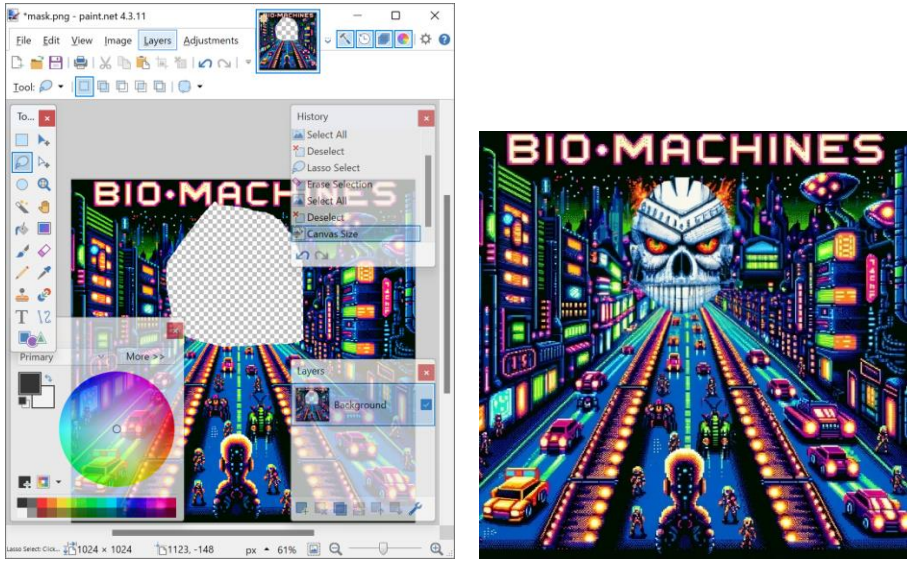


Figure 8-7 Inpainting with a mask created in Paint.NET

"Outpainting" is following the same principal as "inpainting", but you only need to shift the image outside of the canvas in the direction that you want to fill.

[106] https://www.getpaint.net/

VARIATIONS

Variations are easy to understand. You input an image of the right format, and get another flavor of it:

```
imagePath = "biomachines_car.png";
[images,resp] = createVariation(mdl, imagePath, NumImages=2);
if ~isempty(images)
    tiledlayout('flow')
    for ii = 1:numel(images)
        nexttile
        imshow(images{ii})
    end
else
    disp(resp.Body.Data.error)
end
```

You can get several variants, by entering n=2 for instance.

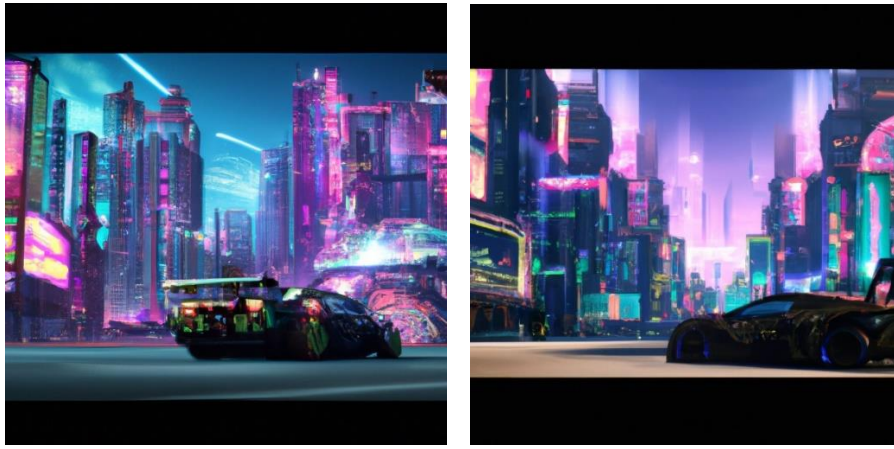


Figure 8-8 Variants of an image with Dall-E 2

9. Appendix

More AI theory

Machine Learning

Machine learning is a branch of artificial intelligence that enables computers to learn from data and experience, without being explicitly programmed for every possible scenario. This is often what is meant when referring to AI. It can be used to solve complex problems that are hard to codify with rules, such as image recognition, natural language processing, or recommendation systems.

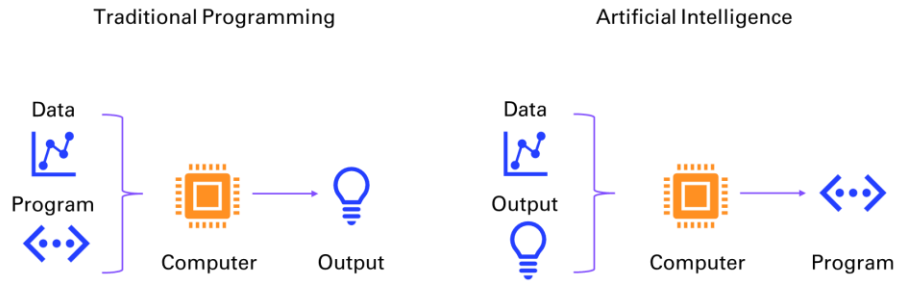


Figure 9-1 Traditional programming vs AI

Machine learning can be broadly divided into two categories: supervised and unsupervised learning. Supervised learning is when the computer learns from labeled data, that is, data that has a known output or target. For example, in image recognition, the computer learns from images that are labeled with their corresponding categories, such as cat, dog, or car. The goal of supervised learning is to train the computer to predict the correct output for new data that it has not seen before.

Supervised learning can be further divided into two types: regression and classification. Regression is when the output is a continuous value, such as

temperature, price, or speed. Classification is when the output is a discrete category, such as spam or not spam, positive or negative, or one of several classes.

Unsupervised learning is when the computer learns from unlabeled data, that is, data that has no known output or target. For example, in text analysis, the computer learns from documents that are not labeled with any topic or sentiment. The goal of unsupervised learning is to discover hidden patterns or structures in the data, such as clusters, outliers, or features.

Unsupervised learning can be mainly divided into two types: clustering and dimensionality reduction. Clustering is when the computer groups similar data points together based on some measure of similarity or distance, such as k-means, hierarchical clustering, or Gaussian mixture models. Dimensionality reduction is when the computer reduces the number of features or dimensions of the data, while preserving as much information as possible, such as principal component analysis, linear discriminant analysis, or autoencoders.

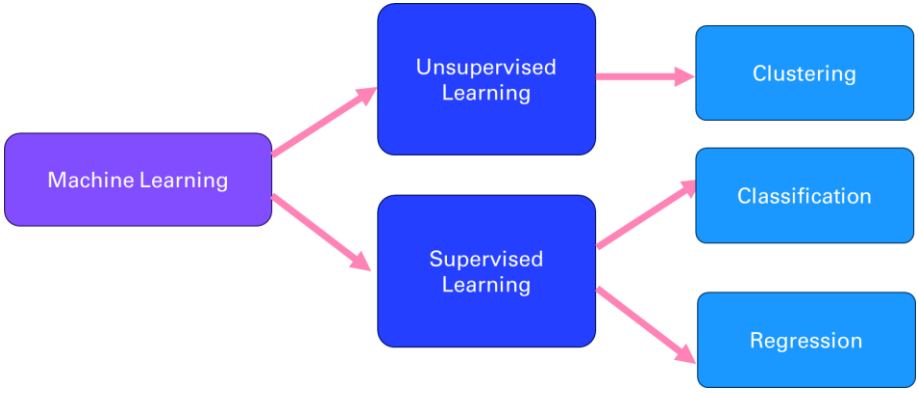


Figure 9-2 Different branches of Machine Learning

Deep Learning

Deep learning is a branch of machine learning that uses neural networks mimicking the structure of the brain in the way it processes information in neurons. Deep learning can handle complex and high-dimensional data, such as images, speech, or natural language, and perform tasks such as object recognition, speech recognition, natural language processing, or machine translation. Deep learning is inspired by the discoveries of neuroscience and cognitive science, and relies on advances in mathematical optimization, parallel computing, and big data.

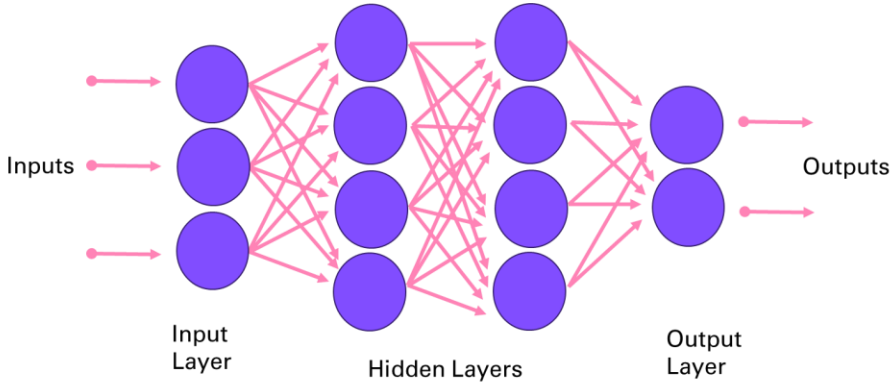


Figure 9-3 Deep Neural Network architecture

A few key research breakthroughs made possible the innovations presented in this book, starting with *Yann LeCun et al* (1989) Backpropagation Applied to Handwritten Zip Code Recognition[107]. This paper that is the same age as I introduced way back the potential of neural network for image processing, on the famous MNIST dataset[108].

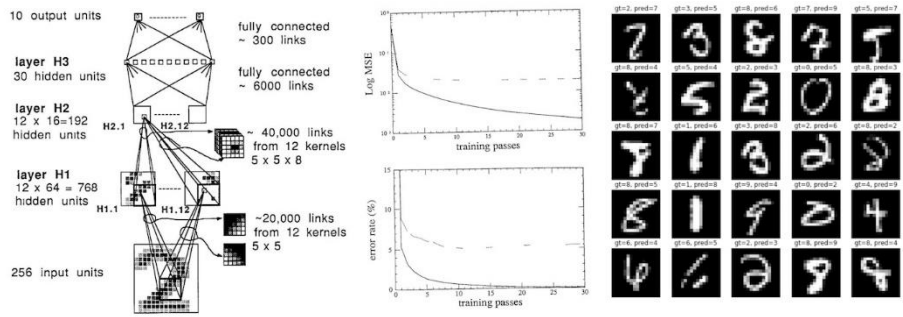


Figure 9-4 LeNet applied to digits recognition

Natural Language Processing

Natural language processing (NLP) is the field of artificial intelligence that deals with understanding and generating natural language, such as text or speech. NLP has many applications, such as question answering, sentiment analysis, machine translation, summarization, dialogue systems, information extraction, and more. NLP

[107] http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf
[108] http://yann.lecun.com/exdb/mnist/

faces many challenges, such as ambiguity, variability, complexity, and diversity of natural language.

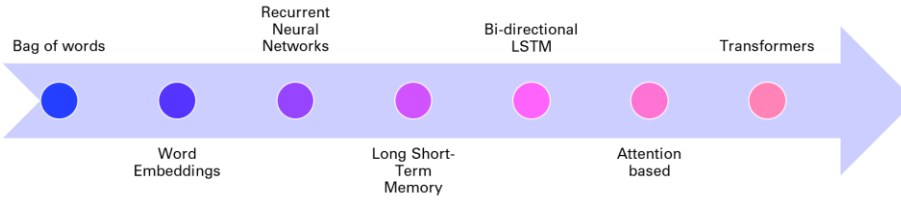


Figure 9-5 History of NLP techniques

One of the key tasks in NLP is to represent natural language in a way that computers can understand and manipulate. Traditionally, this was done by using rule-based or statistical methods to extract features from words, such as their part-of-speech, syntactic structure, semantic role, or frequency. However, these methods often require a lot of human effort and domain knowledge and cannot capture the rich and dynamic nature of natural language.

To overcome these limitations, deep learning methods have been developed to learn distributed representations of natural language, also known as embeddings, from large amounts of data. As mentioned in chapter 4, embeddings are vectors that encode the meaning and usage of words or sentences in a low-dimensional space, and can be used as input or output for various NLP tasks. Embeddings can capture the semantic and syntactic similarities and relationships between words or sentences, and can also adapt to new domains and languages.

One of the first methods to learn word embeddings was the bag-of-words model, which represents a document as a vector of word frequencies, ignoring the order and context of words. The bag-of-words model is simple and efficient, but it suffers from sparsity, dimensionality, and lack of semantics. To address these issues, neural network models such as word2vec and GloVe were proposed to learn word embeddings from the co-occurrence patterns of words in large corpora, using techniques such as skip-gram and negative sampling. These models can learn more expressive and dense word embeddings, but they still treat words as independent units, ignoring their morphology and compositionality.

To account for the sequential and hierarchical structure of natural language, recurrent neural networks (RNNs) were introduced to learn sentence or document embeddings from word embeddings. RNNs are neural networks that process sequential data by maintaining a hidden state that captures the history of previous inputs. RNNs can learn long-term dependencies and generate variable-length outputs, making them suitable for tasks such as language modeling, machine translation, or text generation. However, RNNs also face some challenges, such as vanishing or exploding gradients, difficulty in parallelization, and sensitivity to noise.

To improve the performance and stability of RNNs, variants such as long short-term memory (LSTM) were developed to introduce gates that control the flow of information in the hidden state. These gates can learn to remember or forget relevant or irrelevant information over time and can handle long-term dependencies better than vanilla RNNs. LSTM have achieved state-of-the-art results on many NLP tasks, such as machine translation, speech recognition, or sentiment analysis.

However, even LSTM have some limitations, such as the inability to model long-range dependencies beyond a fixed window, the sequential nature of computation that limits parallelization, and the lack of attention mechanisms that can focus on relevant parts of the input or output. To overcome these limitations, a new paradigm of neural network models was proposed, based on the concept of transformers.

TRANSFORMERS

You might wonder what's happening under the hood of Large Language Models. Let's dive into the different parts of the model architecture discussed in chapter 1. Figure 10-5 shows the components of a transformer, as introduced in the paper "Attention Is All You Need":

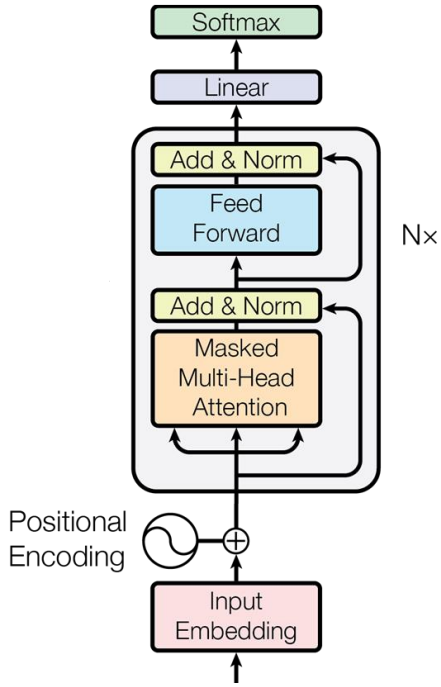


Figure 9-6 Architecture of a generative pre-trained transformer

Input Preprocessing: From Words to Vectors

The preprocessing of the input text proceeds in three steps:

1. *Tokenization:* Turning the words into numbers
2. *Embedding:* Compressing the numbers into a dense vector representation
3. *Positional Encoding:* Storing the position of the words in the sentence

Each step - from tokenization to embedding to positional encoding - plays a critical role in enabling the model to interpret text. Let's take a closer look at these preprocessing steps and their importance in making sense of how LLMs parse their input.

Tokenization

AI models cannot directly understand or process raw text as humans do. Computers only understand numbers. The first step (not shown in Figure 1-7) is breaking down the text into smaller units called *tokens*. These tokens typically represent words, subwords, or even characters, depending on the tokenization strategy. The tokenization step converts the input text into a sequence of tokens.

Embedding

Once tokenized, each token is then transformed into a dense vector of numbers. This process is called embedding. An *embedding* is a learned representation of the tokens in a continuous vector space where semantically similar tokens are closer together. This numeric representation is what the model actually processes.

Positional encoding

Unlike traditional models, transformers do not have a built-in sense of the order of tokens in a sequence. To provide this information, *positional encoding* is added to the token embeddings. This encoding incorporates information about the position of each token in the sequence, allowing the model to understand the order and relationships between tokens over different positions.

Decoder Block: Attention (Plus Compute) Is All You Need

The core of GPT models is a stack of decoder blocks, each composed of two main layers, attention and feed forward, as well as addition and normalization layers completing the two main layers.

The decoder blocks are repeated N times (96 decoder blocks for GPT 3) as displayed on Figure 10-5 to form the main building blocks of the transformer architecture.

Attention

GPT uses a variant of attention known as *causal* or *masked* attention. This type of attention ensures that when generating a token, the model can only attend to previous tokens and not future ones. This is crucial for autoregressive generation, where each token is predicted based on the preceding tokens.

To capture different aspects of the input sequence, the model employs *multiple attention heads* that operate in parallel. Each head focuses on different parts of the input sequence, allowing the model to learn multiple representations of the sequence's information. The outputs from all the heads are then combined and passed on to the next layer.

Feed forward

After the attention mechanism comes a layer of pure compute. The output is passed through a feed-forward network, typically a multi-layer perceptron (MLP). This layer applies a non-linear transformation to the data, allowing the model to capture complex patterns and relationships within the data. The feed-forward network operates independently on each position, meaning it processes each token separately but consistently.

Addition and Normalization

After the multi-head attention and feed-forward layers, the output is combined with the input to that layer, symbolized by the link skipping these layers in Figure 1-7. This addition helps in retaining information from earlier layers, mitigating the vanishing gradient problem, and making it easier for the network to learn.

Following the addition step, normalization helps keep the training process stable and efficient by preventing large shifts in the input distributions as the model learns, which could otherwise disrupt or slow down learning.

Output Postprocessing: Back to Words, with some Probabilities

Once GPT models process input text through multiple layers of transformation and understanding, they need a way to translate their internal representations back into readable words or tokens. These final sets of layers are where abstract vector-based representations from the model's hidden layers are transformed into probabilities that guide token selection. The postprocessing steps involve passing the model's output through a linear layer and softmax function, which together assign probabilities to each possible token in the vocabulary, determining the most likely continuation of the text.

Linear layer

The output from the final decoder block is passed through a linear layer, which maps the high-dimensional representation of each token back to the size of the vocabulary. This step produces a raw score (logits) for each possible token in the vocabulary, indicating how likely each token is to appear next in the sequence.

Softmax function

Finally, the logits are passed through a softmax function, which converts them into probabilities. The softmax function ensures that the output values are between 0 and 1 and that they sum to 1 across the vocabulary. The token with the highest probability is selected as the model's output, representing the most likely next token in the sequence.

This architecture remains surprisingly very similar since the original transformers paper. One notable difference is the application of the layer norm before the attention mechanism in more recent transformers.

OLLAMA

Ollama[109] Ollama is a server for open-source LLMs. Download on your laptop and select the open-source LLMs you want to serve up locally:

```
$ ollama run llama3.2:3b
pulling manifest
pulling dde5aa3fc5ff: 100% ▕████████████████████████████████████▏ 2.0 GB
pulling 966de95ca8a6: 100% ▕████████████████████████████████████▏ 1.4 KB
pulling fcc5a6bec9da: 100% ▕████████████████████████████████████▏ 7.7 KB
pulling a70ff7e570d9: 100% ▕████████████████████████████████████▏ 6.0 KB
pulling 56bb8bd477a5: 100% ▕████████████████████████████████████▏  96 B
pulling 34bb5ab01051: 100% ▕████████████████████████████████████▏ 561 B
verifying sha256 digest
writing manifest
removing any unused layers
success
```

Once you successfully retrieved the weights of the model (here 2Gb for the 3B Llama3.2 model), you can start interacting with the command line:

```
>>> Send a message (/? for help)
```

You can use the Ollama chat client[110] to build local LLMs applications, as an alternative to OpenAI. Depending on your laptop resources (CPU, GPU and RAM) you might have a very slow response compared to what you are used to with GPT-4o-mini. I am serving the model from my mac mini m4, and the throughput is very

[109] https://ollama.com/

[110] https://github.com/matlab-deep-learning/llms-with-matlab/blob/main/doc/functions/ollamaChat.md

decent. Plus I am using a tunnel, in order to make my machine accessible from a public URL (be careful when you do that). Here is an example of a simple chatbot:

```
 ollamaServer = "<your-tunnel>.ngrok-free.app"; % replace with your
ollama server
 chat = ollamaChat("llama3.2", EndPoint=ollamaServer);
 messages = messageHistory;
 while true
     query = input("User: ", "s");
     query = string(query);
     dispWrapped("User", query)
     if query == stopWord
         disp("AI: Closing the chat. Have a great day!")
         break;
     end
     messages = addUserMessage(messages, query);
     [text, response] = generate(chat, messages);

     dispWrapped("AI", text)
     messages = addResponseMessage(messages, response);
 end
```

```
User: hello
AI: Hello! How can I assist you today?
User: what's your name
AI: I don't have a personal name, but I'm an AI designed to assist
and
    communicate with users. You can think of me as a conversational
    partner or a helpful friend. Some people refer to me as
"Assistant"
    or "AI Companion". What would you like to talk about?
…

User: end
AI: Closing the chat. Have a great day!
```

This kind of setup can be useful for batch workflows where you want to process sensitive information without having to share it with a web service. Plus you don’t have to pay for each token that you send, so this can be useful for very large amount of text.

MISTRAL

Mistral AI[111] is a French startup that created an open-source LLM competitive with GPT-3.5 in about 1 year and with a team of 20 engineers. Setting aside the Frenchmanhood, I find this very impressive. The members of the founding team were previously employed at Google DeepMind and at FAIR (Facebook AI Research) working on important projects like the Llama model from Meta.

Mistral is famous for releasing one of the first Mixture of Experts (MoE) model, Mixtral 8x7b and 8x22b, the bigger one outperforming larger model for a long time. MoE[112] are pretrained much faster vs. dense models and have faster inference compared to a model with the same number of parameters, but they require high VRAM as all experts are loaded in memory (not great for the 16Gb of RAM of my mac mini m4).

Mistral 7B is their first dense model. It's a good model for function calling, as seen in chapter 5 to build agentic workflows. Another more recent drop-in replacement is Mistral NeMo, a 12B model built in collaboration with NVIDIA, with a context window of 128k tokens (instead of 32k). I will use this one for the next example, that shows how to detect multiple function calls in a single user prompt and use this to extract information from text data.

This example contains four steps:

- Create an unstructured text document containing fictional customer data.
- Create a prompt, asking Ollama to extract information about different types of customers.
- Create a function that extracts information about customers.
- Combine the prompt and the function. Use Ollama to detect how many function calls are included in the prompt and to generate a JSON object with the function outputs.

The customer record contains fictional information.

```
 record = ["Customer John Doe, 35 years old. Email:
johndoe@email.com";
     "Jane Smith, age 28. Email address: janesmith@email.com";
     "Customer named Alex Lee, 29, with email alexlee@email.com";
     "Evelyn Carter, 32, email: evelyncarter32@email.com";
```

[111] https://mistral.ai/
[112] https://huggingface.co/blog/moe

```
    "Jackson Briggs is 45 years old. Contact email:
jacksonb45@email.com";
    "Aria Patel, 27 years old. Email contact: apatel27@email.com";
    "Liam Tanaka, aged 28. Email: liam.tanaka@email.com";
    "Sofia Russo, 24 years old, email: sofia.russo124@email.com"];
```

Define the function that extracts data from the customers record:

```
f = openAIFunction("extractCustomerData", "Extracts data from
customer records");
f = addParameter(f, "name", type="string", description="customer
name", RequiredParameter=true);
f = addParameter(f, "age", type="number", description="customer
age");
f = addParameter(f, "email", type="string", description="customer
email", RequiredParameter=true);
```

Create a message with the customer record and an instruction.

```
record = join(record);
messages = messageHistory;
messages = addUserMessage(messages,"Extract data from the record: "
+ record);
```

Create a chat object. Specify the model to be "*mistral-nemo*", which supports parallel function calls.

```
model = "mistral-nemo";
ollamaServer = getenv("OLLAMA_SERVER");
% ollamaServer = "<your-tunnel>.ngrok-free.app";
chat = ollamaChat(model,"You are an AI assistant designed to extract
customer data.",EndPoint=ollamaServer,Tools=f);
```

Generate a response and extract the data.

```
[~, singleMessage, response] = generate(chat,messages);
if response.StatusCode == "OK"
    funcData = [singleMessage.tool_calls.function];
    extractedData = struct2table([funcData.arguments])
else
    response.Body.Data.error
end
```

```
extractedData = 8×3 table
```

	age	email	name
1	35	'johndoe@email.com'	'John Doe'
2	28	'janesmith@email.com'	'Jane Smith'
3	29	'alexlee@email.com'	'Alex Lee'
4	32	'evelyncarter32@email.com'	'Evelyn Carter'
5	45	'jacksonb45@email.com'	'Jackson Briggs'
6	27	'apatel27@email.com'	'Aria Patel'
7	28	'liam.tanaka@email.com'	'Liam Tanaka'
8	24	'sofia.russo124@email.com'	'Sofia Russo'

In the second part of the example, use a local function *searchCustomerData* defined at the end of this script.

Create a message with the information you would like to get from the local function.

```
prompt = "Who are our customers under 30 and older than 27?";
messages = messageHistory;
messages = addUserMessage(messages,prompt);
```

Define the function that retrieves customer information:

```
f = openAIFunction("searchCustomerData", "Get the customers who match the specified and age");
f = addParameter(f,"minAge",type="integer",description="The minimum customer age",RequiredParameter=true);
f = addParameter(f,"maxAge",type="integer",description="The maximum customer age",RequiredParameter=true);
```

Create a chat object with a model supporting function calls.

```
model = "mistral-nemo";
chat = ollamaChat(model,"You are an AI assistant designed to search customer data.", EndPoint=ollamaServer,Tools=f);
```

Generate a response

```
[~, singleMessage, response] = generate(chat,messages);
```

Check if the response contains a request for tool calling. Use *jsonencode* to display nested struct compactly.

```
if isfield(singleMessage,'tool_calls')
    tcalls = singleMessage.tool_calls;
    disp(jsonencode(tcalls,PrettyPrint=true))
else
    response.Body.Data.error
end
```

```
{
  "function": {
    "name": "searchCustomerData",
    "arguments": {
      "maxAge": 30,
      "minAge": 28
    }
  }
}
```

And add the tool call to the message history.

```
messages = addResponseMessage(messages,singleMessage);
```

Call the searchCustomerData function and add the results to the messages.

```
messages = processToolCalls(extractedData,messages, tcalls);
```

Finally, generate a response with the function result.

```
[txt, singleMessage, response] = generate(chat,messages);
if response.StatusCode == "OK"
    txt
else
    response.Body.Data.error
end
```

```
txt =
    " Our customers under 30 and older than 27 are:

     - **Jane Smith**: Age: 28, Email: janesmith@email.com
     - **Alex Lee**: Age: 29, Email: alexlee@email.com
     - **Liam Tanaka**: Age: 28, Email: liam.tanaka@email.com"
```