Lu Sun, and many more.

A Notebook on Artificial Intelligence

To my family, friends and communities members who have been dedicating to the presentation of this notebook, and to all students, researchers and faculty members who might find this notebook helpful.

Contents

Foreword

If software and e-books can be made completely open-source, why not a notebook?

This brings me back to the summer of 2009 when I started my third year as a high school student in Harbin No. 3 High School. In the end of August when the results of Gaokao (National College Entrance Examination of China, annually held in July) are released, people from photocopy shops would start selling notebooks photocopies that they claim to be from the top scorers of the exam. Much curious as I was about what these notebooks look like, never have I expected myself to actually learn anything from them, mainly for the following three reasons.

First of all, some (in fact many) of these notebooks were more difficult to understand than the textbooks. I guess we cannot blame the top scorers for being so smart that they sometimes make things extremely brief or overwhelmingly complicated.

Secondly, why would I want to adapt to notebooks of others when I had my own notebooks which in my opinion should be just as good as theirs.

And lastly, as a student in the top-tier high school myself, I knew that the top scorers are probably my schoolmates or even friends. Why would I pay money to a complete stranger in a photocopy shop for my friends' notebook, rather than requesting a copy from them directly?

However, my mind changed after becoming an undergraduate student in 2010. There were so many modules and materials to learn for a college student, and as an unfortunate result, students were often distracted from digging deeply into a module (For those who were still able to do so, you have my highest respect). The situation became worse when I started pursuing my Ph.D. in 2014. As I had to focus on specific research areas entirely, I could hardly split much time on other irrelevant but still important and interesting contents.

In order to make a difference, I started enforcing myself reading articles beyond my comfort zone, which ended up motivating me to take notes to consolidate the knowledge. I used to work with hand-written notebooks. My very first notebook was on *Numerical Analysis*, an entrance level module for engineering background graduate students. Till today I still have dozens of these notebooks on my bookshelf. Eventually, it came to me: why not digitizing them, making them accessible online and open-source and letting everyone read and edit it?

—

As most of the open-source software, this notebook does not come with any "warranty" of any kind, meaning that there is no guarantee that everything in this notebook is correct, and it is not peer reviewed. **Do NOT cite this notebook in your academic research paper or book!** If you find anything helpful here with your research, please trace back to the origin of the knowledge and confirm by yourself.

This notebook is suitable as:

- a quick reference guide;
- a brief introduction for beginners of an area;
- a "cheat sheet" for students to prepare for the exam or for lecturers to prepare the teaching materials.

This notebook is NOT suitable as:

- a direct research reference;
- a replacement of the textbook;

because as explained the notebook is NOT peer reviewed and after all, it is more of a notebook than a book. It is meant to be easy to read, not to be comprehensive.

Although this notebook is open-source, the reference materials of this notebook, including textbooks, journal papers, conference proceedings, etc., may not be open-source. Very likely many of these reference materials are licensed or copyrighted. Please legitimately access these materials and properly use them, should you decided to trace the origin of the knowledge.

Some of the figures in this notebook are plotted using Excalidraw, a very interesting tool to emulate hand drawings. The Excalidraw project can be found on GitHub, *excalidraw/excalidraw*. Other figures may come from MATLAB, R, Python, and other computation engines. The source code to reproduce the results are intended to be included in the same repository of the notebook, but there might be exceptions.

This work might have benefited from the assistance of large language models, which are used exclusively for editing purposes such as correcting grammar and rephrasing sentences, without introducing new content, generating novel information, or changing the original intent of the text.

Preface

Artificial Intelligence (AI) was included as part of the control system notebook, as in early ages it was mostly used as a system identification tool in control systems.

With the advent of graphical processing units (GPU) in the 1990th and the Industry 4.0 initiatives in 2000th, deep learning network with massive training data became possible, which significantly boosted the performance of artificial neural network (ANN)-based AI systems. Nowadays, AI has been growing rapidly with successful demonstrations of use cases such as computer vision and natural language processing.

Seeing that trend, AI relevant contents have been separated from the control system notebook and collected here.

Special thanks go to the following materials, all of which have been very useful when drafting this notebook. Notice that the referenced contents from the following materials will not be listed separately in the Reference section in the end of the notebook.

- Russell, Stuart J., and Peter Norvig. Artificial Intelligence: a Modern Approach. Pearson, 2016.
- Lakshmanan, Valliappa, Martin Görner, and Ryan Gillard. Practical Machine Learning for Computer Vision. "O'Reilly Media, Inc.", 2021.
- Ed Donner, Ligency Team, LLM Engineering: Master AI, Large Language Models & Agents. Udemy, 2025

List of Figures

List of Tables

Part I

Artificial Intelligence

Part II

Basic Artificial Neural Networks

1

Regression

CONTENTS

2

Perceptron

CONTENTS

3

Multi-layer Perceptron

CONTENTS

Part III

Convolution and Recurrent Networks

4

Convolutional Neural Network

CONTENTS

“nobreak

4.1 Brief Review of CNN

CNN is a type of ANN structure designed to handle grid-like data. It is effective when dealing when data spatially correlated, thus becomes very popular in computer vision. It defines “kernel” that aggregates nearby pixels information before sending it to a dense network.

A CNN kernel, also known as a filter, is a small matrix of weights that slides across the input image or feature map to perform a mathematical operation called convolution. An demonstration of CNN kernel is given in Fig. 4.1. Multiple kernels can be defined on the same layer to handle the same feature map, each kernel associated with an output channel. In practice, each kernel or channel is designed to detect a specific features in the feature map. For example, there might be a kernel detecting edges, while a second kernel detect color codes.

Notice that CNN differs quite largely from transformer in the problems they are expected to address. CNN is more for spatial data processing such as image processing, while transformer targets more on sequential data processing such as natural language processing and machine translation.

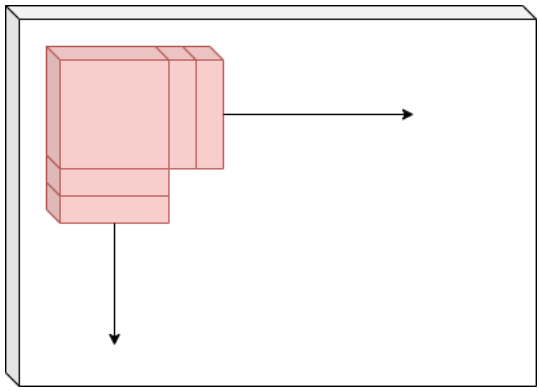

FIGURE 4.1
A demonstration of CNN kernel. The input is given by the white box (3D), and the kernel by the red box.

5

Recurrent Neural Network

CONTENTS

"nobreak

5.1 Brief Review of RNN

RNN is a connectionist model with the ability to selectively pass information across sequence steps [8]. It is good at handling sequence of data such as voice message, text contents, or a flow of images (videos). It is worth mentioning that the "sequence" does not necessarily mean a time sequence. Nevertheless, without losing generality, we will consider time sequence in the review for simplicity and convenience.

Denote inputs $x(1), x(2), ..., x(k), ...$ where $x(k)$ is a vector sampled at time instant k. The length of the sequence may be finite or infinite. In the case of finite sequence, its maximum sample index is denoted by T. For example, in the context of natural language processing, each input might be a word in a dictionary. For example, $x(1)$ = "Pandas", $x(2)$ = "are", $x(3)$ = "so", $x(4)$ = "cute", $x(5)$ = "!". The corresponding target output sequence is given by $y(1), y(2), ..., y(k), ...$, respectively.

RNN differs from the conventional dense ANN by introducing "recurrent edges", which allows the output of hidden layers at $k-1$ be used as additional inputs to the system at k. This means, at any time k, the input of the system includes both $x(k)$ and also selected $h(k-1)$, where $h(\cdot)$ is the outputs of hidden layers. We can think of the "weights" of a trained RNN the "long-term memory" that does not change with specific sequence of inputs, while the information passing through recurrent edges the "short-term memory" that links previous inputs with future inputs.

A demonstration is given in Fig. 5.1. Notice that each hidden layer is a multi-input-multi-output subsystem containing multiple nodes. Different from a conventional dense ANN, each hidden layer takes additional inputs from its corresponding hidden layer in the previous instant. It is also common to see

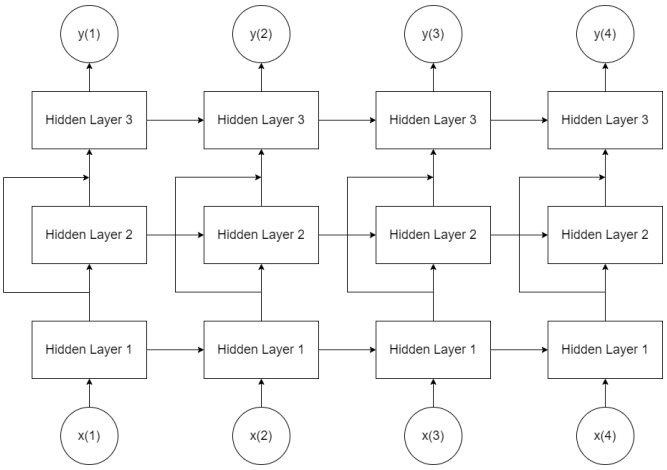

FIGURE 5.1
A example of RNN.

"bypass" (this is widely used in different ANN structures, not unique to RNN) for better performance of the system.

RNN has some limitations. One of the major problem is that it is difficult to train an RNN even for the basic standard feedforward networks. The optimization of RNN is NP-complete. It is especially difficult for RNN to learn long-range dependencies due to the vanishing and exploding gradients problem that could occur when backpropagating errors across many timestamps (long sequence) [8]. This is one of the main challenges why RNN has difficulties building on long-range dependencies. The vanishing and exploding gradient problems are caused by the structure of the system as well as the backpropagation-based training methods.

Different approaches have been proposed to prevent vanishing and exploding gradient problems. Famous ones among these approaches include strategical weight initialization, long short term memory (LSTM), gated recurrent units (GRUs), skip connections, and more. Many of these approaches try to reduce the effect of vanishing and exploding gradient problems by carefully design the ANN structures. For example, both LSTM and GRUs introduce "memory cells" with built-in "gates" that balance and control the flow of information from previous cells versus current inputs. The cells are used to replace the traditional perceptron nodes. These approaches have made the training of RNN a feasible problem. LSTM and GRUs are almost certainly used in modern RNNs.

Bidirectional RNN (BRNN) is proposed at about the same time with LSTM. It allows information to travel not only from previous hidden layers to future layers, but also from future hidden layers to previous layers. LSTM and BRNN can be used together to boost the RNN performance. Notice that

BRNN cannot run continuously as it requires fixed endpoints in both the future and the past. It is useful for prediction over a sequence of fixed length, such as part-of-speech tagging in natural language processing.

Another problem that people have found during the training of RNN is local optima. However, recent studies have shown that local optima is not as serious issue as we might thought when the network is large, since many critical points are actually saddle points rather than local minima.

Successful implementations of the above RNN structures include natural language translation such as [14] where a encoder-decoder structure is used, each is a LSTM. Another example is image captioning, where the AI tries to explain what is in an image using texts. A solution to this is to use CNN to encode the image, and use LSTM to decode to generate texts. Following similar ideas is hand-writing reorganization.

Part IV

Large Language Model

6

Transformer

CONTENTS

Large language model (LLM) is one of the many solutions to NLP, and NLP is a type of problem under sequential data processing. In the past few years we have seen revolutionary development in LLM, and hence it forms a dedicated part in the notebook. With the introduction of multimodal LLM, the capability of LLM expands beyond language processing towards image and even video processing.

Modern LLM is built on top of Transformer, an AI framework proposed in the landmark paper "Attention Is All You Need" [16]. Transformer is introduced in this chapter.

A brief review of RNN, the state-of-the-art framework priori to Transformer, is given, and its shortages are addressed. Transformer framework and its key components are introduced. Lastly, the development trend of Transformer is discussed.

6.1 RNN and Its Limitations

RNN has been introduced in earlier Chapter 5. A brief review is given in this section, focusing on its shortages in NLP.

6.1.1 A Brief Review of RNN

Comparing with a dense ANN, RNN is more efficient in building up correlation between consecutive inputs, and hence it becomes a powerful framework when comes to sequential data processing.

In a conventional dense ANN, the correlation cannot be captured efficiently. In contrast, a dense ANN would treat all the inputs "equally" in a symmetric manner. To enforce the ANN to "address" the correlation, it would require a lot more layers and nodes (which is often considered low-efficient), and requires more data points during the training. Other problems of conventional dense ANN include, for example, the lack of ability in handling data with arbitrary length.

CNN and RNN try to tackle the above problems by implementing a "preprocessing" stage, where the correlation of the spacial and sequential data is first abstracted using some mechanism, and the correlation information is sent as (additional) inputs to the followed dense ANN.

6.1.2 Limitations of RNN

"Attention" was not a new concept invented in [16]. It was proposed a few years earlier than the transformer, and had been integrated into several CNN and RNN models to boost their performances to capture long-range dependencies. In these researches, attention mechanism is more often a supporting role than the main character. It is quite surprising indeed that when getting rid of CNN and RNN and purely relying on attention mechanism, everything gets better: more efficient in training and better in performance.

Does this mean that the RNN, which we had been relying on so much, is in fact the drawback to NLP? The following is the answer of ChatGPT-4 of this question.

Q: By getting rid of RNN and using attention mechanism alone, natural language processing performance gets better. Is this correct? Does this mean that RNN is a drawback to natural language processing?

A: Yes, by getting rid of RNNs and using the attention mechanism alone, the Transformer model demonstrated improved performance in natural language processing tasks. However, it's important to note that RNNs are not inherently a drawback to NLP; they were successfully used for many tasks before the advent of Transformers.

RNNs, particularly LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) variants, were designed to handle sequences and capture dependencies in the data. They have been effective in many NLP tasks, such as language modeling, machine translation, and sentiment analysis. However, RNNs do have certain limitations, such as difficulty in

capturing long-range dependencies and sequential processing, which can lead to slower training and inference times.

The attention mechanism, as employed in the Transformer model, addresses some of these limitations. It enables parallel processing of input sequences, leading to faster training and inference, and effectively captures long-range dependencies without the vanishing gradient problem that can affect RNNs.

While Transformers have shown superior performance in many NLP tasks, RNNs still have their place in certain scenarios, especially when working with resource-constrained environments or smaller datasets. They can be more lightweight and computationally efficient compared to large-scale Transformer models.

In summary, RNNs are not a drawback to NLP per se, but their limitations have been mitigated by the introduction of the attention mechanism in Transformer models, which has led to improved performance in a wide range of NLP tasks.

(April 14, 2023, ChatGPT-4)

6.2 Transformer Framework

A summary of the concepts, components and technologies in [16] is given in this section.

6.2.1 Tokenizer

In most NLP solutions, human language components need to be mapped to numbers before inputting into a model. The outputs of the model must then be mapped back to human-readable components so that a human can interpret them. This is no exception for LLMs. This mapping process is known as **tokenization**.

There are multiple ways to perform tokenization. Different models, and models for different languages, may apply different tokenization schemes. In this section, English is used as an example. Commonly seen tokenization methods are discussed.

Character-Based Tokenization

The most intuitive approach is character-based tokenization. For example, we can use the ASCII standard as the mapping table, where each English character, including spaces and most commonly used special characters, is mapped to an integer.

The benefits of using character-based mapping include:

- Flexibility. Everything can be mapped without exception, including typos or invented words.
- Compact vocabulary. To cover all characters, we need only a few dozen entries in the mapping table.

The main drawback of character-based mapping is inefficiency. For instance, consider the character sequence "t", "h", and "e". When they appear together, they form the word "the" which carries a specific and frequent meaning. Ideally, such frequent patterns should be recognized as a whole. By storing this combination in the token mapping (rather than learning it from model parameters), we can improve modeling efficiency and reduce sequence length.

Word-Based Tokenization

To improve efficiency and reduce the number of input tokens, word-based tokenization was proposed. A dictionary containing around 30,000 to 50,000 words is often used in common applications. Any unrecognized word such as typo or extremely rare word is mapped to a special token that represents an "unknown" word.

The advantages of word-based mapping include:

- Efficiency. Fewer tokens are needed to represent a sentence compared to character-level mapping.
- Better contextual grounding. Each token usually corresponds to a semantically meaningful unit.

However, word-based tokenization sacrifices flexibility. Unseen or misspelled words provide no meaningful signal other than "unknown".

Additionally, word-based schemes often ignore the semantic structure within a word. For example, consider the word "handcraft". It is clearly composed of two recognizable words "hand" and "craft". Semantically, "handcraft" relates to both. Yet, in the word-based mapping, "hand", "craft" and"handcraft" are treated as entirely unrelated tokens. In such cases, it would be more effective to break the word into meaningful segments and tokenize those instead.

Subword-Based Tokenization

Modern LLMs typically use subword-based tokenization. This approach lies between character-based and word-based methods. It tries to achieve optimum flexibility and efficiency by smartly breaking a word into semantically meaningful chunks known as "subwords".

An example is given in Fig. 6.1, where a small piece of text is tokenized using a tokenizer. From the example, we can see that while basic and commonly used words such as "I", "am", "demonstrating", "sentence" remain in one

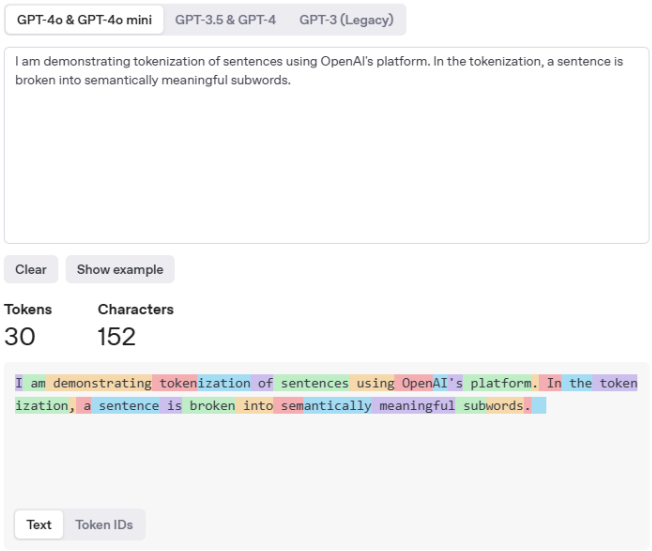

FIGURE 6.1
A demonstrative example where a piece of text is tokenized using GPT-4o's tokenizer.

token, the sophisticated words such as "OpenAI", "tokenization", "semantically" are broken into subwords. Notice that spaces and special characters such as period "." are also considered as part of tokens, as they carry meanings.

Each token is then assigned with a token ID. In the example in Fig. 6.1, they are

```
[40, 939, 73405, 6602, 2860, 328, 40536, 2360, 7788, 17527, 885, 6361,
    13, 730, 290, 6602, 2860, 11, 261, 21872, 382, 17162, 1511, 2743,
    175665, 33329, 1543, 10020, 13, 256]
```

Subword tokenization allows the model to:

- Represent rare or unknown words by combining familiar parts (e.g., prefixes, stems, suffixes).
- Reduce the number of out-of-vocabulary tokens. Even if a new word or typo is detected in the text, it is possible that it still contains meaningful subwords.
- Maintain a compact vocabulary while preserving semantic structure.

Just to put things into perspective, for a typical tokenizer on English writing, 1 token is roughly 4 characters, or 0.75 word, and 1000 tokens are roughly 750 words. Nowadays, math equations, scientific terms and codes are

also used as inputs and outputs to LLMs for domain knowledge related tasks. They usually consumes more tokens than English writings.

Different LLMs may use different tokenizers. Popular subword techniques include Byte Pair Encoding, WordPiece, and Unigram tokenization. The example in Fig. 6.1 demonstrates the tokenizer of OpenAI's GPT-4o. There are many other tokenizers used by variety of models. At this point, there is not a universally best tokenizer.

The maximum number of tokens an LLM can process, both as input and output, is limited by its **context window**. For example, consider a continuous dialogue with an LLM-based chatbot. Each time the user provides a new input, the entire chat history (including all of the user's previous inputs and the LLM's previous responses) is bundled together and sent as a new stateless input to the model. Based on this complete input sequence, the LLM then generates its latest response. This stateless nature means that the LLM does not "remember" anything beyond what is explicitly included in the current input.

As the conversation grows longer, more and more tokens are needed to represent the chat history. For example, GPT-4o model has a context window of $128K$ tokens, roughly one-tenth the length of the complete works of Shakespeare. Once this limit is reached, older parts of the conversation must be truncated or summarized.

In some scenarios, the input cannot be easily compressed or truncated and must be passed to the LLM as a whole. For example, when an LLM is used to troubleshoot a piece of code, the entire code block, possibly including multiple files, configurations, and logs, needs to be input at once to maintain context and correctness. In such cases, the size of the context window becomes a critical constraint. If the total input exceeds the model's maximum token limit, the model will either fail to process it entirely or may be forced to omit important parts of the input, potentially leading to incorrect or incomplete responses.

6.2.2 Encoder and Decoder

TBA

6.2.3 Attention Mechanism

TBA

6.2.4 Transformer-Based NLP

"nobreak

6.3 Transformer Development Trend

“nobreak

6.3.1 Transformer Variants

“nobreak

6.3.2 Trend

With the sizes and capabilities of the frontier models scaling up over the past years, the models are more and more appear to be intelligent and human-like.

A new job opportunity, “prompt engineer”, has emerged. The workscope of a prompt engineer is to design the appropriate prompt that instructs an LLM to complete certain tasks accurately and efficiently. People starts to take advantage of LLMs in their production.

Today, some most popular LLM applications include at least the following.

- Chatbot and customer service.
- Copilot
- Agentization

7

Large Language Model: Theory

CONTENTS

This chapter serves as an overview of the large language model from the theory and technical perspective. LLM is a successful practice of NLP and beyond. It can not only process human languages but also generate new contents based on human language written requirements. The introduction of multimodal LLM further allows LLM to understand and generate not just writings but also pictures and videos.

Nowadays, it is possible for a regular user to leverage on commercialized models to develop his own applications. Given a powerful machine, a user can even deploy LLMs locally in a few steps. Open-source LLMs are available so that the user does not need to train a model from scratch. The deployment of LLMs and the interaction with them are introduced in the next Chapter 8.

Modern LLMs are built on top of Transformer framework which has been introduced in earlier Chapter 6. It is worth emphasizing the relationship between the Transformer and LLM. Transformer is an ANN framework designed for processing any sequential data so long as it can be encoded, and appar-

ently NLP is an important use case of it. LLM is a practice of Transformer on NLP.

7.1 Introduction to LLM

This section reviews existing NLP methods and their limitations, and discusses the gaps that LLM bridges.

7.1.1 Existing NLP Models and Gaps

There have been variety of ways to model natural languages. For example, consider the following sentence:

"I am thirsty. Please give me a bottle of _____."

It is quite natural that a human would likely to put "water" or "tea" in the blank. This is obvious because human has a dictionary of words that he can choose in his mind, and he has been reinforced of learning "a bottle of water" expression in many occasions. In addition, it makes sense to a human that when someone is thirsty, he would look for water.

The challenge using AI on the above task is to build the "dictionary" in the machine and quantitatively analyze what word or phrase would make the most sense to be filled into the blank. Many models have been proposed to solve this problem, some of which have been evolving over time in the past decades since 1990th.

Statistical Language Models

In the early days when AI and ANN were not popular, **statistical language models** (SLM) have been the most popular tool to model natural languages. SLM assumes that a sentence is a Markov process, and the last word depends on the context created by the most recent n words. SLM with a fixed context length of n is also called an n-gram language model.

Conventionally, the paring information from n words to the next word is obtained from data corpus and stored in a table-like structure. When running the model, it looks up the table for the most probable next word based on the earlier n words. Smoothing technologies are used to handle zeros, i.e., when the record is not found in the table.

An obvious issue of SLM is that the computation and storage of the model increase exponentially with the size of n. This limits the context information that the model can use for prediction, hence setting a low performance ceiling. Not to mention that even with a large data corpus, zeros can still happen and the model performance is always an issue in such occasions.

Neural Language Models

With the introduction of ANN, in particular RNN, **Neural Language Models** (NLM) became popular. RNN-based NLM builds the word prediction function conditioned on the aggregated context features abstracted and passed recurrently from current and previous input sequence. More about RNN has been introduced in Chapter 5.

RNN is not a perfect one-stop solution either. The training of RNN can be difficult due to vanishing and exploding gradient problems. This limits the depth that RNN can go. When handling long sequence of words, the performance of RNN drops significantly because it is weak at building long-term dependencies.

Pre-trained Language Models and Large Language Models

As introduced in details in earlier chapters, attention mechanism has been proposed to tackle the long-term dependency problem of RNN. Transformer architecture, which relies purely on encoder, decoder and attention mechanism without using RNN is then proposed. It has been verified that transformer architecture is good at abstracting information from sequential data, in particular, natural languages. With a transformer, it becomes possible to build very deep neural networks and have it trained efficiently with big-size data corpus. The outcome is known as the **large language model** (LLM).

Language models based on different transformer-based architectures are often called **Pre-trained Language Models** (PLM). LLM can be taken as a subset of PLM. The main difference between an LLM and a regular PLM is the size of the model. The scaling of the model from hundreds of millions of parameters for PLMs to tens or hundreds of billions of parameters for LLMs introduces emergent abilities such as in-context learning to the model, significantly enhancing its capability and intelligence. As of this writing, it is not very clear how the these abilities suddenly emerge with the size of the model.

7.1.2 LLM Features and Capabilities

When the size of the model becomes large, usually to the order of at least a few billions parameters, they suddenly gain emergent abilities. Details are discussed as follows.

In-context Learning

In-Context Learning (ICL) allows the behavior of the model to be manipulated via not training or fine-tuning of the parameters, but via instructions and demonstrations given as part of the input.

ICL plays an important part in LLM implementation, as it is the basis of prompt engineering. When the LLM is large, it is possible to use prompt engineering instead of fine-tuning to complete a task following user defined

instructions. This reduces the training cost and makes the implementation more flexible.

Instruction Following

Supervised learning is commonly used in training and fine-tuning a model. In the training set, we need to provide the model "examples". An example includes the input, some good responses, and some bad responses.

When comes to LLM, it is possible to fine-tune a model for a specific task without using the aforementioned examples. Instead, just give it step-by-step instructions. LLM is able to perform well with these tasks described only by instructions. This is known as instruction tuning.

It is worth mentioning that instruction following is also possible in ICL. Give the LLM instructions in prompt engineering without examples, and the LLM is likely to be able to finish the tasks.

Step-By-Step Reasoning

When a model is asked to complete a complicated task that involves multiple steps, it may fail to accomplish the task. With a well fine-tuned LLM, the model might be able to break the task into multiple sub-tasks via chain-of-thought (CoT) prompting strategy, and solve them step-by-step till the final result is obtained.

It has been observed that an LLM with at least $100B$ parameters is likely to have good step-by-step reasoning abilities.

The performance of an LLM, usually referring to its capability in accurately and correctly complete a task, is affect by many factors such as the model architecture, model size, training data set size and quality, etc. Though it is clear that with the scaling up of the system the performance is usually improved, there is no analytical expression that gives full insights about how these factors affect the performance quantitatively.

A **scaling law** tries to quantitatively describe the performance of an LLM as a function of model size and other factors. Many scaling laws have been proposed, many of which obtained from empirical experiments and they may work only within a given range of model size.

Just as an example, OpenAI proposed KM scaling law in 2020 that describes LLM cross entropy loss as a function of model size, training data set size and training computation as follows.

$$\begin{aligned} L(N) &= \left(\frac{N_c}{N}\right)^{\alpha_N} \\ L(D) &= \left(\frac{D_c}{D}\right)^{\alpha_D} \\ L(C) &= \left(\frac{C_c}{C}\right)^{\alpha_C} \end{aligned}$$

where N, D and C denote the model size, dataset size and training compu-

tation, respectively. The rests are constants whose value can be obtained via calibration.

This scaling law works for models with $22M$ to $23B$ parameters. It is assumed that the analysis of a factor can be done independently without other parameters being a bottleneck.

7.1.3 LLM Performing Benchmarks

“nobreak

7.1.4 LLM Development Timeline

A brief timeline of the development of LLMs since the publication of [16] is given below.

2017

“Attention Is All You Need” [16] introduced a new type of NLP model called the “Transformer”. Unlike traditional RNN-based models, the Transformer removes all recurrent components and adopts an encoder-decoder architecture that relies entirely on self-attention mechanisms to model dependencies within and across sequences.

2018-2020

OpenAI released the GPT-1, GPT-2, and GPT-3 models in consecutive years from 2018 to 2020, showcasing the scalability of Transformer-based architectures for autoregressive language modeling.

2022

Reinforcement Learning from Human Feedback (RLHF) became a widely adopted training strategy. OpenAI released ChatGPT, a chatbot fine-tuned from the GPT-3.5 model using RLHF. Its human-like conversational abilities captured public attention and spurred a surge of interest and development in LLM-based applications.

2023-2024

GPT-4, GPT-4o are published in 2023 and 2024 respectively, along with many other frontier models are published.

7.1.5 LLM-Relevant Milestone Technologies

This section looks back into the progression tree of LLM, and lists down milestone techniques that make LLM what it is today. It is the breakthrough in these areas that revolutionizes LLM development.

Big Data

The performance of LLM relies on both the modal size and the training data size. It is the advent in internet, Web 3.0, Industry 4.0, IoT and cloud computing/storage that makes collection and aggregation of big data possible.

Almost every large-size enterprise, both IT companies and conventional industrial companies, have their internal databases. The database can be used to train domain-knowledge LLM. Nowadays, there are many open data sources of community LLMs. Such examples include BookCorpus, CommonCrawl, Reddit posts (with high upvotes), Wikipedia, and many more. These open-source datasets make training LLM for community projects possible.

Large Model and Efficient Training

The invention of CPU-based neural networks and transformer architecture making creating and training large scale LLM possible. Both closed and open-source LLMs have been proposed, including GPT series by OpenAI and LLaMA family by Meta AI and the community.

Manly libraries have been released to the public to help building, training and fine-tuning LLMs, such as `transformers`, a Python library for building transformer models. Many such libraries are tying up with PyTorch and TensorFlow to provide LLM related functions.

More about these models and libraries are introduced in later sections.

Fine-Tuning

Technologies such as LoRA has made fine-tuning easier than before.

Prompt Engineering

There have been a lot of practices on how to make LLM flexible and more efficient in solving particular tasks via prompt engineering.

LLM on Edge Devices

Many efforts have been put into edge-device based LLMs. The target is to develop LLM that consumes less memory, storage and computation while not sacrificing a lot of performance.

API and Interface

Multi-modal LLM has enabled different types of inputs to the LLM, not limited to natural language but also sequential signals and even pictures.

Many tools and software have developed APIs for LLM. These tools enhance computation and online information retrieval capabilities of LLM.

7.2 Existing LLMs Features

As of this writing, there are countless LLMs available on the market or within the open-source community, and the list continues to evolve rapidly.

This section introduces the shared properties and general characteristics of existing LLMs, without delving into detailed performance metrics or task-specific comparisons. The goal here is not to promote one model over another, but rather to provide a broad overview of what defines an LLM in practice.

A more detailed performance review of selected models is provided in Section 8.4.1.

Many LLMs are capable of engaging in human-like conversations and answering questions in depth. Some can even search the internet for the up-to-date relevant information to support their responses. Most well-known LLMs can also follow user instructions to perform specific tasks, such as completing a piece of code, provided the prompts are sufficiently clear and aligned with the model's training data.

That said, many LLMs (unless specifically trained on domain-specific corpora) do not perform as well as experienced professionals in specialized fields. It is often claimed that top-performing LLMs can achieve a level of performance comparable to fresh PhD graduates in certain tasks, although a noticeable gap remains when compared to experienced researchers.

LLMs may generate inaccurate or misleading information while expressing it in a fluent and confident manner. This behavior stems from the their lack of self-awareness and inability to judge the factual correctness of its outputs.

LLMs can also struggle with detailed quantitative tasks. This limitation arises from how they process and generate language: input is tokenized into subword or word-like units, and outputs are generated based on statistical patterns rather than precise symbolic reasoning. For example, an LLM might fail to correctly answer a seemingly simple question like "How many letters 'a' are there in this sentence?", because its training does not emphasize accurate character-level counting.

Multimodal LLMs can generate images or videos based on textual descriptions. However, they often struggle to accurately render fine-grained visual details. For example, if prompted to generate an image of a car with a license plate that reads "3.14159265358979", the resulting image may omit or distort some of the digits. This limitation stems from the fact that image generation is driven by probabilistic pattern synthesis, not precise symbolic encoding or controlled rendering of visual elements. An example is given in Fig. 7.1.

It is true that each LLM is trained and fine-tuned with different data corpus and may behave differently on different types of tasks. Consider LLMs with the same model structure and training strategy. In such case, models with larger training data sets and more parameters often outperform those with smaller training data sets and less parameters. From application perspective,

FIGURE 7.1
A figure from a GPT when it is asked to generate a car with a license plate that reads "3.14159265358979".

the trend is that the frontier models' performances are converging for regular tasks, and the costs are going to be the main differentiator.

OpenAI's GPT models and Meta's LLaMA models are the first models that gain popularity. A brief review is given in the remaining of the section.

7.2.1 OpenAI Family

OpenAI started investigating language models before the proposition of transformer. In its early days, RNN was explored as the most promising model for natural language. In 2017 when the transformer model was proposed, OpenAI quickly adapted their language model to this new architecture, and as a result generative pre-training (GPT) series has been proposed.

GPT-1

GPT-1, OpenAI's first transformer based PLM was proposed in 2018. GPT-1 has $117M$ parameters in the model and it adopts a decoder-only architecture, which is different from the original transformer proposal which has a encoder-decoder architecture. GPT-1 was trained via a two-stage procedure, the first stage unsupervised pre-training and the second stage supervised fine-tuning. This two-stage training pipeline, or something of the similar kind, has been adopted by many LLMs coming after.

GPT-2

GPT-2 is an improvement of GPT-1. It uses much larger number of parameters of $1.5B$ in the model, and it was trained on a much larger dataset WebText. With larger model and training data size, GPT-2 is targeted to be a multi-task solver. The model can be formulated by the following probabilistic form

$$\text{Pre-trained LLM} \quad \equiv \quad P\left(\text{output}|\text{input}, \text{task}\right)$$

In the above formulation, each NLP task can be considered as the word prediction problem based on a subset of the word next, and can be trained during the unsupervised learning stage. Unsupervised pre-training has since then become the most important stage for the LLM to gain knowledge for general tasks.

GPT-3, GPT-3.5 and Chat-GPT

It is clear now that GPT-2 has $1.5B$ parameters which is too few for an LLM to gain emergent abilities. It is GPT-3 with $175B$ parameters trained on $300B$ tokens that made a capability leap and bring LLM to everyone's attention.

It is GPT-3 that for the first time introduces emergent abilities such as ICL. GPT-3 not only accomplishes commonly seen tasks to test LLM capabilities with flying color, but also demonstrates features not shown by other models before, such as reasoning and domain adaption.

OpenAI has developed many task-oriented models that use GPT-3 as the base model. For example, for coding, Codex was introduced. Codex is basically GPT-3 fine-tuned using code database such as GitHub. Comparing with GPT-3, Codex is able to reason and solve complex mathematical problems, and realize them in codes. RL had already been used to fine-tune and improve performance for GPT-2. The same has been applied on GPT-3. Furthermore, reinforcement learning with human feedback (RLHF) is introduced for GPT-3 that allows the model to continue learning from human demonstrations.

With the above enhancements, i.e. code-based fine-tuning, RL and RLHF, GPT-3.5 has been developed. GPT-3.5 is an enhanced version of GPT-3 and it is obtained from GPT-3 via transfer learning. GPT-3.5 is also scaled up in size, with a parameter number of $335B$.

Chat-GPT was developed on top of GPT-3.5 (and later another version was released developed on top of with GPT-4). It is GPT-3.5 fine-tuned specially optimized for dialogue. It demonstrates how LLM can be used in everyone's daily life as an assistant, consultant and a copilot. Chat-GPT demonstrated good performance in communicating of humans than other GPT models. This is because Chat-GPT is trained on human-generated conversations of both human and AI roles. Later on, plug-ins and APIs are enabled on Chat-GPT, to allow it to communicate with other software.

GPT-4

GPT-4 is a multimodal LLM, meaning that it can not only take in natural languages, but also other signals such as images. Comparing with GPT-3.5, GPT-4 demonstrates further enhanced capabilities. It is claimed that GPT-4 is safer to use and is less likely to generate toxic and misleading information.

Just as a quick demonstration of its multimodal feature, pass the following Fig. 7.2 (notice that this figure itself is generated by an AI model) to GPT-4 with instruction "What is in this picture?". GPT-4 is able to response the following:

"The image showcases a majestic cat standing in a fantastical landscape. The cat emits a luminous, ethereal glow, and its fur and tail appear to be infused with swirling, fiery or cosmic energies in brilliant hues of red and gold. These radiant swirls surround the cat, adding to its supernatural aura. Behind the cat, there's a dramatic night sky with clouds illuminated by a hidden light source, possibly lightning or a celestial event. In the foreground, the terrain appears to be a combination of grassy knolls and fiery, ember-lit ground, further emphasizing the magical and otherworldly nature of the scene. The entire setting evokes a sense of wonder and fantasy."

FIGURE 7.2
A cat with superpower. This picture is generated by DALL·E 3.

Other OpenAI Models

DALL·E is the model OpenAI uses to generate images. As of this writing, its latest version, DALL·E 3, has been integrated with the latest ChatGPT. Figure 7.2 gives an example where DALL·E 3 is used to generate a image of a superpower cat.

OpenAI Codex, as introduced earlier, is the GPT model optimized for code generation. As of this writing, Codex has been deprecated because its capability has been integrated into ChatGPT. It is possible to ask ChatGPT to generate a piece of code following user's instruction and description.

OpenAI Codex is the engine that powers GitHub copilot. In this sense, OpenAI Codex remains a live in a different format: not as a standalone software or model, but more as an API.

7.2.2 LLaMA Family

Large Language Model Meta AI (LLaMA) is the LLM model developed by Meta AI. Different from most of the AI models (GPT-3 and onward) developed by OpenAI, LLaMA is open-source hence has a wide availability. Many efforts in the community have made modifications and improvements to LLaMA, making a big family of models with different characteristics.

As of this writing, a family tree of LLaMA is shown in Figure 7.3. The picture is from [17]. The source of the picture is given in the GitHub repository of the paper. Only a small portion of models in the family tree is briefly

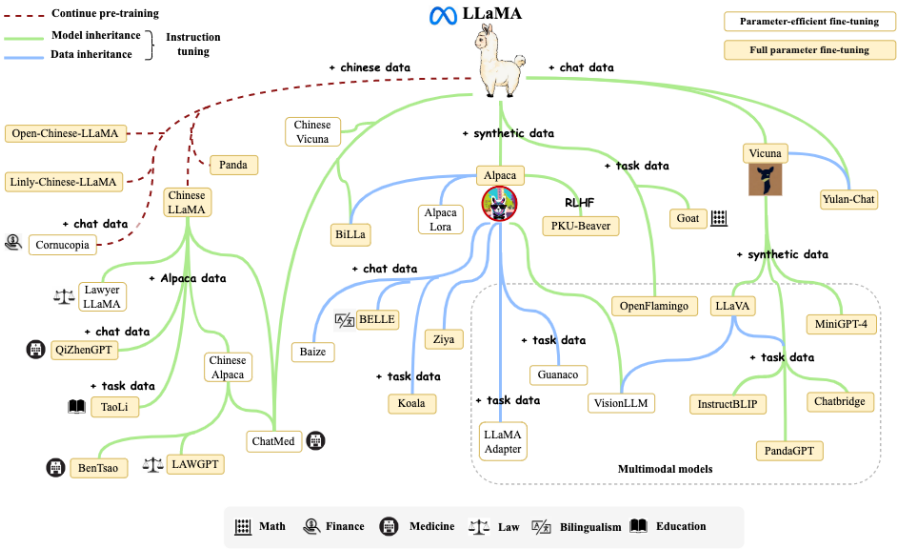

FIGURE 7.3
LLaMA family tree.

introduced here.

LLaMA

LLaMA, comparing with GPT-3 which was used as a benchmark, is smaller in model size (maximum $65B$ parameters VS $175B$ parameters in GPT-3) but larger and better in training data size and quality (maximum $1.4T$ tokens VS $300B$ tokens in GPT-3). As a result, LLaMA is able to achieve generally better performance than GPT-3 with less implementation cost due to the small size. LLaMA demonstrates that training data is equality important as model size. It is possible to reach the same level of performance with a small ($< 100B$ parameters) but well-trained model.

LLaMA's first release includes 4 models of different model and training sizes. Details are summarized in Table 7.1. The training dataset is purely open-

TABLE 7.1
LLaMA models.

Name	Model Size	Training Dataset Size
LLaMA 7B	$6.7B$	$1T$
LLaMA 13B	$13.0B$	$1T$
LLaMA 33B	$32.5B$	$1.4T$
LLaMA 65B	$65.2B$	$1.4T$

Model size is given in number of parameters in billion. Training data size is given in number of tokens.

source, including Common Crawl, C4 Dataset, GitHub, Wikipedia, public domain books, arXiv and Stack Exchange.

Technical wise, LLaMA has some innovations on top of the original transformer proposition [16] n the normalization method, activation function, and embedding methods. More details will be introduced in later sections.

Many open-source tools and packages have been developed to fine-tune LLaMA and its variations. More about fine-tuning, such as LoRA[7] and QLoRA[5], are introduced in more details in later sections. Notice that fine-tuning or re-training of a model often cost a lot of computational power and vector memory.

Stanford Alpaca

Stanford Alpaca 7B is Standford's practice in fine-tuning LLaMA 7B. The running cost of this fine-tuning is impressively small (hundreds of USD), and the resulted model has a performance comparable with GPT-3.5 which has $335B$ parameters. It is impressive to see that a small $7B$ model can compete with a large $335B$ model by careful training and fine-tuning. This might be partially because Alpaca 7B is fine-tuned from LLaMA 7B using knowledge distillation from GPT-3.5 (also known as text-davinci-003).

The details of training Stanford Alpaca can be found in [15]. A brief highlight is given below. The fine-tuning pipeline of Alpaca is shown in Fig. 7.4. It is a 2-stage process as follows.

1. Generating instruction-following examples using GPT-3.5 (text-davinci-003).

2. Fine-tune LLaMA 7B using the examples.

Examples of the generated instruction and response pairs are given below. The full list is available from the GitHub repository.

```
{
        "instruction": "What are the three primary colors?",
        "input": "",
        "output": "The three primary colors are red, blue, and yellow."
```

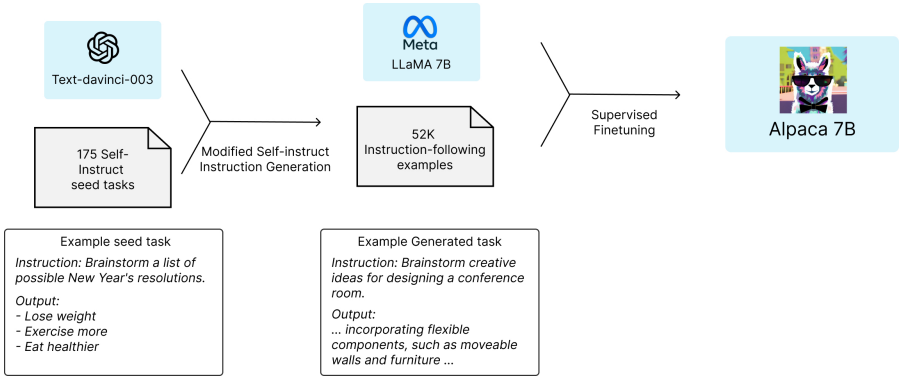

FIGURE 7.4
Alpaca fine-tuning pipeline.

```
},
{
        "instruction": "Give three tips for staying healthy.",
        "input": "",
        "output": "1.Eat a balanced diet and make sure to include plenty
             of fruits and vegetables. \n2. Exercise regularly to keep
            your body active and strong. \n3. Get enough sleep and
            maintain a consistent sleep schedule."
},
{
        "instruction": "Extract the facts from the paragraph.",
        "input": "Online education continues to become more popular for
            schools and students alike. Its advantages are generally
            lower costs, less commitment and the ability to study at a
            time, place and pace that suits the student.",
        "output": "- Online education is becoming increasingly popular.\
            n- It has several advantages such as lower costs, less
            commitment and the ability to study at one\u2019s own time
            and pace."
},
{
        "instruction": "Arrange the given numbers in ascending order.",
        "input": "2, 4, 0, 8, 3",
        "output": "0, 2, 3, 4, 8"
}
```

The fine-tuning code of Alpaca is from Hugging Face. Python packages such as `numpy`, `torch`, `transformers`, etc., are required. Check `requirements.txt` and `train.py` in the repository for more details. Call `train.py` using something like the following

```
torchrun --nproc_per_node=4 --master_port=<your_random_port> train.py \
```

```
    --model_name_or_path <
        your_path_to_hf_converted_llama_ckpt_and_tokenizer> \
    --data_path ./alpaca_data.json \
    --bf16 True \
    --output_dir <your_output_dir> \
    --num_train_epochs 3 \
    --per_device_train_batch_size 4 \
    --per_device_eval_batch_size 4 \
    --gradient_accumulation_steps 8 \
    --evaluation_strategy "no" \
    --save_strategy "steps" \
    --save_steps 2000 \
    --save_total_limit 1 \
    --learning_rate 2e-5 \
    --weight_decay 0. \
    --warmup_ratio 0.03 \
    --lr_scheduler_type "cosine" \
    --logging_steps 1 \
    --fsdp "full_shard auto_wrap" \
    --fsdp_transformer_layer_cls_to_wrap 'LlamaDecoderLayer' \
    --tf32 True
```

where `./alpaca_data.json` is the JSON file that contains all the instruction and response pairs.

Notice that later on, Stanford Alpaca 13B is also developed based on LLaMA 13B.

Vicuna

Vicuna 13B is an open-source chatbot fine-tuned on top of LLaMA using $70K$ user-shared conversations collected from ShareGPT (a platform where users share their ChatGPT conversation histories). Its performance is comparable with GPT-3.5 and Bard, and it is slightly better than LLaMA 13B and Alpaca 13B in general.

See [4] for more details.

LLaVA and MiniGPT-4

Large Language and Vision Assistant (LLaVA) is a multimodal LLM that combines a vision encoder and Vicuna for general purpose visual and language understanding. More details are given in [10] and [9]. Similar with GPT-4, its multimodal encoders allows it take both natural language and image as inputs. An example is given below.

Pass the same Fig. 7.2 to LLaVA with the same instruction "What is in this picture?". LLaVA is able to give the following response:

"The image features a cat standing on a hill, with a fiery background and a cloudy sky. The cat appears to be looking at the viewer, and there is a sense of motion in the scene."

Similar with LLaVA, MiniGPT-4 is a also multimodal LLM that enhances vision interpretation. More details are given in [18]. It uses different techniques and realizations from LLaVA. Pass the same Fig. 7.2 to MiniGPT-4 with the same instruction "What is in this picture?". MiniGPT-4 is able to give the following response:

"The image shows a white cat with flames coming out of its fur, standing on a hill at sunset. The cat's eyes are closed, and it appears to be in a meditative state. The background is a dark sky with clouds and a full moon. The overall mood of the image is peaceful and mystical."

Notice that both LLaVA and MiniGPT-4 model can be downloaded from their GitHub and HuggingFace repositories. As of this writing, live demos are also available for these models.

Multimodal LLM and "Any-to-Any" LLM are not the main forecast of this chapter. More details will be introduced in later chapters.

LLaMA 2

LLaMA 2 is the next generation open source LLM following the original LLaMA. The model is free for research and commercial use. LLaMA 2 70B has $70B$ parameters and it is pre-trained with $2T$ tokens of training data.

7.2.3 Other LLMs

"nobreak

7.3 LLM Development

This section introduces the basic steps in the LLM development workflow, should the user want to develop an LLM from scratch.

7.3.1 Architecture Design

Encoder and Decoder

Normalization Method

Activation Function

Position Embedding Method

Attention Mechanism

7.3.2 Training Corpus Preparation

Data corpus is available online for LLM training. Many libraries such as PyTorch compatible Python packages have been developed to automate the training procedures and to convenient the users.

The existing resources and solutions are briefly introduced in this section.

7.3.3 Pre-training

“nobreak

7.3.4 Fine-Tuning

“nobreak

7.3.5 Model Evaluation

“nobreak

7.4 Multimodal LLM

TBA

8

Large Language Model: Practice

CONTENTS

This chapter studies the selection, deployment and use of LLMs in production environment for different tasks.

8.1 Python Environment Setup

LLM provides APIs to interact with many different programming languages or platforms such as Python, JavaScript, and many more. In the scope of this chapter, we focus on Python-based LLM interaction, as Python is one of the most widely used language for ANN and LLM studies and applications development.

It is recommended to collect all the necessary libraries in a file, such as `environment.yml`, and use

```
conda env create -f environment.yml --name <environment name>
```

to create an environment and install the packages all together. Anaconda

shall figure out the dependencies and compatibility of the packages, and have everything installed correctly.

An example of such YAML file is given below. This example is taken from [6].

```
channels:
  - conda-forge
  - defaults
dependencies:
  - python=3.11
  - pip
  - python-dotenv
  - requests
  - numpy
  - pandas
  - scipy
  - pytorch
  - jupyterlab
  - ipywidgets
  - matplotlib
  - scikit-learn
  - chromadb
  - jupyter-dash
  - sentencepiece
  - pyarrow
  - faiss-cpu
  - pip:
    - beautifulsoup4
    - plotly
    - bitsandbytes
    - transformers
    - sentence-transformers
    - datasets
    - accelerate
    - openai
    - anthropic
    - google-generativeai
    - gradio
    - gensim
    - modal
    - ollama
    - psutil
    - setuptools
    - speedtest-cli
    - langchain
    - langchain-core
    - langchain-text-splitters
    - langchain-openai
    - langchain-chroma
```

```
  - langchain-community
  - faiss-cpu
  - feedparser
  - twilio
  - pydub
```

Among the libraries shown above, some are commonly used across all Python and machine learning projects such as `numpy`, `pandas` and `scikit-learn`, while others are LLM specific packages such as `transformer`, `ollama` and `langchain`.

Environment Solutions of Anaconda Versus PyPA

Anaconda is a Python distribution developed and maintained by Anaconda Inc. It provides `conda`, a useful tool, to manage packages and environments. Anaconda provides reliable services to professional data scientists, cooperations as well as the free-of-charge services to the community. Python also has its native packages and environments management tools known as `pip` and `venv` developed and maintained by Python Packaging Authority (PyPA).

Both `conda` and `pip` allow users to create an environment and install packages from a file. The commands are

```
conda env create -f <filename> --name <environment name>
```

and

```
python -m venv <environment name>
pip install -r <filename>
```

respectively.

It is of the user's choice whether to use `conda` (from Anaconda, or its light version, Miniconda) or `pip`/`venv` to manage the packages. Under the scope of this notebook, both of them should fulfill the needs. As of this writing, the trend seems to be that data scientists, researchers and lectures would more often use `conda`, while software engineers, `pip`/`venv`.

8.2 LLM Local Deployment

Ollama and Langflow are introduced. Ollama allows fast deployment of an open-source LLM on a local computer, while Langflow is a low-code tool that allows a developer to easily integrate and test an LLM in a data pipeline.

8.2.1 Quick LLM Deployment with Ollama

Nowadays it is convenient to quickly deploy an open-source LLM on a PC with necessary hardware. One of the simplest way of doing that is to leverage on **Ollama**, an open-source tool that allows the deployment of commonly seen open-source LLMs such as LLaMA, DeepSeek, etc., on a local machine. Notice that the usable LLMs are limited by the hardware power of the machine. During the installation of Ollama, it automatically detects and configures the GPU of the machine accordingly.

Once Ollama is installed, use

```
ollama run <model name>
```

in the terminal to download and start a model. A screenshot is given in Fig. 8.1 as an example that runs LLaMA 3.2 with Ollama on a PC using the above command.

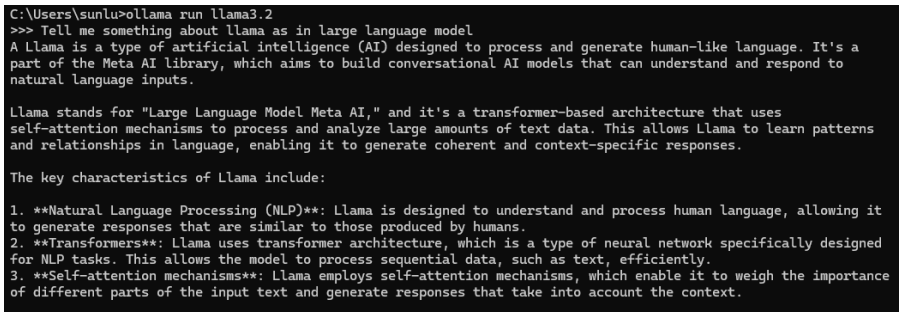

FIGURE 8.1
An example of running Ollama with LLaMA 3.2.

The deployed LLM does not come with any fancy graphical interface, but instead with the basic CLI. The user has the freedom to further deploy interfaces for applications on top of the basic interface.

A list of Ollama supported LLMs are given in [12]. As of this writing, famous ones include `deepseek-r1` (1.5B to 671B), `llama3.3` (70B), `llama3.2` (1B, 3B), `gemma3` (1B to 27B), and a lot more.

8.2.2 Interaction with Locally Deployed LLM

In the previous section, we learned that Ollama can be used to conveniently deploy an open-source LLM on the local machine, and it also provides a CLI where the user can chat with the LLM. It is possible to connect a Python program to that LLM using the interface that Ollama provides.

Make sure that the model is running in the backend. Start the model and check the model status using `ollma serve` and `ollama ps` respectively.

Python program can connect to the model either via HTTP request to `http://localhost:11434/api/chat` as follows.

```
import requests

OLLAMA_API = "http://localhost:11434/api/chat"
HEADERS = {"Content-Type": "application/json"}
MODEL = "llama3.2"

messages = [
    {"role": "user", "content": "<content of the message>"}
]
payload = {
        "model": MODEL,
        "messages": messages,
        "stream": False
    }
response = requests.post(OLLAMA_API, json=payload, headers=HEADERS)
print(response.json()['message']['content'])
```

Alternatively, use `ollama` package as follows.

```
import ollama

MODEL = "llama3.2"
messages = [
    {"role": "user", "content": "<content of the message>"}
]
response = ollama.chat(model=MODEL, messages=messages)
print(response['message']['content'])
```

OpenAI's Python package also provides tools to connect to local LLMs like what has been deployed via Ollama.

```
from openai import OpenAI

MODEL = "llama3.2"
messages = [
    {"role": "user", "content": "<content of the message>"}
]
ollama_via_openai = OpenAI(base_url='http://localhost:11434/v1',
    api_key='ollama')
response = ollama_via_openai.chat.completions.create(
    model=MODEL,
    messages=messages
)
print(response.choices[0].message.content)
```

8.2.3 Naive Deployment

Ollama uses C++ to compile an LLM and deploy it on the local machine. The compiled model is easy to use and runs efficiently, but it is generally difficult to modify—such as for fine-tuning or architectural changes.

For users seeking more flexibility, it is possible to download the raw parameters of open-source LLMs and run them independently. Many such models are available from platforms like Hugging Face, allowing users to experiment with customization, fine-tuning, or integration into custom pipelines. But of course, there will be additional steps for the users to execute the models, and they are introduced in this section.

8.3 LLM Cloud Deployment

Many companies provide the user with cloud-based LLMs. These LLMs are often more powerful and robust than locally deployed ones.

8.3.1 Browser-Based Chatbot

Companies such as OpenAI, Google, Microsoft and Deepseek provide web-based chatbot interface where a user can directly chat with the model. Many of these companies also provide APIs that allow user program to connect to models running on the cloud.

8.3.2 Cloud-Service-Provider-Managed LLM Deployment

Nowadays many cloud providers including AWS, Microsoft Azure, Google Cloud Service, etc., that allow the user to deploy LLM models on their servers. For example, AWS provides Amazon Bedrock which is its frontier model that can be easily deployed on AWS. Similar applies to other major cloud service providers.

8.3.3 API-Based LLM

Many LLM service providers, such as OpenAI, allows the user to interact with their cloud-based LLMs using APIs through command lines. Notice that the API-based LLM has a completely different business model compared with the chatbot, and should be treated as different services.

If the user has decided to use a commercialized LLM model via its API key, he needs to register an account with the LLM provider, such as OpenAI, and create a new API key, and added it to the project as an environmental variable.

TABLE 8.1
OpenAI's API Calls Pricing as of this writing, in USD per $1M$ tokens.

Model	Input Cost	Output Cost
`gpt-5`	1.25	10.00
`gpt-5-mini`	0.25	2.00
`gpt-5-nano`	0.05	0.40
`gpt-4.1`	2.00	8.00
`gpt-4.1-mini`	0.40	1.60
`gpt-4.1-nano`	0.10	0.40
`gpt-4o`	2.50	10.00
`gpt-4o-mini`	0.15	0.60
`o1`	15.00	60.00
`o1-pro`	150.00	600.00

Notice that the use of the API key often introduces costs to be paid to the LLM provider. The cost depends on the model and the number of input and output tokens in a call. To give a perspective, as of this writing the cost of OpenAI's API calls are given in Table 8.1 as of this writing. Powerful models such as `o1` are significantly more expansive than less powerful ones such as `gpt-4o-mini`. This is different from chatbot service where the user pays a fixed monthly subscription fee and get almost unlimited access to most of the models of his choice.

The details about the registration of the account and the creation of the API key are not included in this notebook.

The Python library `openai` provides a quick way to interact with its models, given that the user has a valid API key. A basic realization looks like the following. An example is given later. The "completions" API is used, which asks the LLM to complete a conversation.

```
from openai import OpenAI

api_key = '<sk-proj-...>' # put api key here

openai = OpenAI(api_key=api_key)
messages = [
{"role": "system", "content": "<system prompt>"},
{"role": "user", "content": "<user prompt>"}
]
response = openai.chat.completions.create(model="<model>", messages=
    messages)
```

The format of `message`, originally defined by OpenAI, has now become a convention for LLM API calls. In the message, `<system prompt>` tells LLM the basic setup, such as what role the LLM shall play and what it should do, whereas `user prompt` gives the user specific data that the LLM needs to process.

An example connecting to `gpt-4o-mini` is given below.

```
import os
from openai import OpenAI
from dotenv import load_dotenv

load_dotenv(override=True) # api key is loaded as an environment
    variable

openai = OpenAI()
message = "Hello, GPT! I am connecting you via your API. Let's see how
    it works."
response = openai.chat.completions.create(model="gpt-4o-mini", messages
    =[{"role":"user", "content":message}])
print(response.choices[0].message.content)
```

For the above code to work, make sure that `.env` file exists, and environment variable `OPENAI_API_KEY` has been setup inside.

A response similar to the following can be obtained.

```
Hello! It's great to hear that you're connecting via the API. How can I
     assist you today?
```

A typical `messages` that is passed to OpenAI often looks like the following

```
messages = [
    {"role": "system", "content": "<system prompt>"},
    {"role": "user", "content": "<user prompt>"},
    {"role": "assistant", "content": "<LLM historical response>"},
    {"role": "user", "content": "<user prompt>"},
    {"role": "assistant", "content": "<LLM historical response>"},
    {"role": "user", "content": "<user prompt>"}
]
```

where `<system prompt>` tells LLM the basic setup, such as what role the LLM shall play and what it should do, whereas `<user prompt>` gives the user specific data that the LLM needs to process. LLM API by itself is stateless. The historical conversations need to be passed to it to retain a long conversation. In that case, `assistant` role is used to store LLM's historical responses, so that it knows which part comes from the user and which part from the historical itself.

OpenAI and other companies such as Google, Deepseek, etc., provide variety of APIs for different functions. More details are introduced in later sections.

8.3.4 Commonly Seen APIs

OpenAI GPT, Anthropic Claude and Google Gemini are used as examples. Their supported commonly used APIs are introduced below, with examples to demonstrate the syntax.

8.4 Model Fine-Tuning

Many LLM service providers such as OpenAI allows a user to fine-tune a clone of their base models using his own data. The user pays the computation resources consumed during the fine tuning, and the fine-tuned model needs to reside at the servers of the service provider. Alternatively, a user can also download open-source models from the community, such as Hugging Face, and fine-tune a model locally.

This section introduces the practices in model fine-tuning. To start, a brief review of some of the frontier models is given as of this writing.

8.4.1 A Review of Frontier Models

Models to be reviewed in this section are summarized in Table 8.2. It is also indicated in the table whether the model is propriety or open-source, whether the corresponding company provides a browser-based chatbot where the user can quickly test its performance, and whether the corresponding company provides API call services where the user can link an application to the LLM running at the company's servers.

TABLE 8.2
Frontier LLMs and their accessibility features.

Model (Company)	Proprietary / Open-Source[1]	Chatbot[2]	API[3]
GPT (OpenAI)	P	Y	Y
Claude (Anthropic)	P	Y	Y
Gemini (Google)	P	Y	Y
LLaMA (Meta)	O	N	N
Qwen (Alibaba)	O	Y	Y
DeepSeek (DeepSeek)	O	Y	Y

Note: A model may have multiple versions or variants with different accessibility features. This table reflects the latest version of each model line as of this writing.
[1] Proprietary / Open-Source: Whether the model weights are publicly released and can be deployed locally.
[2] Chatbot: Whether an official website hosted by the developer allows users to interact with the model. Costs may apply, even for open-source models.
[3] API: Whether the developer provides official API access. Costs may apply, even for open-source models.

OpenAI's GPT model is probably the earliest and most well known LLM among the general public. It was also one of the first models to support multi-modal inputs. Today, many third-party applications developed by independent developers default to using GPT when integrating an LLM. The latest version, GPT-4o, is highly capable and performs well across a wide range of general tasks.

Claude is well regarded among data scientists and is considered one of

the most powerful LLMs on the market. Claude models are famous for their capability in mathematics, programming and logic reasoning.

Gemini is Google's flagship LLM family, designed to integrate tightly with its broader ecosystem. The latest versions of Gemini support multimodal inputs—including text, images, audio, and video—and are positioned to compete at the frontier of model capabilities.

Meta's LLaMA series is one of the most influential open-source LLM families. It is commonly used as a foundation for fine-tuned or quantized models in local deployments. Many research institutions, particularly those not training models from scratch, choose LLaMA as their base, making it one of the most popular models in academia.

Qwen is Alibaba's open-source LLM series, notable for its strong multilingual support and solid performance in both base and chat variants. It has achieved top rankings on several Chinese language benchmarks. As of this writing, Alibaba has released multiple specialized versions of Qwen for different downstream tasks.

DeepSeek models are open-source and gaining traction in both research and developer communities due to their strong performance and ease of access. Released under the permissive MIT license, DeepSeek stands out for its efficient architectural design and training strategy, enabling it to compete with top-tier models at a relatively low computational cost.

8.4.2 Proprietary Model Fine-Tuning

"nobreak

8.4.3 Open-Source Model Fine-Tuning

"nobreak

8.5 Resource Augmented Generator

9

Agentic AI

CONTENTS

Agentic AI describes a comprehensive AI system that can make decisions and perform actions for either general or specific tasks, with minimal human intervention. It is a fundamental building block of autonomous systems.

The concept of agentic AI is not new. However, in recent years it has drawn increasing attention due to the rapid advancements in LLM. LLMs provide a general-purpose language interface that makes agentic AI significantly more scalable and cost-effective. As a result, agentic AI is included as a key topic in this part of the notebook.

9.1 Introduction to Agentic AI

The term "agent" refers to an entity that acts on behalf of a human. By this definition, agentic AI refers to AI-based systems that can solve complex problems with limited human oversight. This idea predates LLMs. However, with the rise of LLMs, the term "agentic AI" has been re-contextualized and is experiencing renewed popularity.

We explore agentic AI from two perspectives:

- Conventional intelligent agents. Classical agents developed in traditional AI literature before the rise of LLMs.
- LLM-driven agentic AI. Modern, language-enabled agents powered by large language models.

More details are given in the rest of the section.

9.1.1 Conventional Intelligent Agent

TBA

9.1.2 Agentic AI in the Context of LLM

Agentic AI in the context of LLMs refers to an autonomous system in which LLMs not only perform tasks, but also participate in decision-making and workflow orchestration. In such systems, LLMs serve both as computational units that solve sub-tasks and as the high-level orchestrator that determines how the problem should be approached.

An LLM in an agentic AI system may take on one or more of the following roles:

- Decompose the original problem into smaller sub-problems.
- Assign these sub-problems to LLM instances, either sequentially or in parallel.
- Evaluate the outputs of LLMs and either accept the results or reject them with feedback and revision instructions.
- Integrate the outputs from sub-problems into a final, coherent solution to the original problem.

In contrast, conventional LLM-integrated applications rely on humans or pre-written deterministic code to perform all of the above tasks. From this perspective, agentic AI further reduces human intervention and increases the level of automation.

A comparison is illustrated in Fig. 9.1. In the conventional architecture shown in (a), the workflow is either hard-coded or defined externally, and the LLM merely serves as a problem solver. In the agentic AI architecture shown in (b), the LLM autonomously orchestrates the task, determines the workflow, and governs the solution process.

Figure 9.1 is not the only architectural pattern for agentic AI. Another commonly used architecture is shown in Fig. 9.2. In this design, one LLM generates an output while another LLM evaluates its quality, decides whether to accept or reject it, and provides revision instructions if necessary. The two LLMs collaborate in an iterative loop, refining the solution step by step.

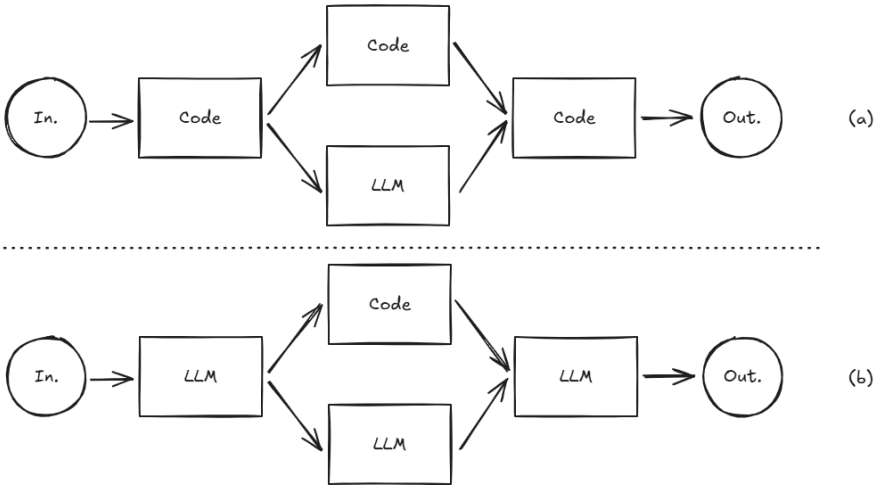

FIGURE 9.1
Conventional architecture (a) versus agentic AI architecture (b).

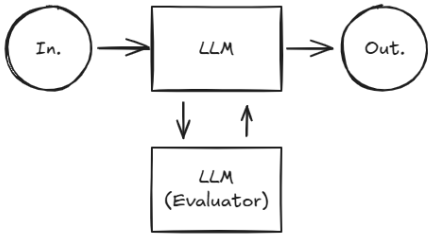

FIGURE 9.2
Self-evaluative agentic AI architecture.

This self-evaluative architecture has been shown to significantly reduce the likelihood of low-quality or hallucinated outputs.

Another key feature of LLM-based agentic AI is that it does not rely solely on the internal knowledge encoded in the model. Instead, it can be configured to interact with external tools such as databases, web browsers, calculators, etc., to enhance its problem-solving capabilities. This tool-augmented reasoning is essential for tasks that require up-to-date information, precise computation, or direct interaction with external environments. It also helps with reducing hallucination. A representative architecture is given in Fig. 9.3, where the LLM autonomously decides when to invoke an external tool and generates the appropriate commands using pre-defined protocols known as

the **Message Communication Protocol** (MCP). More details are given in later sections.

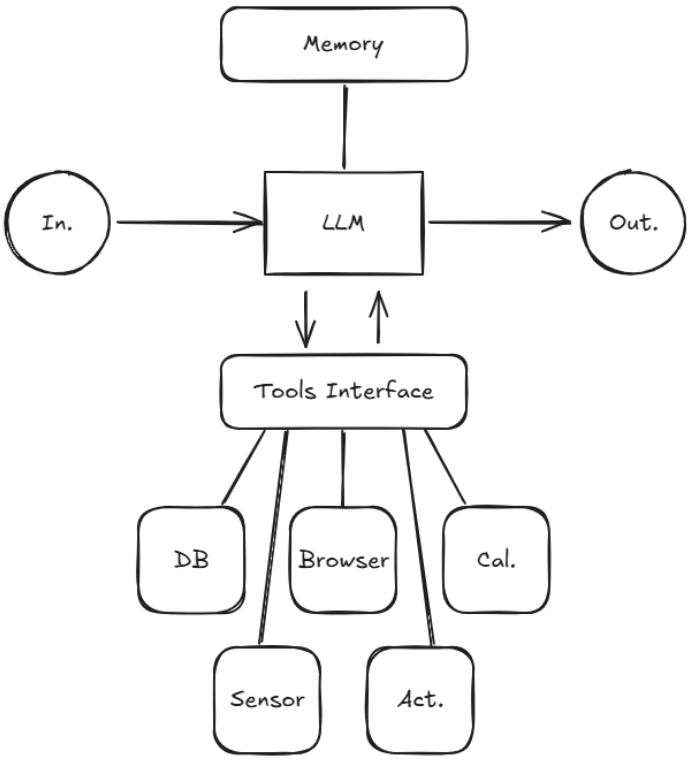

FIGURE 9.3
LLM with memory and tool interfaces to databases, web browsers, calculators, sensors and actuators, and other components that can interact with the environment.

Recall RAG which focuses on database-integrated LLMs to expand knowledge boundaries and suppress hallucinations. A fully developed agentic AI can be taken as an extension of RAG.

In addition, the LLM is equipped with a memory module, which stores historical inputs and outputs to maintain context and support long-term reasoning. This is also shown in Fig. 9.3.

9.2 Agentic AI Framework

Agentic AI can be understood as a form of LLM orchestration. The user specifies the overall task and provides a list of available tools, while the agentic AI system determines how to structure the workflow, which tools to invoke, how to prompt the LLMs, and how to evaluate their outputs.

An **agentic AI framework** refers to the infrastructure that supports the creation, configuration, and execution of such systems. It provides the necessary abstractions and utilities that allow users to define and deploy agentic AI pipelines more easily. While it is always possible to manually coordinate LLMs, prompts, and tool calls, using a mature agentic AI framework significantly simplifies the process of building scalable, multi-agent, or tool-augmented systems.

In this sense, an agentic AI framework plays a role analogous to that of Kubernetes in the context of containerized applications: it abstracts away low-level orchestration logic and enables the declarative specification and runtime management of complex, distributed components.

Common examples of agentic AI frameworks include the OpenAI Agents SDK, CrewAI, LangGraph, AutoGen, and many more. Details are introduced in later sections.

9.2.1 Asynchronous Python

Before introducing different agentic AI frameworks, it is worth reviewing asynchronous Python programming, as it is widely used across many agentic AI platforms.

Asynchronous programming (often abbreviated as async programming) is a well-established concept that predates agentic AI. In a nutshell, it creates a "multi-thread-like" framework where functions do not necessarily run in the order they are defined, but instead follow a "first-ready, first-served" execution model. In the context of async Python programming, an event loop switches between tasks whenever one is waiting (typically due to I/O). It is multi-thread-like in behavior, but there is no true CPU-level parallelism and no true multithreading. By itself, `asyncio` runs on a single thread and a single CPU core; true CPU-bound parallelism requires multiprocessing or native threads.

Even before the emergence of agentic AI frameworks, async programming had already proven valuable in many applications, such as web servers handling HTTP requests or user interfaces monitoring human actions on dashboards. By using async programming, an application does not need to freeze while a task is waiting for inputs—usually from a user or over the internet—and can continue executing other tasks in the meantime.

The `asyncio` library is the standard Python library for writing concurrent code using the `async`/`await` syntax, and many other async libraries are built

on top of it. A detailed introduction to `asyncio` can be found at [13]. A brief review of the basic syntax is given below.

The basic syntax to define and run a Python **coroutine function** is as follows:

```
import asyncio

async def <coroutine_function_name>(input) -> output:
    <do something; can contain nested coroutines>
    [return <something>]

[result = ]asyncio.run(<coroutine_function_name>(input))
```

Here, `async def` defines a coroutine function, and calling it returns a coroutine object. The function `asyncio.run()` runs the given coroutine object as the main entry point of the program. It creates and manages the event loop, runs the coroutine until it is complete, and then closes the loop. This approach is particularly useful when multiple coroutines are executed inside the main coroutine.

Calling a coroutine function directly, as in the example below, does not run it immediately; instead, it returns a coroutine object:

```
[<coroutine_object> = ]<coroutine_function_name>(input) # does not
    execute
```

In this case, the result is a coroutine object rather than the calculation's output. This coroutine object can then be scheduled and executed as follows:

```
[result = ]asyncio.run(<coroutine_object>)
```

When executing a coroutine object like this, there is no need to pass its input arguments again. A coroutine object can be awaited only once.

Another way to execute a coroutine is with the `await` expression:

```
[result = ]await <coroutine_function_name>(input)
```

However, this syntax can only be used inside another coroutine. It cannot be used at the top level, because `await` requires an already running event loop. What it does is suspend the current coroutine until the awaited coroutine completes, allowing the event loop to run other tasks in the meantime.

The `asyncio.gather()` function is another way to schedule multiple coroutines concurrently on the same event loop. For example:

```
x = await cx()
y = await cy()

x, y = await asyncio.gather(
    cx(),
    cy()
)
```

In the first approach, `cx` and `cy` are executed sequentially. If `cx` is delayed due to I/O, `cy` must wait until `cx` finishes. During this time, the event loop may run other ready-to-execute tasks. In the second approach, `cx` and `cy` are scheduled concurrently. While `cx` is waiting for inputs, `cy` can still execute. If both are waiting, `asyncio.gather` pauses, and the event loop switches to other tasks.

The same principle applies to the entire event loop. If the whole event loop is paused (its ready queue is empty), it uses the OS-level selector to wait for I/O readiness, and the operating system schedules other work until new events are ready.

9.2.2 OpenAI Agents SDK

The OpenAI Agents SDK simplifies the deployment of an agentic AI system without sacrificing flexibility. In fact, it enables more versatile agentic AI pipelines, such as chaining multiple AI agents using handoffs and nesting AI agents within tools-capabilities that would be almost impossible to implement manually due to the complexity of such systems.

The OpenAI Agents SDK offers at least the following features:

- Simplified deployment of a single AI agent or tool.

  For example, when defining a tool manually, the user must create a detailed "user manual" in JSON object format and pass it to the LLM. This JSON must include the tool's name, description, and input arguments with their types—carefully specified to ensure correct behavior. With the OpenAI Agents SDK, the user can simply add a decorator to a Python function to turn it into a tool, without needing to write a separate descriptive JSON object.

- Easy conversion of AI agents into tools.

  A single-line command can convert an AI agent into a tool, enabling nested structures. This significantly increases the capabilities of higher-level AI agents that use these tools.

- Chaining AI agents using handoffs.

  Information processed by an upstream AI agent can be passed to a downstream agent easily via **handoffs**, a mechanism for explicitly transferring control and data between agents during execution. The upstream agent decides when and where to pass the data. This allows true cooperation between AI agents and enables highly flexible agentic AI pipelines.

Figure 9.4 illustrates the key features of the OpenAI Agents SDK compared with a manual implementation. As shown, in the SDK-based pipeline, an AI agent can be converted into a tool and respond to tool calls (red arrows). Multiple AI agents can also be chained to form an upstream–downstream pipeline via handoffs (blue arrows). When an AI agent has multiple possible

downstream agents, it can decide at runtime which one to hand over data and control to.

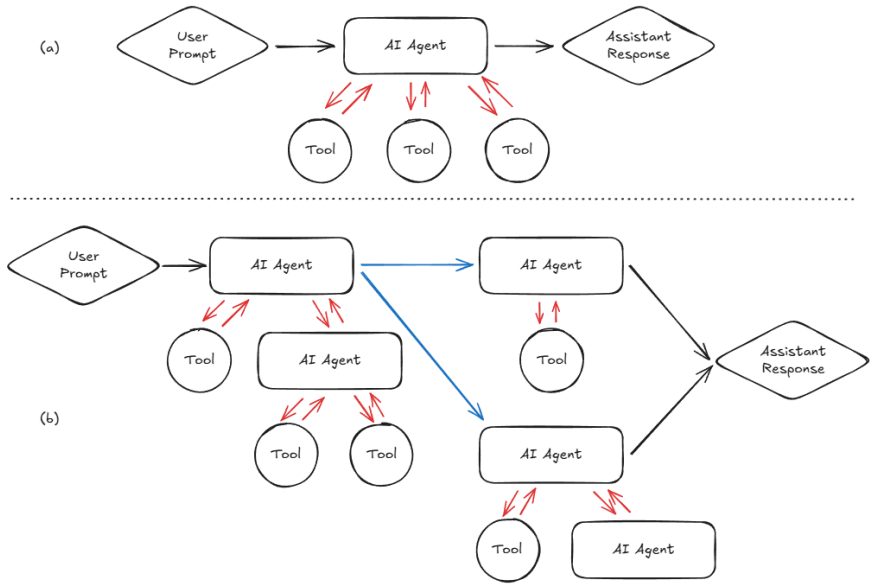

FIGURE 9.4
Manually implemented (a) versus OpenAI Agents SDK–based (b) agentic AI pipelines. Red arrows represent tool calls, and blue arrows represent handoffs.

The basic syntax is introduced below. All code examples are assumed to be executed in an environment with a running event loop. One can assume a wrapper as follows. Note that `OPENAI_API_KEY` is also assumed to be loaded as an environment variable.

Agent

```
import asyncio
from dotenv import load_dotenv

async def main():
    load_dotenv(override=True)

    <the code to be introduced>

if __name__ == "__main__":
    asyncio.run(main())
```

The following example demonstrates how to create an agent with a system prompt and a user prompt.

```
from agents import Agent, Runner, trace, function_tool

system_prompt = "<system prompt>"

agent_instance = Agent(
    name="<agent name>",
    instructions=system_prompt,
    model="<model>"
)
response = Runner.run(<agent name>, "<user prompt>")
output = response.final_output
```

The following code is useful when running multiple agents concurrently:

```
with trace("<trace name>"):
    results = await asyncio.gather(
        Runner.run(<agent name 1>, "user prompt 1"),
        Runner.run(<agent name 2>, "user prompt 2"),
        Runner.run(<agent name 3>, "user prompt 3"),
    )
outputs = [result.final_output for result in results]
```

The call to `await asyncio.gather()` works here because an event loop wrapper is already assumed. The function `asyncio.gather()` schedules all provided coroutines to run concurrently. Notice that the results will be returned in the same order as the input arguments regardless of which coroutine completes first.

The `trace()` context manager allows the user to monitor all LLM-related calls made to OpenAI, and these traces can be viewed on the OpenAI dashboard.

Tool

There are at least two ways to define a tool for an agent, as shown below.

```
@function_tool
def <tool 1>(<input 1>: <input type 1>, <input 2>: <input type 2>, ...)
    :
    """<description>"""
    <do something>
    return {"status": "success", <other returns>}

<tool 2> = <low tier agent name>.as_tool(
    tool_name="<tool name>",
    tool_description="tool description"
)

tools = [<tool 1>, <tool 2>]

agent = Agent(
    name="<high tier agent name>",
```

```
    instructions="<system prompt>",
    tools=tools,
    model="<model>"
)
```

The first method uses the `@function_tool` decorator and is intended for precise tools such as calculators or database query functions. It requires an explicit list of typed input arguments and returns well-defined, structured results.

The second method uses another low-tier agent as a smart tool via `as_tool`. The input to this tool is simply a string, which is treated as the user prompt for the low-tier agent. This creates a nested structure. The low-tier agent can call its own precise or smart tools, enabling more flexible and modular agentic AI pipelines.

Handoff

Handoffs are used to transfer data between agents. In this context, a handoff allows data generated by an upstream agent to be passed to a downstream agent. Downstream agents are registered as available handoffs, and the upstream agent can decide when and where to pass the data.

For a downstream agent, use `handoff_description` as a "self introduction" which will be presented to the upstream agent. An example is shown below:

```
downstream_agent = Agent(
    name="<name>",
    instructions="<instructions>",
    tools=[<tool 1>, <tool 2>, ...],
    model="<model>",
    handoff_description="<handoff description>"
)
```

List all available downstream agents and provide this list to an upstream agent. The upstream agent can then decide at runtime when to pass data and which downstream agent to use.

```
handoffs = [<downstream_agent 1>, <downstream_agent 2>, ...]
upstream_agent = Agent(
    name="<name>",
    instructions="<instructions>",
    tools=[<tool 1>, <tool 2>, ...],
    model="<model>",
    handoffs=handoffs
)
```

It is recommended to include clear instructions in the `instructions` (system prompt) of the upstream agent on how and when to use the downstream agents.

Run the upstream agent, and it will automatically trigger the downstream

agents if, based on its instructions and reasoning, it decides to do so. The return value will be the final result of the entire pipeline. Note that handoffs do not force an upstream agent to pass data. It is possible for the upstream agent to return its response directly to the user, completely bypassing the downstream pipeline. In this sense, an upstream agent determines the execution path of the pipeline from itself onward.

Guardrail

It is recommended to double-check both the initial user input and the final output of an agentic AI system to ensure that the input meets the required criteria or contains the necessary information, and that the output does not include harmful content.

There are several ways to achieve this. For example, OpenAI provides a moderation API that checks whether a piece of multimodal content contains material related to sexual content, hate, self-harm, and other restricted categories. More about this API will be introduced later in this section, and it will be tested in the project examples.

Within the OpenAI Agents SDK, guardrails can be implemented by defining dedicated AI agents to check the input and output of the agentic AI system according to customized requirements. Such guardrail agents return a boolean flag indicating whether the guardrail was triggered, along with details about what content caused the trigger. If a guardrail is triggered, it will raise an exception.

The following syntax demonstrates the minimum implementation of a guardrail. It involves the following steps.

1. Define a class to hold the return value of the guardrail agent.
2. Define the guardrail agent itself.
3. Define a coroutine function, decorated with `@input_guardrail` or `@output_guardrail`, that triggers the guardrail agent.

```
from pydantic import BaseModel
from openai_agents import Agent, Runner, input_guardrail,
    output_guardrail, GuardrailFunctionOutput


class <guardrail return class>(BaseModel):
       <is_triggered>: bool
       <reason or content>: str

guardrail_agent = Agent(
       name="<guardrail check agent name>",
       instructions="<instructions, often the things to check>",
       output_type=<guardrail return class>,
)
```

```
@output_guardrail
async def <guardrail coroutine name>(ctx: RunContextWrapper, agent:
    Agent, output: MessageOutput) -> GuardrailFunctionOutput:
       result = await Runner.run(guardrail_agent, output.response,
           context=ctx.context)
       return GuardrailFunctionOutput(output_info=result.final_output,
           tripwire_triggered=result.final_output.<is_triggered>)
```

When defining an agent in the pipeline, include the above coroutine function as a guardrail:

```
agent = Agent(
       name="Customer support agent",
       instructions="<instructions>",
       output_guardrails=[<guardrail coroutine name>],
)
```

The above example defines an output guardrail. An input guardrail can be defined in a similar way.

Notice that in OpenAI Agents SDK, `input_guardrails` only run when attached to the *first* agent in a chain (on the initial user input), and `output_guardrails` only run when attached to the *last* agent (on the final agent output). Guardrails attached to intermediate agents do not trigger.

Recall that OpenAI provides moderate API to check whether a content contains harmful information. It can be used as the output guardrail as follows.

```
@output_guardrail
async def moderation_guardrail(ctx, agent: Agent, message):
       response = client.moderations.create(
              model="omni-moderation-latest",
              input=message.response
       )
       flagged = response.results[0].flagged
       return GuardrailFunctionOutput(output_info=response.results[0],
           tripwire_triggered=flagged)
```

An example using OpenAI Agents ADK is given in Section 9.3.2.

9.2.3 CrewAI

CrewAI is not just a framework but also a company and an ecosystem. Depending on the context, CrewAI can refer to the following:

- A company that develops and provides agentic AI solutions.
- A platform offered by CrewAI to deploy agentic AI systems on the cloud.
- An application with a graphical interface that allows users to design and deploy agentic AI pipelines with minimal coding.

- An open-source agentic AI framework.

In this notebook, we will focus on the CrewAI framework.

Recall the OpenAI Agents SDK introduced in Section 9.2.2, where the user defines agents, tools, and handoffs, and links them together, while the agents decide which tools and handoffs to use. The CrewAI framework follows a very different philosophy. A "Crew" in this context is a team (e.g., engineering, data science, or marketing), where each AI agent acts as an employee. A job is assigned to the team, with specified context, inputs, and outputs. The user plays the role of hiring manager: defining the roles and characteristics of each AI agent, the context and goals of each task, and assembling the team from scratch. Once all agents and tasks are defined, the team completes the job.

In this sense, while the OpenAI Agents SDK requires the user to micromanage pipeline details, CrewAI emphasizes macro-management by defining context, inputs and outputs, and assigning agents to tasks. The OpenAI Agents SDK is more flexible, but when a job aligns well with CrewAI's philosophy, implementing it in CrewAI is often much easier.

CrewAI as UV Tool

CrewAI provides not only Python libraries but also CLI tools. It is recommended to install both the Python library and the CLI tool when using CrewAI. The CLI tool conveniently creates a CrewAI project directory tree with template Python scripts that users can adapt for their applications. Manually creating such a directory tree from scratch would be tedious.

To install and update CrewAI as a tool with `uv`, use

```
uv tool install crewai
uv tool install crewai --upgrade
```

9.2.4 LangGraph

"nobreak

9.2.5 AutoGen

"nobreak

9.3 Examples

Demonstrative projects are given as examples for a variety of agentic AI applications.

9.3.1 Manual: Semantic Search RAG

This section demonstrates a manually implemented retrieval-augmented agent. Documents (papers, reports, web pages, etc.) are stored locally. The agent answers user questions about these documents. Because the total corpus can be large, it is impractical to send all text to the LLM on every turn. Instead, the system performs semantic search to select only the most relevant chunks.

On first run, the program checks whether embeddings already exist locally. If not found, it chunks the documents into pieces with a specified token budget and uses OpenAI's vector embeddings API to compute vector representations. These vectors are cached locally. On subsequent runs, the cached embeddings are loaded.

On each user request, the query is embedded and used to retrieve the most relevant chunks. These chunks are appended to the prompt and sent to the LLM. The LLM is given an explicit "retrieval" tool. If it determines that more context is needed, it may trigger another semantic search with refined keywords or increase the number of returned chunks, or both. This can be performed as many times as the LLM requires.

Dialogues are stored locally so conversations persist across sessions. Only the user's messages and the assistant's replies are saved. Retrieved chunks are treated as ephemeral context and are not stored in the conversation log.

The implementation is framework-free (no agent framework) and can be found at [11]. Key components are outlined below.

Chunk Documents

The following codes are relevant to the chunk of documents.

```
def set_tokenizer(self):
    self.tokenizer = tiktoken.encoding_for_model(self.
        LLM_EMBEDDING_MODEL)

def tokenize(self, text):
    return self.tokenizer.encode(text)

def count_tokens(self, text):
    return len(self.tokenize(text))

def chunk_text(self, text):
    words = text.split()
    chunks = []
    chunk = []
    tokens_so_far = 0
    for word in words:
        token_count = self.count_tokens(word)
        if tokens_so_far + token_count > self.MAX_TOKENS:
            chunks.append(" ".join(chunk))
            if self.OVERLAP > 0:
```

```
                chunk = chunk[-self.OVERLAP:]
                tokens_so_far = self.count_tokens(" ".join(chunk))
            else:
                chunk = []
                tokens_so_far = 0
        chunk.append(word)
        tokens_so_far += token_count

    if chunk:
        chunks.append(" ".join(chunk))

    return chunks

def chunk_documents(self):
    chunks = []
    for filename in os.listdir(self.DOC_DIR_PATH):
        if filename.endswith('.pdf'):
            doc_path = os.path.join(self.DOC_DIR_PATH, filename)
            doc = fitz.open(doc_path)
            text = ""
            for page in doc:
                text += page.get_text()
            doc.close()
            chunks.extend(self.chunk_text(text))
        elif filename.endswith('.html') or filename.endswith('.htm'):
            with open(os.path.join(self.DOC_DIR_PATH, filename), 'r',
                 encoding='utf-8') as f:
                soup = BeautifulSoup(f.read(), 'html.parser')
                text = soup.get_text()
                chunks.extend(self.chunk_text(text))
    return chunks
```

The idea is simple and straight forward. For a given block of text, it first splits it into words, and then accumulates the words into a chunk while counting the total token size. Once the maximum token budget is reached, the words are packaged into a chunk. If chunk overlap is enabled, the next chunk starts by including the overlapped words. Otherwise, the next chunk starts fresh.

Embed Chunks and Perform Semantic Search

The following codes are relevant to the embedding of the chunks.

```
def generate_embeddings(self, chunks):
    embeddings = []
    for chunk in chunks:
        response = self.client.embeddings.create(
            model=self.LLM_EMBEDDING_MODEL,
            input=chunk,
            encoding_format="float"
        )
```

```
        embedding = response.data[0].embedding
        embeddings.append(embedding)
    return embeddings

def save_chunks_embeddings(self, chunks, embeddings):
    with open('parsed_chunks.json', 'w', encoding='utf-8') as f:
        json.dump(chunks, f, ensure_ascii=False, indent=2)
    np.save('embeddings.npy', embeddings)

def load_chunks_embeddings(self):
    if os.path.exists('parsed_chunks.json') and os.path.exists('
        embeddings.npy'):
        with open('parsed_chunks.json', 'r', encoding='utf-8') as f:
            chunks = json.load(f)
        embeddings = np.load('embeddings.npy')
        return chunks, embeddings
    return None, None
```

OpenAI's vector embedding API, `OpenAI.embeddings.create` is used to generate the embeddings. Both the embeddings and chunks that contain the original text are saved in ordered list. The idea is to use the embeddings for semantic search, and use the chunks corresponding to the top matched result for LLM analysis.

Once the user raise a request or the agentic AI decides to perform a new round of semantic search with some specified keywords, the following is executed.

```
def semantic_search(self, query, embeddings, chunks, top_n=0):
    similarities = []
    query_embedding = self.client.embeddings.create(
        model=self.LLM_EMBEDDING_MODEL,
        input=query,
        encoding_format="float"
    ).data[0].embedding
    for embedding in embeddings:
        similarity = np.dot(query_embedding, embedding) / (np.linalg.
            norm(query_embedding) * np.linalg.norm(embedding))
        similarities.append(similarity)
    top_indices = np.argsort(similarities)[::-1][:top_n]
    top_chunks = [chunks[i] for i in top_indices]
    return top_chunks
```

The query is embedded in the same manner, and the resulted vector compared with the embeddings vectors from the chunk. The chunks with the top similarity is returned.

Tools

Several tools are defined for the agentic AI. The most important tools include that the agentic AI is able to perform semantic search with customized

keywords and to increase the number of returned chunks. These tools allow the system to autonomously decide what and how many times to retrieve information from the documents to respond to user's question.

The following JSON objects are created and used as the "user manuals" for the LLM.

```
request_more_info_json = {
    "name": "request_more_info",
    "description": "Use this tool to request larger number of results
        from the semantic search",
    "parameters": {
        "type": "object",
        "properties": {
            "is_more_results_required": {
                "type": "boolean",
                "description": "Whether to return more results"
            }
        },
        "required": ["is_more_results_required"],
        "additionalProperties": False
    }
}

request_semantic_search_json = {
    "name": "request_semantic_search",
    "description": "Use this tool to request performing semantic search
         to the documents with a key sentence",
    "parameters": {
        "type": "object",
        "properties": {
            "semantic_search_key": {
                "type": "string",
                "description": "The key sentence for the semantic search
                    "
            }
        },
        "required": ["semantic_search_key"],
        "additionalProperties": False
    }
}
```

The above information is passed to LLM via role `function` as follows. Notice that in addition to the aforementioned tools, other tools are defined as well to record questions that the LLM cannot answer (likely due to that the information is missing from the document) and the suggestions from the user such as further information to be included to the documents or useful features the user would like the application to have.

```
def tools(self):
    return [
```

```
        {"type": "function", "function": record_unknown_question_json},
        {"type": "function", "function": record_suggestion_json},
        {"type": "function", "function": request_semantic_search_json},
        {"type": "function", "function": request_more_info_json}
    ]
```

Lastly, when the agentic AI decides to trigger the tools, the following codes handle the call.

```
def request_more_info(self, is_more_results_required):
    if is_more_results_required:
        self.TOP_N += 5
        if self.TOP_N > self.TOP_N_MAX:
            self.TOP_N = self.TOP_N_MAX
            return {"status": "error", "message": f"Cannot increase
                TOP_N beyond {self.TOP_N_MAX}."}
    return {"status": "success", "message": f"Top N increased to {self.
        TOP_N}."}

def request_semantic_search(self, semantic_search_key):
    if not semantic_search_key:
        return {"status": "error", "message": "Semantic search key is
            required."}
    chunks, embeddings = self.load_chunks_embeddings()
    if chunks is None or embeddings is None:
        return {"status": "error", "message": "Failed to load document
            chunks or embeddings."}
    context = self.semantic_search(semantic_search_key, embeddings,
        chunks, top_n=self.TOP_N)
    context = "\n\n".join(context)
    if not context:
        return {"status": "error", "message": "No relevant chunks found
            for the semantic search key."}
    else:
        return {"status": "success", "message": "Semantic search
            completed successfully.", "data": context}

def handle_tool_call(self, tool_calls):
    results = []
    for tool_call in tool_calls:
        tool_name = tool_call.function.name
        arguments = json.loads(tool_call.function.arguments)
        print(f"AI is calling tool: {tool_name}", flush=True)
        tool = getattr(self, tool_name, None)
        result = tool(**arguments) if tool else {"status": "error", "
            message": f"Tool {tool_name} not found"}
        results.append({"role": "tool", "content": json.dumps(result), "
            tool_call_id": tool_call.id})
    return results
```

Tools

Several tools are defined for the agentic AI. The most important capabilities are the ability to perform semantic search with customized keywords and to increase the number of returned chunks. These tools allow the system to autonomously decide what to retrieve, how often to retrieve it within a turn, and how much context to bring into the answer.

The following JSON objects are created and used as concise "user manuals" for the LLM. They describe the tool names, the intent of each tool, and the parameter schemas that the model should supply when invoking them.

```
request_more_info_json = {
    "name": "request_more_info",
    "description": "Use this tool to request larger number of results
        from the semantic search",
    "parameters": {
        "type": "object",
        "properties": {
            "is_more_results_required": {
                "type": "boolean",
                "description": "Whether to return more results"
            }
        },
        "required": ["is_more_results_required"],
        "additionalProperties": False
    }
}

request_semantic_search_json = {
    "name": "request_semantic_search",
    "description": "Use this tool to request performing semantic search
         to the documents with a key sentence",
    "parameters": {
        "type": "object",
        "properties": {
            "semantic_search_key": {
                "type": "string",
                "description": "The key sentence for the semantic search
                    "
            }
        },
        "required": ["semantic_search_key"],
        "additionalProperties": False
    }
}
```

The above information is passed to the LLM via the function/tool interface as follows. In addition to these retrieval tools, other tools are provided to record questions the LLM cannot answer because the information is miss-

ing, and to collect user suggestions about documents to add or features to implement.

```
def tools(self):
    return [
        {"type": "function", "function": record_unknown_question_json},
        {"type": "function", "function": record_suggestion_json},
        {"type": "function", "function": request_semantic_search_json},
        {"type": "function", "function": request_more_info_json}
    ]
```

Lastly, when the agentic AI decides to trigger the tools, the following code handles the calls.

```
def request_more_info(self, is_more_results_required):
    if is_more_results_required:
        self.TOP_N += 5
        if self.TOP_N > self.TOP_N_MAX:
            self.TOP_N = self.TOP_N_MAX
            return {"status": "error", "message": f"Cannot increase
                TOP_N beyond {self.TOP_N_MAX}."}
    return {"status": "success", "message": f"Top N increased to {self.
        TOP_N}."}

def request_semantic_search(self, semantic_search_key):
    if not semantic_search_key:
        return {"status": "error", "message": "Semantic search key is
            required."}
    chunks, embeddings = self.load_chunks_embeddings()
    if chunks is None or embeddings is None:
        return {"status": "error", "message": "Failed to load document
            chunks or embeddings."}
    context = self.semantic_search(semantic_search_key, embeddings,
        chunks, top_n=self.TOP_N)
    context = "\n\n".join(context)
    if not context:
        return {"status": "error", "message": "No relevant chunks found
            for the semantic search key."}
    else:
        return {"status": "success", "message": "Semantic search
            completed successfully.", "data": context}

def handle_tool_call(self, tool_calls):
    results = []
    for tool_call in tool_calls:
        tool_name = tool_call.function.name
        arguments = json.loads(tool_call.function.arguments)
        print(f"AI is calling tool: {tool_name}", flush=True)
        tool = getattr(self, tool_name, None)
        result = tool(**arguments) if tool else {"status": "error", "
            message": f"Tool {tool_name} not found"}
```

```
        results.append({"role": "tool", "content": json.dumps(result), "
            tool_call_id": tool_call.id})
    return results
```

Chat and Record Conversation

The following code is executed to initiate and maintain the chat session. It loads historical conversation records if they exist and, upon quitting, saves the most recent conversation history.

The functions below handle saving and loading conversation history from the local drive.

```
def save_history(self, history):
    with open('history.json', 'w', encoding='utf-8') as f:
        json.dump(history, f, ensure_ascii=False, indent=2)

def load_history(self):
    if os.path.exists('history.json'):
        with open('history.json', 'r', encoding='utf-8') as f:
            return json.load(f)
    return []
```

The following function generates the system prompt, which provides the LLM with role instructions, context, and an overview of the available tools.

```
def system_prompt(self):
    system_prompt = (
        f"You are a document explainer system. You are asked to explain
            variety of documents that is saved in the user's system.\n"
        f"Semantic search has been implemented prior to this request.
            The most relevant chunks relevant to the user's latest
            question have been identified. These chunks will be given
            to you shortly. The total number of chunks will also be
            given. \n"
        f"Your task is to provide a concise and accurate explanation of
            the document based on the provided chunks and the user's
            questions. \n"
        f"Use the following tools when necessary:\n"
        f"- record_unknown_question: Use this tool to record any
            question that couldn't be answered from the chunks, even
            after you have requested the maximum number of chunks.\n"
        f"- record_suggestion: Use this tool to record a suggestion for
            enriching the system, such as adding more documents or
            improving the search functionality.\n"
        f"- request_semantic_search: Use this tool to request performing
             semantic search to the documents with a key sentence of
            your choice. Use this tool when you think you need to query
             something from the documents for further information.\n"
        f"- request_more_info: Use this tool to request larger number of
             chunks returned from the semantic search. Notice that
```

```
            there is a limit on the number of {self.TOP_N_MAX} chunks
            that can be returned. Do not use this function to request
            more chunks if that limit is hit. Notice that in the
            beginning of each conversation, the chunk number is reset
            to {self.TOP_N_DEFAULT} upon the completion of a round of
            conversation.\n"
        f"Remember to always provide a clear and concise explanation,
            and use the tools only when necessary.\n"
        f"If you cannot find the answer in the chunks, let the user know
             honestly, especially if the user requires you to answer
            based on the chunks.\n"
        f"If you cannot find the answer in the chunks, and you think you
             can answer based on your own knowledge, let the user know
            that you are answering based on your own knowledge.\n\n"
    )
    return system_prompt
```

The following function block starts the chat session. It performs all actions introduced so far: it ensures embeddings are available, retrieves the most relevant chunks for the query, constructs the initial system prompt, and handles iterative interaction with the LLM.

```
def chat(self, query, history):
    chunks, embeddings = self.load_chunks_embeddings()
    if chunks is None or embeddings is None:
        chunks = self.chunk_documents()
        embeddings = self.generate_embeddings(chunks)
        self.save_chunks_embeddings(chunks, embeddings)
    context = self.semantic_search(query, embeddings, chunks, top_n=
        self.TOP_N)
    intro_prompt = self.system_prompt() + (
        f"Below are information chunks extracted from various documents
            .\n"
        f"Current number of chunks: {len(chunks)}.\n\n"
    )
    content_prompt = intro_prompt + "\n\n".join(context)
    messages = (
        [{"role": "system", "content": content_prompt}] +
        history +
        [{"role": "user", "content": query}]
    )
    done = False
    tools = self.tools()
    while not done:
        response = self.client.chat.completions.create(model=self.
            LLM_MODEL, messages=messages, tools=tools)
        if response.choices[0].finish_reason == "tool_calls":
            message = response.choices[0].message
            tool_calls = message.tool_calls
            results = self.handle_tool_call(tool_calls)
```

```
            messages.append(message)
            messages.extend(results)
        else:
            done = True
    return response.choices[0].message.content

def main(self):
    history = self.load_history()
    while True:
        query = input("You: ")
        if query.lower() in ['exit', 'quit']:
            break
        response = self.chat(query, history)
        self.TOP_N = self.TOP_N_DEFAULT
        print(f"Bot: {response}")
        print("---")
        history.append({"role": "user", "content": query})
        history.append({"role": "assistant", "content": response})
    self.save_history(history)
```

When the agentic AI triggers a tool, the LLM returns a structured function call in the `response` object generated by

```
response = self.client.chat.completions.create(model=self.LLM_MODEL,
    messages=messages, tools=tools)
```

The response for tool invocations contains one or more function call objects, each including a unique tool call ID, the function name, and its arguments. When a tool call is processed, its results must be appended to the `messages` list as a new entry with the role `tool`, along with the matching tool call ID. This explicit linking allows the LLM to correlate each tool's return with its original request, maintaining coherence in multi-step reasoning and tool use.

9.3.2 OpenAI Agents SDK: Semantic Search RAG with Online Cross Check

This section demonstrates the use of OpenAI Agents SDK to build a semantic search RAG with online cross check function. It is a rebuild and enhancement on top of Section 9.3.1. The agentic AI system contains two AI agents, the first Agent 1 carrying out semantic search on local chunks and answers the user's questions, while the second Agent 2 cross checks what the first agent provides against a one-time Internet query.

Details are given below.

Structured Output of Agent 1

The output of Agent 1 is not a plain text string, but a structured JSON that contains two fields, `is_require_cross_check` and `search_result`. Notice that it is up to Agent 1's decision whether to trigger Agent 2 Internet

cross check, the later of which is required only when Agent 1 is providing facts from the documents or from its own knowledge. When Agent 1 is casually chatting with the user without providing any information, or when it is recording user's suggestions, it does not need to trigger Agent 2.

The structured output is defined as follows.

```
from pydantic import BaseModel

class SearchAgentOutput(BaseModel):
    is_require_cross_check: bool
    search_result: str
```

Chunk Documents and Embed Chunks

This part is identical with Section 9.3.1.

Notice that in Section 9.3.1, a semantic search is performed before triggering any AI agent, and the result is sent to the AI agent with the user message. This is no longer the case in this example. In this example, all semantic searches are carried out by the AI agent with tools, and the AI agent choose the query keywords. The system does not offer the one-time initial semantic search.

Tools

The following tools are defined for Agent 1. They all have corresponding contour parts in Section 9.3.1, and are packed into function tools as required by OpenAI Agents SDK. The detailed explanation is neglected.

```
@staticmethod
@function_tool
def semantic_search(query: str):
    """Perform semantic search on the document chunks with the
        specified query."""
    print(f"SYSTEM: Performing semantic search with query: {query}")
    instance = explainer
    if instance.semantic_search_num >= SEMANTIC_SEARCH_MAX:
        return {"status": "error", "message": f"Maximum number of
            semantic searches ({SEMANTIC_SEARCH_MAX}) reached."}
    similarities = []
    query_embedding = instance.client.embeddings.create(
        model=EMBEDDING_MODEL,
        input=query,
        encoding_format="float"
    ).data[0].embedding
    for embedding in instance.embeddings:
        similarity = np.dot(query_embedding, embedding) / (np.linalg.
            norm(query_embedding) * np.linalg.norm(embedding))
        similarities.append(similarity)
    top_indices = np.argsort(similarities)[::-1][:instance.top_n]
    top_chunks = [instance.chunks[i] for i in top_indices]
```

```
    context = "\n\n".join(top_chunks)
    instance.semantic_search_num += 1
    if not context:
        return {"status": "error", "message": "No relevant chunks found
            for the semantic search key."}
    else:
        return {"status": "success", "message": f"Semantic search
            completed successfully and {instance.top_n} chunks
            retrieved.", "data": context}

@staticmethod
@function_tool
def request_increasing_top_n():
    """Request to increase the number of chunks returned."""
    print(f"SYSTEM: Requesting to increase the number of chunks.")
    instance = explainer
    instance.top_n += 5
    if instance.top_n > TOP_N_MAX:
        instance.top_n = TOP_N_MAX
        return {"status": "error", "message": f"Maximum value of top_n {
            TOP_N_MAX} reached and cannot be increased further."}
    return {"status": "success", "message": f"Maximum number of chunks
        returned increased to {instance.top_n}."}

@staticmethod
@function_tool
def record_unknown_question(question: str):
    """Record an unknown question, if the relevant information is
        missing from the chunks."""
    with open('unknown_questions.json', 'a', encoding='utf-8') as f:
        json.dump({"question": question}, f, ensure_ascii=False, indent
            =2)
        f.write('\n')
    return {"status": "success", "message": "Question recorded."}

@staticmethod
@function_tool
def record_suggestion(suggestion: str):
    """Record a suggestion for improving the document or the search
        process."""
    with open('suggestions.json', 'a', encoding='utf-8') as f:
        json.dump({"suggestion": suggestion}, f, ensure_ascii=False,
            indent=2)
        f.write('\n')
    return {"status": "success", "message": "Suggestion recorded."}
```

Agent 2 uses web search tool offered by OpenAI. The realization is not given. The use of the tool is explained in later part where the agents are introduced.

Agents

Agents 1 and 2 are defined below. Agent 1 queries local documents based on the user's message and generate the response. When the response contains factual information from the documents or from its own knowledge, it passes the output to Agent 2, who then perform web search to verify the facts.

```
def define_search_agent(self):
    system_prompt = (
        f"You are a helpful document explainer assistant.\n"
        f"Your task is to answer the user's questions based on the
            information recorded in the documents.\n"
        f"You can use tools to access the documents. You are allowed to
            perform semantic searches on the documents, "
        f"and you may perform multiple searches with different query
            contents. You are also allowed to increase "
        f"the number of returned chunks when necessary.\n"
        f"Always provide concise and accurate responses to the user's
            questions based on the documents.\n\n"
        f"Use the following tools when appropriate:\n"
        f"- record_unknown_question: Use this tool to record any
            question that cannot be answered from the chunks, "
        f"even after you have performed several searches.\n"
        f"- record_suggestion: Use this tool to record suggestions for
            enriching the system, such as adding more "
        f"documents or improving the search functionality.\n"
        f"- semantic_search: Use this tool to query the documents for
            further information. You can generate your own "
        f"queries when needed. You may perform at most {
            SEMANTIC_SEARCH_MAX} searches per user request.\n"
        f"- request_increasing_top_n: Use this tool to request a larger
            number of chunks returned from semantic search. "
        f"You may request this multiple times, with each increase adding
             5 chunks, capped at {TOP_N_MAX}.\n\n"
        f"Guidelines:\n"
        f"- Always provide a clear and concise answer.\n"
        f"- Use tools only when necessary.\n"
        f"- If the user's question is unclear or ambiguous, ask for
            clarification.\n"
        f"- Always try to answer based on the information in the
            documents. For this reason, you are encouraged to "
        f"perform at least one semantic search per user query.\n"
        f"- If you cannot find the answer in the returned chunks, you
            are encouraged to use semantic_search or "
        f"request_increasing_top_n before concluding, until the limits
            are reached or you believe no further relevant "
        f"information can be found.\n"
        f"- Do not add your own knowledge unless explicitly asked to, or
             if you believe the documents are lacking and "
```

```
        f"your knowledge can meaningfully improve the answer. When you
            do so, make it clear to the user.\n"
        f"- You may chat with the user without accessing the documents
            only if the user's message is not a question "
        f"(e.g., a suggestion for the system, or a request to summarize
            the conversation history).\n\n"
        f"Output Format:\n"
        f"- Always return a JSON object that matches this schema:\n"
        f"  {{\n"
        f'    "is_require_cross_check": true or false,\n'
        f'    "search_result": "<your answer text here>"\n'
        f"  }}\n\n"
        f"- If the users question involves factual claims (retrieval,
            summarization, knowledge-based), "
        f"set is_require_cross_check = true.\n"
        f"- If the users question is meta (e.g., summarizing
            conversation history, casual chat), "
        f"set is_require_cross_check = false.\n"
        f"- Put your full, clear, concise answer into search_result.\n"
        f"- Do not output anything except the JSON object.\n"
    )
    self.search_agent = Agent(
        name="Search agent",
        instructions=system_prompt,
        model=LLM_MODEL,
        tools=[
            self.record_unknown_question,
            self.record_suggestion,
            self.semantic_search,
            self.request_increasing_top_n
        ],
        output_type=SearchAgentOutput,
    )

def define_cross_check_agent(self):
    system_prompt = (
        f"You are a fact-checking assistant.\n"
        f"You will receive the final text output of an upstream
            assistant whose job is to "
        f"retrieve information from documents and summarize it for the
            user.\n\n"
        f"Your task:\n"
        f"- Review the answer for factual errors, misleading claims,
            outdated information, or statements "
        f"that conflict with your own knowledge.\n"
        f"- If you suspect a claim is wrong or outdated, you may perform
             at most one web search to verify.\n"
        f"- Output your findings as bullet points. For each issue, write
            :\n"
```

```
        f"  INCORRECT: <copied statement>\n"
        f"  CORRECTED: <the corrected or updated information>\n"
        f"- If you have a suspicion but cannot verify it with a single
            search, note it as:\n"
        f"  UNVERIFIED SUSPICION: <statement> - <why it might be wrong>\
            n"
        f"- If everything appears correct, output a single line: 'No
            factual errors found.'\n\n"
        f"Guidelines:\n"
        f"- Do not repeat or restate the full upstream answer.\n"
        f"- Keep your output limited to bullet points or the 'No factual
             errors found.' line.\n"
        f"- Be concise and clear in your corrections.\n"
    )
    self.cross_check_agent = Agent(
        name="Cross-check agent",
        instructions=system_prompt,
        model=LLM_MODEL,
        tools=[WebSearchTool(search_context_size="low")],
        model_settings=ModelSettings(
            parallel_tool_calls=False
        )
    )
```

Chat

Last but not least, the agents are put into a pipeline. Historical conversations are recorded. Documents chunk and embedding are performed in the first run of the system.

```
async def chat(self, query, history):
    messages = history + [{"role": "user", "content": query}]
    response = await Runner.run(self.search_agent, messages)
    return response.final_output

async def cross_check(self, query):
    messages = [{"role": "user", "content": query}]
    response = await Runner.run(self.cross_check_agent, messages)
    return response.final_output

def save_history(self, history):
    with open('history.json', 'w', encoding='utf-8') as f:
        json.dump(history, f, ensure_ascii=False, indent=2)

def load_history(self):
    if os.path.exists('history.json'):
        with open('history.json', 'r', encoding='utf-8') as f:
            return json.load(f)
    return []
```

```
def main(self):
    history = self.load_history()
    while True:
        query = input("You: ")
        if query.lower() in ['exit', 'quit']:
            break
        response = asyncio.run(self.chat(query, history))
        self.top_n = TOP_N_DEFAULT
        self.semantic_search_num = 0
        if response.is_require_cross_check:
            print("SYSTEM: Cross-checking information...")
            cross_check_response = asyncio.run(self.cross_check(response
                .search_result))
            final_response = (
                f"Part 1 - Information Retrieval & Summary Agent's
                    Answer:\n"
                f"{response.search_result}\n\n"
                f"Part 2 - Online Cross-Check Result:\n"
                f"{cross_check_response}"
            )
        else:
            final_response = response.search_result
        print(f"Bot: \n {final_response}")
        print("---")
        history.append({"role": "user", "content": query})
        history.append({"role": "assistant", "content": final_response})
        self.save_history(history)
```

Notice that `asyncio.run` is used to start the two agents.

A

Brief Introduction to Python Package Manager

CONTENTS

Two package managers, `conda` and `uv`, are introduced.

A.1 Conda

Developed and maintained by Anaconda Inc., `conda` is a free and open-source program for package and environment management. When installing and updating packages with `conda`, it automatically detects the hardware of the machine and the dependencies of packages, and applies the latest compatible versions.

The purpose of this chapter is not to give a detailed introduction to `conda` which can be found at [1]. This chapter is merely a summary of commonly used configurations and commands. Majority of contents in this chapter come from [2].

A.1.1 Installation

It is recommended that the user shall install Miniconda, a variation of Anaconda distribution, to use `conda`.

Miniconda is a light version of Anaconda distribution. The former of which contains only `conda`, `python` and a small number of other packages, while the latter contains more than three hundred packages, some of which are proprietary to Anaconda and may require a license for use in production environments.

A.1.2 Configuration of Channels

Channels refer to the cloud archive from where `conda` downloads and upgrades packages. There are default channels coming with `conda` installation, and the user can add channels or change their priorities in the list.

The channels are stored in file `.condarc`, which is usually stored in the user's directory. The user can add or remove channels by directly editing that file. Alternatively, there are commands to add channels or to check channel status.

To list all channels, use

```
conda config --show channels
```

Commonly seen channels include `defaults` and `conda-forge`. Notice that `defaults` is a collection of three Anaconda defined channels, and it contains propriety packages developed by Anaconda, and it may require licensing to be used in commercial environment.

To add a channel, either use

```
conda config --add channels <new channel>
```

to add a new channel to the top (highest priority) of the channel list, or

```
conda config --append channels <new channel>
```

to append a new channel to the end (lowest priority) of the channel list.

A.1.3 Environment Management

To list all environments, use

```
conda info --envs
```

To activate or deactivate an environment, use

```
conda activate <env name>
conda deactivate
```

respectively.

To list all packages and their source channels, use

```
conda list --name <env name> --show-channel-urls
```

where notice that the additional `--show-channel-urls` displays where each package come from.

To create an environment, use

```
conda create --name <env name> [python=<version>]
```

where additional `--file` followed by the list of packages in TXT or YAML file can be used to create an environment and install required packages. In this case, `conda` automatically checks and configures the machine setup and handles package dependencies.

To clone an environment, use

```
conda create --clone <source env name> --name <env name>
```

To remove an environment, use

```
conda remove --name <env name> --all
```

The environment, including the platform, packages and channels information can be exported as a plain text file, usually either a YAML file or a TXT file. This file can be used to quickly setup an identical environment in a later stage.

It is recommended to name the file after the environment name so that the name is preserved.

To export the environment, use

```
conda export --from-history><env name>.yml
```

A.1.4 Package Management

To install a package in specified environment, use

```
conda install --name <environment name> <package name>
```

To update all packages in an environment, use

```
conda update --all --name <environment name>
```

To remove a package from specific environment, use

```
conda uninstall --name <environment name> <package name>
```

If no environment name is specified in the above commands, current environment will be used.

A.2 UV

`uv` is another open-source Python package manager that has gained increasing popularity recently. Compared with `conda`, it has the following features:

- Fast. This is because it is implemented in Rust, a programming language whose performance characteristics are comparable to C/C++, whereas many other Python package managers such as `conda` are implemented in

the much slower languages such as Python. It is said that `uv` is typically 10 to 100 times faster than `conda` when comes to massive library installation.

- Git-style management. The user can use `uv` to initialize a Python project and create a virtual environment. Unlike `venv` or `conda` which store virtual environments and associated packages in a dedicated location outside the project folder, `uv` stores all virtual environment data inside the project root folder by default, typically in the `.venv` directory.

Details of `uv` can be found at [3]. A brief is given in the remaining of this section.

A.2.1 Installation

To install `uv`, follow the instructions in [3]. Installation is performed with a single-line PowerShell or Bash command on both Windows and Linux/macOS, which downloads the installation script and installs `uv` on the system.

For Linux and macOS:

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

For Windows:

```
> powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/
    install.ps1 | iex"
```

A.2.2 Python Interpreter Installation

Once `uv` is installed, the user may want to install a Python runtime interpreter for basic script execution.

Unlike `conda`, where each virtual environment is self-contained and includes its own `python.exe` (on Windows) or `python` binary (on Linux/macOS), `uv` manages Python interpreters in a centralized manner. In `uv`, the Python runtime is downloaded once and stored in a global cache directory, shared by all projects.

Project folders do not contain the interpreter itself. Instead, they include a `.python-version` file, which specifies the version of Python required for that project. When a virtual environment is created, it links to the corresponding cached interpreter.

To install the latest Python interpreter and make it available as `python3.<minor>` on your `PATH`, run:

```
uv python install
```

To install a specific version:

```
uv python install <version>
```

To also make this version the default `python` and `python3` executable in your shell, use:

```
uv python install <version> --default
```

With the above steps completed, the user can run basic Python scripts without creating a project or a virtual environment by using:

```
uv run <script>.py
```

In addition to the Python interpreter itself, similar commands can be used to install Python-related standalone executables such as `ruff`. These are not Python libraries, but tools that developers commonly use when developing Python applications. To install such tools, use:

```
uv tool install <tool>
```

The tool will be installed in a centralized cache.

Python interpreters and tools can be upgraded or removed using similar commands.

A.2.3 Environment Management

While both Python interpreters and standalone executable tools are installed in a centralized manner, Python libraries are installed within their corresponding projects. Each project defines its own environment, and the libraries and dependencies are managed by files stored locally inside the project root folder. This is where the package management becomes Git-style.

To initiate a new project, use

```
uv init <project name>
```

which will create a new directory with the project name, and inside the project, a demo "hello-world" python script. If the project folder already exists, set it as the current working directory and use

```
uv init
```

Once a project is initiated, `uv` automatically applies the following file system

```
|- .gitignore
|- .python-version
|- README.md
|- main.py
|- pyproject.toml
```

Notice that though `.gitignore` is automatically created and caches used by `uv` added, the user still needs to use `git init` should he wants to make it a git repository.

With Python interpreter installed, the user can run the script, by default "hello-world", using

```
uv run main.py
```

Once done, a `.venv` should be created as follows.

```
.
|- .venv
|    |- bin
|    |- lib
|    |- pyvenv.cfg
|- .python-version
|- README.md
|- main.py
|- pyproject.toml
|- uv.lock
```

Important files and subdirectories are explained below.

- `.venv` contains the actual code and binaries of the Python libraries installed for the project, along with scripts and environment-specific configuration.
- `uv.lock` is a human-readable lockfile that lists the fully resolved dependency graph for the project, including the exact names, versions, sources, and hashes of all libraries.
- `pyproject.toml` declares the minimum set of required libraries and dependencies for the project, without necessarily specifying exact versions or the complete dependency tree.
- `.python-version` specifies the required Python interpreter version for the project, which `uv` uses to select the appropriate cached runtime.

The `uv.lock` file contains libraries and dependencies information. It is automatically created and updated when the user runs the program or update the libraries. Share the project with `uv.lock` so that the receiver can re-create the environment. There is no need to specifically export a `requirements.txt` file so long as the receiver also uses `uv`.

While the lockfile is created automatically, the lockfile may also be explicitly created or updated using

```
uv lock
```

While the lockfile is automatically updated. the user can check whether it is up to date by

```
uv lock --check
```

The user can export `requirements.txt` specifically if it needs to be shared with someone with `conda`. To do that, use

```
uv export --format requirements.txt
```

To install files from `uv.lock` simply run the script and `uv` will automatically update the installed libraries. Alternatively, use

```
uv sync
```

to manually trigger an installation.

A.2.4 Package Management

By default, `uv` uses the Python Package Index (PyPI) for dependency resolution and package installation. However, `uv` can be configured to use other package indexes; details are not given here.

To install all packages specified in the `uv.lock` file, use `uv sync` as explained earlier.

To install one or more specific packages, use:

```
uv add <package>
```

This command not only downloads and installs the package(s) into the project's virtual environment, but also updates the `pyproject.toml` and `uv.lock` files accordingly.

Similarly, use:

```
uv remove <package>
```

to remove a package from the environment and update the dependency files.

Use:

```
uv tree
```

to display the project's dependency tree.

In addition to `uv add`, `uv` provides:

```
uv pip install <package>
```

which behaves like `pip install` inside the project virtual environment, and adds libraries to `.venv`. Unlike `uv add`, it does not modify the `pyproject.toml` or `uv.lock` files. This makes `uv pip install` suitable for local (usually experimental) installations that should not be traced as part of the project permanent dependencies. (Notice that `.venv` is in `.gitignore` by default, and it is not meant to share across platforms.) For reproducible dependencies, `uv add` is the recommended command.

Bibliography

[1] Inc. Anaconda. Anaconda org. https://anaconda.org/. Accessed: 2025-07-15.

[2] Inc. Anaconda. Cheatsheet. https://docs.conda.io/projects/conda/en/stable/user-guide/cheatsheet.html. Accessed: 2025-07-15.

[3] Astral. Uv. https://docs.astral.sh/uv/. Accessed: 2025-08-14.

[4] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*, 2023.

[6] Ed Donner. LLM engineering (ed donner). https://github.com/ed-donner/llm_engineering. Accessed: 2025-04-15.

[7] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[8] Zachary C. Lipton, John Berkowitz, and Charles Elkan. A critical review of recurrent neural networks for sequence learning, 2015.

[9] Haotian Liu, Chunyuan Li, Yuheng Li, and Yong Jae Lee. Improved baselines with visual instruction tuning, 2023.

[10] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. In *NeurIPS*, 2023.

[11] Sun Lu. Document explainer. https://github.com/sunluelectric/document-explainer. Accessed: 2025-08-11.

[12] Ollama. Ollama library. https://ollama.com/library. Accessed: 2025-04-14.

[13] Python Org. asyncio — asynchronous i/o. https://docs.python.org/3/library/asyncio.html. Accessed: 2025-08-11.

[14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[15] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`, 2023.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[17] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.

[18] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.