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) were released, people from photocopy shops would start selling notebooks photocopies that they claimed 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 read than the textbooks. I guess we cannot blame the top scorers for being so smart that they sometimes made 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 were probably my schoolmates. Why would I pay money to a stranger in a photocopy shop for my friends' notebooks, 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 (and 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 enough time on other irrelevant but still important and interesting contents.

To make a difference, I enforced 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?

Similar with most open-source software, this notebook does not come with any "warranty" of any kind, meaning that there is no guarantee that every-

thing 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.

The notebook is NOT peer reviewed, thus is more of a notebook than a book. It is meant to be easy to read, not to be comprehensive and very rigorous.

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 convenient 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 they are collected here in this notebook.

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.

- Andrew Ng, CS229 Machine Learning, Stanford, 2018
- 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
- Ed Donner, Ligency Team, The Complete Agentic AI Engineering Course. Udemy, 2025

List of Figures

List of Tables

Part I

General Artificial Intelligence Studies

1

Introduction to Artificial Intelligence

CONTENTS

This chapter discusses the concept and scope of artificial intelligence and gives a brief review of its development trend. Machine learning is a broader concept than artificial intelligence. Before the introduction of artificial intelligence, machine learning is briefly reviewed.

1.1 Artificial Intelligence

Artificial intelligence (AI) refers to the intelligent entities we human have built to mimic ourselves, or part of ourselves.

The scope and commonly seen approach is introduced in this section.

1.1.1 Scope

It is not easy to give a universally consistent definition to AI because our understanding of intelligence is evolving over time.

We look at ourselves and try to find out what makes human intelligent. Many concepts, models and even research areas have been proposed to explain what human intelligence is composed of. For example, human has **knowledge** which is a mechanism that allows us to remember facts and experience, and human can obtain new knowledge by either **reasoning** which derives new knowledge from existing knowledge, or **learning** which gets knowledge from examples, datasets or experiences. Human is usually **rational** when making decisions, which means we do the "right thing" that is most beneficial to

individuals or teams. Everything above, and probably many other factors that we are not aware of yet, jointly make human intelligent.

We want to build a system that can mimic human behavior and complete human jobs with the same or better quality. Ideally, the system should have human-level intelligence or even beyond. This level of intelligence is often known as **artificial general intelligence** (AGI).

Turing test, proposed by Alan Turing, is a widely accepted operational definition of (a portion of) AGI. A computer-based system passes the test if a human interrogator, after posing some questions, cannot tell whether the responses come from a computer or a human. Obviously, to pass the Turing test, the computer should be able to process natural languages, have knowledge representations, can do human-level reasoning, can make rational decisions, can learn from conversations, and can be generative and creative. A practically more useful AGI system should be able to process not just writing languages, but also audio and video.

As of this writing, we have not yet achieved AGI, although recent advent in large language model (LLM) (see Part IV of this notebook for more details) has for a short time made us believe that we might be close to it.

It is worth mentioning that although there is not yet a comprehensive AI that is comparable with a human at every aspect, we do have AIs that are very good at specific tasks, such as classifications, processing videos and audios, playing board games, autonomous driving, etc. They are not AGI, yet still useful and productive in practice, and have started playing important roles in human industry.

1.1.2 AI with Machine Learning

AI can be constructed in many ways. Imagine a rule-based system with very comprehensive rules. If done correctly, it may mimic a portion of human behavior. However, this is very difficult to achieve in practice due to the complexity in human. It is more often that we use machine learning to achieve AI.

Machine learning refers to the field of study that gives computers the ability to learn without being explicitly programmed to do so. Instead, it systematically learns from the data it sees. Compared with a explicitly programmed AI, machine learning is often less labor insensitive and more robust.

Depending on the application requirement and training method, there are at least the following 3 types of machine learning problems.

Supervised Learning

In **supervised learning**, samples are given in the form of (x, y), where x is the input features (usually in vector form) and y the label or reference. The purpose of supervised learning is to build a machine learning system that estimates y when receiving input x.

Example: on a house price guide website, given the size and location of a house, estimate its price on the market.

There are broadly divided two types of supervised learning problem, depending on the format of output y. If y is a continuous value , supervise learning becomes a **regression** problem which tries to map $y = f(x)$ with a (non-)linear function. If y is a discrete value taken from a finite set, supervised learning becomes a **classification** problem which tries to find the most likely label given the input.

Example: given a picture of an animal, classify what animal it is; given the size and location of a house, estimate its price guide.

Reinforcement Learning

In **reinforcement learning**, a sample is given in the form of (x, a, r) where x is the state of the system, a (optional) an action that the system takes, and r the reward. The purpose of reinforcement learning is to build a machine learning system that determines the optimal action to take that maximizes the utility of the system. The utility often includes not only the immediate reward, but also future foreseeable benefits.

Notice that different from supervised learning, the optimal action is not included directly in the sample during training. The machine needs to find out what the optimal action is by exploring the action space.

Example: in an autonomous driving vehicle, given the current status of the road environment and the vehicle, decide what actions to take; given a board game (such as chess, go, etc.) and its current board configuration, decide the best next move.

Unsupervised Learning

In **unsupervised learning**, a sample is given in the form of x only the input corpus without any label, reference or rewards. The machine is expected to find the patterns hidden in x. The purpose of unsupervised learning is to build a machine learning system that categories a new input based on the detected patterns.

Example: in a auto-fill text completion tool, given the first part of a sentence, fill in the remainder of the sentence; based on the movies a user has recently viewed, recommend them new movies that they are likely to enjoy.

1.2 AI Agent

“nobreak

1.2.1 Expectation

“nobreak

1.2.2 Agent Structure

2

Searching

CONTENTS

3

Knowledge and Logic Reasoning

CONTENTS

4

Decision Making

CONTENTS

An AI agent should be capable of making decisions. Based on what it observes, what it believes, and what it desires, the agent must determine the most beneficial action or sequence of actions to take. Different from a conventional optimal control problem where the plant and the cost function are assumed known, one of the biggest challenges for the AI agent is that it does not possess sufficient prior knowledge about the plant, and in many cases it needs to learn from successful and failed trails using both offline and online data, and gradually improve its performance.

This chapter studies decision making with AI. Markov decision process is discussed in detail, as it is one of the most widely used decision-making frameworks.

4.1 Utility

To formulate a decision-making problem mathematically, the first step is to define the **utility function** that quantifies the value of an action or a system state. Decision making is essentially the process of maximizing utility. This section introduces the concept of utility and its formulation, and how utility is used in decision making with AI.

4.1.1 Utility Theory

Intuitively, a rational AI agent should always choose the action that maximizes expected utility among all available actions. This is known as the **maximum expected utility** (MEU) principle. Below, MEU is formulated as an optimization problem.

When an AI agent decides to perform an action among all the actions it can take, the system transitions from one state to another. Let the **transition model** be denoted by $P(s'|a, e)$, where e represents the evidence or observation of the system, incorporating the agent's awareness of its current origin state, a the action taken, and s' the destination state. Notice that $P(s'|a, e)$ is given in the form of probability, which captures uncertainty in the system.

The benefit of reaching a state is quantitatively described by the **utility of a state** $U(s)$. The utility $U(s)$ usually includes not only the immediate reward gained from state s, but also the foreseeable future benefits that may arise from reaching s as an intermediate state. For now, do not bother how $U(s)$ can be calculated. Later in Sections 4.2 and 4.3, the systematic calculation of $U(s)$ will be introduced.

The **expected utility of action** $EU(a|e)$, given evidence e, over possible landing states $s_i \in S$, can be given by

$$EU(a|e) = \sum_{s_i \in S'} P(s_i|a, e) \left(R(s_i, a, e) + \gamma U(s_i)\right) \tag{4.1}$$

where S' is the set of all probable destination states by taking action a given observation e, $R(s_i, a, e)$ the reward gained by taking the action a to reach s_i given the observation e, $U(s_i)$ the utility of state s_i, and $\gamma \leq 1$ the discount factor. More about discount factor will be introduced in later sections.

Reward versus Utility

Reward and utility are two different concepts in the context of this chapter. Reward represents the instantaneous profit gained by the system by performing an action or a transition, or by leaving or landing on a state. Utility, on the other hand, is a comprehensive evaluation of "how much an action or a state worth", by considering not only the instanta-

neous reward, but also expected maximum reward (often with discount rate applied) to be received in the future.

As will be shown later, utility can be derived from fixed rewards and transition model.

Both reward and utility are relative values and they can be positive, zero or negative.

Given the evidence e, an AI agent may take one of several actions $a_i \in A$. MEU suggests that the optimal action is

$$a^* = \arg\max_{a_i \in A} EU(a_i|e) \tag{4.2}$$

An example of MEU is given by Fig. 4.1. Consider a maze in which a robotic AI agent is randomly placed in one of several locations (blocks). Each location corresponds to a state represented by the coordinate (x, y). The agent can move horizontally or vertically one step at a time. Let the utility of a state be given by its shortest distance to the goal, as shown in Fig. 4.1. It can be interpreted as the minimum turn required to reach the goal from its current location, if optimal actions are taken. (Notice that this is not how utility of a state is often calculated. It is used only in this demonstrative example.)

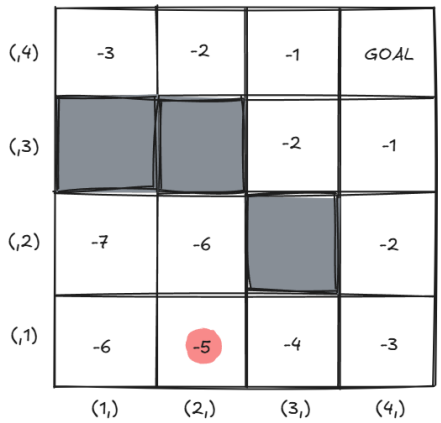

FIGURE 4.1
State utility map in the MEU example.

Let e represent the agent's current location, assumed to be the red dot at $(2, 1)$. The agent can take four actions and they are listed below together with

their possible resulting states and associated probabilities.

$$\begin{aligned} a_1 &= \text{UP} \\ P((2,1)|a_1,e) &= 0.2 \\ P((2,2)|a_1,e) &= 0.6 \\ P((1,1)|a_1,e) &= 0.1 \\ P((3,1)|a_1,e) &= 0.1 \end{aligned}$$

$$\begin{aligned} a_2 &= \text{LEFT} \\ P((2,1)|a_2,e) &= 0.2 \\ P((2,2)|a_2,e) &= 0.1 \\ P((1,1)|a_2,e) &= 0.6 \\ P((3,1)|a_2,e) &= 0.1 \end{aligned}$$

$$\begin{aligned} a_3 &= \text{DOWN} \\ P((2,1)|a_3,e) &= 0.7 \\ P((2,2)|a_3,e) &= 0.1 \\ P((1,1)|a_3,e) &= 0.1 \\ P((3,1)|a_3,e) &= 0.1 \end{aligned}$$

$$\begin{aligned} a_4 &= \text{RIGHT} \\ P((2,1)|a_4,e) &= 0.2 \\ P((2,2)|a_4,e) &= 0.1 \\ P((1,1)|a_4,e) &= 0.1 \\ P((3,1)|a_4,e) &= 0.6 \end{aligned}$$

Note that the outcome of an action can be nondeterministic, as is demonstrated by this example. In practice, this may result from signal transmission errors or environmental uncertainty. The agent cannot move downward from its current location $(2,1)$ since it is already at the bottom boundary of the maze. When it chooses the "DOWN" action a_3, it will likely remain in the same place after running into the wall.

Using (4.1), the expected utility of a_1 can be computed as

$$\begin{aligned} EU(a_1|e) &= 0.2 \times (-5) + 0.6 \times (-6) + 0.1 \times (-6) + 0.1 \times (-4) \\ &= -5.6 \end{aligned}$$

Similarly, $EU(a_2|e) = -5.6$, $EU(a_3|e) = -5.1$, and $EU(a_4|e) = -4.6$. Therefore, according to (4.2), the rational decision is to take action a_4, i.e., move "RIGHT."

In this example, all states s_i, transition probabilities $P(s_i|a, e)$, and utilities $U(s_i)$ are assumed known from the very beginning of the game. This is rarely the case in practical problems. Sections 4.2 and 4.3 will introduce variety of methods such as value iteration and Q-learning which will become handy when the aforementioned information is unknown from the beginning.

Are there alternatives to MEU?

The expected utility of an action is calculated using (4.1). An AI agent using (4.1) and (4.2) to make decisions. However, can we base decision making on other measures of utility? For example, consider the worst-case utility defined by

$$U^{\text{worst}}(a|e) \quad = \quad \min\{R(s_i, a, e) + \gamma U(s_i) \,|\, P(s_i|a, e) > 0\}$$

Instead of maximizing expected utility, could we maximize worst-case utility?

This question concerns the definition of a rational agent. To formalize this, the following **axioms of utility theory** is defined.

- **Orderability.** For any two actions a_1 and a_2, exactly one of the following statements must hold: "a_1 is preferred over a_2," "a_1 and a_2 are indifferent," or "a_2 is preferred over a_1." Thus, any two actions can be ranked.
- **Transitivity.** If a_1 is preferred over a_2 and a_2 is preferred over a_3, then a_1 must be preferred over a_3.
- **Continuity.** For three actions a_1, a_2, and a_3, where a_1 is preferred over a_2 and a_2 over a_3, there exists a probability p such that a compound action $a_{1,3,p,1-p}$ (which randomly chooses a_1 with probability p and a_3 with $1 - p$) is indifferent to a_2.
- **Substituability.** If a_1 and a_2 are indifferent and a_3 is any action, then for any p, the compound actions $a_{1,3,p,1-p}$ and $a_{2,3,p,1-p}$ must be indifferent.
- **Monotonicity.** If a_1 is preferred over a_2, then for compound actions $a_{1,2,p_1,1-p_1}$ and $a_{1,2,p_2,1-p_2}$, if $p_1 > p_2$, then $a_{1,2,p_1,1-p_1}$ is preferred over $a_{1,2,p_2,1-p_2}$.
- **Decomposability.** Compound actions can be nested or decomposed. For example, in compound action $a_{1,2,p,1-p}$, if a_2 is itself a compound action $a_2 = a_{21,22,q,1-q}$, then $a_{1,2,p,1-p}$ is indifferent to $a_{1,21,22,p,(1-p)q,(1-p)(1-q)}$.

Let $U(a|e)$ (or simply $U(a)$) denote the utility of action a. The specific form of $U(a)$, whether it represents expected utility or another valid measure, is acceptable as long as the following conditions are satisfied:

- $U(a)$ must exist for every action.
- $U(a)$ must reflect preference: if $U(a_1) > U(a_2)$, then a_1 is preferred over a_2; if $U(a_1) = U(a_2)$, they are indifferent.
- The preferences reflected by $U(a)$ satisfy the axioms of utility theory.

For any utility $U(a)$ satisfying these criteria, the utility of a compound action composed of actions $a_1, \ldots, a_n$ with corresponding probabilities $p_1, \ldots, p_n$ is given by

$$U(a_{1,2,\ldots,n,p_1,p_2,\ldots,p_n}) \quad = \quad \sum_i p_i U(a_i)$$

Note that the utility function is not unique. For instance, given a utility function $U(a)$,

$$U'(a) \quad = \quad \alpha U(a) + \beta, \quad \alpha > 0$$

is also a valid utility function.

4.1.2 Utility Function

As introduced in the beginning of this chapter, the utility function maps an action or a system state to a real number. The expected utility of an action $U(a|e)$ is an example of the utility function. As noted earlier, the expected utility of an action is not the only valid measure of an action. Nonetheless, it is the most widely used utility measure in commonly seen decision making frameworks. For the remainder of this chapter, unless otherwise specified, we consider expected utility as the utility function of an action.

From (4.1), the expected utility of an action $EU(a|e)$ depends on the transition model $P(s'|a, e)$ and the destination state utility $U(s')$. State utility typically includes both the immediate reward (from reaching that state) and the anticipated future utility obtainable from using that state as an intermediate step.

The computation of utility functions for actions and states is often referred to as **preference elicitation**. Different models have different ways of defining and calibrating utility functions, some of which will be introduced in later Sections 4.2 and 4.3.

Some general principles for assigning rewards and calculating utility function values are given below.

- Although rewards are relative and can take any number, it is often helpful

to define global upper and lower bounds, where the lower bound represents "immediate loss" and the upper bound represents "goal achieved". Rewards and utilities should be scaled accordingly.

- In many real-world problems, reward is expressed in monetary terms (financial gain or cost). Rewards and penalties of various types are converted to monetary values and normalized by a scaling factor.
- Decision makers may differ in risk preference. Risk-averse and risk-seeking agents can have different rewards even under identical circumstances.
- Mathematically derived rewards and utilities sometimes contradict human intuition, as humans are not always rational and do not always conform to the axioms of utility theory.

An **influence diagram** or **decision network** represents the structure of rewards or utility functions. It is a graphical model illustrating the relationships between utility, its contributing attributes, and the factors influencing those attributes, as well as the entities responsible for making decisions. Understanding this structure helps decision makers identify which factors influence outcomes and should be considered when determine the utility values.

An example is shown in Fig. 4.2, where a decision network assists a buyer in choosing a house. Here, the utility function depends on three attributes—location, construction quality, and price—each determined by several underlying factors. The entity "buyer" controls these attributes, selecting actions (purchase choices) that maximize utility.

4.2 Markov Decision Process

Up to this point, we have introduced the basic principles of decision making. In summary, utility must be defined for each action, and it should be designed to satisfy the axioms of utility theory so that the agent behaves rationally. In practice, the utility of system states is specified, and the utility of an action can be derived from state utilities and the transition model. The decision network helps formulate the utility of a state. The action with the highest expected utility is often regarded as optimal, and this principle is known as MEU.

Equations (4.1) and (4.2) are a realization of the above, with Fig. 4.1 providing an example. In that example, the utilities of all states and the transition model are assumed known, which is not always true in reality. While the example in Fig. 4.1 is simple, real-world problems are often more complex. Consider a similar setting to Fig. 4.1, except that

- The state space is unknown in advance.

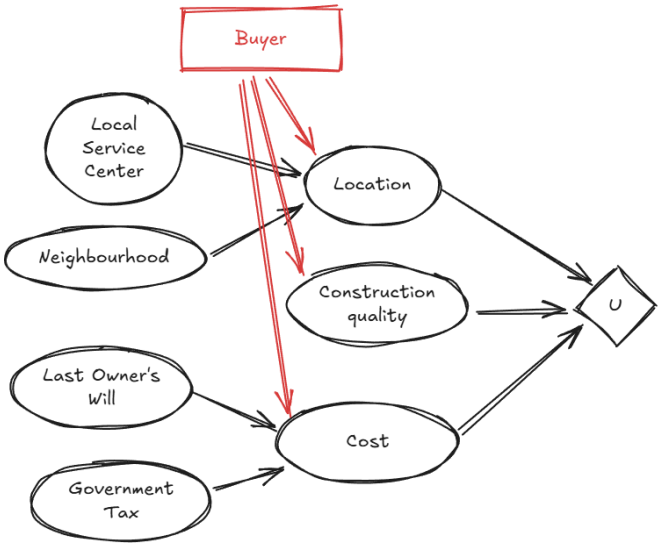

FIGURE 4.2
A simple decision network in which a buyer must decide which house to purchase.

The size, shape, and boundary of the maze are initially unknown, and the agent must explore by trial and error.

- The transition model is unknown in advance.

  There will be a list of available actions provided to the agent. However, the agent does not know which and how each action will lead it into the next state, and the agent must explore by trail and error.

- Both the reward and the utility of states and actions are unknown in advance.

  The agent does not know the reward it will gain for each transition, until that transition is performed during the trial and error. And since both reward and transition model is unknown, the agent cannot calculate the utilities of states and actions in advance. All the information needs to be gained through exploration.

These factors substantially increase the difficulty of the problem.

This section and the next Section 4.3 study MDP and discuss the calibration of state utilities and the transition model.

4.2.1 General Review of MDP

A **Markov Decision Process** (MDP) refers to the following decision-making problem formulation:

- It defines the state, which is a minimal set of information that reflects the current status of the system.
- It defines the transmission model, which describes the relationship between the current state, the action taken, and the resulting next state. The transition model is often given by the stochastic model $P(s'|s, a)$ (essentially the same as $P(s'|a, e)$ in (4.1), where current-state information is incorporated into the evidence e). The transition model is Markovian, i.e., it depends only on the current state and action, not on the full history.
- Agent utility is defined, which is the accumulation of rewards that agent gains by traveling among states. For each travel, the agent get some reward which is determined by the destination state, the action taken, the origin state, or all of them jointly.
- Actions are taken sequentially until a time horizon is reached or certain amount of utility is accumulated (often when the agent reaches a particular state).
- The objective is to determine the policy at each state. The policy maps each state to the best action that maximizes the expected utility.

Depending on whether the agent has the full picture of the current state, the MDP formulations can be divided into two types, namely fully observable MDP and partially observable MDP (POMDP). This section studies fully observable while the next Section 4.3 studies POMDP.

Known MDP versus Fully Observable MDP

Notice that a fully observable MDP assumes that the agent always knows its current state but does not necessarily know the transition model or reward function. It does not need to know all possible states either. These quantities can be learned or calibrated through trial and error using reinforcement learning.

If a fully observable MDP also has complete knowledge of all transition models at every state and all possible rewards for each state–action pair, it becomes a **known MDP**. Figure 4.1 illustrates an example of a known MDP, where the maze layout and the goal position are both clearly marked. In this case, the agent does not need to explore, and it can compute the optimal policy before taking any action.

Now consider a similar scenario in which the robot does not know the size or shape of the maze, nor the positions of walls or goals in advance. This means the agent does not know the set of reachable states, the transition model (i.e., which actions are possible in each state and their

outcomes), or the reward function. It still, however, knows its current position. In this case, the system remains a fully observable MDP, but it is no longer a known MDP.

How to Define the State

A natural question that follows is how to determine what information should be included in the state. After all, if we were to define the transition model or reward function themselves as part of the state (which we should not), a fully observable MDP would degenerate into a known MDP.

The state should include the minimum set of information, so that

- The state is Markovian. The available actions, reward and the transition model are functions of only the current state, but not any historical states or actions taken.
- The reward and transition model are stationary to the state.

If the state definition cannot fulfill the above requirements, consider augment the state with additional information.

The following examples illustrate how to decide what information should be included in the state. Consider a robot navigating a maze. It knows its current position, represented by coordinates, but it does not know the maze layout.

Consider the following different scenarios for the robot.

- Scenario 1: the robot needs to find the goal.

  In this case, the state should include only the robot's current location. With the current position as the state, a stationary transition model can describe how each action leads to a resulting state and what reward the agent receives if it reaches the goal or a trap. The agent may not initially know which positions correspond to walls or traps, but that information is stationary in the sense that when the agent revisits the same location, the same outcomes apply. Through repeated trials, the agent can learn these relationships.

- Scenario 2: The robot needs to travel as many unique locations as possible within a time horizon.

  In this example, it is insufficient for the state to include only the current location. Although the transition model can still describe how actions lead to new positions, this information alone cannot determine the reward, since the reward depends on whether the robot visits new locations or revisits old ones.

  Consider defining the state as the current location together with the

total number of unique positions it has visited. Indeed, the state provides enough information to calculate the reward. However, this time it does not possess enough information to calculate the transition model. Assume that the agent takes an action and arrives at a new location. Should that number increase by one? It does not know, because the agent is not sure whether it has arrived the location earlier. The transition model is not Markovian.

Therefore, to preserve the Markov property, the state must include the current location along with all the unique locations it has visited. Only then can both the transition model and the reward function be expressed in a Markovian and stationary form.

- Scenario 3: The robot needs to find the goal, but the layout of the maze changes over time and other factors.

  In this scenario, the positions of walls change according to certain external conditions (e.g., time, temperature, or other factors). The wall configuration directly affects the transition model. If the state includes only the current location of the robot, the transition model becomes non-stationary, violating the MDP assumption.

  The state must therefore include the external conditions that influence the maze layout, ensuring that the transition model remains stationary.

As demonstrated by Scenario 3, even if the transition model or reward function of a system is non-stationary (for example, changing with time or other external conditions), the problem can always be reformulated as a stationary MDP by augmenting the state with all relevant variables that cause the non-stationary, such as time or system mode. Therefore, in this notebook, it is assumed without loss of generality that the transition model and the reward function of an MDP are stationary. This assumption is standard in most theoretical formulations and practical implementations of MDP.

If the dynamics of the system is too complicated and there are "unknown" factors affecting the transition model and the reward, consider still augmenting those unknown factors in the state and formulate it as a POMDP problem.

4.2.2 Problem Formulation

An MDP is often described by

$$(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \mathcal{R})$$

where $\mathcal{S}$ is the set of states, $\mathcal{A}$ actions, $\mathcal{P}$ the transition kernel (denoted by $\mathcal{T}$ in some literatures), γ the discount factor, and $\mathcal{R}$ the reward.

State and actions have introduced in details in the earlier Section 4.2.1. The following concepts are introduced or revisited in this section.

- Reward
- Utility, Utility of State, Utility of Action
- Policy, Utility of Policy, Optimal Policy

Some of these concepts have been introduced previously. They are re-emphasized in the specific context of MDP.

Reward

A reward is the instantaneous gain or loss obtained by taking an action or staying at a state. Rewards are assumed to be stationary. If rewards vary dynamically (e.g., with time or environmental conditions), the factor causing this variation should be included into the MDP state so that the reward function remains stationary.

Rewards can take several common forms as follows.

- Reward by staying or reaching a state, $R(s)$

  In some problems, the agent gains a reward or penalty by reaching or staying at a particular state. The previous state or the action taken to reach that state is irrelevant. For example, in the robot-in-a-maze problem, the robot receives a positive reward when it reaches the goal and a negative reward when it reaches or stays inside a trap. The reward depends only on the destination state, not on the path taken.

- Reward by taking an action, $R(a)$

  In some problems, rewards are associated with performing actions. For example, a robot navigating a maze may receive a small negative reward for each move to represent battery consumption.

  In general, a reward can depend on both the origin and destination states, and the action taken. Such cases are denoted by $R(s, a, s')$, where the first s is the origin state, a the action, and s' the destination state.

Utility, Utility of State, Utility of Action

Utility, also known as **value function** (VF) in the context of MDP, d-iffers from reward. While reward represents an immediate benefit gained at a specific step, utility is the cumulative or comprehensive measure of value that reflects the long-term desirability of a state or an action. In other words, utility captures both the current and the expected future rewards achievable from a given state or action. Utility is typically derived or calibrated from rewards but embodies the notion of potential benefit rather than immediate gain.

The following example illustrates the distinction. Consider a game of chess. A reward is obtained only when the game ends in checkmate, positive for the winner and negative for the loser. However, checkmate cannot occur on the first move. If snapshots of the board are taken after each round, the probability of winning gradually increases for one player, from about 50% in the beginning to nearly 100% at checkmate. Although the actual reward is given only at the end, every board configuration during the game has a certain utility. The closer a configuration is to a winning position, the more likely the reward will be gained from that configuration, hence the higher its utility.

The calculation of utility from rewards will be introduced shortly, but before that it is worth mentioning the two types of MDPs with finite and infinite maximum allowed number of actions, as they are treated slightly differently when calculating utility.

- Finite horizon

  There is a fixed time horizon N. After N actions, the game ends and no further utility is accumulated. The goal is to maximize total utility within these N steps.

  In this case, although the transition model and reward function can remain stationary, the optimal policy may be non-stationary and vary with the remaining time. Suppose the agent returns to the same state after $k < N$ actions. With only $N - k$ steps left, the optimal policy may differ from what it would have been initially when all N steps were available.

  The optimal policy can be made stationary by augmenting the remaining time horizon into the state. When that value becomes zero, no more rewards can be received.

- Infinite horizon

  The game has no explicit time limit, and it can run indefinitely.

  In this case, the optimal policy is typically stationary. The optimal action at a given state remains fixed and does not depend on when the state is reached. Care must be taken in defining the utility function, however, because if rewards are unbounded or poorly scaled, the agent may loop indefinitely to accumulate infinite utility, rendering the problem ill-defined.

In whichever the case, assume the following scenario. An agent starts with zero utility or reward. It receives R_0 right away at the current state s_0. It performs action $a_{0,1}$ to transform state from s_0 to s_1. The action awards a reward $R_{0,1}$. After reaching state R_1, it receives a reward R_1. It then performs $a_{0,2}$ to transform state from s_1 to s_2 and it gains $R_{1,2}$. After landing at state s_2, it receives R_2, and so on. Stationary rewards and transition models are assumed.

The utility the agent collects can be formulated as a collective of rewards as follows.

$$U = R_0 + R_{0,1} + \gamma\left(R_1 + R_{1,2} + \gamma\left(R_2 + R_{2,3} + \gamma(\ldots)\right)\right) \tag{4.3}$$

where $0 < \gamma \leq 1$ is known as the **discount factor**. The discount factor has at least two purposes. When $\gamma < 1$ is used, it guarantees that the utility is bounded even in an infinite horizon MDP, so that the agent cannot gain infinite utility by circulating around a few state. It also reflects a widely adopted assumption in finance, where a future reward of the same value is often less worthy than an immediate reward.

Notice that one may formulate (4.3) as follows

$$U = R_0 + \gamma \left(R_{0,1} + R_1 + \gamma \left(R_{1,2} + R_2 + \gamma(\ldots) \right) \right)$$

where the action reward $R_{a,b}$ is treated the same way as R_b when applying the discount factor. This expression lives in a parallel world of (4.3) and everything should work just alright so long as everything is consistent. Nevertheless, (4.3) is more commonly adopted, where we assume that the reward of an action is received together with the origin state reward, not the destination state reward.

Following the spirit in (4.3), the utility of a state can be recurrently defined as follows.

$$U(s) = R(s) + \max_{a \in A(s)} \left(\sum_{s_i \in S} P(s_i|s,a) \left(R(s,a,s_i) + \gamma U(s_i) \right) \right) \quad (4.4)$$

where s is the state of interest, $U(s)$ the utility of state s, $R(s)$ the reward received immediately when reaching the state, $a \in A(s)$ all the actions that can be taken at the state, $s_i \in S$ all the states or all the states reachable from s with action a, $P(s_i|s,a)$ the probability of reaching s_i from s with action a, $R(s,a,s_i)$ the immediate reward obtained by action a transforming state from s to s_i, and $U(s_i)$ the utility of state s_i. Equation (4.4) is known as the **Bellman equation**. Notice that in Bellman equation, it is assumed that the agent is rational and always takes the action that maximizes the utility.

Bellman equation (4.4) can be used recurrently to calculate the utility of states. More details will be given in Section 4.2.3.

The utility of an action a at state s is given by

$$U(a|s) = \sum_{s_i \in S} P(s_i|s,a) \left(R(s,a,s_i) + \gamma U(s_i) \right) \quad (4.5)$$

which describes the expected return from taking an action at a state (under a policy). Equation (4.5) also known as the **Q-function**. Q-function (4.5) is related to utility of state (4.4) as follows.

$$U(s) = R(s) + \max_{a \in A(s)} U(a|s)$$

Equation (4.4) and (4.5) reveals the correlation between the utility of a state and the utility of an action. The utility of state s is the reward of arriving the state plus the maximum utility of action among all the actions the agent can take from the state.

Policy, Utility of Policy, Optimal Policy

A policy refers to a "rule book" that says what action to take at each and every state. A policy is often denoted by

$$\pi(s): \qquad s \to a \tag{4.6}$$

as a map from the state s to a. The utility of a policy, giving an initial state, can be calculated using (4.3). With stationary reward and transmission model and with a properly chosen discount factor, the utility of a policy is always bounded.

Among all the probable policies, the policy that gives the highest utility is known as the optimal policy, often denoted by $\pi^*(s)$. It is fairly easy to prove that with stationary reward and transmission model, the optimal policy is also stationary, meaning that when the system is at the same state, the optimal policy should always suggest the same action. In the remainder, only stationary policies are considered.

When the optimal policy is applied, the agent choose the action with the maximum utility from (4.5), i.e.,

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s_i \in S} P(s_i|s,a) \left(R(s,a,s_i) + \gamma U(s_i)\right) \tag{4.7}$$

When there are multiple actions that give the same largest utility, the agent randomly choose one action from them.

4.2.3 Value Iteration

To get the optimal policy $\pi^*(s)$ using (4.7), the key is to get the utility of state $U(s)$. The Bellman equation (4.4) provides a way to recursively calculate the utility of state as follows.

$$U_{i+1}(s) \quad \leftarrow \quad R(s) + \max_{a \in A(s)} \left(\sum_{s_i \in S} P(s_i|s,a) \left(R(s,a,s_i) + \gamma U_i(s_i)\right) \right) \tag{4.8}$$

which is known as the **value iteration** of MDP.

The following example is given to demonstrate the use of Bellman equation to update the utility of states.

A Robot in a Maze with Dynamic Rewards and Traps

Place a robot in a 4×4 maze.

The robot can move in all directions for one block, or stay at its current space in each round of action. The robot cannot move outside the maze. This defines 9 actions for the robot if it is in the center area of the maze, 6 actions if it is beside an edge, and 4 sections if it is at the corner. When the robot makes an action, there is a probability of P that the command is interpreted correctly and the robot will behave accordingly. There is a probability of $1-P$ that the robot fails to interpret the command, in which case it will randomly select an action from what it can do.

Some locations of the maze are assigned with rewards or penalties. There are two sets of different rewards configurations and they switch every 5 actions. The two setups are given in Fig. 4.3. The reward map is known in advance.

The robot is able to sense its location and also the time, meaning that it knows not only the current reward setup, but also the future reward map swaps.

The target is to design an MDP and calculate the utility of state using Bellman update.

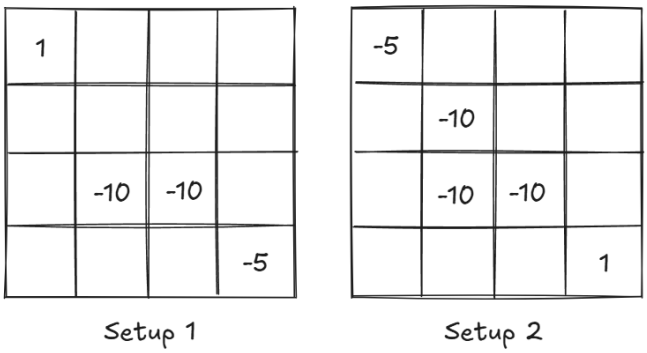

FIGURE 4.3
Reward in the example, where the reward change periodically over time.

Define the state. Notice that the rewards are determined jointly by the location and time cycle. To make the reward stationary, the time cycle information must be included in the MDP state definition. The two reward setups swap every 5 actions. Therefore, the reward configuration cycle is 10 actions. There are a total of $4 \times 4 \times 10 = 160$ states ($4 \times 4 = 16$ the distinct location of the robot, and 10 the time index in the reward cycle).

For the convenience, let x a 3×1 vector that can be used to denote the states. The first element $x(1) \in \{1, 2, 3, 4\}$ is the horizontal axis, the

second element $x(2) \in \{1, 2, 3, 4\}$ the vertical axis, and the third element $x(3) = \{1, \ldots, 10\}\}$ the time index.

Define the transition model. In this example, different states have different available actions. When the robot is at the edge or at the corner, it will have less moving options. For simplicity, only two actions from two states as follows are given as examples below. The rest actions can be formulated similarly.

Consider state $x = [2, 2, 1]$. This is one of the center area of the maze, and the robot can move 9 directions from this state (one of them being stay where it is). A total of 9 actions can be defined for this state. Take "up" action as an example. The destination state and the probability for this action is given below.

$$\begin{aligned}
P\left([1,1,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([1,2,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([1,3,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([2,1,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([2,2,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([2,3,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} + P \\
P\left([3,1,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([3,2,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9} \\
P\left([3,3,2]|[2,2,1],\text{up}\right) &= \frac{1-P}{9}
\end{aligned}$$

There are a total of 9 possible destination state for this action at this state. Notice that the time index increases by 1 then the action is performed. Consider another state $x = [4, 4, 10]$. This is a corner state, and the robot can move only 4 directions from this state. A total of 4 actions are defined. Take "left" as an example. The destination state and the probability for this action is given below.

$$\begin{aligned}
P\left([3,3,1]|[4,4,10],\text{left}\right) &= \frac{1-P}{4} \\
P\left([3,4,1]|[4,4,10],\text{left}\right) &= \frac{1-P}{4} + P \\
P\left([4,3,1]|[4,4,10],\text{left}\right) &= \frac{1-P}{4} \\
P\left([4,4,1]|[4,4,10],\text{left}\right) &= \frac{1-P}{4}
\end{aligned}$$

The time index loops to 1 after taking the action.

The reward is formulated as follows. There is no reward for actions $R(s, a, s')$, and only rewards for certain states $R(s)$ as follows.

$$\begin{aligned}
R([1,4,i]) &= 1, i = 1, ..., 5 \\
R([2,2,i]) &= -10, i = 1, ..., 5 \\
R([3,2,i]) &= -10, i = 1, ..., 5 \\
R([4,1,i]) &= -5, i = 1, ..., 5 \\
R([1,4,i]) &= -5, i = 6, ..., 10 \\
R([2,2,i]) &= -10, i = 6, ..., 10 \\
R([2,3,i]) &= -10, i = 6, ..., 10 \\
R([3,2,i]) &= -10, i = 6, ..., 10 \\
R([4,1,i]) &= -10, i = 6, ..., 1
\end{aligned}$$

Set discount rate to be $\gamma = 0.8$ and success rate of action to be $P = 0.9$.

Looping over the entire 160 states using (4.8) is known as one iteration. After 42 iterations, the utility of the state converges. Fig 4.4 gives some of the results.

Some highlights are as follows.

- The agent tries to stay far away from the "traps". There is a fail rate for a move. If the agent stay adjacent to a trap, it may fall into the trap by accident. For safety, the agent prefers to stay or move around at safer areas far away from the traps.
- The utility of state is a function of timestamps. Notice that although the reward configurations at timestamp 1 and 5 are identical, the utility of states are different. At timestamp 5, the agent is aware that the reward configuration is going to change, and it is preparing for it. There are several motivations. The original reward sweet point will be a trap, and the agent wants to stay away from it. There is a discount factor, which motivates the agent to move to the new reward point as quickly as possible.

When the reward configuration and transition model are known and stationary, we can use value iteration to calculate the utility of state. With utility of state known, it is straight forward to decide the optimal policy using (4.7).

It can be proved that if infinite iteration is used, Bellman update will converge to the unique global optimum. Proof is not given here.

4.2.4 Policy Iteration

Policy iteration is an alternative way of finding the optimal policy given the reward configuration and transition model. Unlike the value iteration where the utility of the state is first calculated, in policy iteration, it is started with assuming a policy $\pi^0(s)$ which is not necessarily optimal.

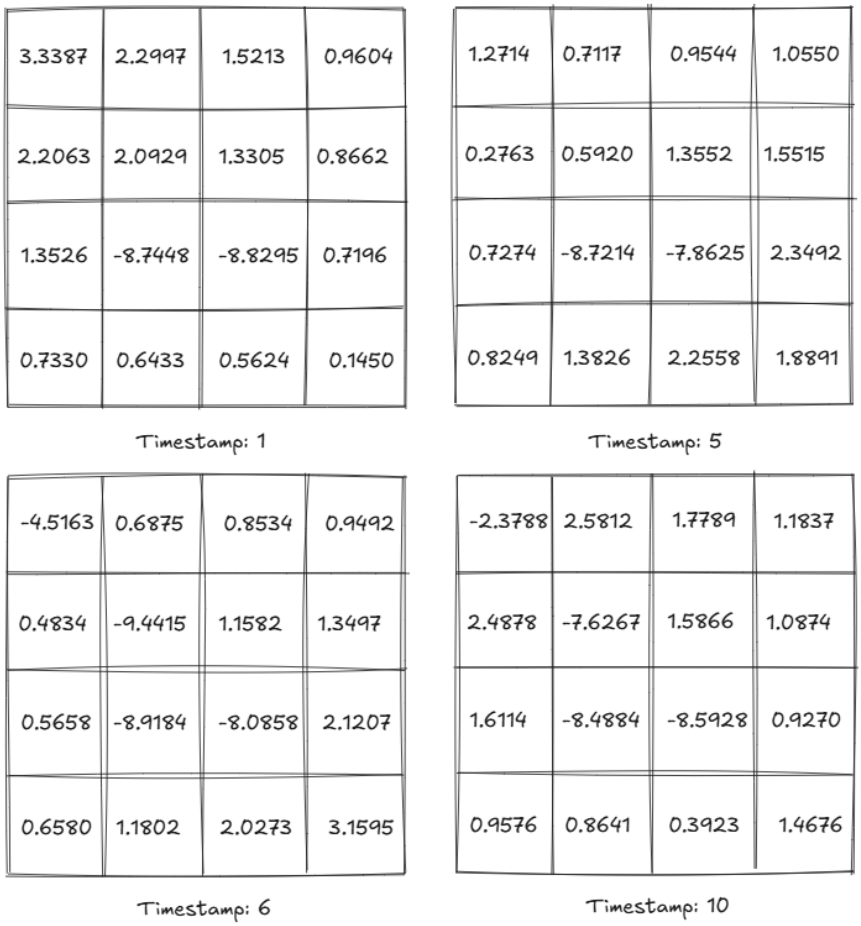

FIGURE 4.4
Converged utility of state at timestamps 1,5,6 and 10.

With that $\pi^0(s)$, the utility of state can be calculated using (4.3). With the utility of state calculated, use (4.7) to revise the policy to get $\pi^1(s)$. Repeat the above steps iteratively until the utility of state and the policy converge.

Policy iteration and value iteration should lead to the same result. It is reported that sometimes policy iteration is easier to implement.

4.2.5 MDP Online Reinforcement Learning

Both value iteration and policy iteration assume known reward configuration and transition model, which is often not true. In practice, the agent needs to explore and find the transition model associated with each action at each

state as well as the rewards received from each action or state. Inspired by policy iteration, the following **Q-Learning** algorithm can be used.

1. In the beginning when there is no information of reward and transition model, initialize utility of state (4.4) with 0 for all states. Initialize the Q-function with 0 for all actions at all states.

2. Let the agent freely transition from the current state until the end of the session, either when it reaches the goal, runs into a trap, or after certain time. Notice that the action with the largest utility is not necessarily selected. It uses a exploit-explore model instead, which will be introduced later.

3. For each state visited in the trail, update the Q-function of the actions taken together with the utility of the state as follows, whenever an action is taken.

$$\begin{aligned} U_{k+1}(a|s) &\leftarrow U_k(a|s) + \alpha_k \left(R(s,a,s_i) + \gamma U_k(s_i) - U_k(a|s)\right) \quad (4.9) \\ U_{k+1}(s) &\leftarrow R(s) + \max_{a \in A(s)} U_{k+1}(a|s) \end{aligned}$$

where steps 2 and 3 are repeated so that the agent keeps learning from different trails, until the agent has gained enough information about each and every action and each and every state, and no further learning is required.

In (4.9), α_k should fulfill the following criteria, so that the Q-function and utility of state converge to the optimum.

$$\sum_{k=0}^{\infty} \alpha_k^2 < \infty$$
$$\sum_{k=0}^{\infty} \alpha_k \to \infty$$

which implies that until everything converges, the agent must keep learning. An example of α_k can be $\alpha_k = 1/k\alpha_0$ with a finite α_0.

As mentioned earlier, for the agent to keep learning, it should not always stay in its comfort zone by running only the estimated optimal policy. It is encouraged to always explore new actions that is not the best action under existing estimated optimal policy. It is important to maintain a proper exploit-explore balance.

The **ϵ-greedy exploration** model is recommended. In the kth trail, for each action the agent is taken, it has a $1-\epsilon_k$ probability of choosing the optimal action under the current estimated optimal policy, and a ϵ_k probability of choosing a random action, regardless of its Q-function value. The value of ϵ_k decreases as more and more trails carry out, so that the model will perform

more conservative as the learned information accumulates. A common choice is $\epsilon = 1/k$.

Q-Learning is similar with value iteration or policy iteration, except that it is model-free. It is assumed that the reward configuration and transition model are unknown. It uses trails and reinforcement learning to update the utility of actions and utility of states by exploring the different states and transitions of the system.

4.3 Partially Observable MDP

Partially Observable Markov Decision Process (POMDP) refers to the case where the agent is not always clear which state it is in. Imagine in the example given by Fig 4.1, the agent has no sense of its location in coordinate, but can detect how many walls it is adjacent to. The problem then becomes a typical POMDP.

POMDP differs from fully observable MDP in several aspects. Though the spirit remains the same where the decision is made based on a Markov process, many details in the execution needs to be adjusted. Details are introduced in this section.

4.3.1 Belief of State

The agent, in this case, has a **belief of state** often denoted by

$$b(s) \quad = \quad [p_1, \ldots, p_n]$$

where n is the total number of state and p_i the belief of the agent currently in state i. One can think of $b(s)$ as a way to denote the state. The agent makes decisions based on $b(s)$.

The evidence e is the measurement available to the agent. The probability $P(e|s)$ associates the state belief with the evidence. The agent can adjust its belief as follows. Let $b(s)$ be the agent's initial belief. The agent takes action a, and after that it observes e. The new belief becomes

$$b'(s') = \alpha P(e|s') \sum_s P(s'|s, a) b(s)$$

where α is used for probability normalization so that $\sum b_i(s) = 1$ is ensured. The decision making is Markovian just like the fully observable MDP, where the optimal action depends on only the current belief of state $b(s)$. It does not depend on past states, and it does not depend on actual state.

The biggest challenge is that the $b(s)$ is a group of probability, and hence continuous and has infinite values. For example, the policy $\pi(s)$ (now $\pi(b)$)

will be difficult to define, as if it were done in the conventional manner, there will be infinite possibilities. For instance, we cannot assign an action for $b(s) = [0.5, 0.5]$ and another action for $b(s) = [0.501, 0.499]$. Also, the transition model which is used to be $P(s'|s, a)$, is now $P(b'|b, a)$ with b a continuous vector.

In the remainder of the section, it is introduced how POMDP defines the transition model and optimal policy in the context of belief of state.

4.3.2 Transition Model with Belief of State

The transition model of actual state $P(s'|s, a)$ and the evidence and actual state correlation $P(e|s)$ are used to derive the transition model of state of belief $P(b'|b, a)$ as follows.

$$\begin{aligned} P(b'|b,a) &= \sum_e P(b'|e,a,b)P(e|a,b) \\ &= \sum_e P(b'|e,a,b)\sum_{s\prime} P(e|s')\sum_s P(s'|s,a)b(s) \end{aligned}$$

where

$$\begin{aligned} P(e|a,b) &= \sum_{s\prime} P(e|s')P(s'|a,b) \\ &= \sum_{s\prime} P(e|s')\sum_s P(s'|s,a)b(s) \end{aligned}$$

4.3.3 Value Iteration and Optimal Policy with Belief of State

“nobreak

4.3.4 POMDP Online Reinforcement Learning

“nobreak

4.4 Continuous-State Markov Decision Process

Up to this point, both the fully observable MDP and POMDP assume that the number of states is finite. In a more general and practical control problem, however, the number of state can be infinite, mostly because the state variables in the state space vector can take continuous values.

Continuous-state MDP describes an MDP formulation where the state variables can take continuous values and hence there are infinite number of states. More details are discussed in the remainder of this section.

4.4.1 Problem Formulation

Consider a control problem where the state space is given by $s \in S = \mathbb{R}^n$. While the order of the system, i.e., the number of the state variable, is a finite number n, s can take infinite and continuous values.

The purpose is to decide $a \in A(s)$. There is no parametric model that describes how a affects s, and there is no reference signal for the state s either. However, there is a reward $R(s)$ corresponds with each s. The reward is perceivable by the system.

From the above description, it is obvious that the problem can hardly be formulated into a parametric model based control problem or an adaptive control problem. It is a reinforcement learning problem by nature. Thus, consider using MDP to formulate and solve the problem. Conventional MDP assumes finite number of state. In this problem the state can take infinite values. Continuous-state MDP formulation is proposed to solve the problem as follows.

4.4.2 Discretization Approach

Intuitively, we can discretize the continuous values the state vector can take, and map them into discrete values. For small scale system where the length of state vector and the ranges each state variable can take is limited, discretization approach may work. However, it has the following problems preventing it from being implemented in large scale systems.

- Information loss due to the staircase fitting.
- Curse of dimensionality. The total number of states grows exponentially larger as the system scale grows. Consider a system with n state variables, each discretized into k values. The total number of states is k^n.

The probable solution to the above two problems, under the scope of discretization approach, conflict with each other. To reduce the information loss due to the staircase fitting, the discretization resolution for each state variable needs to be increased, which would cause more severe curse of dimensionality.

In practice, discretization approach only works for small-scale system. For system with order $n \leq 3$, it is often safe to use discretization approach. For system with order $3 < n \leq 6$, there is a chance that it may work. In these cases, the user needs to choose the discretization strategy carefully by using low resolution for state variables that matter less. For system with order $n > 6$, discretization approach is generally not recommended.

4.4.3 Utility Function Approximation approach

Notice that in the remainder of this Section 4.4.3 finite number of actions is assumed while state vector can take infinite continuous values. This is a fair assumption in many control problems, as the action space is often limited

compared with the state vector. In the case where the actions take continuous values, it is often fine to discretize only the action space.

Consider approximating the utility function of a state directly without discretization, i.e.,

$$U(s) \approx \theta^T \phi(s) \tag{4.10}$$

where θ is the linear regression coefficient, and $\phi(s)$ the feature of state s. With (4.10), we can then use (4.5) and (4.7) to choose the optimal action and decide the optimal policy.

As the first step to implement utility function approximation, consider building a model of the plant as shown in Fig. 4.5. The model is also known as the simulator.

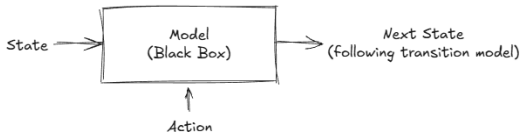

FIGURE 4.5
Model (simulator) of the plaint in the continuous state MDP.

Notice that the model is treated like a black box and the user does not need to possess parametric information of the plant. In practice, the model can be constructed purely from data derived from past trajectories when an human operator was running the system. Consider using supervised learning to estimate $s' = f(s, a)$, where $f(\cdot)$ is the plant black box. More about supervised learning is given in Chapters 5, 6 and 7.

Formulations of the Model

The formulation of the model is not unique especially when it comes from the data. For the same dataset, the user may choose between linear or nonlinear models, and deterministic or stochastic models. There is no globally correct way of formulation for all different applications. In practice, stochastic models like (4.11) are often preferred over deterministic models for enhanced system robustness.

$$s' = f(s, a) + \epsilon \tag{4.11}$$

With the built model, we can then train the MDP using reinforcement learning. This is known as the **model-based reinforcement learning**. This is popular in many practical problems when the true plant is too expensive to run. The user can use the model to generate massive amount

of synthetic data which gives him flexibility in choosing the feature of the state.

With the defined mode, we need to select finite number of random states $s_1, \ldots, s_m$ from the infinite continuous-value state space. Apply value iteration as follows. For each sampled state s_i, let $a \in A(s_i)$ be an action applicable to the state. Iteratively calculate the utility of the action and the state as follows.

$$U(a|s_i) \leftarrow \frac{1}{j}\sum_{i=1}^{j}[R(s_i, a, s_j) + \gamma U(s_j)] \tag{4.12}$$

$$U(s_i) \leftarrow R(s_i) + \max_a U(a|s_i) \tag{4.13}$$

where j is the number of trial that action a is taken, and s_k the destination state. This can be emulated with the aforementioned model. Massive data is required for the training, as there is often a large number of state samples m and a large number of trials of actions j. This is why the model can become handy. The utility $U(s_k)$ is calculated with $U(s_k) = \theta^T\phi(s_k)$ with θ from the last iteration.

With $U(s_i)$ obtained through value iteration (4.13) for the sampled discrete states, consider mapping it with continuous function (4.10) via linear regression, i.e.,

$$\theta = \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}\left(\theta^T\phi(s) - U(s_i)\right)^2 \tag{4.14}$$

We need to choose the feature of the state $\phi(s)$. This is often experience based, where $\phi(s)$ can be a subset or an extension of the original s together with nonlinear components derived from components of s that may jointly contribute to the utility. With the massive synthetic data generated from the model, the user can choose a large set of features without necessarily worrying about over-fitting. This is another benefit of using a model.

The number of samples m required by the fitting (4.14), which is also the number of sampled states, depends on the formulation of $\phi(s)$. For example, for a feature that contains 50 elements in $\phi(s)$, consider m of at least between 500 and 1000.

For online application, the optimal policy is given by

$$\pi^*(s) = \arg\max_a [R(s, a, s_j) + \gamma U(s_j)] \tag{4.15}$$

where $s_j = f(s, a)$ is obtained from the model (4.11) with $\epsilon = 0$. Notice that although in the training discrete action has been used in the value iteration, (4.15) can be formulated with continuous action space. Details are not discussed here.

Replace (4.12) and (4.13) with the online greedy-search based counterparts for online learning. Details are not introduced here.

LQR as a Special Case of Continuous-State MDP

In a special case of continuous-state MDP where the state transition function and the model can be described by the following linear equation

$$s' = Fs + Ga + w$$

with $s \in \mathbb{R}^n$, $a \in \mathbb{R}^m$ and the reward

$$R(s,a) = -\left(s^T Q^s s + a^T Q^a a\right)$$

where both Q^s and Q^a positive definite matrices, the problem becomes a **Linear Quadratic Regulator** (LQR) problem. Details of LQR can be found elsewhere in optimal control related notebooks and is are not discussed here.

4.5 Multi-attribute Utility Function

To this point, we have been assuming that all the attributes that affect utility can converted and put on the same scale. When maximizing the utility, only that single figure is considered. However, this is not always true in the reality. For example, consider personal protective equipment design. In practice, there is always a chance that the protection fails. It will take infinite money to make the equipment infinitely safe. To maximize the utility, we need to put financial cost and human life on the same scale, which is difficult and can be immoral.

A problem whose outcomes are characterized by two or more attributes that cannot be easily converted into a single figure are handled by **multi-attribute utility theory**. There are several ways to handle the situation, and they are briefly introduced as follows.

In the heat map approach, the actions are converted into coordinates in a hyper space where each dimension representing an attribute. A heat map is constricted, representing the safe and dangerous zones in the hyper space. Actions inside the safe zone are selected.

In the statistic dominance approach, we investigate the chance that an action may perform better than any other actions in every aspects. The action that statistically more likely to dominant (perform better than other actions in all attributes) is selected.

Sometimes the attributes follow joint distribution. We can set a lower bound for one of the attributes, hence reducing the problem to a single-attribute problem. That attribute is used to formulate the utility function. Consider the earlier personal protective equipment example. The more expensive the gear, the more likely it is safe. The safety and the cost form a positively correlated joint distribution, where each sample in the distribution is an ac-

tion. We can set a lower bound for safety requirements, for example, safety probability of 99.999%, and reduce the problem to a single attribute problem where we find the action with the minimum financial cost that guarantees the safety probability.

4.6 Game Theory

Multi-agent control system.

Part II

Artificial Neural Networks

5

Regression

CONTENTS

Regression is one of the most fundamental and important applications of AI. This chapter introduces regression and corresponding artificial neural networks structures.

5.1 Linear Regression

Linear regression is one of the most basic problems of AI. Though simple, it plays a significant part in the history of AI. The methodologies used in linear regression, such as gradient descent, inspire the further development of more sophisticated artificial neural networks structures.

5.1.1 Problem Statement

Let

$$y \quad = \quad f(x, \epsilon)$$

where $y \in \mathbb{R}$ the output, $x \in \mathbb{R}^n$ the inputs in vector form (also known as the features), n the number of features, ϵ the stochasticity of the model, and $f(\cdot)$ an unknown function of x. The purpose of linear regression is to find such $\theta \in \mathbb{R}^n$ and $\theta_0 \in \mathbb{R}$ for

$$h(x) \quad = \quad \theta^T x + \theta_0 \tag{5.1}$$

such that $h(x)$ is an approximation of y with minimum $\|y - h(x)\|$.

Linear Regression Equation with Augmented State

Technically speaking, (5.1) is not a linear function of x due to the bias term θ_0. One can make it a linear function using augmented state vector as follows.

Let

$$\begin{aligned} \bar{x} &= \begin{bmatrix} x_0 & x^T \end{bmatrix}^T \\ \bar{\theta} &= \begin{bmatrix} \theta_0 & \theta^T \end{bmatrix}^T \end{aligned}$$

with $x_0 = 1$ a constant augmented parameter. Equation (5.1) then becomes

$$h(x) = \bar{\theta}^T \bar{x} \tag{5.2}$$

which is a linear function of $\bar{x}$.

If often does not batter which notation to use, either (5.1) or (5.2), so long as it is used consistently.

5.1.2 Solution with Gradient Descent

As a first step, collect training samples. Let there be m training samples denoted by $(x^{(i)}, y^{(i)})$, $i = 1, \ldots, m$.

With the m samples, θ and θ_0 can be determined as follows.

$$\theta, \theta_0 = \arg\min_{\theta, \theta_0} J \tag{5.3}$$

$$\begin{aligned} J &= \frac{1}{2} \sum_{i=1}^{m} \left(h(x^i) - y^i \right)^2 \\ &= \frac{1}{2} \sum_{i=1}^{m} \left(\theta^T x^i + \theta_0 - y^i \right)^2 \end{aligned} \tag{5.4}$$

with J the quadratic cost function.

There are different ways to solve (5.3). For example, since the analytical form of J in (5.4) is known, we can solve $\frac{\partial J}{\partial \theta} = 0$ and $\frac{\partial J}{\partial \theta_0} = 0$ to get θ and θ_0. The result is given by the **normal equation** below.

$$\begin{aligned} X^T X \theta &= X^T Y \\ X_{m \times n} &= \begin{bmatrix} x_1^1 & \ldots & x_n^1 \\ \vdots & \ddots & \vdots \\ x_1^m & \ldots & x_n^m \end{bmatrix} \\ Y_{m \times 1} &= \begin{bmatrix} y^1 \\ \vdots \\ y^m \end{bmatrix} \end{aligned}$$

solving which gives

$$\theta = \left(X^T X\right)^{-1} X^T Y \tag{5.5}$$

which is known as the **least squares estimation** of θ.

For the scope of this chapter, consider using gradient decent as follows. Gradient decent may seem overkill for linear regression problem, but it is so widely used in machine learning and it is worth introducing here.

In **gradient descent**, initialize θ and θ_0 randomly or to be zero, and use them as a starting point. Iteratively do the following. Calculate the gradient of J at (θ, θ_0) and revised θ and θ_0 in the inverse direction of the gradient to get a smaller value of J as follows.

$$\begin{aligned} \theta &\leftarrow \theta - \alpha \frac{\partial}{\partial \theta} J \\ \theta_0 &\leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J \end{aligned}$$

where α is the **learning rate** which is often a small value and can by dynamic as the number of iterations increases. Iterate until θ and θ_0 converge, or until a maximum iteration number is reached, or until J does not decrease further.

Notice that in each iteration, all the training samples are used to calculate $\frac{\partial}{\partial \theta} J$ and $\frac{\partial}{\partial \theta_0} J$ in the gradient descent. This is known as **batch gradient descent**. When the total number of samples m is extremely large, batch gradient descent requires large computer memory. To save computational resources in the case the training set is very large, consider using **mini-batch gradient descent**, where the m training samples are divided into smaller mini batches. In each iteration, one of the mini-batch is used. It will take several iterations before the entire training samples are gone through. In the special case where there is only one sample in each mini-batch, the method is known as the **stochastic gradient descent**.

In general, batch gradient descent consumes more computational resources in each iteration than mini-batch or stochastic gradient descent and hence it is often very slow. However, it is more robust and helps with more stable parameter converges. When using mini-batch and stochastic gradient descent, on the other hand, the parameters may never truly asymptotically converge. Instead, they may circulate in a very small zone. Nowadays for many applications the training set is often very large, making it impractical to use batch gradient descent. Mini-batch and stochastic gradient descent are more often used. Though there is the risk that they may not asymptotically converge to the minimum, the result is often acceptable.

Each time the entire training set is scanned, it is called an **epoch**. In batch gradient descent, each iteration corresponds with an epoch, whereas in mini-batch gradient descent, multiple iterations correspond with an epoch.

It is worth mentioning that the quadratic cost function J given by (5.4) is

a convex function of θ and θ_0 (proof is neglected in this notebook), and there is no local minimum. With a small learning rate α, global minimum of J can be achieved.

5.2 Locally Weighted Regression

Linear regression introduced in Section 5.1 is an example of **parametric learning algorithm**, which has a fixed set of parameters θ, θ_0 in (5.1) and the purpose of the machine learning is to fit them with most suitable values. In parametric learning algorithm, since the number of parameters is fixed regardless of the size of the training set, the memory consumption to trace and update the parameters is more or less limited.

Locally weighted regression, on the other hand, is a **non-parametric learning algorithm**. The amount of parameters to be traced grows during the training process.

5.3 Logistic Regression

Logistic regression, though has "regression" in its name, is a basic and widely used classification method. Details are introduced below.

5.3.1 Problem Statement

Let

$$y = f(x, \epsilon)$$

where $y \in \{0, 1\}$ the label, $x \in \mathbb{R}^n$ the features, n the number of features, ϵ the stochasticity of the model, and $f(\cdot)$ an unknown function of x. Notice that the label categories the samples into two categories, 0 (negative) and 1 (positive).

The purpose is to find a specific non-linear mapping that can categorize a new sample with into either the negative set or the positive set.

5.3.2 Solution with Newton's Method

TBA

6

Perceptron

CONTENTS

7

Multi-layer Perceptron

CONTENTS

Part III

Convolution and Recurrent Networks

8

Convolutional Neural Network

CONTENTS

"nobreak

8.1 Brief Review of CNN

CNN is a type of ANN structure designed to handle grid-like data. It is effective when dealing with spatially correlated data, thus it has become 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. A demonstration of CNN kernel is given in Fig. 8.1. 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 specific features in the feature map. For example, there might be a kernel detecting edges, while a second kernel detects color patterns.

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.

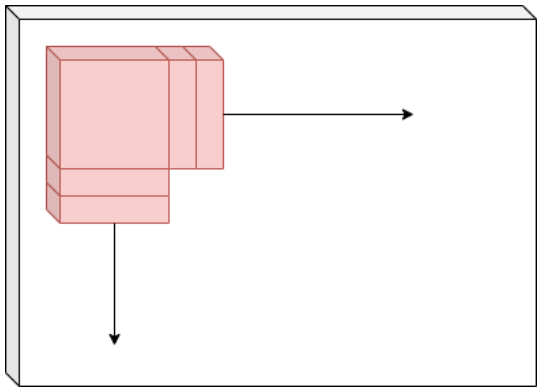

FIGURE 8.1
A demonstration of CNN kernel. The input is given by the white box (3D), and the kernel by the red box.

9

Recurrent Neural Network

CONTENTS

"nobreak

9.1 Brief Review of RNN

RNN is a connectionist model with the ability to selectively pass information across sequence steps [16]. 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. 9.1. 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

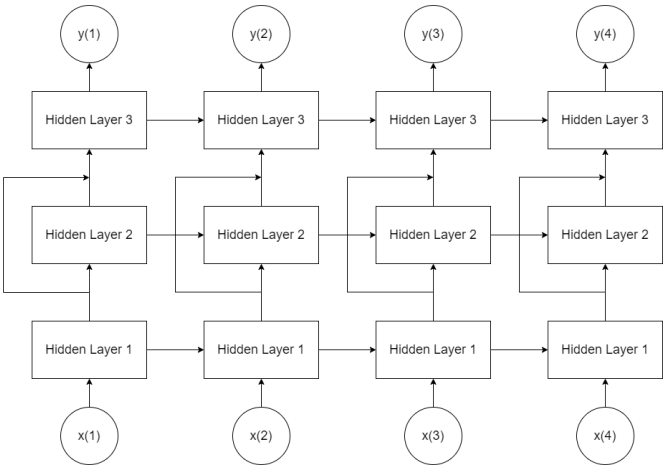

FIGURE 9.1
An example of RNN.

"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) [16]. 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 [24] where an encoder-decoder structure is used, each is an 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 recognition.

Part IV

Large Language Model

10

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" [26]. Transformer is introduced in this chapter.

A brief review of RNN, the state-of-the-art framework prior 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.

10.1 RNN and Its Limitations

RNN has been introduced in earlier Chapter 9. A brief review is given in this section, focusing on its shortages in NLP.

10.1.1 A Brief Review of RNN

Compared with a dense ANN, RNN is more efficient in building up correlation between consecutive inputs, and hence it becomes a powerful framework when it 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 spatial and sequential data is first abstracted using some mechanism, and the correlation information is sent as (additional) inputs to the following dense ANN.

10.1.2 Limitations of RNN

"Attention" was not a new concept invented in [26]. 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)

10.2 Transformer Framework

A summary of the concepts, components and technologies in [26] is given in this section.

10.2.1 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. 10.1, 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

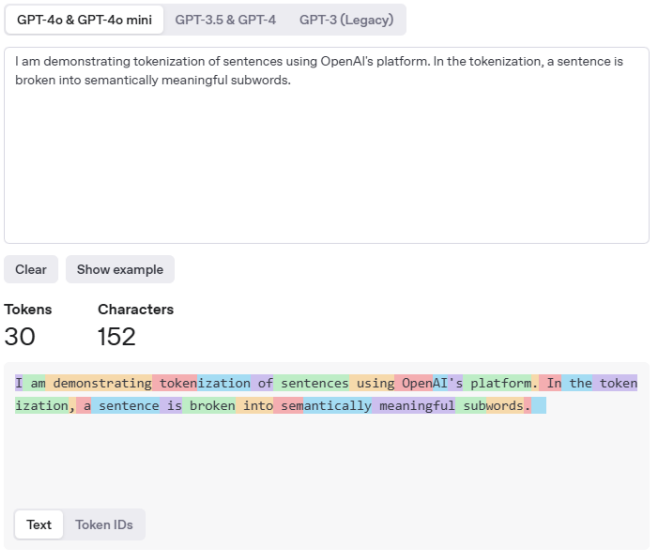

FIGURE 10.1
A demonstrative example where a piece of text is tokenized using GPT-4o's tokenizer.

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. 10.1, 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. 10.1 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.

10.2.2 Encoder and Decoder

TBA

10.2.3 Attention Mechanism

TBA

10.2.4 Transformer-Based NLP

"nobreak

10.3 Transformer Development Trend

“nobreak

10.3.1 Transformer Variants

“nobreak

10.3.2 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

11

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 commercialized models to develop their 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 12.

Modern LLMs are built on top of Transformer framework which has been introduced in earlier Chapter 10. 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.

11.1 Introduction to LLM

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

11.1.1 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 be likely to put "water" or "tea" in the blank. This is obvious because humans have a dictionary of words that they can choose from in their mind, and they have been reinforced by learning the "a bottle of water" expression on many occasions. In addition, it makes sense to a human that when someone is thirsty, they 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 the 1990s.

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 pairing 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 9.

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.

11.1.2 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 it 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 to accurately and correctly complete a task, is affected 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.

11.1.3 LLM Performing Benchmarks

"nobreak

11.1.4 LLM Development Timeline

A brief timeline of the development of LLMs since the publication of [26] is given below.

2017

"Attention Is All You Need" [26] 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.

11.1.5 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, has its 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.

Many libraries have been released to the public to help with building, training and fine-tuning LLMs, such as `transformers`, a Python library for building transformer models. Many such libraries are integrating 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.

11.2 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 12.4.1.

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. 11.1.

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 11.1
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.

11.2.1 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. 11.2 (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 11.2
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 11.2 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.

11.2.2 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 11.3. The picture is from [27]. 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

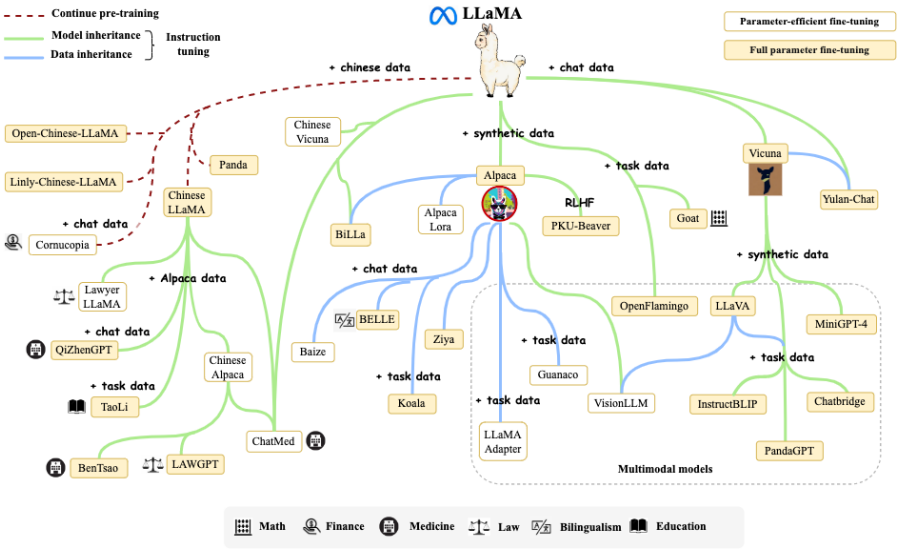

FIGURE 11.3
LLaMA family tree.

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 11.1. The training dataset is pure-

TABLE 11.1
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.

ly open-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 [26] 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[12] and QLoRA[10], 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 [25]. A brief highlight is given below. The fine-tuning pipeline of Alpaca is shown in Fig. 11.4. 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."
```

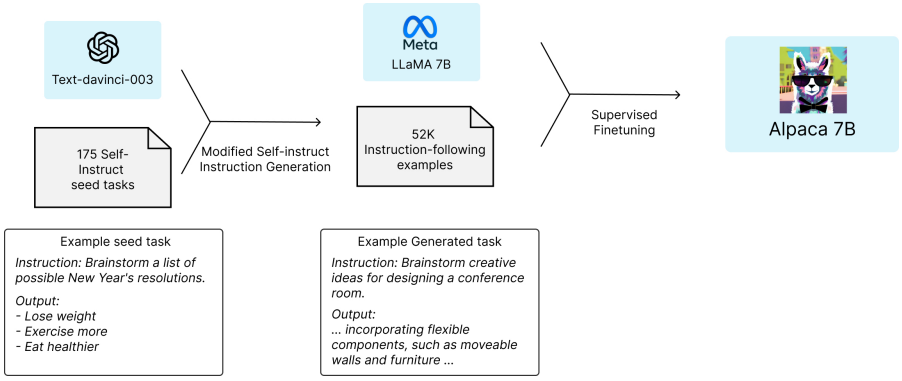

FIGURE 11.4
Alpaca fine-tuning pipeline.

```
},
{
        "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 L-LaMA 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 [18] and [17]. 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. 11.2 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 [28]. It uses different techniques and realizations from LLaVA. Pass the same Fig. 11.2 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.

11.2.3 Other LLMs

"nobreak

11.3 LLM Development

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

11.3.1 Architecture Design

Encoder and Decoder

Normalization Method

Activation Function

Position Embedding Method

Attention Mechanism

11.3.2 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.

11.3.3 Pre-training

“nobreak

11.3.4 Fine-Tuning

“nobreak

11.3.5 Model Evaluation

“nobreak

11.4 Multimodal LLM

TBA

12

Large Language Model: Practice

CONTENTS

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

12.1 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 languages 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 [11].

```
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 `transformers`, `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, corporations as well as 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 lecturers would more often use `conda`, while software engineers use `pip`/`venv`.

12.2 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.

12.2.1 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 ways of doing that is to leverage **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. 12.1 as an example that runs LLaMA 3.2 with Ollama on a PC using the above command.

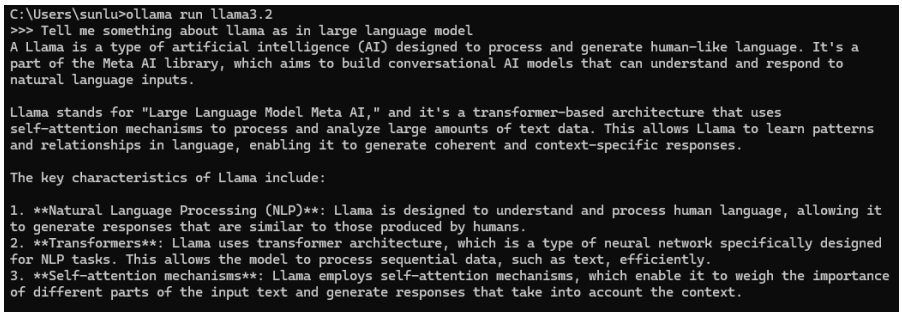

FIGURE 12.1
An example of running Ollama with LLaMA 3.2.

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 [21]. 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.

12.2.2 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)
```

12.2.3 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.

12.3 LLM Cloud Deployment

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

12.3.1 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.

12.3.2 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.

12.3.3 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 12.1
OpenAI's API Calls Pricing as of this writing per $1M$ tokens, in USD.

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, the cost of OpenAI's API calls are given in Table 12.1 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.

12.3.4 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.

12.4 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.

12.4.1 A Review of Frontier Models

Models to be reviewed in this section are summarized in Table 12.2. 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 12.2
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.

12.4.2 Proprietary Model Fine-Tuning

“nobreak

12.4.3 Open-Source Model Fine-Tuning

“nobreak

12.5 Resource Augmented Generator

13

Agentic AI

CONTENTS

Agentic AI describes a comprehensive AI system that can make decisions and perform actions for either general or specific tasks with minimum human intervention. It is an important and fundamental building block of AI-driven 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. For this reason, agentic AI is included as a key topic in this part of the notebook.

13.1 Introduction to Agentic AI

The term "agent" refers to an entity that functions like a human. By this definition, **agentic AI** refers to AI-based systems that can solve complex

problems with minimum human oversight. It is possible to build an agentic AI system without LLM and use only conventional AI frameworks such as MDP and ANN. While being good at specific tasks, the use of such agentic AI systems often lacks of general-purpose interfaces. With the recent rise of LLMs, agentic AI has been re-contextualized and is gaining increasing popularity. LLMs provide a general-purpose language interface that makes agentic AI significantly more scalable and cost-effective.

An LLM-incorporated agentic AI system with LLM agents is often more capable than a conventional agentic AI system in the following aspects.

- An LLM-incorporated agentic AI system can understand human instructions in natural languages.
- An LLM agent can take in unstructured data input and generate structured data output in a scalable manner.
- An LLM agent can make decisions based on its built-in domain knowledge.

LLM-incorporated agentic AI system refers to an autonomous system in which LLM agents 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 solution should be approached. An LLM in an agentic AI system may take on one or more of the following roles.

- Orchestration:
  - Decompose the original problem into smaller sub-problems.
  - Assign these sub-problems to LLM instances, either sequentially or in parallel.
  - Decide the path of the pipeline based on the outputs of each AI agent.
- Sub-tasks solving:
  - Receive and comprehend unstructured inputs.
  - Use "tools" to connect to other resources such as databases, computational units and actuators.
  - Generate structured outputs.
  - Integrate the outputs from sub-problems into a final, coherent solution to the original problem.
- Output guardrails:
  - Evaluate the outputs of LLMs and either accept the results or reject them with feedback and revision instructions.

In contrast, a conventional LLM-integrated application without agentic AI relies on humans or pre-written deterministic code to perform all of the above tasks. From this perspective, agentic AI expands the use of LLM agents and further reduces human intervention and increases the level of automation. A comparison is illustrated in Fig. 13.1. In the conventional architecture shown in (a), the workflow is either hard-coded or defined externally, and the LLM merely serves as a sub-task solver. In the agentic AI architecture shown in (b), the LLM autonomously orchestrates the task, determines the workflow, and governs the solution process.

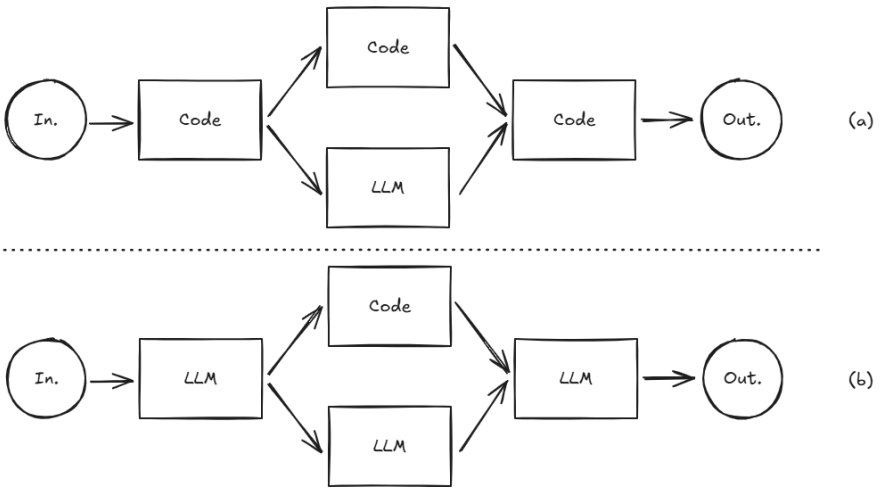

FIGURE 13.1
Conventional architecture (a) versus agentic AI architecture (b).

Another commonly used architecture is shown in Fig. 13.2. 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. This self-evaluative architecture significantly reduces the likelihood of low-quality or hallucinated outputs.

LLM agents in an agentic AI system do not rely solely on the internal knowledge encoded in the model. Instead, they can be configured to interact with external tools such as databases, web browsers, calculators, etc., to enhance their 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. 13.3, where the LLM autonomously decides when to invoke an external tool and generates

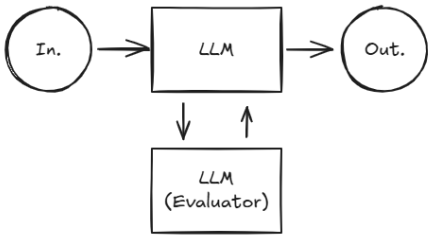

FIGURE 13.2
Self-evaluative agentic AI architecture.

the appropriate commands using pre-defined protocols known as the Model Context Protocol (MCP). MCP is introduced in Section 13.3.

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. Other than databases, agentic AI systems often adopt memory modules which store historical inputs and outputs to maintain context and support long-term reasoning. This is also shown in Fig. 13.3.

13.2 Agentic AI Framework

An **agentic AI framework** refers to the infrastructure that supports the creation, configuration, and execution of an agentic AI system. 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.

On one hand, agentic AI frameworks simplify the deployment of LLM applications, but on the other hand they add additional layers of abstraction and prompts, thus making the system behavior less controllable and sometimes more difficult to debug. The user should choose whether to use agentic AI frameworks wisely based on the need.

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

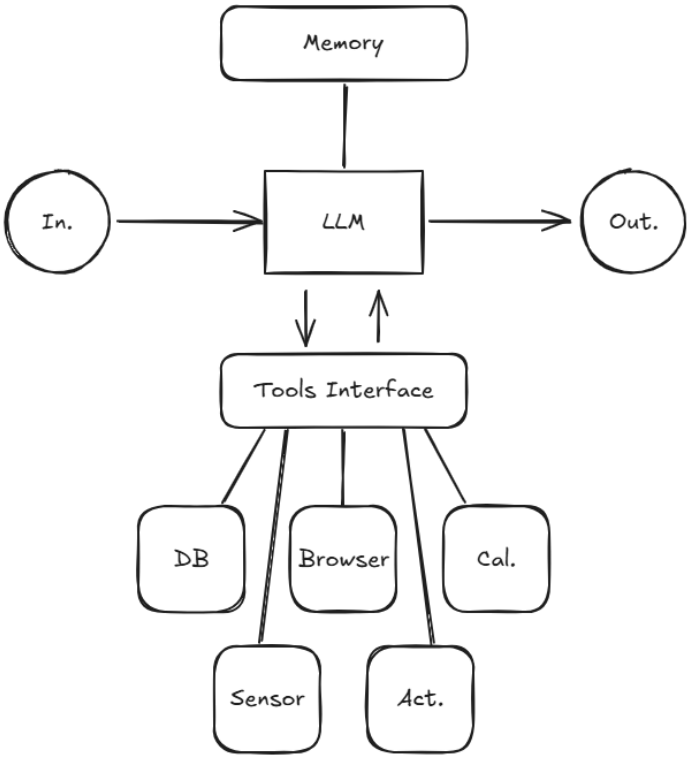

FIGURE 13.3
LLM with memory and tool interfaces to databases, web browsers, calculators, sensors and actuators, and other components that can interact with the environment.

13.2.1 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 tasks (among a group of 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 multi-threading. By itself, `asyncio` runs on a single thread and a single CPU core. While async Python programming does not help with

reducing the total CPU time consumption, it saves overlapped waiting time of the tasks. 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.

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 [23]. 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. Calling a coroutine function directly does not run it immediately. Instead, it returns a coroutine object. The syntax is given below.

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

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 executed only once. 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 event loop

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 existing 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. An example is given below. Assume that there is already an event loop. The following syntax

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

will execute them in sequential manner, not the parallel-like manner. This is because `await` behaves like "there is a coroutine function running and let us pause here until it finishes." If `cx` is delayed due to I/O, `cy` must wait until `cx` finishes. There are at least two ways to let the tasks run in the parallel-like manner. Consider

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

This way, `cx` and `cy` are scheduled concurrently. While `cx` is waiting for inputs, `cy` can still execute.

Alternatively, we can also "register" the coroutine functions as background tasks.

```
x_task = asyncio.create_task(cx())
y_task = asyncio.create_task(cy())
x = await x_task
y = await y_task
```

Tasks are treated differently than coroutine functions. In the above example, when `x_task` is await, the event loop will look for other pending tasks (not coroutine functions) and as a result, `y_task` will be executed. This is what `await` does in general. It pauses the script for the coroutine function or task, and while waiting, it checks other registered tasks and executes those that can run. 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.

The main program can be wrapped as follows so that .

```
import asyncio

async def main():
       <the code to be introduced>

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

13.2.2 OpenAI Agents SDK

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.

Comparing with manually deployment, the OpenAI Agents SDK offers at least the following additional features.

- Simplified deployment of a single AI agent or tool

  Without an agentic AI framework, the user must create a detailed "user manual" in JSON object format and pass it to the LLM. 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. OpenAI Agents SDK will try to interpret the functionality and the input and output formats of the tool without additional help.

- Easy conversion of AI agents into tools

  An AI agent can itself be converted into a tool with a single-line command. 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 13.4 illustrates the key features of OpenAI Agents SDK compared with a manual implementation. In the OpenAI Agents 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.

The basic syntax of creating an OpenAI Agents SDK framework based agentic AI system is introduced in the rest of this section. All code examples are assumed to be executed in an environment with a running event loop. Note that `OPENAI_API_KEY` is also assumed to be loaded as an environment variable.

Agent

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>"
)
```

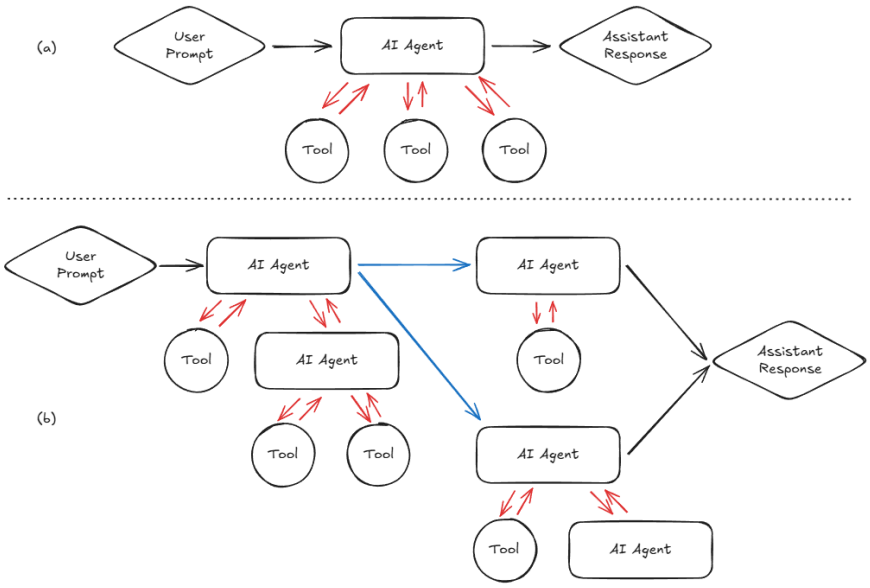

FIGURE 13.4
Manually implemented (a) versus OpenAI Agents SDK–based (b) agentic AI pipelines. Red arrows represent tool calls, and blue arrows represent handoffs.

```
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 (as well as the entire context to generate the content) 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 it decides to do so based on its instructions and reasoning. 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.

In the OpenAI Agents SDK framework, 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 13.4.2.

13.2.3 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 13.2.2 and Fig. 13.4, 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 a "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. This is demonstrated by Fig. 13.5. Notice that a "manager" agent can be assigned to a crew to determine the pipeline flow.

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 and feel more under control, but when a job aligns well with CrewAI's philosophy, implementing it in CrewAI is often much easier. Another interpretation is that CrewAI can be used to build up complicated applications in an easier manner, but it may lack precision.

CrewAI Installation as a UV Tool

CrewAI provides both Python libraries and CLI tools. It is recommended to install both when working with CrewAI. The CLI tool simplifies project setup by generating a directory tree with template Python scripts that can be adapted for specific applications. Manually creating such a structure from scratch would be tedious.

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

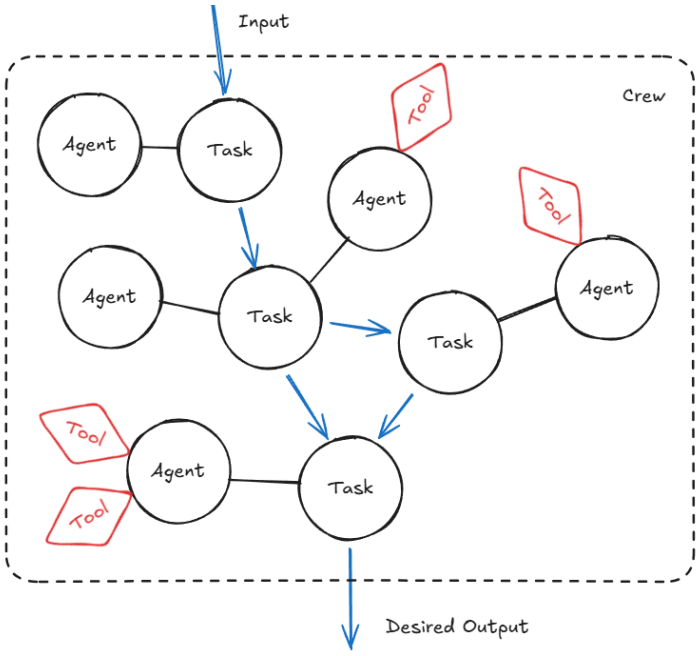

FIGURE 13.5
CrewAI framework pipeline, where the user defines agents, tasks and their associations and the crew get things done automatically.

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

Once CrewAI is installed, a new project can be created with:

```
crewai create crew <project_name>
```

This command generates a project root folder containing a hello-world-style template. During creation, the user is prompted to select a default LLM service provider and model, and to provide an API key. These choices are used in generating the template but can be modified later.

An example of the template directory structure is shown below [6]:

```
my_project/
|-- .gitignore
|-- knowledge/
|-- pyproject.toml
|-- README.md
|-- tests/
|-- .env
\-- src/
  \-- my_project/
```

```
   |-- __init__.py
   |-- main.py
   |-- crew.py
   |-- tools/
   |  |-- custom_tool.py
   |  \-- __init__.py
   \-- config/
     |-- agents.yaml
     \-- tasks.yaml
```

Agents and Tasks Definition

In the project template, `agents.yaml` and `tasks.yaml` are configuration files that specify the properties of agents and tasks. An agent refers to an LLM with defined capabilities and tools, while a task represents a workflow step with input, desired output, and context. These configuration files may include prompts, parameters, and other metadata that shape the agent's behavior and the execution of tasks.

For example, in a document-refinement workflow, the agent might be a "publication editor," and the task could be "correct-grammar-error." The input would be text containing grammatical mistakes, and the output would be error-free text that preserves the original meaning with minimal changes.

Below is an example of `agents.yaml` created during project initialization:

```
researcher:
  role: >
    {topic} Senior Data Researcher
  goal: >
    Uncover cutting-edge developments in {topic}
  backstory: >
    You're a seasoned researcher with a knack for uncovering the latest
         developments in {topic}. Known for your ability to find the
        most relevant information and present it in a clear and concise
         manner.

reporting_analyst:
  role: >
    {topic} Reporting Analyst
  goal: >
    Create detailed reports based on {topic} data analysis and research
         findings
  backstory: >
    You're a meticulous analyst with a keen eye for detail. You're
        known for your ability to turn complex data into clear and
        concise reports, making it easy for others to understand and
        act on the information you provide.
```

Here, two agents, "researcher" and "reporting analyst," are defined. Each agent has three key components: `role`, `goal`, and `backstory`. Together, these

specify the agent's intended capabilities and behavior. The symbol `>` in YAML indicates a folded block scalar, meaning that multi-line text is treated as a single string with line breaks converted to spaces. The placeholder `{topic}` in the example will be filled dynamically by the higher-level script.

In addition to the compulsory `role`, `goal`, and `backstory` fields, other optional fields may be included. A commonly used one is `llm`, which lets the user specify an LLM model other than the default. For example:

```
<agent_name>:
  role: <something>
  goal: <something>
  backstory: <something>
  llm: openai/gpt-5-mini
```

A full list of allowed fields can be found at [7], including tools, delegation, maximum iteration count, rate limits, and others.

Below is an example of `tasks.yaml` created during project initialization:

```
research_task:
  description: >
    Conduct thorough research about {topic}. Make sure you find any
        interesting and relevant information given the current year is
        {current_year}.
  expected_output: >
    A list with 10 bullet points of the most relevant information about
         {topic}
  agent: researcher

reporting_task:
  description: >
    Review the context you got and expand each topic into a full
        section for a report. Make sure the report is detailed and
        contains all relevant information.
  expected_output: >
    A fully fledged report with the main topics, each with a full
        section of information. Formatted as markdown without ’‘‘‘’
  agent: reporting_analyst
```

Here, two tasks are defined, each assigned to an agent. The user provides a description and expected output for each task. In addition to `description` and `expected_output`, other optional fields can also be specified. A full list can be found at [9], including settings that limit the tools available for a task, enforce output formats, and define guardrails.

Crew Definition

A crew is an integration of agents and tasks defined in the YAML files. The configuration files define the "class" of different types of agents and tasks, while the crew Python script reads these configurations and instantiates them. An example is given below. Notice that different decorators are used.

```
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task
from crewai.agents.agent_builder.base_agent import BaseAgent
from typing import List

@CrewBase
class <crew class name>():
"""<description of the crew>"""

  agents: List[BaseAgent]
  tasks: List[Task]

  @agent
  def researcher(self) -> Agent:
    return Agent(
      config=self.agents_config['researcher'],
      verbose=True
    )

  @agent
  def reporting_analyst(self) -> Agent:
    return Agent(
      config=self.agents_config['reporting_analyst'],
      verbose=True
    )

  @task
  def research_task(self) -> Task:
    return Task(
      config=self.tasks_config['research_task'],
    )

  @task
  def reporting_task(self) -> Task:
    return Task(
      config=self.tasks_config['reporting_task'],
      output_file='report.md'
    )

  @crew
  def crew(self) -> Crew:
  """<description>"""
    return Crew(
      agents=self.agents,
      tasks=self.tasks,
      process=Process.sequential,
      verbose=True,
    )
```

When using the `agent` and `task` decorators, the defined components are

automatically added to the `self.agents` and `self.tasks` lists in their defined order. Notice that in this example `Process.sequential` is used. In this case, the sequence of tasks in `self.tasks` matters, as this will be the task pipeline of the crew. In this sense, the user defines the pipeline.

It is possible to assign a **manager agent** to supervise the crew and ensure the quality of generations. In this setup, the manager oversees the behavior of each agent and task, and can determine the crew's execution flow. A manager agent can be assigned in the `@crew` definition as follows:

```
@crew
def crew(self) -> Crew:
"""<description>"""

  manager = Agent(
    config=self.agents_config['manager'],
    allow_delegation=True
  )

  return Crew(
    agents=self.agents,
    tasks=self.tasks,
    process=Process.hierarchical,
    verbose=True,
    manager_agent=manager,
  )
```

Here, a manager is defined, assuming its role description is provided in the `agents` configuration file. It is then assigned to the crew via `manager_agent`. The process is changed from `Process.sequential` to `Process.hierarchical`, allowing the manager to supervise the crew above other agents. The manager can dynamically re-order, route, or request follow-ups across tasks and agents. In hierarchical mode, the pipeline is determined not by the sequence of task definitions but by the manager agent.

It is also possible to assign an LLM model directly as the manager without creating an agent instance or defining it in the `agents` configuration file. However, it has been reported that a properly defined manager agent may outperform a bare model assignment.

It is possible to enforce structured output for a task. This can be done in a manner similar to the OpenAI Agents SDK. A Pydantic model inheriting from `BaseModel` is defined with the desired output structure. An example is shown below:

```
class <ModelName>(BaseModel):
    """ <description> """
    <field>: <type> = Field(description="<description>")
    <field>: <type> = Field(description="<description>")
    <field>: <type> = Field(description="<description>")
```

When defining a task, the model is specified via the `output_pydantic` argument:

```
@task
def <task_name>(self) -> Task:
    return Task(
        config=self.tasks_config['<task name in configuration>'],
        output_pydantic=<ModelName>,
    )
```

Notice that Pydantic models can be nested. For example, one can define a model with a field that is a list of another Pydantic model.

It is not required but can become helpful sometimes to also remind CrewAI of the required structured output in the configuration files for tasks and agents.

Execution

Finally, the crew can be executed as follows:

```
def run():
"""
Run the crew.
"""
  inputs = {
        'topic': 'AI LLMs',
        'current_year': str(datetime.now().year)
  }

  try:
    DocumentRefiner().crew().kickoff(inputs=inputs)
  except Exception as e:
    raise Exception(f"An error occurred while running the crew: {e}")

if __name__ == "__main__":
        run()
```

The above summarizes the basic usage of the CrewAI framework. More advanced materials are introduced in the following sections.

Tool

Tools are one of the most important features of an agentic AI system, as they significantly extend its capabilities. Recall that the OpenAI Agents SDK provides several ways for a user to define tools, including at least the following:

- Using tools from a library, for example, `WebSearchTool` provided by OpenAI, which allows an LLM to query information from the web.
- Defining a custom function tool with the `@function_tool` decorator.
- Supplying a JSON-formatted descriptive tool.

- Using an agent as a tool.

CrewAI likewise provides multiple methods for calling tools from libraries or defining customized tools. More details are given below.

User defined tools need to be saved under the folder `tools` as a function. An example is given below.

```
@tool("<tool name>")
def <tool function name>(<input>) -> str:
    <do something>
    ...
    try:
        <do something>
        return "Tool executed successfully"
    except Exception as e:
        return f"An error occurred: {e}"
```

In the `crew.py` file, import tools as follows.

```
from crewai_tools import <tool function name> # official tool
from <project name>.tools.<tool file name> import <tool function name>
    # user-defined tool
```

When defining an agent, assign tools to it as follows.

```
@agent
def <agent name>(self) -> Agent:
    return Agent(
        config=self.agents_config['<agent name>'],
        tools=[<tool function name>, <tool function name>],
        verbose=True,
        allow_delegation=True
    )
```

Do not forget to introduce the tool to the agent in `agents.yaml`. When introducing the tool, use `<tool name>` in the decorator.

Memory

LLMs are inherently stateless. In an LLM-based chatbot application, the conversation between the user and the model is typically stored and then reused as context whenever the LLM is triggered. Conventionally, the conversation is saved in a JSON file, for example `history.json`, which may look like:

```
[
       {"role": "user", "message": "<something>"},
       {"role": "assistant", "message": "<something>"},
       ...
]
```

When the conversation history grows long, feeding the entire record to the LLM for every call is problematic for at least the following reasons.

- Much of the historical dialogue may be irrelevant to the current query, making it wasteful to process.
- The total token count of the history may exceed the LLM's input limit.

This naturally motivates several strategies as follows.

- Query only for relevant past information before sending it as context to the LLM, rather than including the full history.
- Periodically prune older conversations, as they tend to become less relevant over time.
- Selectively preserve information likely to be useful in the future by tagging it so that it is not removed.

These ideas form the foundation of memory management in CrewAI.

As an agentic AI framework, CrewAI involves multiple LLM agents. Memory is managed at the agent level, the task level and the crew level, and memory sharing across agents can be enabled if desired. CrewAI defines several memory types, such as short-term memory, long-term memory, and entity memory, and each can be activated independently. When a memory type is enabled, CrewAI automatically promotes selected conversations into that memory category. Unlike conventional conversation logs stored solely in JSON format, CrewAI combines JSON (for raw text), embedded vectors (for semantic retrieval), and SQLite (for efficient tagging and metadata queries) to store memory. The storage medium depends on the memory type.

The commonly used memory types and their features are introduced below.

Short-term memory. Short-term memory stores the recent inputs and outputs of an AI agent using embedded vectors. It relies on ChromaDB [5], an open-source, file-based embedding database.

Long-term memory. Long-term memory stores important task results that can be reused across sessions. Both ChromaDB and SQLite are employed for this purpose.

Entity memory. Entity memory tracks entities such as names, places, and concepts. It also uses ChromaDB for storage.

The above three memory types are collectively referred to as the basic memory system, and they can be enabled by:

```
crew = Crew(
    agents=[...],
    tasks=[...],
    process=Process.sequential,
    memory=True, # Enables short-term, long-term, and entity memory
    verbose=True
)
```

The user can choose the storage location and the vector embedding tool. If not specified, a default location inside the project folder is used, and the

OpenAI embedding API is applied. The behavior of the embedding tool can also be configured. More details are given at [8].

Once memory is properly enabled, the overall performance of the agentic system is improved. In general, it becomes more efficient, potentially cheaper—since semantic search is used instead of retrieving the entire chat history as context—and it remembers important concepts more accurately and consistently. Applications benefit from these improvements naturally, without needing to manage the internal details of the memory system.

Coder Agent

On a machine where containerization tools are installed, an agent can trigger the execution of a code that it generates, and it refines the code based on the error message or output or several times. This improves the stability and performance of a coder agent.

The user is able to setup maximum trail number and maximum waiting time for the trail code execution.

13.2.4 LangGraph

It is worth mentioning the differences between LangChain and LangGraph. They are two distinct products from the same company, LangChain. A brief explanation is given below.

LangChain is an open-source, composable framework that provides a standard interface for LLM models and APIs to databases and a variety of tools [13]. Suppose that an agentic AI system with multiple components has been defined. If LangChain is used to connect and integrate these components, the user can switch underlying models (for example, from OpenAI to Anthropic) and tools (for example, from MySQL to PostgreSQL) with relatively little code change. This greatly enhances the portability and robustness of the agentic AI solution.

From that sense, LangChain is to some extend similar with MCP, but in the LangChain ecosystem. Nowadays, some agentic AI frameworks developed by other entities, such as AutoGen, also support connectivity with LangChain.

LangGraph, on the other hand, is an agentic AI framework, broadly comparable to the OpenAI Agents SDK and CrewAI. It focuses on coordinating AI agents and reliably handling complex tasks by representing applications as stateful graphs [14]. It is sufficient to use only LangChain to build an LLM application with deterministic pipeline. LangGraph, on the other hand, can be taken as an abstraction layer built on top of LangChain that provides orchestration intelligence, enriching the system with agentic AI capability.

This section focuses on LangGraph, the agentic AI framework. Technologies offered by LangChain will also be introduced and used. After all, LangGraph uses LangChain under the hood.

Install LangGraph as a Python package as follows.

```
uv add langgraph
```

or

```
uv pip install -U langgraph
```

LangGraph introduces the concepts of states, nodes and edges to visualize the workflow as a graph. A **LangGraph state** represents a snapshot of the application. The state is immutable, meaning that each update of any kind creates a new state derived from the old one. In this way, LangGraph maintains a full history of the system status and supports time-travel debugging and replay. A **LangGraph node** typically represents an atomic unit of computation, such as a function, tool, or an agent call. It receives the current state as input and produces a new state derived from it, with some data processed. Finally, a **LangGraph edge** represents a directed route that connects nodes, thereby defining the pipeline of the data flow.

An example is given in Fig. 13.6 to demonstrate the concept of nodes and edges. In the figure, apart from the start and end nodes, two nodes are defined, namely the LLM chatbot node and the tool node, likely for resource augmentation. The arrows are edges, where the solid arrows represent deterministic paths, and the dashed arrows, conditional paths. When the LLM chatbot node decides to use a tool call, tool node is executed. The result is returned to the LLM chatbot node. This process is iteratively carried out until the LLM node decides to proceed without tool calls. Along the edges, states are passed from one node to the other.

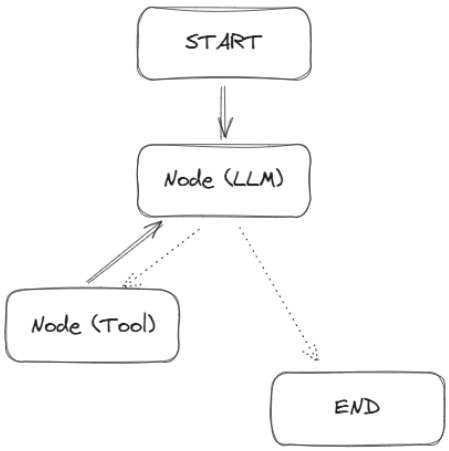

FIGURE 13.6
A simple demonstration of LangGraph with one LLM and one tool.

The following steps are typically required to define a LangGraph-based application.

- Define the state class, often in the form of a Python dataclass or Pydantic model.
- Create a state graph builder with the defined state class.
- Create nodes and add nodes to the graph builder.
- Create edges and add edges to the graph builder.
- Compile the graph builder to obtain the graph.
- Invoke the graph with the initial state.

Detailed introduction is given below.

State Definition

As explained earlier, the state class is essentially a data structure, such as a Pydantic model or a `TypedDict`. The following is an example from the official documentation.

```
from typing import Annotated
from typing_extensions import TypedDict
from operator import add

class State(TypedDict):
    foo: int
    bar: Annotated[list[str], add]
```

Here, `TypedDict` is used as the basis of the state class. Pydantic `BaseModel` is also supported, but it is generally less performant than `TypedDict` [15]. In the rest of this chapter, `TypedDict` will be used for LangGraph.

It is worth mentioning the **reducer function**, which can be defined in a state. As noted earlier, a node creates and outputs a new state based on the existing state it receives as input. By default, a node may overwrite fields in the new state with fresh values, discarding the previous contents of those fields. This does not affect the connectivity of the graph itself, but it may result in the loss of useful historical information.

A reducer function provides a mechanism to preserve such information. If a field in the state is associated with a reducer function, the node must update the field through that function. The reducer function specifies how new values should be merged with existing ones, ensuring that past information is retained. For example, a common reducer function appends new values to a list so that all historical entries for the field are preserved.

There are different ways to define a reducer, one of which is through Annotated. **Python Annotated** allows the programmer to attach metadata to a variable, often specifying its data type and usage. In the example above, `add` from the `operator` library is used as the annotation. This reducer function specifies how `bar` should be updated. In this case, new values should be concatenated to the list.

LangGraph provides commonly used state classes and reducer functions that can be imported and used directly, many of which in `langgraph.graph`.

Once the state is defined, a **state graph builder** can be declared as follows. All the nodes and edges what will be added to this graph later will use the defined state in the data pipeline.

```
class State(TypedDict):
    <defined earlier>

from langgraph.graph import StateGraph

builder = StateGraph(State)
```

Can I use any field name for any purpose?

For most field names, yes. The user has the flexibility to define any field names. In the custom nodes, the user can add any contents to these fields as part of the pipeline.

However, there are so called "system-managed" fields. Some system level functions will try to use particular field names for communication. An example is the `messages` field, which is used for AI agents with LLM to communicate with `ToolNode`.

For that reason, `messages` field is almost always defined in the state as follows.

```
messages: Annotated[list[str], add_message]
```

Node Definition

A node is a function that consumes the current state and produces a new state. For example, assume the state is defined as follows.

```
from typing import Annotated
from typing_extensions import TypedDict
from operator import add

class State(TypedDict):
    foo: int
    bar: Annotated[list[str], add]

from langgraph.graph import StateGraph
builder = StateGraph(State)
```

A node can then be defined as:

```
def update_state(state: State) -> State:
    return {
        "foo": state["foo"] + 1,
        "bar": ["new item"] # appears as an overwrite
    }
```

In this example, the node outputs a dictionary. The update to the `bar` field appears to be an overwrite. However, since a reducer is defined for `bar`, LangGraph intercepts the update and applies the reducer instead, resulting in concatenation of the old and new values.

The node can be added to the graph as follows. The function name is passed directly into the `add_node` method without requiring an additional decorator:

```
<builder name>.add_node("<node name>", <node function name>)
```

For the earlier example:

```
builder.add_node("first_node", update_state)
```

In practice, a node may contain an LLM call. An example is shown below:

```
from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list[str], add_messages]

from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="<model name>")

def update_state(state: State) -> State:
    response = llm.invoke(state["messages"])
    new_state = {
        "messages": [response]
    }
    return new_state
```

Edge (and Handoff) Definition

Edges can be either deterministic or optional.

To add a deterministic edge that connects two nodes, use:

```
<builder name>.add_edge("<upstream node name>", "<downstream node name
    >")
```

In the earlier example, the following edges can be added:

```
from langgraph.graph import START, END

<builder name>.add_edge(START, "first_node")
<builder name>.add_edge("first_node", END)
```

Here, `START` and `END` represent the default entry and exit points of the graph.

For optional edges, a routing function is used to decide which edge to select.

```
<builder name>.add_conditional_edges("<upstream node>", <routing
    function>)
```

where the return of the routing function needs to be a string of the name of the desired downstream node. Boolean routing function can be used as well, in which case the syntax is given below.

```
<builder name>.add_conditional_edges("<upstream node>", <routing
    boolean function>, {True: "<downstream node 1>", False: "<
    downstream node 2>"})
```

Graph Compiling and Invoking

Once all nodes and edges have been added, the graph must be compiled before it can be executed:

```
<graph name> = <builder name>.compile()
```

After compilation, the graph can be visualized as a Mermaid diagram in a Jupyter notebook:

```
from IPython.display import Image, display

display(Image(<graph name>.get_graph().draw_mermaid_png()))
```

To invoke a graph once with an initial state, use the following. This will trigger a **super step** which is the execution of the whole graph.

```
<graph name>.invoke(<initial state>)
```

It is also possible to run a invoke a graph in asynchronous mode as follows.

```
<graph name>.ainvoke(<initial state>)
```

Alternatively, the graph can be executed step by step using `.stream()`, which yields intermediate states and outputs at each stage of the execution.

Does LangGraph necessarily involve LLM?

No. LangGraph and LangChain can be used to develop applications that have nothing to do with LLMs. Data processing is handled by the nodes. Technically, it is possible to build an application in which all nodes perform conventional computations without involving any LLM.

In an LLM-based application, the state class often contains text, and LLMs are introduced in the nodes to process that text.

Note that memory are not automatically synchronized between different super steps. As a result, the user must either use a global parameter to save the chat history and supply it as the initial state for each super step, or persist the information in a database and let the node retrieve it via tools.

Tool

LangGraph allows packaging a function into a tool conveniently. First, define the function you want to package. Then wrap the function with `Tool`. A simple example is shown below:

```
def <function name>(input) -> <output type>:
    <do something>
    return <output>

from langchain.agents import Tool
<tool> = Tool(
    name="<tool name>",
    func=<function name>,
    description="<description>"
)
```

To test the tool, use

```
result = <tool>.invoke(<input>)
```

which should directly triggers the function defined in the tool.

To run a tool in AI agents, use either

```
result = <tool>.run(<input>)
```

or

```
result = await <tool>.arun(<input>)
```

to execute it in synchronous or asynchronous mode respectively.

The LangChain community provides a variety of tools out of the box. An example is shown below:

```
from langchain_community.utilities import GoogleSerperAPIWrapper

serper = GoogleSerperAPIWrapper()

tool_search = Tool(
    name="tool_search",
    func=serper.run, # serper.run() triggers the function
    description="useful for making online queries for additional
        information"
)
```

Recall that a node is essentially a user-defined function. To use tools in a node, simply call the tool functions within the node's source code.

In an agentic AI application, a node is often an LLM, and it decides what tools to trigger in a dynamic manner. In this case, consider binding the tools to the LLM as follows.

```
from langgraph.graph.message import add_messages

class State(TypedDict):
    messages: Annotated[list[str], add_messages]

from langchain_openai import ChatOpenAI
llm = ChatOpenAI(model="<model name>")
llm_with_tools = llm.bind_tools([<tool 1>, <tool 2>, ...])
```

```
def update_state(state: State) -> State:
    response = llm_with_tools.invoke(state["messages"])
    new_state = {
        "messages": [response]
    }
    return new_state
```

By using `.bind_tools()`, the LLM learns the use of those tools. The user does not need to prepare JSON documents to introduce the tools for the LLM.

Add the set of tools as a "tool node" to the graph. Use conditional edges so that the LLM determines whether to call a tool. A typical pattern is as follows.

```
from langgraph.prebuilt import ToolNode, tools_condition

<builder name>.add_node("<tool node>", ToolNode(tools=[<tool 1>, <tool
    2>, ...]))
<builder name>.add_conditional_edges("<llm node>", tools_condition, "<
    tool node>")
```

Notice that when using `tools_condition`, if the LLM decides to not use any tool, the edge is routed to `END`. Hence, there is no need to specifically connect the LLM node to the end node.

Structured Output

The nodes in a graph take the state as input and output. Therefore, there is no need to structure the output of a graph. When comes to individual LLMs, it is possible to enforce the output as follows.

```
from pydantic import BaseModel, Field

class <pydantic model>(BaseModel):
    <field>:<type> = Field(description="<description>")
    <field>:<type> = Field(description="<description>")
    ...

from langchain_openai import ChatOpenAI

<llm> = ChatOpenAI(model="<model name>")
<llm>.with_structured_output(<pydantic model>)
```

where `.with_structured_output()` function is used to associate a Pydantic model with an LLM's output.

Memory Sharing across Super Steps

As explained earlier, each invocation of the graph is known as a super step. By default, the input to a super step contains only the initial state. The memory from the previous super step is not automatically carried over to the next one.

A straightforward way to share memory across super steps is to manually add the conversation or results from earlier steps to the initial state of the next step. While this works, it is tedious and error-prone. LangGraph provides a native and robust mechanism for sharing memory across super steps. A demonstrative example is shown below:

```
from langgraph.checkpoint.memory import MemorySaver

memory = MemorySaver()

<define class, graph builder, add nodes and edges>

graph = builder.compile(checkpointer=memory)
config = {"configurable": {"thread_id": "1"}}
graph.invoke(<initial state>, config=config)
```

In the above example, `MemorySaver` is used to share memory across super steps. When compiling the graph, pass it as the checkpointer. When invoking a super step, set a thread ID in the configuration. All super steps with the same thread ID will share the same memory.

One can use the following command:

```
graph.get_state_history(config)
```

to retrieve all checkpointer values, i.e., the complete sequence of state snapshots since the first invocation of the graph. It is also possible to rewind to a specific checkpoint in this history and restart the graph from that point, which can be useful for certain applications.

13.2.5 AutoGen

AutoGen is an open-source agentic AI solution offered by Microsoft. Notice that just like LangGraph, AutoGen by itself is not just an agentic AI framework, but more of a comprehensive environment that provides variety of agentic AI applications relevant tools and solutions. As of this writing, it includes at least the following solutions [20].

- **AutoGen Core**

  AutoGen Core refers to a collection of concepts and tools that offer an easy way to quickly build event-driven, distributed, scalable, resilient AI agent systems.

- **AutoGen AgentChat**

  AutoGen AgentChat is a high-level API for building multi-agent applications. It is built on top of the AutoGen Core package.

- **AutoGen Extensions**

  AutoGen Extensions refers to a collection of APIs, models or tools that

extends AutoGen Core's capability. It is often used together with AutoGen Core and AutoGen AgentChat.

- **AutoGen Studio**

  AutoGen Studio is a low-code interface for agentic AI applications prototyping. It is built on top of AutoGen Agent Chat.

As far as the scope of this notebook concerns, we focus on AutoGen Core, AutoGen AgentChat and AutoGen Extensions.

Agent

The minimum realization of an AI agent in AutoGen framework is given below.

Agent with Multimodal LLM

Tool

AutoGen allows easy definition of a tool in the form of a function. It is also compatible with tools defined by LangChain and MCP. More details of MCP is introduced in the next Section 13.3. Syntax and examples are given below.

Team

AutoGen allows the user to define a team that consists of multiple AI agents for a task. The role of each AI agent is defined. The overall task is defined. The team tries to achieve the objective by letting the AI agents in the team communicate freely and discuss the assigned task, until the conclusion is drawn. This is to some extent similar with the idea of "crew" in CrewAI.

13.3 Model Context Protocol

Model Context Protocol (MCP) is an open-source standardized protocol to connect AI agents to variety of resources and tools.

In earlier Section 13.2, we have seen how different agentic AI frameworks incorporate tools in their systems. The following is a summary.

- Manual deployment

  The tools are formed as a Python function. JSON "user manuals" are provided to the LLM, with detailed explanation on how to call the function. The LLM agent calls the function when needed.

- OpenAI Agents SDK

The tools are formed as a Python function. Decorator is used to convert the function into a function tool. A collection of function tools is provided to the LLM. The LLM learns to use the function tools using the function description and no additional user manual is required.

Alternatively, an LLM agent can be converged into a tool with a single-line command.

- CrewAI

  The tools are formed as Python functions and saved at specified locations. Decorator is used. Tool names are provided to agents and tasks to specify what tool they can use.

- LangGraph

  The tools are formed as Python functions. Class `Tool()` or `StructuredTool()` are used to wrap the tools. The available tools information is provided to the LLM using `bind_tools()` function. ToolNodes are used. The LangGraph graph builder automatically route the requests between LLM agents and ToolNodes.

  LangGraph has a huge community where people share tools.

From the above description, we can see that it is often easy to wrap a function and build a tool and use it privately. However, it is challenging to share tools from different applications due to the different framework requirement and personal programming habits. The LangGraph community has made some progress with how people can share tools that they have developed. However, the usage of these tools is mostly limited to LangGraph users.

MCP is a standard and a solution that tries to make the sharing and recycling of tools and other resources in agentic AI systems easier across applications.

13.3.1 MCP Structure

The following concepts are introduced in the context of MCP.

- Host: the agentic AI based application.
- **MCP client**: the piece of program that runs in the host that provides an interface to the host of an MCP service.
- **MCP server**: the software corresponds with MCP client that actually provides the MCP service.

Figure 13.7 is used to demonstrate the relationship among host, MCP client and MCP server.

Some highlights are as follows.

- Each MCP client has a corresponding MCP server.

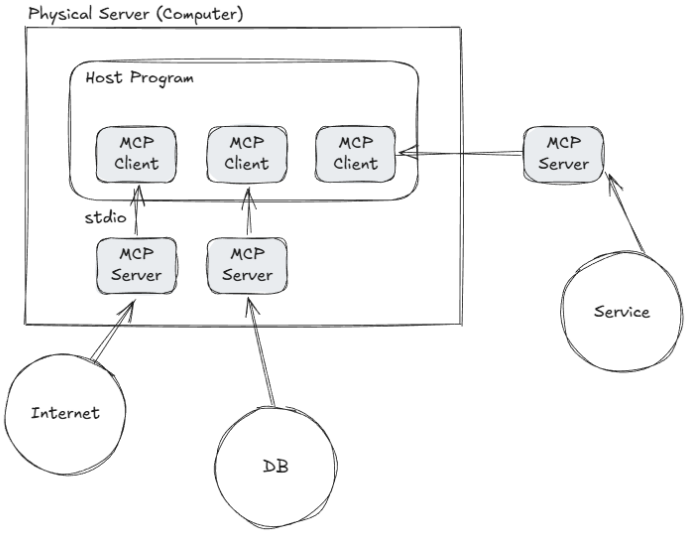

FIGURE 13.7
Demonstration of host program, MCP client and MCP server relationship.

- In most scenarios, MCP servers run on the same physical machine as the host program. It is possible, though rare in practice, that MCP servers run on different machines other than the host program.
- The information source for MCP servers does not have to be located on the same physical machine. For example, an MCP server may access information on the Internet or a remote database.
- There are options of the connectivity channels between MCP servers and clients. In the figure, `stdio` is used, which is indeed a popular choice.

MCP can be used for standardizing and sharing varieties of agentic AI resources. It has been most popular for sharing tools.

13.3.2 MCP Server Creation

As explained in earlier Section 13.3.1, to use MCP services, an MCP server needs to be started and it needs to be connected to the host program. As far as this notebook concerns, Python plant is assumed as the host program.

Notice that an MCP server runs outside the Python host program as an independent software, and it is not necessarily a Python program. The MCP server needs to be installed and started separately inside a dedicated virtual environment, and a connectivity channel needs to be build between the MCP client in the Python host program and the MCP server. Some python libraries

and tools, such as OpenAI Agents SDK, have made things easier. Though running inside Python scripts, they can call terminal commands to install, deploy and configure the MCP server and the connectivity channel.

An example of using OpenAI Agents SDK to start an MCP server is given below [22].

```
from pathlib import Path
from agents import Agent, Runner
from agents.mcp import MCPServerStdio

current_dir = Path(__file__).parent
samples_dir = current_dir / "sample_files"

async with MCPServerStdio(
       name="Filesystem Server via npx",
       params={
              "command": "npx",
              "args": ["-y", "@modelcontextprotocol/server-filesystem
                  ", str(samples_dir)],
       },
) as server:
       agent = Agent(
              name="Assistant",
              instructions="Use the files in the sample directory to
                  answer questions.",
              mcp_servers=[server],
       )
       result = await Runner.run(agent, "List the files available to
           you.")
       print(result.final_output)
```

In this example, a filesystem managing tool is deployed using MCP. The following part

```
async with MCPServerStdio(name="<name>", param=<configuration command>)
     as server:
       <do something>
```

is used to automatically deploy the MCP server and client and build the connectivity. The MCP server configuration is formulated into a dictionary that contains the commands to start and configure the MCP server.

The LLM agent gets to know an MCP server from `mcp_server` field in agent declaration as in

```
async with MCPServerStdio(name="<name>", param=<configuration command>)
     as server:
       agent = Agent(
              name="<name>",
              instructions="<instruction>",
              mcp_servers=[server],
       )
```

which is similar with tools. This way, the LLM agent automatically retrieves all the tools and APIs from the MCP server and learns how to use them.

Are There Risks with MCP Servers?

Running an MCP server is essentially the same with running 3rd party software. To make it worse, MCP server runs outside the Python environment, make it even more difficult to control. There is always an operation risk. Always make sure that the software is retrieved from a legitimate source (there are multiple marketplace for MCP servers) and it is well-known and well-reviewed, in order to mitigate the risk.

13.3.3 MCP Marketplace

Think of MCP marketplaces APP stores for MCP servers. There are many popular MCP market places. Popular MCP servers are often published on multiple MCP marketplaces.

To name one MCP marketplace as an example, *mcp.so*.

13.4 Examples

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

13.4.1 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 [19]. 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 missing, 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 rel-

evant 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.

13.4.2 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 13.4.1. 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 13.4.1.

Notice that in Section 13.4.1, 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 13.4.1, 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.

13.4.3 CrewAI: Credit Card Bill Recorder

The following agentic AI system is built using CrewAI framework. The system is used to parse a credit card bill, and use the information to maintain 3 tables: cards, merchants and transactions. The following AI agents and tasks are defined.

- Agents:
  - Credit Card Parser. Read and credit card PDF, and from the PDF summarizes credit card information in the form of Pydantic object and summarized transaction record in the form of list of Pydantic objects.
  - Card manager. Read the credit card information, and maintain the card table.
  - Merchant manager. Read the merchants from the summarized transactions records, collect the frequently visit merchants, and maintain the merchants table.
  - Transaction manager. Read the summarized transaction and maintain the transactions table.
- A task is associated with each agent.

Details are as follows.

Database Preparation

PostgreSQL database is created and deployed in a podman container. Three tables are created. Details are as follows.

To deploy the database, use

```
#!/bin/bash

source ~/Projects/smart-home/.env

if podman container exists "$PG_CONTAINER"; then
    status=$(podman inspect -f '{{.State.Status}}' "$PG_CONTAINER")
    if [ "$status" = "running" ]; then
```

```
        echo "Container $PG_CONTAINER is already running."
    else
        echo "Starting existing container $PG_CONTAINER..."
        podman start "$PG_CONTAINER"
    fi
else
    echo "Creating and starting container $PG_CONTAINER..."
    podman run -d \
        --name "$PG_CONTAINER" \
        -e POSTGRES_USER="$PG_USER" \
        -e POSTGRES_PASSWORD="$PG_PASSWORD" \
        -e POSTGRES_DB="$PG_DB" \
        -v "$PG_DATA_DIR":/var/lib/postgresql/data:Z \
        -p "$PG_PORT":5432 \
        docker.io/library/postgres:15
fi
```

where the database login credentials and configurations are given in the `.env` file as environmental variables.

To create the database tables, use

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

-- Define enum for card status (safe for re-execution)
DO $$
BEGIN
IF NOT EXISTS (SELECT 1 FROM pg_type WHERE typname = 'card_status')
    THEN
CREATE TYPE card_status AS ENUM ('in_use', 'replaced', 'lost', 'closed
    ');
END IF;
END
$$;

CREATE TABLE IF NOT EXISTS cards (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
name TEXT NOT NULL,
issuer TEXT NOT NULL,
network TEXT NOT NULL,
last4 CHAR(4) NOT NULL,
status card_status NOT NULL,
opened_on DATE,
closed_on DATE,
expires_on DATE NOT NULL,
tags JSONB,
notes TEXT,
created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

```
CREATE TABLE IF NOT EXISTS merchants (
```

```
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),

  name TEXT NOT NULL,                  -- Display name: "Coles", "
      Amazon"
  canonical_name TEXT,                 -- Optional normalized/merged
      name
  merchant_type TEXT,                  -- e.g. 'grocery', 'online_shop
      ', 'restaurant'

  location_label TEXT,                 -- Free-form (e.g. 'Macquarie
      Centre')
  latitude DOUBLE PRECISION,           -- Optional: physical location
  longitude DOUBLE PRECISION,          -- Optional: physical location
  opening_hours JSONB,                 -- Optional: hours for physical
      stores

  is_digital BOOLEAN,                  -- True if goods/services are
      typically digital
  is_recurrent BOOLEAN,                -- True if charges are usually
      recurring/subscription-based

  tags TEXT[],                         -- Program-friendly labels (e.g.
       ['australian', 'food'])
  properties JSONB,                    -- Arbitrary structured merchant
      -specific metadata
  notes TEXT,                          -- Human-facing comments or
      clarifications

  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Insert default "Others" merchant as a fallback
INSERT INTO merchants (
  id, name, canonical_name, merchant_type, tags, notes
)
VALUES (
  gen_random_uuid(),
  'Others',
  'Others',
  'uncategorized',
  ARRAY['fallback', 'unknown'],
  'Catch-all merchant for unmatched or unresolved transactions'
);
```

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";

CREATE TABLE IF NOT EXISTS card_transactions (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
card_id UUID NOT NULL REFERENCES cards(id),
```

```
merchant_id UUID NOT NULL REFERENCES merchants(id),
date DATE NOT NULL,
amount NUMERIC(10, 2) NOT NULL,
currency CHAR(3) NOT NULL DEFAULT 'AUD',
raw_entity TEXT NOT NULL,
tags JSONB,
notes TEXT,
statement_id TEXT,
imported_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Tools

Two tools are defined, namely the PostgreSQL tool that allows an AI agent to execute an SQL command (query or insert), and the web search tool. Notice that the web search tool is used by the merchant manager to query information about a merchant from online to determine its merchant type.

The following PostgreSQL Query Executor is defined.

```
from crewai.tools import tool
import os
import psycopg
from dotenv import load_dotenv

dotenv_path = os.path.expanduser("~/Projects/smart-home/.env")
load_dotenv(dotenv_path=dotenv_path, override=True)

@tool("PostgreSQL Query Executor")
def run_postgres_query(query: str) -> str:
    """
    Executes a SQL query on a PostgreSQL database and returns the
        results.

    Args:
        query (str): The SQL query to execute.

    Returns:
        str: The results of the query or an error message.
    """
    try:
        # Load database connection details from environment variables
        db_host = os.getenv("PG_HOST", "localhost")
        db_port = os.getenv("PG_PORT", "5432")
        db_name = os.getenv("POSTGRES_DB", "mydatabase")
        db_user = os.getenv("POSTGRES_USER", "SUNLU")
        db_password = os.getenv("PG_PASSWORD", "")
        print(f"Connecting to database {db_name} at {db_host}:{db_port}
            as user {db_user}")
        # Connect to the PostgreSQL database
```

```
        with psycopg.connect(
            host=db_host,
            port=db_port,
            dbname=db_name,
            user=db_user,
            password=db_password
        ) as conn:
          # Create a cursor to execute the query
            with conn.cursor() as cur:
                cur.execute(query)
                if cur.description: # If the query returns rows
                    results = cur.fetchall()
                    return str(results)
                else: # If the query does not return rows (e.g., INSERT,
                     UPDATE)
                    conn.commit()
                    return "Query executed successfully."
    except Exception as e:
        return f"An error occurred: {e}"
```

Serper API is used as the web search tool.

```
from crewai_tools import SerperDevTool

search_tool = SerperDevTool()
```

Agents and Tasks

The following agents and tasks are defined.

```
credit_bill_parser:
  role: >
    Credit Bill Parser
  goal: >
    Extract and summarize key information from a credit card bill
  backstory: >
    You are a helpful assistant skilled in analyzing credit card bills.
         You are given a credit card bill in text form, and you need to
         extract the following information --
    - Bank and card basic information, such as customer name, bank name
        , card last four digits, statement date, the status of the card
         (active or inactive), etc.
    - Money to be paid from last month, and money actually paid for
        last month.
    - Money to be paid for this month, minimum payment due date, and
        minimum payment amount.
    - Summary of each line item, including:
      - Date of transaction
      - Merchant name
      - Amount spent
      - Currency
```

```
    You must output the extracted information as a structured JSON
        object that strictly conforms to the ‘BillAnalysis‘ Pydantic
        model. This includes a ‘CardInfo‘ object, a ‘StatementSummary‘
        object, and a list of ‘Transaction‘ objects.
  llm: openai/gpt-4o-mini

credit_card_manager:
  role: >
    Credit Card Manager
  goal:
    Help the user manage credit cards database, including adding new
        cards and updating existing cards
  backstory: >
    You are a helpful assistant skilled in managing the credit cards
        database.
    You receive a ‘CardInfo‘ Pydantic object from the upstream agent.
    You have a tool, 'PostgreSQL Query Executor', that you use to
        generate and execute SQL queries to access the credit card
        table. The name of the table is ‘cards‘.
    The credit card table structure is as follows:
       Column  |          Type          | Collation | Nullable |
           Default
    -----------+------------------------+-----------+----------+-------------------

     id        | uuid                   |           | not null |
         uuid_generate_v4()
     name      | text                   |           | not null |
     issuer    | text                   |           | not null |
     last4     | character(4)           |           | not null |
     status    | card_status            |           | not null |
     opened_on | date                   |           |          |
     closed_on | date                   |           |          |
     expires_on | date                  |           | not null |
     tags      | jsonb                  |           |          |
     notes     | text                   |           |          |
     created_at | timestamp with time zone |      | not null | now()
    Your primary function is to manage credit card records in the ‘
        cards‘ table. You first need to query the table to determine if
         a card already exists. A card is identified as a match if the
        ‘issuer‘ and ‘last4‘ digits from the provided ‘CardInfo‘ object
         match a record in the database.
    If no matching card is found, you must generate an SQL ‘INSERT‘
        query to add the new card. You must use placeholder values for
        any required fields that are not present in the ‘CardInfo‘
        object. For example, use 'unknown' for text fields, '0000' for
        last 4 digits, and '9999-12-31' for date fields like expires_on
        .
    If a matching card is found, you can generate an SQL ‘UPDATE‘ query
        . You need to compare the existing database record with the new
```

```
        `CardInfo` object. You should only update a field in the
       database if the corresponding field in the database is
       currently a placeholder (e.g., 'unknown', '0000', or
       '9999-12-31') and the new data from `CardInfo` is valid.

   Tool usage rules for "PostgreSQL Query Executor":
     - Pass ONLY the raw SQL string to the tool's `query` argument.
     - Do NOT wrap SQL in JSON, Python, code fences, or quotes like
         "{\"query\": \"...\"}".
     - Return the SQL string directly.
     - Forbidden patterns:
       - "{\"query\": \"...\"}"
       - "```sql ... ```"
       - Python dicts or JSON objects instead of plain SQL strings

  llm: openai/gpt-4o-mini

merchant_manager:
  role: >
    Merchant Manager
  goal:
    Help the user manage merchants database, including adding new
        merchants and updating existing merchants
  backstory: >
    You are a helpful assistant skilled in managing the merchants
        database.
    You receive a list of transaction records from the upstream agent.
        Each record contains a merchant name. You have two tools at
        your disposal: 'PostgreSQL Query Executor' for database access,
         and 'Search Internet' to look up merchant information online.
    The merchants table structure is as follows. The name of the table
        is `merchants`.
       Column    |          Type          | Collation | Nullable |
           Default
    ---------------+------------------------+-----------+----------+-------------------

     id           | uuid                   |           | not null |
         gen_random_uuid()
     name         | text                   |           | not null |
     canonical_name | text                 |           |          |
     merchant_type | text                  |           |          |
     location_label | text                 |           |          |
     latitude     | double precision       |           |          |
     longitude    | double precision       |           |          |
     opening_hours | jsonb                 |           |          |
     is_digital   | boolean                |           |          |
     is_recurrent | boolean                |           |          |
     tags         | text[]                 |           |          |
     notes        | text                   |           |          |
```

```
    created_at    | timestamp with time zone |       | not null | now()
   Your primary function is to iterate through each unique merchant in
        the transaction list.
   For each unique merchant, you must first query the `merchants`
       table using your 'PostgreSQL Query Executor' tool to check if
       the merchant already exists in the database.
   - If the merchant exists, no further action is needed for this
       merchant.
   - If the merchant does not exist, you must decide whether to add it
        to the database. A merchant is worth adding if it appears 3 or
        more times in the transaction list or if it is a well-known
       brand. You can use your 'Search Internet' tool to gather
       information and make this decision.
   If you decide to add a new merchant, you must use your 'Search
       Internet' tool to gather necessary information, such as `
       canonical_name`, `merchant_type`, and `is_digital`. Once you
       have the information, you must generate a well-formed SQL `
       INSERT` query and execute it using the 'PostgreSQL Query
       Executor' tool. You should use 'Others' as the default `
       merchant_type` if the type cannot be determined.

   Tool usage rules for "PostgreSQL Query Executor":
     - Pass ONLY the raw SQL string to the tool's `query` argument.
     - Do NOT wrap SQL in JSON, Python, code fences, or quotes like
         "{\"query\": \"...\"}".
     - Return the SQL string directly.
     - Forbidden patterns:
       - "{\"query\": \"...\"}"
       - "```sql ... ```"
       - Python dicts or JSON objects instead of plain SQL strings

  llm: openai/gpt-4o-mini

credit_card_transaction_manager:
  role: >
    Credit Card Transaction Manager
  goal:
    Help the user manage the credit card transactions database, mainly
        adding new transactions to the database
  backstory: >
    You are a helpful assistant skilled in managing the credit card
        transactions database.
    You receive a structured Pydantic object containing a list of `
        Transaction` records.
    You have a tool, 'PostgreSQL Query Executor', that you use to
        generate and execute SQL queries.
    The `card_transactions` table structure is as follows:
       Column   |          Type          | Collation | Nullable |
           Default
```

```
-------------+-------------------------+-----------+----------+--------------------

 id          | uuid                    |           | not null |
     uuid_generate_v4()
 card_id     | uuid                    |           | not null |
 merchant_id | uuid                    |           | not null |
 date        | date                    |           | not null |
 amount      | numeric(10,2)           |           | not null |
 currency    | character(3)            |           | not null |
 raw_entity  | text                    |           | not null |
 tags        | jsonb                   |           |          |
 notes       | text                    |           |          |
 statement_id | text                   |           |          |
 imported_at | timestamp with time zone |       | not null | now()

The ‘cards‘ table structure is as follows:
    Column  |          Type          | Collation | Nullable |
        Default
------------+-------------------------+-----------+----------+--------------------

 id        | uuid                    |           | not null |
     uuid_generate_v4()
 name      | text                    |           | not null |
 issuer    | text                    |           | not null |
 last4     | character(4)            |           | not null |
 status    | card_status             |           | not null |
 opened_on | date                    |           |          |
 closed_on | date                    |           |          |
 expires_on | date                   |           | not null |
 tags      | jsonb                   |           |          |
 notes     | text                    |           |          |
 created_at | timestamp with time zone |      | not null | now()

The ‘merchants‘ table structure is as follows:
    Column    |          Type          | Collation | Nullable |
        Default
---------------+-------------------------+-----------+----------+--------------------

 id           | uuid                    |           | not null |
     gen_random_uuid()
 name         | text                    |           | not null |
 canonical_name | text                  |           |          |
 merchant_type | text                   |           |          |
 location_label | text                  |           |          |
 latitude     | double precision        |           |          |
 longitude    | double precision        |           |          |
 opening_hours | jsonb                  |           |          |
 is_digital   | boolean                 |           |          |
 is_recurrent | boolean                 |           |          |
```

```
    tags          | text[]                   |          |         |
    notes         | text                     |          |         |
    created_at    | timestamp with time zone |       | not null | now()

   Your primary function is to process each transaction in the list.
       For each one, you must:
   1. Check if the transaction already exists in the ‘
       card_transactions‘ table using the date, amount, currency, and
       raw entity.
   2. If it does not exist, you need to find the ‘id‘ of the
       corresponding card from the ‘cards‘ table. You can use the card
       ’s ‘issuer‘ and ‘last4‘ to query for the correct ‘card_id‘.
   3. You must also find the ‘id‘ of the corresponding merchant from
       the ‘merchants‘ table.
   4. Once you have both the ‘card_id‘ and ‘merchant_id‘, generate an
       SQL ‘INSERT‘ query to add the new transaction to the ‘
       card_transactions‘ table.
   - If you cannot find a matching card, you must report this as an
       error.
   - If you cannot find a matching merchant, you should map the
       transaction to a default ’Others’ merchant category.

   Tool usage rules for "PostgreSQL Query Executor":
     - Pass ONLY the raw SQL string to the tool’s ‘query‘ argument.
     - Do NOT wrap SQL in JSON, Python, code fences, or quotes like
         "{\"query\": \"...\"}".
     - Return the SQL string directly.
     - Forbidden patterns:
       - "{\"query\": \"...\"}"
       - "‘‘‘sql ... ‘‘‘"
       - Python dicts or JSON objects instead of plain SQL strings

  llm: openai/gpt-4o-mini

crew_master:
  role: >
    Crew Master
  goal: >
    Orchestrate a team of agents to achieve a common goal.
  backstory: >
    You are an experienced project manager and team lead. Your role is
       to oversee the entire process, delegate tasks to the right
       agents, and ensure they are completed correctly and in the
       proper order. You are the ultimate authority and decision-maker
        for the crew.
  llm: openai/gpt-4o
```

The following tasks are defined.

```
manage_card_database:
```

```
  description: >
    Receive a CardInfo object from the upstream agent. Use the '
        PostgreSQL Query Executor' tool to first check if the card
        already exists in the 'cards' table using the issuer and last4
        digits. If not, generate an SQL INSERT query to add the new
        card. If it exists, generate an SQL UPDATE query to update any
        placeholder values with valid data from the CardInfo object.
  expected_output: >
    A simple confirmation message, such as 'Card added successfully' or
         'Card updated successfully', or 'No changes needed'.
  agent: credit_card_manager
  context:
    - parse_bill

manage_merchants:
  description: >
    Receive a structured Pydantic object from the upstream agent and
        process the `transactions` list within it. For each unique
        merchant, check for its existence in the 'merchants' table. If
        the merchant is new, decide if it should be added to the
        database based on its frequency (3+ times) or prominence. If so
        , use online search to gather details and generate an SQL
        INSERT query for the new merchant.
  expected_output: >
    A summary of the actions taken, for example, 'Analyzed 15 merchants
        . Added 3 new merchants to the database: [list of new merchant
        names]'.
  agent: merchant_manager
  context:
    - parse_bill

manage_transactions:
  description: >
    Receive a structured Pydantic object from the upstream agent and
        process the `transactions` list within it. For each transaction
        , check for its existence in the `card_transactions` table. If
        it's a new transaction, use the 'PostgreSQL Query Executor'
        tool to retrieve the `card_id` from the `cards` table and the `
        merchant_id` from the `merchants` table. Once the required IDs
        are found, generate an SQL INSERT query to add the transaction
        to the database.
  expected_output: >
    A summary of the actions taken, such as 'Processed 25 transactions.
         Added 20 new transactions to the database.'
  agent: credit_card_transaction_manager
  context:
    - parse_bill
    - manage_card_database
    - manage_merchants
```

```
parse_bill:
  description: |
    You are given the full raw text of a credit card bill between the
        delimiters.
    Extract a single JSON object that strictly matches the BillAnalysis
         schema
    (card_info, statement_summary, transactions). Return ONLY JSON (no
        Markdown).

    ---BEGIN BILL TEXT---
    {bill_content}
    ---END BILL TEXT---

    ## Source format & extraction rules (OCBC examples)
    - The issuer is "OCBC Bank" if present. Customer name appears near
        the top (e.g., "SUN, LU").
    - Card number appears like "XXXX-XXXX-XXXX-6684" -> last4 = "6684".
    - Statement date appears as "STATEMENT DATE" followed by either
        "01-07-2023" or "01 JUL 23".
      * Normalize ALL dates to YYYY-MM-DD.
      * If you see DD/MM (e.g., "31/05"), infer the year from the
          statement date's year.
      * If the date is like "01 JUL 23", map month names to numbers and
           expand "23" to "2023".
    - Minimum payment and due date appear as "TOTAL MINIMUM DUE" and "
        PAYMENT DUE DATE".
    - Totals often appear as "TOTAL AMOUNT DUE" and/or "TOTAL".
    - "LAST MONTH'S BALANCE" is the previous balance.
    - Payments/credits may be shown in parentheses (e.g., "(1,657.81
        PAYMENT BY INTERNET)") -> treat as negative amounts for
        payments_and_credits.
    - Transaction section looks like:
       TRANSACTION DATE
       DESCRIPTION
       AMOUNT (SGD)
      Followed by blocks such as:
       "31/05"
       "NINTENDO ... (extra lines like FOREIGN CURRENCY ...)"
       "112.00"
      * Merchant names may span multiple lines; join them into a single
           merchant_name string with spaces.
      * Use the SGD number shown in the AMOUNT column as the
          transaction amount.
      * Use currency "SGD" unless explicitly stated otherwise for the
          final charged amount.
      * Parentheses around amounts mean negative/refund (e.g., "(194.40
           ANNUAL FEE WAIVER)").
```

```
    * Include fees/rebates as transactions with merchant_name like "
        ANNUAL FEE", "ANNUAL FEE WAIVER", "CASH REBATE", etc.
    * Ignore boilerplate text (contact info, headings, totals/
        subtotals rows).
  - Set card_status to "Active" unless the bill explicitly indicates
      otherwise.
  - Set card_network to null unless you can reliably infer it (do NOT
      guess).

  ## Output schema (strict)
  {
    "card_info": {
      "customer_name": "string",
      "issuer": "string",
      "card_network": "string|null",
      "last4": "string",
      "statement_date": "YYYY-MM-DD",
      "card_status": "string"
    },
    "statement_summary": {
      "previous_balance": number,
      "payments_and_credits": number,
      "new_charges": number,
      "new_balance": number,
      "minimum_payment": number,
      "minimum_payment_due_date": "YYYY-MM-DD"
    },
    "transactions": [
      {
        "date": "YYYY-MM-DD",
        "merchant_name": "string",
        "amount": number,
        "currency": "SGD",
        "raw_entity": "string" // a concise slice of the original
            lines used
      }
    ]
  }

  ## Validation & normalization
  - Use null for unknowns rather than inventing values.
  - All numbers must be plain numbers (no currency symbols, commas,
      or parentheses).
    * For negative amounts shown as "(xxx.xx)", output a negative
        number (e.g., -194.40).
  - Dates must be ISO "YYYY-MM-DD".
  - Ensure totals are consistent: new_balance = previous_balance -
      payments_and_credits + new_charges.
```

```
  expected_output: >
    A single JSON object that validates against the BillAnalysis
        Pydantic schema. No additional text.
  agent: credit_bill_parser
```

Crew

Crew is defined as follows.

```
import sys
import os
from datetime import date

from crewai_tools import SerperDevTool
from crewai import Agent, Task, Crew, Process
from crewai.project import CrewBase, agent, crew, task
from pydantic import BaseModel, Field
from typing import List, Optional

# Import the custom tools
from credit_card_bill_injector.tools.postgresql_tools import
    run_postgres_query

from pydantic import BaseModel, Field
from typing import List, Optional

class Transaction(BaseModel):
    """Represents a single transaction line item from a credit card
        bill."""
    date: str = Field(..., description="Date of the transaction in YYYY
        -MM-DD format.")
    merchant_name: str = Field(..., description="Name of the merchant
        as it appears on the bill.")
    amount: float = Field(..., description="Amount of the transaction
        .")
    currency: str = Field(..., description="Currency of the transaction
        , e.g., 'AUD', 'USD'.")
    raw_entity: Optional[str] = Field(None, description="The raw,
        unparsed text line for the transaction.")

class StatementSummary(BaseModel):
    """Summarizes the key financial figures from a credit card
        statement."""
    previous_balance: float = Field(..., description="Balance from the
        previous statement.")
    payments_and_credits: float = Field(..., description="Total amount
        of payments and credits applied.")
    new_charges: float = Field(..., description="Total amount of new
        charges this period.")
```

```
    new_balance: float = Field(..., description="The new total balance
        due for this statement.")
    minimum_payment: float = Field(..., description="Minimum payment
        due.")
    minimum_payment_due_date: str = Field(..., description="Due date
        for the minimum payment in YYYY-MM-DD format.")

class CardInfo(BaseModel):
    """Contains basic information about the credit card and cardholder
        ."""
    customer_name: str = Field(..., description="The name of the
        cardholder.")
    issuer: str = Field(..., description="The name of the bank or card
        issuer.")
    card_network: Optional[str] = Field(None, description="The network
        of the card, e.g., 'Visa', 'Mastercard'.")
    last4: str = Field(..., description="The last four digits of the
        credit card number.")
    statement_date: str = Field(..., description="The date the
        statement was issued in YYYY-MM-DD format.")
    card_status: str = Field(..., description="The status of the card,
        e.g., 'Active'.")

class BillAnalysis(BaseModel):
    """The complete structured output for a parsed credit card bill."""
    card_info: CardInfo = Field(..., description="Basic card and
        statement information.")
    statement_summary: StatementSummary = Field(..., description="
        Summary of statement balances and payments.")
    transactions: List[Transaction] = Field(..., description="A list of
         all transactions for this statement.")


search_tool = SerperDevTool()

@CrewBase
class CreditCardBillInjector():
    """CreditCardBillInjector crew"""

    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    @agent
    def credit_bill_parser(self) -> Agent:
        return Agent(
            config=self.agents_config['credit_bill_parser'],
            verbose=True,
            allow_delegation=False
        )
```

```
    @agent
    def credit_card_manager(self) -> Agent:
        return Agent(
            config=self.agents_config['credit_card_manager'],
            tools=[run_postgres_query],
            verbose=True,
            allow_delegation=True
        )

    @agent
    def merchant_manager(self) -> Agent:
        return Agent(
            config=self.agents_config['merchant_manager'],
            tools=[run_postgres_query, search_tool],
            verbose=True,
            allow_delegation=True
        )

    @agent
    def credit_card_transaction_manager(self) -> Agent:
        return Agent(
            config=self.agents_config['credit_card_transaction_manager
                '],
            tools=[run_postgres_query],
            verbose=True,
            allow_delegation=True
        )


    @task
    def parse_bill(self) -> Task:
        return Task(
            config=self.tasks_config['parse_bill'],
            agent=self.credit_bill_parser(),
            output_pydantic=BillAnalysis
        )

    @task
    def manage_card_database(self) -> Task:
        return Task(
            config=self.tasks_config['manage_card_database'],
            agent=self.credit_card_manager(),
            context=[self.parse_bill()]
        )

    @task
    def manage_merchants(self) -> Task:
        return Task(
```

```
            config=self.tasks_config['manage_merchants'],
            agent=self.merchant_manager(),
            context=[self.parse_bill()]
        )

    @task
    def manage_transactions(self) -> Task:
        return Task(
            config=self.tasks_config['manage_transactions'],
            agent=self.credit_card_transaction_manager(),
            context=[self.parse_bill(), self.manage_card_database(),
                self.manage_merchants()]
        )

    @crew
    def crew(self) -> Crew:
        #manager = Agent(
        #    config=self.agents_config['crew_master'],
        #    allow_delegation=True
        #)
        return Crew(
            agents = self.agents,
            tasks = self.tasks,
            verbose=True,
            process=Process.sequential,
        )
```

And finally, in the main program,

```
import sys
import warnings

from datetime import datetime

from credit_card_bill_injector.crew import CreditCardBillInjector

warnings.filterwarnings("ignore", category=SyntaxWarning, module="pysbd
    ")

from dotenv import load_dotenv
import os

import fitz

def run():
    """
    This function sets up the environment and runs the CrewAI process.
    """
    sys.path.append(os.path.abspath(os.path.join(os.path.dirname(
        __file__), '..', '..')))
```

```
    dotenv_path = os.path.expanduser('~/Projects/smart-home/.env')
    load_dotenv(dotenv_path=dotenv_path, override=True)

    doc = fitz.open('/data/Projects/smart-home/input-document/
        credit_card_bill.pdf')
    bill_content = ""
    for page in doc:
        bill_content += page.get_text()
    doc.close()

    injector_crew = CreditCardBillInjector()

    result = injector_crew.crew().kickoff(
        inputs={
            'bill_content': bill_content
        }
    )

    print("\n\n########################")
    print("## Here is your Crew's result:")
    print("########################\n")
    print(result)

if __name__ == '__main__':
    run()
```

A

Brief Introduction to Python Package Manager

CONTENTS

As far as this notebook concerns, almost all the codes are in Python. There are many environment and package managers for Python. Two popular ones, `conda` and `uv`, are introduced in this appendix chapter.

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 resolves the dependencies issue and applies the latest compatible versions of libraries.

Details of `conda` can be found in [1]. This section 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 install Miniconda, a variation of Anaconda distribution, to use `conda`.

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

Miniconda can be installed on varieties of operating systems. Download the installer from the official website and follow the instructions just like installing any other software.

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 edit channels and their priorities.

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

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 proprietary packages developed by Anaconda, and it may require license if used in commercial environments.

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 comes from.

To create an environment, use

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

which will create a fresh-start environment.

Alternatively, if there is a file inside which details of the environment and packages are given, consider using

```
conda env create -f <file>
```

where additional `-f` followed by the list of packages in TXT or YAML file (conventionally, `environment.yml`) 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. To export the current environment, use

```
conda export > <file name>.yml
```

It is possible to configure the output format to be YAML, TXT or JSON using `--format`, and to perform cross-platform exporting using `--from-history`. Details can be found in the official website and are not given here.

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 `PATH`, run:

```
uv python install
```

To install a specific version, use

```
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

```
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.

  Note: `.venv` may contain massive small-size files that are not suitable for cloud synchronization. It is recommended to exclude `.venv` from cloud synchronization.

- `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 the following command

```
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] chroma-core. chroma-core/chroma: Open-source search and retrieval database for ai applications. https://github.com/chroma-core/chroma. Accessed: 2025-08-25.

[6] CrewAI. Installation. https://docs.crewai.com/en/installation/. Accessed: 2025-08-21.

[7] CrewAI Documentation. Agents – crewai concepts. https://docs.crewai.com/en/concepts/agents. Accessed: 2025-08-21.

[8] CrewAI Documentation. Memory – crewai core concepts. https://docs.crewai.com/en/concepts/memory. Accessed: 2025-08-25.

[9] CrewAI Documentation. Tasks – crewai core concepts. https://docs.crewai.com/en/concepts/tasks. Accessed: 2025-08-22.

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

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

[12] 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.

[13] LangChain. Langchain. https://www.langchain.com/langchain. Accessed: 2025-09-09.

[14] LangChain. Langgraph. `https://www.langchain.com/langgraph`. Accessed: 2025-09-09.

[15] LangGraph Documentation. Compiling your graph – langgraph guides: Low-level graph api concepts. `https://langchain-ai.github.io/langgraph/concepts/low_level/#compiling-your-graph`. Accessed: 2025-09-14.

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

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

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

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

[20] Microsoft. Autogen: A framework for building ai agents and applications. `https://microsoft.github.io/autogen/dev/index.html`. Accessed: 2025-10-07.

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

[22] OpenAI. Model context protocol (mcp) – openai agents sdk. `https://openai.github.io/openai-agents-python/mcp/`. Accessed: 2025-11-05.

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

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

[25] 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.

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

[27] 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.

[28] 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.