Chapter 7. Learning in Agentic Systems

This chapter covers different techniques for approaching and integrating learning into agentic systems. Adding the capability for agents to learn and improve over time is an incredibly useful addition, but is not necessary when designing agents. Implementing learning capabilities takes additional design, evaluation, and monitoring, which may or may not be worth the investment depending on the application. By learning, we mean improving the performance of the agentic system through interaction with the environment. This process enables agents to adapt to changing conditions, refine their strategies, and enhance their overall effectiveness.

Nonparametric learning refers to techniques to change and improve performance automatically without changing the parameters of the models involved. In contrast, parametric learning refers to techniques in which we specifically train or fine-tune the parameters of the foundation model. We will start by exploring nonparametric learning techniques, then cover parametric fine-tuning approaches, including supervised fine-tuning and direct preference optimization, that adapt model weights for targeted improvements.

Nonparametric Learning

Multiple techniques exist to do this, and we will explore several of the most common and useful approaches.

Nonparametric Exemplar Learning

The simplest of these techniques is exemplar learning. In this approach, as the agent performs its task, it is provided with a measure of quality, and those examples are used to improve future performance. These examples are used as few-shot examples for in-context learning. In the simplest version, fixed few-shot examples, they are hardcoded into the prompt and do not change (the left side of Figure 7-1).

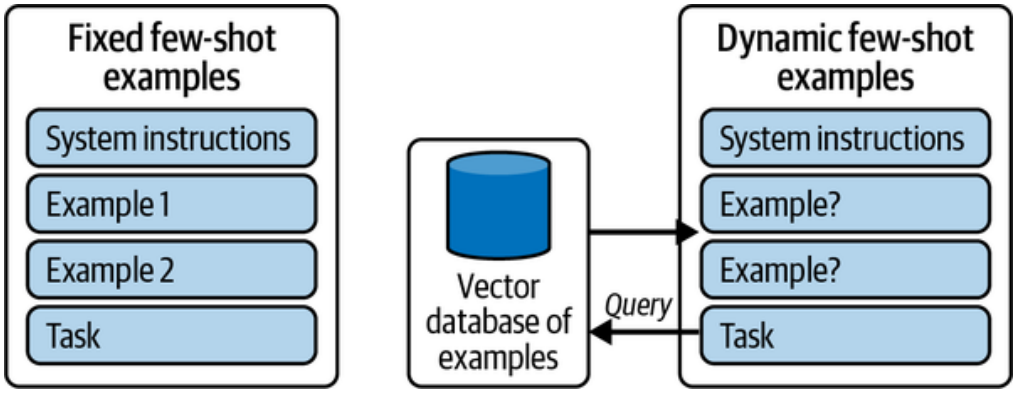

Figure 7-1. Fixed versus dynamic few-shot example selection. On the left, the model prompt uses a static set of few-shot examples embedded in the system prompt. On the right, dynamic few-shot selection retrieves the most relevant examples from a vector database at runtime, enabling more adaptive and contextually appropriate task prompting.

If we have more examples, we can continue adding them into the prompt, but that eventually comes with increases in cost and latency. In addition, not all examples might be useful for all inputs. A common way to address this is to dynamically select the most relevant examples to include in the prompt (see on the right side of Figure 7-1). These experiences, as examples, are then stored in a way that makes them accessible for future reference. This typically involves building a memory bank where details of each interaction—such as the context, actions taken, outcomes, and any feedback received—are stored. This database acts much like human memory, where past experiences shape understanding and guide future actions. Each experience provides a data point that the agent can reference to make better decisions when encountering similar situations. This method enables agents to build a repository of knowledge that can be drawn upon to improve performance.

The agent retrieves information from its database of past cases to solve new problems. Each stored case consists of a problem description, a solution that was applied, and the outcome of that solution. When faced with a new situation, the agent searches its memory to find similar past cases, analyzes the solutions that were applied, and adapts them if necessary to fit the new circumstances. This method allows for high flexibility, as the agent can modify its approach based on what has or has not worked in the past, thus continually refining its problem-solving strategies.

When successful examples are saved in persistent storage, then retrieved and provided as examples in the prompt, performance increases significantly on a range of tasks. This is a well-established finding and has been confirmed across a variety of domains. In practice, this provides us with a simple, transparent, and lightweight way to rapidly improve the agent performance on given tasks. As the number of successful examples increases, it then becomes

wise to retrieve the most relevant successful examples by type, text retrieval, or semantic retrieval. Note that this technique can be applied to the agentic task execution as a whole, or it can be performed independently on subsets of the task.

Reflexion

Reflexion equips an agent with a simple, language-based habit of self-critique: after each unsuccessful attempt, the agent writes a brief reflection on what went wrong and how to improve its next try. Over time, these reflections live in a "memory buffer" alongside the agent's prior actions and observations. Before each new attempt, the agent rereads its most recent reflections, allowing it to adjust its strategy without ever retraining the model.

At a high level, the Reflexion loop works like this:

1. *Perform an action sequence.* The agent interacts with the environment using its usual prompt-driven planning.
2. *Log the trial.* Every step—actions taken, observations received, success or failure—is appended to a log in persistent storage (for example, a JSON file or database table).
3. *Generate a reflection.* If the trial fails, the agent constructs a short "reflection prompt" that includes the recent interaction history plus a template asking: "What strategy did I miss? What should I do differently next time?" The LLM produces a concise plan.
4. *Update memory.* A helper function (update_memory) reads the trial logs, invokes the LLM on the reflection prompt, and then saves the new reflection back into the agent's memory structure.
5. *Inject reflections on the next run.* When the agent attempts the same (or a similar) task again, it prepends its most recent reflections into the prompt, guiding the model toward the improved strategy.

Reflexion is very lightweight. You don't touch model weights; you simply use the foundation model as its own coach. Reflexion accommodates both numerical feedback (e.g., a success flag) and free-form comments, and it has been shown to boost performance on tasks ranging from code debugging to multistep reasoning. You can see how this works in Figure 7-2.

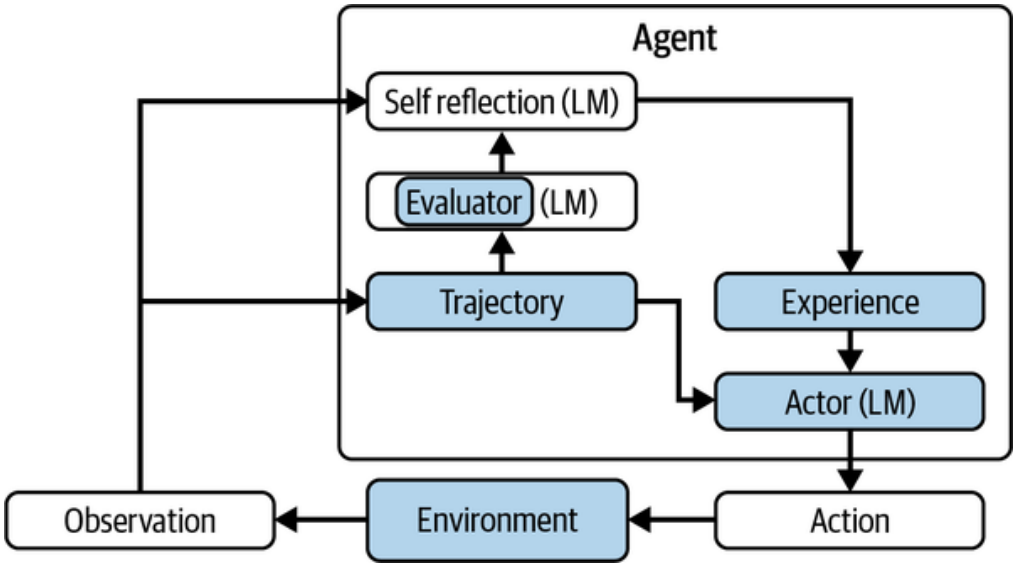

Figure 7-2. Reflexion agent.

Despite the significant improvement that Reflexion can add to agents, this approach can be implemented with just a few lines of code:

```
from typing import Annotated, List, Dict
from typing_extensions import TypedDict
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState,
from langchain_core.messages import HumanMessage

llm = ChatOpenAI(model="gpt-5")

reflections = []

def call_model(state: MessagesState):
    response = llm.invoke(state["messages"])
    return {"messages": response}

reflexion_prompt = f"""You will be given the history of
which you were placed in an environment and given a tas
were unsuccessful in completing the task. Do not summar
but rather think about the strategy and path you took t
the task. Devise a concise, new plan of action that acc
with reference to specific actions that you should have
if you tried A and B but  forgot C, then devise a plan
environment-specific actions. You will need this later
the same task. Give your plan after "Plan".

Instruction:
I am looking for dairy-free and apple variety packs of
lower than 30.00 dollars
[Search]
```

Action: search[dairy free and apple variety pack of chi
Observation:
[Back to Search]
Page 1 (Total results: 50)
[Next >]
[B07HRFJWP8]
Enjoy Life Foods Soft Baked Ovals, Breakfast Bars, Nut
Dairy Free, Non GMO, Gluten Free, Vegan, Variety Pack,
$100.0
[B01KMHY5PG]
Enjoy Life Soft Baked Chewy Bars, Variety Pack, Nut Fre
Free, Gluten Free, 6 Boxes (30 Total Bars)
$21.49
[B008D2X8C4]
Enjoy Life Lentil Chips Variety Pack, Dairy Free Chips,
Non GMO, Vegan, Gluten Free, 24 Bags (0.8 oz)
$100.0

Action: think[The first two items are not what I want.
the dairy free and apple variety pack of chips I want.]
Observation: OK.

Action: click[B008D2X8C4]
Observation:
[Back to Search]
[< Prev]
flavor name [dill and sour cream][garlic & parmesan]
[light sea salt][margherita pizza][thai chili lime][var
size [0.8 ounce (pack of 24)][4 ounce (pack of 12)]
Price: $100.0
Rating: N.A.
[Description]
[Features]
[Reviews]
[Buy Now]

Action: think[The item has options 'variety pack' and '
and seems good to buy.]
Observation: OK.

Action: click[variety pack]
Observation: You have clicked variety pack.

Action: click[0.8 ounce (pack of 24)]
Observation: You have clicked 0.8 ounce (pack of 24).

```
Action: click[Buy Now]

STATUS: FAIL

Plan:
"""
```

The prompt is built in three sections to turn the model into its own coach: first, a brief framing instruction tells the model "you failed your task—focus on strategic missteps rather than summarizing the environment and output your corrective plan after the word 'Plan,'" which ensures a concise, parseable response. Next, under "Instruction:" we restate the original goal ("find a dairy-free, apple variety pack of chips under $30"), anchoring the reflection in the true objective. Finally, we include the complete Action/Observation transcript of the failed run—every search, click, and internal thought ending with `STATUS: FAIL` —so the model has concrete evidence of what went wrong. By ending with the cue "Plan:" we signal the model to shift from diagnosis to prescription, yielding a focused set of next-step recommendations. Here's the Python implementation that sets up our three-part coaching prompt—framing instruction, restated goal under "Instruction:" and the full Action/Observation transcript—ending with the cue "Plan:":

```
def get_completion(prompt: str) -> str:
    # Wraps our `call_model` helper for one-off text cc
    result = llm.invoke([{"role":"user","content":promp
    return result[0].content

def _generate_reflection_query(trial_log: str, recent_r
    history = "\n\n".join(recent_reflections)
    return f'''{history}
        {trial_log}
        Based on the above, what plan would you follow

def update_memory(trial_log_path: str, env_configs: Lis
    """Updates the given env_config with the appropriat
    with open(trial_log_path, 'r') as f:
        full_log: str = f.read()

    env_logs: List[str] = full_log.split('#####\n\n####
    assert len(env_logs) == len(env_configs), print(f'l
    for i, env in enumerate(env_configs):
        # if unsolved, get reflection and update env cc
```

```
            if not env['is_success'] and not env['skip']:
                if len(env['memory']) > 3:
                    memory: List[str] = env['memory'][-3:]
                else:
                    memory: List[str] = env['memory']
                reflection_query = _generate_reflection_que
                reflection = get_completion(reflection_quer
                env_configs[i]['memory'] += [reflection]


builder = StateGraph(MessagesState)
builder.add_node("reflexion", call_model)
builder.add_edge(START, "reflexion")
graph = builder.compile()

result = graph.invoke(
    {
        "messages": [
            HumanMessage(
                reflexion_prompt
            )
        ]
    }
)
reflections.append(result)
print(result)
update_memory(trial_log_path, env_configs)
```

The preceding example is built around a handful of core ideas woven together in under 20 lines of code. First, we isolate every call to the LLM behind a simple wrapper— `call_model(state)` —so that our graph nodes remain focused and reusable. Next, we craft one multiline “reflection prompt” that tells the model: “You attempted this task and failed. Don’t rehash the environment; focus on what strategic step you missed, and output a concise plan after the word ‘Plan’.” We then log each trial’s full transcript to disk, and after a failure we invoke `update_memory(...)` to read those logs, pull in the last few stored reflections to bound context, and ask the LLM to generate a new self-critique, which we append back into our in-memory list. Finally, by adding a single “reflexion” node to our `StateGraph` (wired from `START`), every run of the agent automatically invokes this prompt and enriches its state with the latest “Plan: …” output. Over repeated runs, the model effectively becomes its own coach—continually refining its strategy without touching a single parameter.

Experiential Learning

Experiential learning takes nonparametric learning a step further. In this approach, the agent still gathers its experiences into a database, but now it applies a new step of aggregating insights across those experiences to improve its future policy. This is especially valuable for reflecting on past failures and attempting to develop new techniques to improve performance in similar situations in the future. As the agent extracts insights from its experience bank, it maintains this list of insights over time, and it dynamically modifies these insights, promoting the most valuable insights, downvoting the least useful ones, and revising insights based on new experiences.

This work builds on Reflexion by adding a process for cross-task learning. This allows the agent to improve its performance when it moves across different tasks and helps identify good practices that can transfer. In this approach, ExpeL maintains a list of insights that are extracted from past experiences. Over time, new insights can be added, and existing insights can be edited, upvoted, downvoted, or removed, as can be seen in Figure 7-3.

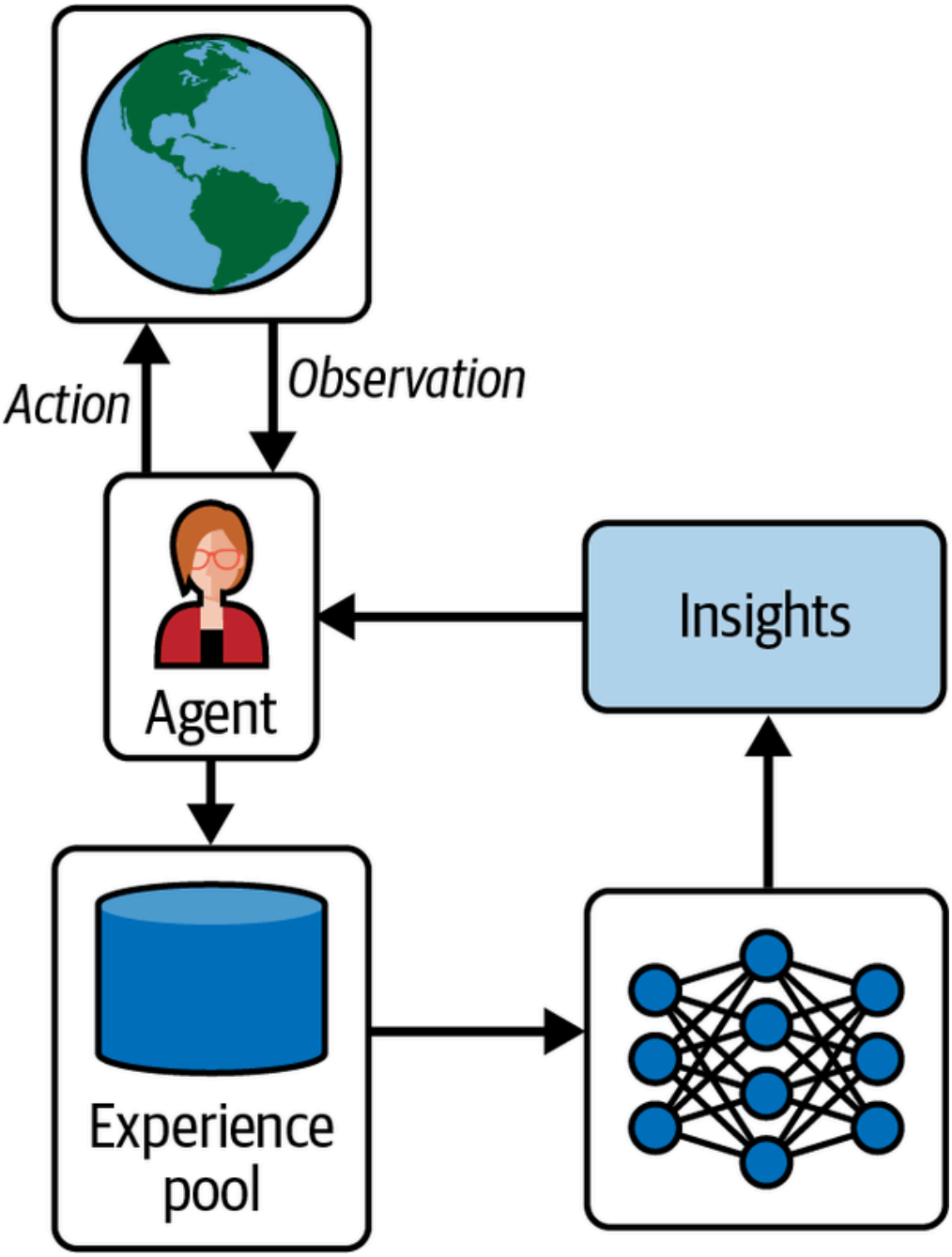

Figure 7-3. Experiential learning agents.

This process begins with a simple effort of asking the foundation model to reflect on the observation returned from the environment, with the goal of identifying insights that can lead to better performance on the task in the future:

```
from typing import Annotated
from typing_extensions import TypedDict
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState,
from langchain_core.messages import HumanMessage

# Initialize the LLM
llm = ChatOpenAI(model="gpt-5")
# Function to call the LLM
def call_model(state: MessagesState):
```

```
    response = llm.invoke(state["messages"])
    return {"messages": response}

class InsightAgent:
    def __init__(self):
        self.insights = []
        self.promoted_insights = []
        self.demoted_insights = []
        self.reflections = []

    def generate_insight(self, observation):
        # Use the LLM to generate an insight based on th
        messages = [HumanMessage(content=f'''Generate ar
            on the following observation: '{observation}

        # Build the state graph
        builder = StateGraph(MessagesState)
        builder.add_node("generate_insight", call_model)
        builder.add_edge(START, "generate_insight")
        graph = builder.compile()

        # Invoke the graph with the messages
        result = graph.invoke({"messages": messages})
        # Extract the generated insight
        generated_insight = result["messages"][-1].conte
        self.insights.append(generated_insight)
        print(f"Generated: {generated_insight}")
        return generated_insight
```

This may work well when we have a small number of examples to learn from, but what if we have many? This technique offers a simple but effective way to manage this: the insights generated are regularly reevaluated and adjusted in relative importance to the other rules. For example, a sample prompt to reflect on previous actions to generate new rules that improve performance on future trials could be:

> *By examining and contrasting to the successful trial, and the list of existing rules, you can perform the following operations: add, edit, remove, or agree so that the new list of rules is GENERAL and HIGH LEVEL critiques of the failed trial or proposed way of Thought so they can be used to avoid similar failures when encountered with different questions in the future. Have an emphasis on critiquing how to perform better Thought and Action. (ExpeL)*

These learned rules are then regularly reevaluated and adjusted in importance relative to the other rules derived from experience. The methodology for evaluating and improving the existing rules is as follows:

> *The available operations are:* `AGREE` *(if the existing rule is strongly relevant for the task),* `REMOVE` *(if one existing rule is contradictory or similar/duplicated to other existing rules),* `EDIT` *(if any existing rule is not general enough or can be enhanced),* `ADD` *(introduce new rules that are distinct from existing rules and relevant for other tasks). Each needs to closely follow their corresponding formatting as follows (any existing rule not edited, not agreed upon, or not removed is considered copied):*

```
AGREE <EXISTING RULE NUMBER>: <EXISTING RULE>
REMOVE <EXISTING RULE NUMBER>: <EXISTING RULE>
EDIT <EXISTING RULE NUMBER>: <NEW MODIFIED RULE>
ADD <NEW RULE NUMBER>: <NEW RULE>
```

This process is a bit more involved, but it still relies on manageable logic. Specifically, this process enables helpful insights to be dynamically improved upon in subsequent experiences. This process is illustrated in Figure 7-4, in which the model is used to extract insights from pairs of successful and unsuccessful examples, and in which insights are promoted and demoted over time, distilling out a small list of insights that are used to guide and improve the performance of the agent.

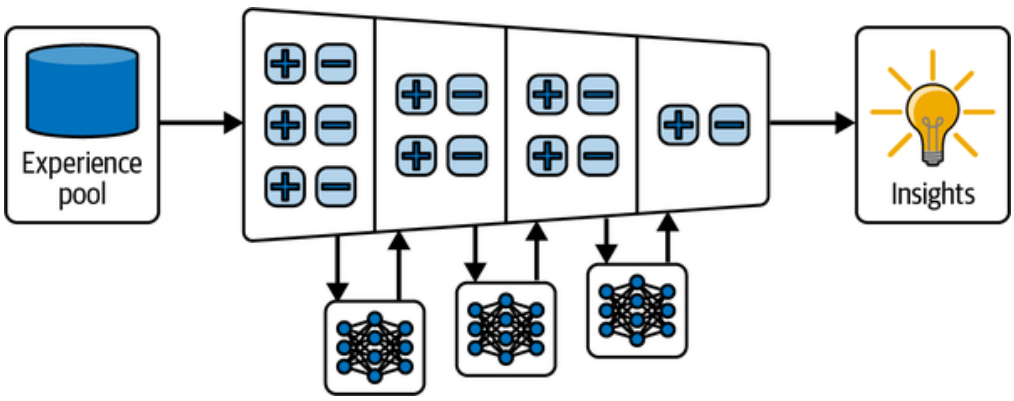

Figure 7-4. Experiential learning with insight extraction and distillation. The agent begins by gathering experiences into an experience pool. Multiple model evaluations are then used to extract insights from these experiences, aggregating and distilling them into a concise set of general, high-level critiques and rules. These distilled insights guide future decisions, enabling the agent to improve its performance across tasks over time.

In this next section, we see how these rules are actually created, promoted, modified, and removed to enable the agent to improve its performance on the

task over time:

```
    def promote_insight(self, insight):
            if insight in self.insights:
                self.insights.remove(insight)
                self.promoted_insights.append(insight)
                print(f"Promoted: {insight}")
            else:
                print(f"Insight '{insight}' not found in ins

        def demote_insight(self, insight):
            if insight in self.promoted_insights:
                self.promoted_insights.remove(insight)
                self.demoted_insights.append(insight)
                print(f"Demoted: {insight}")
            else:
                print(f"Insight '{insight}' not found in prc

        def edit_insight(self, old_insight, new_insight):
            # Check in all lists
            if old_insight in self.insights:
                index = self.insights.index(old_insight)
                self.insights[index] = new_insight
            elif old_insight in self.promoted_insights:
                index = self.promoted_insights.index(old_ins
                self.promoted_insights[index] = new_insight
            elif old_insight in self.demoted_insights:
                index = self.demoted_insights.index(old_insi
                self.demoted_insights[index] = new_insight
            else:
                print(f"Insight '{old_insight}' not found.")
                return
            print(f"Edited: '{old_insight}' to '{new_insight
        def show_insights(self):
            print("\nCurrent Insights:")
            print(f"Insights: {self.insights}")
            print(f"Promoted Insights: {self.promoted_insigh
            print(f"Demoted Insights: {self.demoted_insights

        def reflect(self, reflexion_prompt):
            # Build the state graph for reflection
            builder = StateGraph(MessagesState)
            builder.add_node("reflection", call_model)
            builder.add_edge(START, "reflection")
            graph = builder.compile()
```

```
        # Invoke the graph with the reflection prompt
        result = graph.invoke(
            {
                "messages": [
                    HumanMessage(
                        content=reflexion_prompt
                    )
                ]
            }
        )
        reflection = result["messages"][-1].content
        self.reflections.append(reflection)
        print(f"Reflection: {reflection}")
```

With sufficient feedback, this process provides an efficient way to learn from interactions with the environment and improve performance over time. An added advantage of this approach is its capability to facilitate the agent's gradual adaptation to nonstationary environments. Thus, if your agent needs to adjust its policy to a changing environment, this approach enables it to do so effectively. Let's now take a look at some example usage:

```
agent = InsightAgent()
# Simulated sequence of observations and whether the KP
reports = [
    ("Website traffic rose by 15%, but bounce rate jump
        False),
    ("Email open rates improved to 25%, exceeding our 2
    ("Cart abandonment increased from 60% to 68%, missi
        False),
    ("Average order value climbed 8%, surpassing our 5%
    ("New subscription sign-ups dipped by 5%, just belo
        False),
]
# 1) Generate and prioritize insights over the reportin
for text, hit_target in reports:
    insight = agent.generate_insight(text)
    if hit_target:
        agent.promote_insight(insight)
    else:
        agent.demote_insight(insight)
# 2) Refine one of the promoted insights with human-in-
if agent.promoted_insights:
    original = agent.promoted_insights[0]
        agent.edit_insight(original, f'''Refined: {orig
```

```
        landing-page UX changes to reduce bounce.''')
# 3) Display the agent's final insights state
agent.show_insights()
# 4) Reflect on the top insights to plan improvements
reflection_prompt = (
    "Based on our promoted insights, suggest one high-i
    run next quarter:"
    f"\n{agent.promoted_insights}"
)
agent.reflect(reflection_prompt)
```

As you can see, even a small number of lines of code can enable an agent to continually learn from experience to improve performance on a specific task. These approaches are very practical, affordable, easy to implement, and enable continual adaptation from experience. In some cases, though, and especially when we have a large number of samples to learn from, it can make sense to consider fine-tuning.

Parametric Learning: Fine-Tuning

Parametric learning involves adjusting the parameters of a predefined model to improve its performance on specific tasks. When we have evaluation data, we can use it to improve the performance of our system. It often makes sense to start with nonparametric approaches, because they are simpler and faster to implement. Adding examples and insights into the prompt takes time and computational resources, though. When we have a sufficient number of examples, it might be worth considering fine-tuning your models as well to improve your agentic performance on your tasks. Fine-tuning is a common approach where a pretrained model is adapted to new tasks or datasets by making small adjustments to its parameters.

Fine-Tuning Large Foundation Models

Most developers begin building agentic systems with generic large foundation models such as GPT-5, Claude Opus, Gemini, and other similar classes of models because these offer an exceptional level of performance across a variety of tasks. These models are pretrained on extensive, general-purpose datasets, which equip them with a vast amount of linguistic and conceptual knowledge. These companies invest a great deal of effort in their own post-

training processes. Fine-tuning these models involves making targeted adjustments to their parameters, tailoring them to specific tasks or domains. This process allows developers to adapt the model's extensive knowledge to specialized applications, boosting its relevance and effectiveness on specific tasks while retaining its general capabilities. Figure 7-5 illustrates the generic fine-tuning process, showing how a large pretrained model is further adapted to specific tasks using curated domain datasets.

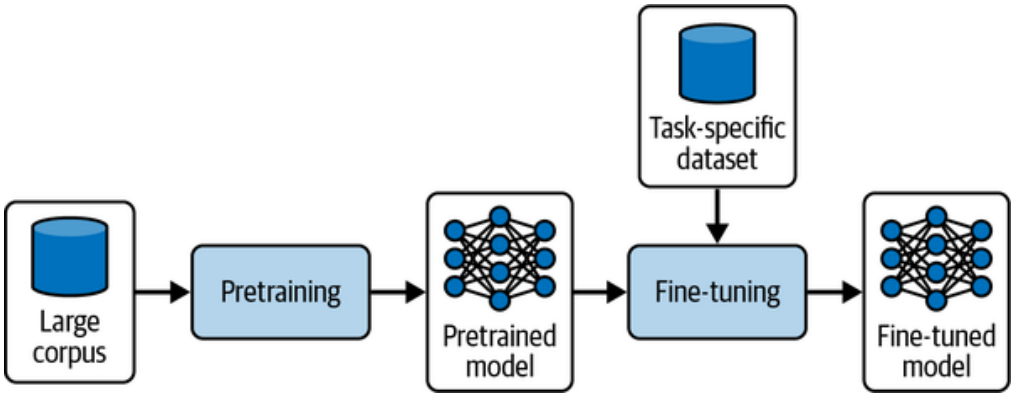

Figure 7-5. Fine-tuning workflow. A language model is first pretrained on a broad corpus to build general capabilities, then fine-tuned on a smaller, task-specific dataset to produce a specialized model aligned with domain needs.

Deciding whether to invest in fine-tuning hinges on your specific needs, resources, and longer-term maintenance plans. Consider fine-tuning in the following scenarios:

Domain specialization is critical

You need the model to speak your organization's jargon, follow a strict style guide, or handle highly sensitive content with minimal errors. Off-the-shelf models often struggle with narrow domains, and supervised fine-tuning (SFT) or direct preference optimization (DPO) can lock in that expertise.

Consistent tone and format matter

If every response must adhere to a precise template—say, financial disclosures or legal disclaimers—fine-tuning ensures the model reliably produces the correct structure without elaborate prompt engineering.

Tool and API calls must be precise

When your agent regularly invokes external functions or services (e.g., medical dosages, trading APIs), function-calling fine-tuning can

drastically reduce miscalls and handle edge-case errors more gracefully than in-context prompts alone.

You have sufficient high-quality data and budget

Fine-tuning large models demands hundreds to thousands of curated examples, expert graders (for reinforcement fine-tuning [RFT]), and GPU hours. If you lack data or compute, nonparametric methods like Reflexion or exemplar retrieval may offer better ROI.

Retraining frequency is manageable

Fine-tuned models require version management, retraining schedules, and compatibility checks. If your domain changes frequently, the upkeep cost can outweigh the performance gains.

When to hold off:

You're in rapid prototyping or low-volume use

Early in development, nonparametric learning or prompt engineering lets you iterate at zero retraining cost. Only commit to fine-tuning once your use case and data pipelines are stable.

Model evolution could invalidate your effort

Proprietary LLM providers regularly release improved base models. A new GPT-5 update may outperform your fine-tuned GPT-4, wiping out months of retraining work. Always weigh your fine-tuning investment against the pace of upstream model advances.

You're experiencing resource constraints

If GPU availability is limited, annotation is expensive, or inference speed is a priority, consider nonparametric strategies like retrieval-augmented generation. They can deliver many of the same benefits at a fraction of the cost and with far lower initial investment and ongoing maintenance.

In short, fine-tune a model only when your performance requirements, data availability, and operational capacity align—and always maintain a clear plan for retraining or migrating when the next generation of base models arrives. It's important to note that pretraining—training a model from scratch on trillions of tokens—is an undertaking reserved for major AI labs with vast

compute resources and proprietary data. For nearly all teams, the best approach is to start with high-quality open source models that have appropriate licenses for your use case. Often, these models already include post-training or instruction tuning that aligns closely with your task needs. In many cases, this eliminates the need for additional fine-tuning altogether, or at least reduces it to minimal targeted updates. Before investing in fine-tuning, always explore whether an existing pretrained or instruction-tuned model can meet your requirements with prompt engineering, nonparametric learning, or lightweight adaptation techniques. When in doubt, don't fine-tune your model. There are often lower-cost, higher-leverage activities you can take to improve your product. Table 7-1 shows the primary methods for fine-tuning language models.

Table 7-1. Primary methods for fine-tuning language models

Method	How it works	Best for
Supervised fine-tuning (SFT)	Provide (prompt, ideal-response) pairs as "ground truth" examples. Call the OpenAI fine-tuning API to adjust model weights.	Classification, structured output, correcting instruction failures
Vision fine-tuning	Supply image-label pairs for supervised training on visual inputs. This improves image understanding and multimodal instruction following.	Image classification, multimodal instruction robustness
Direct preference optimization	Give both a "good" and a "bad" response per prompt and indicate the preferred one. The model learns to rank and prefer higher-quality outputs.	Summarization focus, tone/style control
Reinforcement fine-tuning (RFT)	Generate candidate outputs and have expert graders score them. Then use a policy gradient-style update to reinforce high-scoring chains of thought.	Complex reasoning, domain-specific tasks (legal, medical)

Fine-tuning offers four distinct levers for adapting pretrained models to your needs:

Supervised fine-tuning (SFT)

SFT uses curated (prompt, response) pairs to teach the model exactly how it should behave, making it ideal for classification tasks, structured outputs, or correcting instruction-following errors.

Vision fine-tuning

Vision fine-tuning injects labeled image-label pairs to sharpen a model's multimodal understanding—perfect when you need robust image classification or more reliable handling of visual inputs.

Direct preference optimization (DPO)

DPO trains the model on paired "good versus bad" responses, helping it learn to favor higher-quality outputs, which is especially useful for tuning tone, style, or summarization priorities.

Reinforcement fine-tuning (RFT)

RFT leverages expert-graded outputs and policy-gradient updates to reinforce complex reasoning chains, making it the go-to for high-stakes domains like legal analysis or medical decision support.

Large foundation models excel at absorbing vast amounts of general knowledge, but their true power emerges when you fine-tune them on domain-specific data. A GPT-5 model customized for financial documents, for example, will not only parse jargon correctly but also adhere to your organization's precise reporting conventions. Similarly, a legal-tuned model can surface case law insights with the right tone of voice, while a customer-support tune can ensure every reply follows your corporate guidelines. This tight alignment between the model's internal representations and your real-world context is why fine-tuning remains indispensable for mission-critical applications.

That said, fine-tuning large models demands serious resources. Billions of parameters translate into heavy GPU requirements, lengthy training runs, and nontrivial cloud costs. Retraining to keep up with evolving data or to correct drift can multiply these expenses, and real-time deployments may suffer from higher inference latency as a result. For organizations without dedicated ML infrastructure, these barriers can make large-model fine-tuning impractical.

Equally important is the need for high-quality, task-specific training data. Large models only become “better” in your domain when they see enough representative examples—often in the thousands—to internalize subtle patterns. Curating, labeling, and validating these datasets is time-consuming and can introduce bias if not handled carefully. Without rigorous data governance and robust hold-out testing, you risk overfitting your model to stale or unrepresentative examples, limiting its ability to generalize and retain fairness.

Despite these challenges, fine-tuning large models remains a powerful approach, especially in cases where high performance is critical and the resources to support such models are available. The unparalleled capacity of large models enables them to perform at exceptional levels when fine-tuned for specific tasks, often surpassing the performance of smaller, task-specific models. This makes them ideal for applications where accuracy, depth of understanding, and nuanced language handling are necessary, such as healthcare diagnostics, legal analysis, or complex technical support.

Fine-tuning language models is a large and complex domain, encompassing a wide range of techniques, architectures, and trade-offs. In this section, we are not attempting to cover every nuance or training approach in depth. Instead, the examples provided here are intended as an introduction to the topic—offering practical illustrations to help you assess whether fine-tuning might be worth deeper investment for your own projects. If you find that these methods align with your goals, there are many excellent resources, papers, and open source toolkits available to continue your learning journey into fine-tuning strategies, scalable optimization, and production deployment.

Large foundation models offer a powerful solution for applications requiring high accuracy, adaptability, and nuanced understanding. Fine-tuning these models enables developers to harness their extensive pretrained knowledge while optimizing performance for specialized tasks or domains. While the computational and data requirements are significant, the benefits of fine-tuning large models can justify the investment for applications demanding peak performance and robust language comprehension, but it is only recommended for a small number of use cases.

The Promise of Small Models

In contrast to large foundation models, small models offer a more resource-efficient alternative, making them suitable for many applications where computational resources are limited or response time is critical. While small models inherently have fewer parameters and simpler architectures, they can still be surprisingly effective when finely tuned to a specific task. This adaptability stems from their simplicity, which not only allows for faster adaptation but also enables rapid experimentation with different training configurations. Small models are particularly advantageous in environments where deploying larger, more complex models would be costly, impractical, or excessive given the task requirements.

The lean architecture of small models offers unique advantages in transparency and interpretability. Because they have fewer layers and parameters, it is easier to analyze their decision-making processes and to understand the factors influencing their outputs. This interpretability is invaluable in applications where explainability is essential—such as finance, healthcare, and regulatory domains—as stakeholders need clear insights into how and why decisions are made. For instance, a small model fine-tuned for medical image classification can be more straightforward to debug and validate, providing assurance to medical practitioners who rely on its predictions. In these contexts, smaller models contribute to increased accountability and trust, particularly in high-stakes applications where the reasoning behind decisions must be understandable and accessible.

Small models also enable Agile development workflows. Their lightweight structure allows for faster iterations during fine-tuning, which can lead to quicker insights and adjustments. For developers working in Agile environments or with limited access to high-performance computing, small models provide a flexible, responsive solution. They are ideal for tasks requiring continuous or incremental learning, where models must be frequently updated with new data to maintain relevance. Moreover, small models can be deployed effectively in real-time systems, such as embedded devices, mobile applications, or Internet of Things networks, where low latency is essential. In these applications, the reduced computational footprint of small models enables efficient processing without compromising the overall system’s responsiveness.

Another key advantage of small models is their accessibility, both in terms of cost and availability. Many high-performing small models are open source and freely available, including models like Llama and Phi, which can be modified to suit various use cases. This accessibility lowers barriers for organizations and developers who may not have the budget or infrastructure to support large-scale models. Small models allow these teams to experiment, innovate, and deploy ML solutions without incurring significant operational costs. This democratization of ML technology enables more organizations to harness the benefits of AI, contributing to a more inclusive development ecosystem.

In terms of performance, fine-tuned small models can achieve results comparable to those of larger models on specific, narrowly defined tasks. For example, a small model fine-tuned for sentiment analysis within a particular domain, such as financial reports, can achieve high accuracy because it specializes in recognizing patterns specific to that context. When applied to well-defined tasks with clear data boundaries, small models can match, or even surpass, the performance of larger models by focusing all of their capacity on the relevant aspects of the task. This efficiency is particularly valuable in applications with high accuracy demands but limited data, where small models can be customized to perform effectively without overfitting.

In addition to their efficiency, small models support a sustainable approach to AI development. Training and deploying large models consume significant energy and computational resources, which contribute to environmental impacts. Small models, however, require substantially less energy for training and inference, making them a more sustainable choice for applications where resource consumption is a concern. Organizations prioritizing environmental sustainability can integrate small models as part of their green AI strategies, contributing to reduced carbon footprints without compromising on innovation.

The promise of small models extends to settings where frequent updates or retraining are needed. In scenarios where the data landscape changes rapidly —such as social media sentiment analysis, real-time fraud detection, or personalized recommendations—small models can be quickly retrained or fine-tuned with new data, adapting rapidly to changing patterns. This ability to frequently update without high retraining costs makes small models ideal for applications where adaptability is crucial. Additionally, small models can be deployed in federated learning environments, where data privacy concerns require models to be trained across decentralized data sources. In these

settings, small models can be efficiently fine-tuned on edge devices, enabling privacy-preserving AI solutions.

Fine-tuning smaller models represents a rapidly evolving landscape—a kaleidoscope of architectures, sizes, and capabilities that can deliver near-state-of-the-art performance at a fraction of the compute and cost. In early 2025, benchmarks like Stanford's HELM (Holistic Evaluation of Language Models) showcased open weight models such as DeepSeek-v3 and Llama 3.1 Instruct Turbo (70B) achieving mean scores above 66% on MMLU, and even 8B-parameter variants like Gemini 2.0 Flash-Lite began to crack the 64% threshold. In addition, Baytech Consulting reported that Phi-3-mini (3.8B) matched 540B-parameter PaLM's 60% MMLU score, a 142× size reduction in two years. Mobile-MMLU further highlighted that models under 9B can excel on edge-focused tasks, although variance grows as parameter counts fall.

This pace means that the "best" small model family today—be it Llama 3 (8B–70B), Qwen2.5 Turbo (72B), or the emerging Palmyra and DeepSeek lines—may be eclipsed within months. To stay current, practitioners should rely on trusted third-party leaderboards:

- Stanford HELM publishes live MMLU, GPQA, and IFEval scores across dozens of models.
- Papers With Code aggregates benchmarks and provides downloadable artifacts for comparative analysis.
- Hugging Face's Evaluation on the Hub offers an API to fetch up-to-date results on common tasks like GSM8K and HumanEval.
- BigBench Leaderboard tracks performance on the BBH suite, complementing HELM's broader scope.

When choosing a small model, consider your deployment constraints—latency, hardware, budget—and task demands. Models with fewer than eight billion parameters are unbeatable for on-device or low-cost inference; 8B–70B families strike a sweet spot for general reasoning; above that, proprietary giants like GPT-5 still lead in high-stakes accuracy. By combining these resources with periodic leaderboard checks, you can navigate this shifting terrain and select the optimal small-model family for your agentic application—while acknowledging that the field's rapid churn will likely deposit a new champion by the time you finish reading this chapter.

Supervised Fine-Tuning

Among parametric approaches, supervised fine-tuning remains the foundational technique, enabling precise behavioral shaping through curated input/output examples. SFT is the foundational approach for precisely steering an agent's behavior by showing it explicit examples of how to respond. One powerful use case is teaching an agent exactly when and how to invoke external APIs—fine-tuning function calling so the agent not only formats tool calls correctly but also reasons whether a call should happen at all. This extends what standard hosted function calling offers, providing more control and consistency when prompt engineering alone falls short. While off-the-shelf foundation models continue to improve at generating function calls, you may encounter stubborn cases where your prompts grow unwieldy, parameters are repeatedly mis-parsed, or accuracy lags behind your domain's strict requirements. In those scenarios—especially if you're driving high-volume traffic and every percentage point of reliability matters—fine-tuning on curated examples can both boost performance and, over time, reduce your per call costs compared with token-expensive proprietary endpoints. In essence, SFT uses carefully curated (prompt, response) pairs to help the model learn the desired output style, structure, or behavior. The same technique can adapt an agent for consistent tone, structured output, or—in this example—precise tool use. You can see this process illustrated in Figure 7-6.

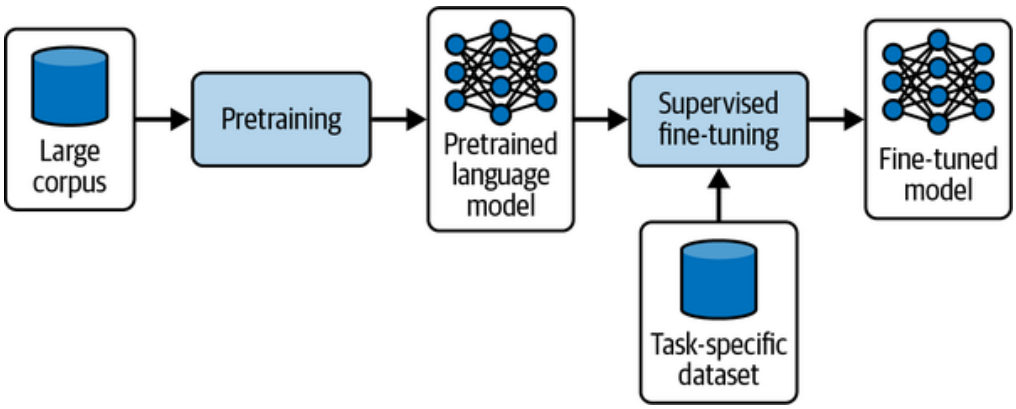

Figure 7-6. SFT workflow. A foundation model is first pretrained on a broad corpus to build general capabilities, then further fine-tuned using a task-specific supervised dataset to adapt it for specialized applications.

To make function calls robust, you'll typically define an explicit schema for each API you expose—specifying function names, valid arguments, types, and return formats. This ensures your examples teach the agent the *contract* it must follow. To do this, you assemble a fine-tuning dataset of structured examples that mirror your exact API schema—function names, argument types, and return formats—so the model internalizes your toolset's contract.

The result is a model that not only formats calls correctly on the first try, but also makes contextual judgments about whether a function should be invoked at all. Because this approach demands extra data curation, compute resources, and maintenance, we recommend starting with the pretrained models' built-in function-calling and runtime schema validation. Only once you've confirmed that prompt engineering and standard APIs fall short, should you consider this more heavyweight investment—ideally when your scale and precision requirements justify the up-front effort.

This involves presenting the model with structured examples where the agent must choose whether to make a function call, populate arguments accurately, and wrap the result appropriately. For example, if a user asks, "What's the weather in Boston?", a well-tuned agent should call a `get_weather(location="Boston")` function, and then incorporate the result into its reply. But if the user says, "Imagine it's snowing in Boston—what should I wear?", the agent should reason hypothetically without triggering a real call. This type of contextual judgment is learned through targeted examples.

To ensure your fine-tuned agent generates only well-formed, safe function invocations, it's critical to define and enforce a clear schema for every API or tool you expose. By codifying each function's name, argument types, and return structure in a machine-readable format—such as JSON Schema or a TypeScript/Zod schema—you give the model a precise contract to follow. During fine-tuning, include these schemas alongside your examples so the model learns not just what to call but exactly how to structure its JSON payload. At runtime, validate every proposed call against the same schema (using libraries like Zod, Ajv, or Pydantic) before executing it; any mismatch can be caught early and either corrected or rejected, preventing malformed or malicious requests. This end-to-end schema discipline drastically reduces errors, simplifies debugging, and hardens your system against unexpected inputs.

Fine-tuning also helps the model learn how to parse user inputs into valid arguments, recover from errors (like missing parameters), and gracefully fall back if the function call fails. Special tokens and formatting—such as wrapping the agent's internal reasoning in `<think>…</think>` or enclosing a call in `<tool_call>…</tool_call>` —can help the model distinguish between dialogue, thought, and action.

The following is a minimal working pattern for the supervised fine-tuning of a language model with LoRA (Low-Rank Adaptation) adapters for function calling. This includes preprocessing conversations into a consistent *chat template*:

1. Attaching special tokens for `<think>` or `<tool_call>` segments
2. Using LoRA to adapt only targeted layers efficiently
3. Training with `SFTTrainer` to update the model on your dataset of correct (prompt, response) pairs

We start with the preprocess function, which structures the data appropriately for training:

```
def build_preprocess_fn(tokenizer):
    """Returns a function that maps raw samples to toke
    def _preprocess(sample):
        messages = sample["messages"].copy()
        _merge_system_into_first_user(messages)
        prompt = tokenizer.apply_chat_template(messages
        return {"text": prompt}

    return _preprocess
```

Here, we wrap the model's internal reasoning and external tool calls in special tokens, like `<think>…</think>` and `<tool_call>…</tool_call>`. This makes it easy for the model to separate its "thoughts" from its API actions:

```
def build_tokenizer(model_name: str):
    tokenizer = AutoTokenizer.from_pretrained(
        model_name,
        pad_token=ChatmlSpecialTokens.pad_token.value,
        additional_special_tokens=ChatmlSpecialTokens.l
    )
    tokenizer.chat_template = CHAT_TEMPLATE
    return tokenizer


def build_model(model_name: str, tokenizer, load_4bit:
    kwargs = {
```

```
        "attn_implementation": "eager",
        "device_map": "auto",
    }
    kwargs["quantization_config"] = BitsAndBytesConfig(
        load_in_4bit=True,
        bnb_4bit_compute_dtype=torch.bfloat16,
        bnb_4bit_quant_type="nf4",
        bnb_4bit_use_double_quant=True,
    )
    model = AutoModelForCausalLM.from_pretrained(model_
    model.resize_token_embeddings(len(tokenizer))
    return model
```

Each example is tokenized and added to a training dataset, and then fine-tuned using standard supervised learning techniques with LoRA for efficiency. The training loop uses `SFTTrainer` from Hugging Face's TRL library, which supports features like sequence packing and gradient checkpointing:

```
def load_and_prepare_dataset(ds_name: str, tokenizer, m
    int, max_eval: int) -> DatasetDict:
    """Loads the dataset and applies preprocessing & tr
    raw = load_dataset(ds_name).rename_column("conversa
    processed = raw.map(build_preprocess_fn(tokenizer),
        remove_columns="messages")
    split = processed["train"].train_test_split(test_si
    split["train"] = split["train"].select(range(max_tr
    split["test"] = split["test"].select(range(max_eval
    return split


def train(
    model,
    tokenizer,
    dataset: DatasetDict,
    peft_cfg: LoraConfig,
    output_dir: str,
    epochs: int = 1,
    lr: float = 1e-4,
    batch_size: int = 1,
    grad_accum: int = 4,
    max_seq_len: int = 1500,
):
    train_args = SFTConfig(
        output_dir=output_dir,
```

```
        per_device_train_batch_size=batch_size,
        per_device_eval_batch_size=batch_size,
        gradient_accumulation_steps=grad_accum,
        save_strategy="no",
        eval_strategy="epoch",
        logging_steps=5,
        learning_rate=lr,
        num_train_epochs=epochs,
        max_grad_norm=1.0,
        warmup_ratio=0.1,
        lr_scheduler_type="cosine",
        report_to=None,
        bf16=True,
        gradient_checkpointing=True,
        gradient_checkpointing_kwargs={"use_reentrant":
        packing=True,
        max_seq_length=max_seq_len,
    )

    trainer = SFTTrainer(
        model=model,
        args=train_args,
        train_dataset=dataset["train"],
        eval_dataset=dataset["test"],
        processing_class=tokenizer,
        peft_config=peft_cfg,
    )

    trainer.train()
    trainer.save_model()
    return trainer
```

When agents depend on reliable tool use—retrieving calendar entries, executing commands, or querying databases—SFT makes these calls dramatically more robust than prompt engineering alone. It lowers error rates, teaches contextual judgment (when *not* to call), and reduces your token cost by cutting retries and malformed calls.

It also introduces a layer of reasoning: the model can choose when not to call a tool. For example, if the user says “If it rains tomorrow, I’ll stay in,” the agent can reason that no API call is needed and simply reply.

Finally, this method improves user experience by enabling agents to handle complex tasks with reliability. As agents take on more responsibility—especially in automation and decision-making roles—structured function calling becomes a foundational skill worth fine-tuning.

Direct Preference Optimization

Building on SFT, direct preference optimization introduces preference learning, aligning outputs more closely with human-ranked quality judgments. DPO is a fine-tuning technique that trains a model to prefer better outputs over worse ones by learning from ranked pairs. Unlike standard SFT, which simply teaches the model to replicate a "gold" output, DPO helps the model internalize preference judgments—improving its ability to rank and select high-quality completions at inference time. Figure 7-7 illustrates the DPO workflow, showing how models are trained on human preference data to learn to produce outputs that align with ranked quality judgments.

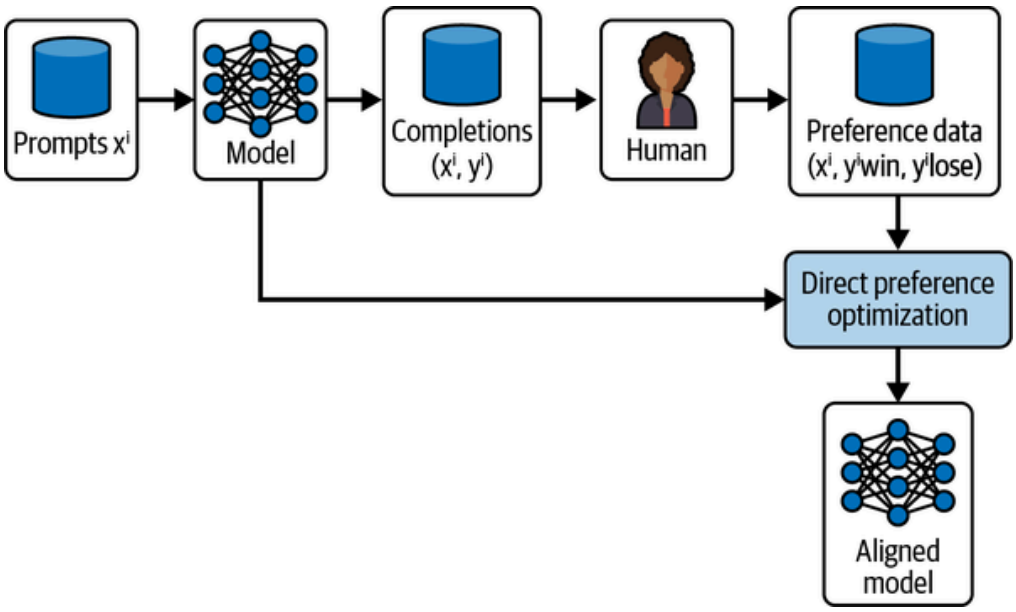

Figure 7-7. DPO workflow. Prompts are fed to the model to generate multiple completions, which are then evaluated by humans to produce preference data indicating the better response. These preferences are indicated as `y_win` and `y_lose` under "preference data." This data is used in DPO training to directly optimize the model toward preferred outputs, resulting in an aligned model that better reflects human preferences.

The following is a minimal working example using a small Phi-3 model to fine-tune help desk response quality:

```
import torch, os
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForCau
    BitsAndBytesConfig
from peft import LoraConfig, get_peft_model
```

```
from trl import DPOConfig, DPOTrainer
import logging

BASE_SFT_CKPT = "microsoft/Phi-3-mini-4k-instruct"
DPO_DATA      = "training_data/dpo_it_help_desk_trainir
OUTPUT_DIR    = "phi3-mini-helpdesk-dpo"

# 1️⃣ Model + tokenizer
tok = AutoTokenizer.from_pretrained(BASE_SFT_CKPT, padc
                                    trust_remote_code=1

logger = logging.getLogger(__name__)
if not os.path.exists(BASE_SFT_CKPT):
    logger.warning(f'''Local path not found; will
        attempt to download {BASE_SFT_CKPT} from the Hu

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.bfloat16
)

base = AutoModelForCausalLM.from_pretrained(
    BASE_SFT_CKPT,
    device_map="auto",
    torch_dtype=torch.bfloat16,
    quantization_config=bnb_config
)

lora_cfg = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.05,
   target_modules=["q_proj", "k_proj", "v_proj", "o_prc
                   "up_proj", "down_proj"],
    bias="none",
    task_type="CAUSAL_LM",
)
model = get_peft_model(base, lora_cfg)
print("✅  Phi-3 loaded:", model.config.hidden_size, "
```

Next, we load our dataset containing ranked pairs. Each example includes a prompt, a preferred (“chosen”) response, and a less preferred (“rejected”)

response. This structure enables the model to learn which outputs to favor during training:

```
# Load DPO dataset with ranked pairs
#    Each row should include: {"prompt": ..., "chosen":
dataset = load_dataset("json",
    data_files="training_data/dpo_it_help_desk_training
    split="train")
```

With our data prepared, we define training hyperparameters and configure DPO. The `beta` parameter adjusts how strongly the model prioritizes the preferred response during optimization:

```
# 4️⃣ Trainer
train_args = TrainingArguments(
    output_dir      = OUTPUT_DIR,
    per_device_train_batch_size = 4,
    gradient_accumulation_steps = 4,
    learning_rate   = 5e-6,
    num_train_epochs= 3,
    logging_steps   = 10,
    save_strategy   = "epoch",
    bf16            = True,
    report_to       = None,
)

dpo_args = DPOConfig(
    output_dir              = "phi3-mini-helpdesk-dpo",
    per_device_train_batch_size  = 4,
    gradient_accumulation_steps  = 4,
    learning_rate           = 5e-6,
    num_train_epochs        = 3.0,
    bf16                    = True,
    logging_steps           = 10,
    save_strategy           = "epoch",
    report_to               = None,
    beta                    = 0.1,
    loss_type               = "sigmoid",
    label_smoothing         = 0.0,
    max_prompt_length       = 4096,
    max_completion_length   = 4096,
    max_length              = 8192,
    padding_value           = tok.pad_token_id,
```

```
        label_pad_token_id      = tok.pad_token_id,
        truncation_mode         = "keep_end",
        generate_during_eval    = False,
        disable_dropout         = False,
        reference_free          = True,
        model_init_kwargs       = None,
        ref_model_init_kwargs   = None,
    )

    trainer = DPOTrainer(
        model,
        ref_model=None,
        args=dpo_args,
        train_dataset=ds,
    )

    trainer.train()
    trainer.save_model()
    tok.save_pretrained(OUTPUT_DIR)
```

In summary, this script loads a base Phi-3 model with LoRA adapters, prepares a dataset of preference-ranked examples, and fine-tunes the model using `DPOTrainer`. After training, the model can produce higher-quality outputs that reflect your defined preferences more reliably than standard SFT alone.

DPO is especially useful when your primary goal is to shape output *quality* rather than simply replicate examples. It complements SFT by adding a preference-learning dimension, helping your agents produce outputs that are not only correct but also aligned with nuanced human expectations.

Reinforcement Learning with Verifiable Rewards

Building on preference-based fine-tuning, reinforcement learning with verifiable rewards (RLVR) introduces policy optimization against an explicit, measurable reward function.

Unlike preference-based approaches, RLVR enables you to connect any grader you can build—automated metrics, rule-based validators, external scoring models, or human evaluators—and directly optimize your model toward those rewards. This unlocks scalable, targeted improvement for virtually any task where you can define a verifiable evaluation signal.

Whether optimizing summarization quality, correctness of tool calls, factuality of knowledge retrieval, or even adherence to safety constraints, RLVR transforms static preference learning into a general, extensible reinforcement learning framework.

Unlike DPO, which directly optimizes for pairwise preferences, RLVR combines preference learning with reinforcement learning, enabling the model to generalize beyond observed rankings by predicting value scores and optimizing its outputs accordingly. Figure 7-8 illustrates the RLVR workflow, showing how models learn from graded completions to iteratively improve their performance on target tasks which then guide policy updates to produce outputs that maximize predicted quality and utility.

In the interest of readability, we will not include the full code for RLVR here, but it can be found in the accompanying repository for those who wish to implement it in practice.

Benefits of RLVR include its flexibility to optimize against any measurable signal, its ability to generalize beyond observed examples through value prediction, and its suitability for tasks where automated grading or scalable human evaluation is available. RLVR is particularly effective when you have ranked preference data or when you can build a reliable scoring function to evaluate outputs. It is ideal for scenarios requiring continual quality improvement, especially when rewards are sparse or evaluation is too costly to obtain at scale through direct human labeling alone.

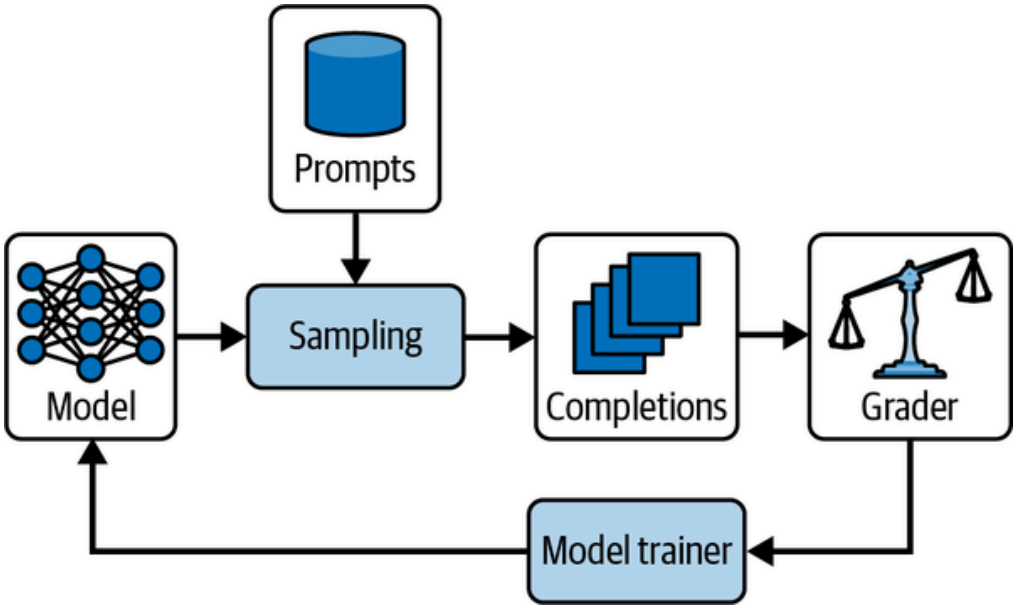

Figure 7-8. Reinforcement fine-tuning with verifiable rewards. Prompts are sampled to generate multiple completions, which are evaluated by a grader (either automated or human). These rewards are fed into the model trainer to update the policy, improving future outputs based on observed performance.

In summary, RLVR expands the possibilities of RFT by combining preference learning with value-based policy optimization. This allows your models not just to imitate preferred outputs, but to predict and optimize for what will be most useful, accurate, or aligned—paving the way for self-improving, task-specialized foundation models.

Conclusion

Learning in agentic systems encompasses a variety of approaches, each offering distinct advantages for improving performance and adaptability. Nonparametric learning enables agents to learn dynamically from experience without modifying underlying model parameters, emphasizing simplicity, speed, and real-world responsiveness. Parametric learning, by contrast, directly fine-tunes model weights to achieve deeper specialization—whether through supervised fine-tuning for structured outputs and function calling, or through direct preference optimization to shape output quality according to nuanced human judgments. Together, these learning methods form a powerful toolkit. By combining nonparametric agility with targeted parametric adaptation, developers can create intelligent, robust agents capable of evolving alongside changing tasks and environments—while ensuring each investment in learning aligns with operational constraints and performance goals.