AI Agents — Appendices

Michael Bücker Michael Hewing

Table of contents

Preface

These are the appendices to *AI Agents: Designing, Orchestrating, and Governing LLM-Based Systems* by Michael Bücker and Michael Hewing.

They are published here rather than in the printed book, so that the parts that age fastest, the setup steps, the package versions, and the provider-specific API details, can be corrected while the edition is in use.

Appendix A, Python Setup is the one to read first if you want to run the book's code. It covers installing Python and `uv`, the project's dependencies, environment variables and API keys, and running a first example.

Appendix B, Python and API basics is a reference rather than a full course: enough Python to read, run, and adapt the book's listings, and enough about web APIs and JSON to follow the tool-calling, retrieval, and orchestration examples that depend on them.

Appendix C, Deep learning sits one level below the models chapter, covering the vector and matrix operations, network structure, and training dynamics that the book uses without pausing to define.

The runnable code from the book, one notebook per chapter, is at https://github.com/ai-agents-book/code.

A. Python Setup

This appendix explains how you can set up Python and all required libraries so that you can run the code examples from this book on your own machine.

The goal is that you can read a listing, run it, change it, and see what changes — using the same environment the examples were written and executed in, so that the output you get matches the output printed in the text.

We assume basic familiarity with the command line (Terminal on macOS / Linux, PowerShell on Windows).

A.1. What You Need

To follow along, you should have:

- A computer with **Windows 10+, macOS, or a recent Linux distribution**
- **Python 3.11 or 3.12**
- The **uv** tool for managing environments and dependencies
- A text editor or IDE (VS Code, PyCharm, etc.)

A.1.1. Check Your Python Version

```
python --version
```

If the output shows a version below `3.11`, please install a newer Python release from the official Python website or via your system's package manager.

A.1.2. Install uv

Follow the instructions on the `uv` website for your platform to install the `uv` command. After installation, verify:

```
uv --version
```

If this prints a version number without an error, `uv` is ready.

A.2. Getting the Example Code

All code used in this book is available in a public Git repository at https://github.com/ai-agents-book/code.

```
git clone https://github.com/ai-agents-book/code
cd code
```

In the remainder of this appendix, we assume that your current working directory is this project folder.

A.3. Project Configuration with uv (pyproject.toml)

The dependencies for all examples are defined in a standard `pyproject.toml` file, which is understood by `uv`.

```
[project]
name = "agentic-ai"
version = "0.1.0"
description = "Python environment to execute code from 'AI Agents' by Michael Bücker and Mic
readme = "README.md"
requires-python = ">=3.11"
dependencies = [
    "dotenv>=0.9.9",
    "fastapi>=0.139.0",
    "fastmcp>=2.13.0",
    "jupyter>=1.1.1",
    "langgraph>=1.2.7",
    "matplotlib>=3.10.7",
    "nltk>=3.9.2",
    "numpy>=2.3.5",
    "openai>=2.9.0",
    "pandas>=3.0.3",
    "pip>=25.3",
    "pydantic>=2.12.5",
    "requests>=2.32.5",
    "scikit-learn>=1.8.0",
    "spacy>=3.8.11",
    "sqlalchemy>=2.0.51",
    "textblob>=0.19.0",
    "torch>=2.12.1",
    "transformers>=4.57.3",
]
```

Most of these libraries support a single chapter's examples: `fastmcp` implements the Model Context Protocol server and client used in the tools chapter, `langgraph` supports the orchestration examples, and `pandas`, `numpy`, and `sqlalchemy` back the data-handling code in later chapters. The important part is that **you do not have to install packages manually**, `uv` uses this file as the single source of truth, and any future addition to it is picked up the next time you run `uv sync`.

A.4. Creating and Using the Python Environment with uv

With `pyproject.toml` in place, the easiest way to get a matching Python environment is:

```
# Create (or update) a virtual environment based on pyproject.toml
uv sync

# Optionally: activate the virtual environment (recommended)
source .venv/bin/activate        # macOS / Linux
# .venv\Scripts\activate         # Windows (PowerShell)
```

What this does:

- `uv sync` creates a virtual environment (by default in `.venv/`) and installs all required packages according to `pyproject.toml`.
- Activating `.venv` ensures that when you run `python`, the correct interpreter and libraries are used.

From now on, you should see something like `(.venv)` at the beginning of your shell prompt.

You can confirm that you are using the right Python:

```
which python      # macOS / Linux
# or
where python      # Windows
```

The path should point to the `.venv` directory.

A.5. Optional: Jupyter for Interactive Exploration

`jupyter` is already listed in `dependencies`, so `uv sync` installs it along with everything else. Start it with:

```
uv run jupyter lab
```

or:

```
uv run jupyter notebook
```

This will run Jupyter using the same environment that `uv sync` created. You can then open the notebooks or create new ones to experiment with the code from the book.

A.6. API Keys and Environment Variables

The LLM examples throughout this book call a chat completions API through the standard `openai` Python package. This client interface is not tied to a single vendor: the same method calls work against OpenAI directly, against a self-hosted model server, and against any other provider implementing the OpenAI-compatible API standard, so you are not restricted to the exact setup used while writing the book. The code as

printed constructs the client with no arguments and takes its configuration from the environment, so the same listings address whichever endpoint you point them at. The web-search example in the tools chapter additionally calls Serper.dev. You must provide your own credentials for whichever provider you use.

We **never** hard-code keys into the code. Instead, we read them from environment variables, typically loaded from a local `.env` file.

A.6.1. Step 1: Create a `.env` File

In the project root folder, create a `.env` file and insert your own credentials. To run the examples exactly as printed:

```
OPENAI_API_KEY="..."
OPENAI_BASE_URL="..."     # omit when calling OpenAI itself
CHAT_MODEL="..."
EMBED_MODEL="..."
SERPER_API_KEY="..."
```

The listings deliberately carry no default model, so the book never pins a choice that providers retire on their own schedule. Set `CHAT_MODEL` and `EMBED_MODEL` to identifiers your provider actually serves; naming schemes differ from one provider to the next even for the same underlying model. `OPENAI_BASE_URL` is needed only when you call something other than OpenAI itself, such as a self-hosted model server or a gateway placed in front of one.

> **Which model produced the printed outputs**
>
> The outputs shown throughout this book were generated with GPT-4o. A different model will word its answers differently and may make different tool-calling choices; what should carry over is the structure

> of each result, not its exact phrasing. Where an example depends on a specific capability rather than on wording, the text says so.

The `.env` file stays on your machine and is **ignored by Git**, so your credentials are not accidentally published.

A.6.2. Step 2: Loading Environment Variables in Python

The examples use the `python-dotenv` package to load environment variables. This is the client construction exactly as the chapters print it:

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

# Load variables from the .env file into the environment
load_dotenv()

client = OpenAI()

CHAT_MODEL = os.environ["CHAT_MODEL"]
EMBED_MODEL = os.environ["EMBED_MODEL"]
```

`OpenAI()` takes no arguments here because it reads `OPENAI_API_KEY` from the environment on its own, and `OPENAI_BASE_URL` whenever that is set. The same three lines therefore reach OpenAI itself, a self-hosted model server, or a gateway in front of either, without any change to the code. The core idea holds regardless of which provider a given example calls: **configuration via environment variables, not via hard-coded strings.**

A.6.3. Using Azure OpenAI

Organizations frequently permit Azure OpenAI and nothing else. Azure addresses deployments rather than models and pins an API version, so it needs a different client class — but everything past client construction (tool schemas, message handling, the tool-calling loop) stays identical, because it relies only on the standard chat completions interface.

Replace the client construction above with:

```
import os
from dotenv import load_dotenv
from openai import AzureOpenAI

load_dotenv()

client = AzureOpenAI(
    azure_endpoint=os.getenv("AZURE_OPENAI_ENDPOINT"),
    api_key=os.getenv("AZURE_OPENAI_API_KEY"),
    api_version="<current-api-version>",
)

CHAT_MODEL = os.environ["CHAT_MODEL"]
EMBED_MODEL = os.environ["EMBED_MODEL"]
```

and add the corresponding variables to your `.env`:

```
AZURE_OPENAI_ENDPOINT="https://<your-resource>.openai.azure.com/"
AZURE_OPENAI_API_KEY="..."
```

Here `CHAT_MODEL` and `EMBED_MODEL` hold *deployment* names, which you choose when creating the deployment and which need not match the underlying model's name. The API version is account-specific and changes

on the provider's schedule, which is why it appears as a placeholder rather than a fixed value.

A.6.4. Alternative: Setting Environment Variables in the Shell

If you prefer not to use a `.env` file, you can export the variables directly in your shell session before running a script:

```
export OPENAI_API_KEY="..."
export CHAT_MODEL="..."
export EMBED_MODEL="..."

python examples/01_first_agent.py
```

On Windows PowerShell, use:

```
$env:OPENAI_API_KEY = "..."
$env:CHAT_MODEL = "..."
$env:EMBED_MODEL = "..."
python examples/01_first_agent.py
```

A.7. Running an Example

Once your environment is set up (`uv sync` run, `.venv` activated, `.env` configured), you can run any example script, for instance:

```
python examples/01_first_agent.py
```

If the script runs without import errors and the API calls succeed, your setup matches the one used for this book and you are ready to explore the other chapters.

B. Python and API basics

This appendix is a reference, not a full Python course. Its goal is narrow: enough Python to read, run, and adapt the code listings in this book, and enough about web APIs and JSON to follow the tool-calling, retrieval, and orchestration examples that depend on them. Coverage is deliberately shaped by what the book's own code actually uses, rather than by a generic language tour — constructs that never appear in a listing (e.g., operator overloading, metaclasses) are left out, while constructs that recur throughout (type hints, dictionaries that mirror JSON, decorators, `try/except`) get dedicated treatment. For installing Python and the packages used in this book, see Appendix A.

B.1. Python as an interpreted language

Python is commonly described as an interpreted language. In practice, a typical implementation (CPython) compiles source code into bytecode, which is then executed by a virtual machine. The operational consequence is that programs can be run interactively, executed incrementally, and developed with short edit-run cycles — the same code cell can be re-run after a small edit without a separate build step.

Compiled language toolchains translate source code into machine code before execution. This separation frequently enables stronger ahead-of-time optimization and reduces runtime overhead, but it also changes the development workflow: compilation becomes a distinct build step, and interactive execution requires additional infrastructure.

B.2. Working environments

Python programs are commonly written and executed in an integrated development environment (IDE), or in notebook systems that combine code and narrative text — the format used throughout this book.

Jupyter Notebook environments are frequently used for exploratory work, reproducible analyses, and documentation that interleaves prose with executable code. Notebook execution is stateful: cells can be executed out of order, which makes it necessary to manage dependencies between cells explicitly when producing reproducible results. Every executable listing in this book is a notebook cell in exactly this sense: later cells assume that earlier cells in the same chapter have already run.

For projects that involve multiple files and larger codebases, an IDE often provides stronger navigation, refactoring support, and debugging capabilities. A practical workflow uses notebooks for experimentation and moves stable components into modules that can be imported in both notebooks and scripts.

B.3. Language fundamentals

Python is case-sensitive, uses indentation to structure blocks, and is dynamically typed. Dynamic typing means that the type of a variable is associated with the runtime value rather than with an explicit declaration.

Indentation is part of the syntax and replaces block delimiters such as braces. A consistent indentation style is required, since inconsistent indentation can cause syntax errors and subtle behavioral changes.

B.3.1. Getting help interactively

The built-in functions `help()` and `dir()` provide documentation and introspection information in interactive sessions.

```
help(abs)
dir(str)
```

B.3.2. Variables and basic data types

Variables bind names to objects. Assignments introduce or update a binding, and the runtime determines the type from the assigned value.

```
x = 10
x = "Hello"
```

Basic data types include integers, floating-point numbers, Booleans, strings, and the special value `None`. `None` is Python's explicit absence-of-a-value marker and appears constantly in this book's code wherever a field is optional — a tool argument that was not supplied, a cache lookup that missed, a step that produced no result.

```
x = 3
pi = 3.14
flag = True
text = "Python"
missing = None
type(x), type(text), type(missing)
```

Identity testing with `is` is distinct from equality testing. Identity checks whether two references point to the same object; `is None` is the idiomatic way to test for absence, preferred over `== None`. Type checks are usually expressed with `isinstance`.

```
x = 3.14
isinstance(x, float)
```

B.3.3. Type hints

Type hints annotate variables, function parameters, and return values with the type they are expected to hold. The interpreter does not enforce them at runtime; they exist for readers, editors, and separate static-analysis tools. This book's function signatures use them throughout, because a hinted signature documents a tool's or a function's contract without requiring a separate specification.

```
def add(x: int, y: int) -> int:
    return x + y
```

Container types are parameterized with the type of their elements or entries: `list[str]` is a list of strings, `dict[str, float]` maps strings to floats. `Optional[str]` (or, equivalently, `str | None`) marks a value that may be absent.

```
from typing import Optional

def lookup_price(item: str, catalog: dict[str, float]) -> Optional[float]:
    return catalog.get(item)
```

B.3.4. Strings and formatting

Strings are sequences of Unicode characters. Indexing and slicing operations produce substrings without modifying the original value.

```
name = "Alice"
name[0]
name[1:4]
```

An f-string (formatted string literal) embeds expressions inside string templates. This is the dominant way this book constructs prompts and formats output, since it keeps the template and the interpolated values in one readable line.

```
name = "Alice"
age = 30
msg = f"Name: {name}, age: {age}"
msg
```

```
price = 19.99
qty = 3
total = price * qty
f"Total: {total:.2f}"
```

A handful of string methods account for most of the text handling in this book's code: `.split()` breaks a string into a list on whitespace or a given separator, `.join()` is the reverse, concatenating a list of strings with a separator between them, `.strip()` removes leading and trailing whitespace, `.lower()` normalizes case for comparison, and `.startswith()` tests a prefix.

```
line = "  §4.2 Refunds are issued within 5 business days  "
line.strip().lower().startswith("§4.2")
```

```
parts = "policy,refund,shipping".split(",")
", ".join(parts)
```

Exercise: strings

Create `sentence = "I am learning Python"`. Compute its length, extract the substring `"Python"` via slicing, and replace `"Python"` with `"APIs"`.

B.3.5. Collections and indexing

Python provides several built-in collection types. Lists are ordered and mutable, tuples are ordered and immutable, dictionaries map keys to values, and sets represent unordered collections of unique elements. Dictionaries are the most important of these for this book: a Python dictionary and a JSON object have the same shape, and tool arguments, API payloads, and configuration are all passed around as dictionaries.

```
values = [10, 20, 30, 40]
pair = (10, 20)
record = {"name": "Alice", "age": 30}
unique = {1, 2, 3}
values[0], pair[1], record["name"]
```

The dictionary method `.get(key, default)` looks up a key without raising an exception when it is missing, which is safer than `[key]` for data whose shape is not fully guaranteed — a common situation when the dictionary came from a retrieved document or an API response.

```
record.get("email", "not provided")
```

Sets do not support positional indexing. Membership tests express the most common access pattern.

```
10 in {10, 20, 30}
```

> Exercise: list indexing
>
> Create a list that contains all even numbers from 2 to 20 and print the seventh element.

B.3.6. Unpacking with * and **

The ** operator spreads a dictionary's entries as keyword arguments at a function call. This single idiom is how every tool-calling example in this book turns a parsed JSON object into an ordinary function call: an LLM's tool call arrives as a JSON string, `json.loads` turns it into a dictionary, and ** turns that dictionary into the keyword arguments the underlying Python function expects.

```
def add_numbers(x, y):
    return x + y

args = {"x": 13, "y": 29}
add_numbers(**args)   # identical to add_numbers(x=13, y=29)
```

The single * operator does the same for a list or tuple, spreading its elements as positional arguments. Both operators also work in the other direction, inside a function *definition*: `*args` collects any extra positional arguments into a tuple, and `**kwargs` collects any extra keyword arguments into a dictionary. This is what lets a decorator's wrapper function, shown below, accept and forward arguments for a function it has never seen.

```
def describe(*args, **kwargs):
    return args, kwargs

describe(1, 2, city="Berlin")
```

B.3.7. Comprehensions

A comprehension builds a new list, dictionary, or set from an existing iterable in a single expression, instead of appending to an empty collection inside an explicit loop. This book uses list comprehensions routinely, for instance to pull one field out of a list of records returned by an API or a retrieval step.

```
words = ["alpha", "beta", "gamma", "delta"]
lengths = [len(w) for w in words]
long_words = [w for w in words if len(w) > 4]
lengths, long_words
```

A comprehension is a direct replacement for the equivalent `for` loop below — more compact, but expressing exactly the same computation.

```
long_words = []
for w in words:
    if len(w) > 4:
        long_words.append(w)
long_words
```

B.4. Control structures

Conditional statements select code paths based on Boolean expressions. The `if` / `elif` / `else` chain evaluates conditions in order and executes the

first matching block.

```
x = 5
if x > 10:
    label = "greater than 10"
elif x == 10:
    label = "equal to 10"
else:
    label = "less than 10"
label
```

Loops perform repeated execution. A `for` loop iterates over items in an iterable. A `while` loop repeats while a condition holds. Termination conditions must be explicit to avoid infinite loops — a concern that reappears in the agent flow chapter, where an agent loop's stopping condition plays exactly this role.

```
fruits = ["Apple", "Banana", "Cherry"]
for fruit in fruits:
    print(fruit)
```

Two built-ins extend the plain `for` loop and appear throughout this book's plotting and data-preparation code. `enumerate` yields each item together with its position, avoiding a manually maintained counter; `zip` steps through two or more iterables in parallel, pairing up their elements.

```
words = ["alpha", "beta", "gamma"]
for i, word in enumerate(words):
    print(i, word)

lengths = [5, 4, 5]
for word, length in zip(words, lengths):
    print(f"{word}: {length}")
```

```
count = 0
while count < 3:
    print(count)
    count += 1
```

Exercise: conditional statements

Read an integer temperature in degrees Celsius from `input()` and print one of four labels: `"very cold"` (below 0), `"cold"` (0 to 15), `"pleasant"` (16 to 25), and `"hot"` (above 25).

```
temperature = int(input("Temperature (°C): "))
```

Exercise: loops (FizzBuzz)

Iterate from 1 to 20. Print `"Fizz"` for multiples of 3, `"Buzz"` for multiples of 5, `"FizzBuzz"` for multiples of both, and the number otherwise.

B.5. Functions and modules

Functions package reusable behavior behind a stable interface. Parameters declare inputs, and `return` provides a result. If no explicit return statement is present, the function returns `None`.

```
def add(x, y):
    return x + y

add(3, 4)
```

Docstrings attach structured documentation to objects. They can be accessed programmatically and extracted by documentation tools. In this book, a function's docstring frequently doubles as the natural-language description an LLM sees when the function is exposed as a tool.

```
def compute_area(width: float, height: float) -> float:
    """Compute the area of a rectangle."""
    return width * height

compute_area.__doc__
```

A lambda is a small, unnamed function defined inline, typically passed as an argument to another function — for instance, as the sort key below.

```
records = [{"name": "Bob", "age": 30}, {"name": "Anna", "age": 23}]
records_by_age = sorted(records, key=lambda r: r["age"])
records_by_age
```

Modules package functions, classes, and constants into importable units. The standard library provides functionality for common tasks such as mathematical computations, file I/O, and structured data processing.

```
import math
math.sqrt(16)
```

B.5.1. Decorators

A decorator is a function that wraps another function or method to add behavior without changing its body — logging, registration, or, as in the examples below, exposing a plain function as a web endpoint or as a tool an agent can call. `@app.post(...)` and `@mcp.tool(...)`, both used elsewhere in this book, are decorators in exactly this sense: the decorated

function is unchanged Python, but the decorator registers it with a framework.

```
def log_call(func):
    def wrapper(*args, **kwargs):
        print(f"calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log_call
def greet(name):
    return f"Hello, {name}"

greet("Alice")
```

A context manager, entered with `with`, pairs a setup and a teardown step, guaranteeing that the teardown runs even if the code inside raises an exception. `open()` is the canonical built-in example; a custom one is written with the `@contextmanager` decorator from `contextlib`.

```
from contextlib import contextmanager

@contextmanager
def timer(label):
    import time
    start = time.time()
    try:
        yield
    finally:
        print(f"{label}: {time.time() - start:.3f}s")

with timer("work"):
    total = sum(range(1_000_000))
```

B.5.2. A note on `async` and `await`

A function defined with `async def` is a coroutine: calling it does not run its body immediately but produces an awaitable object, which must itself be awaited (with `await`) inside another coroutine to actually execute and yield its result. The point is concurrency: while one coroutine is awaiting a slow operation such as a network call, the program can make progress on other work instead of blocking. `async with` pairs this with the context-manager pattern above, for a setup and teardown that themselves involve awaiting something — such as opening and closing a connection to a tool server.

```
import asyncio

async def fetch_value():
    await asyncio.sleep(0.1)
    return 42

asyncio.run(fetch_value())
```

This book uses `async`/`await` sparingly, in the MCP client and server examples of the tools chapter, where a tool call is naturally something to wait on. Elsewhere, functions are ordinary synchronous Python.

B.6. Error handling

A `try` block runs code that may raise an exception; matching `except` clauses catch specific exception types and decide how to recover. `finally` runs regardless of whether an exception occurred, making it the right place for cleanup. This is the standard shape for anything that can fail for reasons outside the program's control, most notably a network call to an external API.

```python
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
    finally:
        print("division attempted")

safe_divide(10, 0)
```

Catching a broad `Exception` hides bugs that a narrower `except` would surface; this book's API-calling examples catch specific exceptions (a timeout, a connection error) rather than every possible failure, so that an unexpected bug does not silently disappear into an error handler meant for network conditions.

B.7. Classes and structured data

Classes define user-defined types. Instances encapsulate state (attributes) and behavior (methods). Object orientation in Python supports encapsulation, inheritance, and polymorphism as common mechanisms for organizing code.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name}"
```

```
person = Person("Alice", 30)
person.greet()
```

Inheritance is expressed by defining a subclass that names its parent class. The `super()` helper delegates initialization and method resolution to parent classes.

```
class Vehicle:
    def __init__(self, maker, year):
        self.maker = maker
        self.year = year

    def drive(self):
        return f"{self.maker} vehicle is driving"


class Car(Vehicle):
    def __init__(self, maker, year, seats):
        super().__init__(maker, year)
        self.seats = seats

    def drive(self):
        return f"{self.maker} car is driving"


Car("BMW", 2021, 5).drive()
```

B.7.1. Data classes and Pydantic models

A plain class written as above needs an explicit `__init__` for every attribute, which is repetitive for classes that only hold data. The

`@dataclass` decorator generates that boilerplate from a list of typed attributes.

```
from dataclasses import dataclass

@dataclass
class ToolResult:
    status: str
    value: float

ToolResult(status="ok", value=42.0)
```

Pydantic models go one step further: like a data class, a `BaseModel` subclass declares its fields with type hints, but it additionally validates values at construction time and can convert directly to and from JSON. This book uses Pydantic models to define the input and output shape of a web endpoint (the integration chapter) and, under the hood, to define tool argument schemas — the same declaration serves as documentation, as a runtime check, and as the schema an LLM is shown.

```
from pydantic import BaseModel

class TriageResult(BaseModel):
    category: str
    confidence: float
    escalate: bool = False

result = TriageResult(category="refund", confidence=0.87)
result.model_dump()
```

If a field is given a value of the wrong type, Pydantic raises a validation error immediately rather than allowing a malformed object to propagate further into the program — the same guarantee a tool schema is meant to provide for an LLM's arguments.

B.8. Working with JSON

JavaScript Object Notation (JSON) is a text-based format for serializing structured values: JSON objects encode key-value mappings, JSON arrays encode ordered lists, and values are limited to numbers, strings, Booleans, `null`, arrays, and other objects. JSON has no dedicated syntax for anything else — no dates, no sets, no custom classes — which is part of why it is a safe interchange format between systems that do not share a type system.

In Python, the structural match is exact: a JSON object is a dictionary, a JSON array is a list, and `null` is `None`. This correspondence is why JSON shows up constantly in this book, in tool call arguments, API request and response bodies, and configuration files — in each case, the JSON on the wire and the Python dictionary in memory are the same data, just at different stages of being read or written.

```
import json

payload = {
    "name": "Alice",
    "age": 25,
    "tags": ["premium", "beta-tester"],
    "address": None,
}
text = json.dumps(payload)
text
```

`json.dumps` converts a Python object to a JSON string; `json.loads` does the reverse. Round-tripping a value through both functions reproduces the original dictionary — the basis for testing that a tool call's arguments or an API's response were parsed correctly.

```
roundtrip = json.loads(text)
roundtrip == payload
```

`json.dumps(..., indent=2)` produces the pretty-printed, multi-line form used for readability in this book's own listings; without `indent`, the same call produces a single compact line suited for sending over a network.

```
print(json.dumps(payload, indent=2))
```

Nesting follows the same rules recursively: a value inside a JSON object can itself be an object or an array, which is how a single tool call argument or API response can carry an entire structured record rather than one flat value.

```
tool_call = {
    "name": "get_weather",
    "arguments": {
        "city": "Berlin",
        "unit": "celsius",
        "forecast_days": [1, 2, 3],
    },
}
json.loads(json.dumps(tool_call))["arguments"]["city"]
```

Exercise: JSON round-trip

Build a Python dictionary describing a book (title, author, and a list of chapter titles). Serialize it with `json.dumps`, parse it back with `json.loads`, and confirm the parsed chapter list has the same length as the original.

B.9. Calling web APIs from Python

An Application Programming Interface (API) specifies how software components interact. A web API is an API accessed over a network, typically by sending HTTP requests to endpoints and receiving responses that contain status information and a JSON payload — the OpenAI, Azure, and other model-provider calls used throughout this book are all web APIs in this sense. Figure B.1 summarizes the exchange: a client constructs a request with a method, a URL, and headers; a server validates the request, performs application logic, and returns a response with a status code and a body.

Figure B.1.: HTTP request and response exchange.

The client-server model separates request initiation from request handling. Statelessness is a common design goal: a single request contains all the information required for the server to process it, rather than relying on server-side session state, which is part of what makes it possible to scale a service across many independent server processes.

B.9.1. HTTP methods and status codes

HTTP methods determine the action performed on a resource. API designs often map method semantics to resource operations, but the mapping

is a convention and not a guarantee. Table B.1 lists the methods with their safety and idempotency properties.

Table B.1.: HTTP method properties used in API design.

Method	Safe	Idempotent	Typical use
`GET`	yes	yes	retrieve representation
`POST`	no	no	create subordinate resource
`PUT`	no	yes	replace resource state
`PATCH`	no	not required	partial update
`DELETE`	no	yes	delete resource

Safe methods are intended not to change server state; `GET` is the primary safe method. Idempotency describes whether repeating a request yields the same effect as sending it once — relevant when a client must decide whether it is safe to retry a failed call.

HTTP status codes describe the outcome of a request and are grouped by their first digit. Table B.2 gives the groups and the codes an API client meets most often.

Table B.2.: Frequently encountered status codes in API clients.

Code	Name	Typical meaning
200	OK	request succeeded
201	Created	resource created
204	No Content	success without body
400	Bad Request	invalid request
401	Unauthorized	authentication missing
403	Forbidden	insufficient permissions
404	Not Found	resource missing
409	Conflict	incompatible state
429	Too Many Requests	rate limit hit

Code	Name	Typical meaning
500	Internal Server Error	server failure
503	Service Unavailable	temporary outage

B.9.2. URLs and query parameters

A URL identifies a resource and can include additional parameters. Query parameters are appended after `?` and are commonly used for filtering, pagination, and field selection. Table B.3 names each part of a URL and what it carries.

Table B.3.: URL components in web APIs.

Component	Example	Role
scheme	`https`	transport and security
host	`api.example.com`	service address
path	`/v1/items/42`	resource selection
query	`?limit=10&offset=0`	parameterization

B.9.3. The `requests` library

The `requests` library provides a high-level interface for sending HTTP requests and processing responses. Network calls should use explicit timeouts, since the default behavior may block indefinitely; an API that stops responding should not be allowed to stop the calling program along with it.

```
import requests

base_url = "https://jsonplaceholder.typicode.com"
```

```
resp = requests.get(f"{base_url}/posts", params={"userId": 1}, timeout=10)
resp.status_code, resp.json()[0]
```

`params` handles query-string encoding; `json=payload` on a `POST` serializes a Python dictionary to a JSON body and sets the matching content-type header.

```
payload = {"title": "foo", "body": "bar", "userId": 1}
resp = requests.post(f"{base_url}/posts", json=payload, timeout=10)
resp.status_code, resp.json()
```

Repeated calls to the same service should reuse a `requests.Session()`, which maintains connection pooling and shared configuration such as default headers across calls — the same pattern used implicitly by the `OpenAI` client throughout this book.

```
sess = requests.Session()
sess.headers.update({"Accept": "application/json"})
resp = sess.get(f"{base_url}/posts", timeout=10)
resp.status_code
```

API clients should treat network failures and error responses as normal operating conditions, not exceptional ones. `raise_for_status()` turns an error status code into a Python exception, which can then be caught alongside timeout and connection errors.

```
try:
    resp = requests.get(f"{base_url}/posts", timeout=10)
    resp.raise_for_status()
except requests.Timeout:
    print("timeout")
except requests.RequestException as exc:
```

```
    print(f"request failed: {exc}")
else:
    print(len(resp.json()))
```

Retry logic is appropriate for transient conditions such as timeouts and temporary server failures, but retrying a non-idempotent request (a `POST`) can create duplicate side effects unless the endpoint supports an idempotency key. Rate limiting is typically signaled by status `429`, often with a `Retry-After` header indicating how long to back off before trying again.

B.9.4. Authentication and secrets

Authentication controls access to protected resources. A bearer token is the most common mechanism used by the APIs in this book, sent via the `Authorization` header.

```
token = "REDACTED"
headers = {"Authorization": f"Bearer {token}"}
```

Credentials and API keys must never be embedded in source code. The standard pattern loads them from environment variables, with `python-dotenv` populating those variables from a local `.env` file that is excluded from version control — the setup used by every API-calling listing in this book.

```
import os
from dotenv import load_dotenv

load_dotenv()

api_key = os.getenv("API_KEY")
api_key is not None
```

Configuration via environment variables keeps credentials out of the repository and lets different environments (a laptop, a CI pipeline, a production server) use different values without any code change.

B.9.5. Pagination

Collection endpoints frequently return lists and support pagination to bound response size. Offset-based pagination uses `limit` and `offset` parameters; cursor-based pagination returns an opaque token that must be supplied to fetch the next page, which stays stable even if items are inserted or removed between requests. Table B.4 compares the two schemes.

Table B.4.: Common pagination schemes in web APIs.

Pagination	Client parameters	Server response
offset	`limit`, `offset`	list
cursor	`limit`, `cursor`	list + cursor

B.10. Building APIs in Python

Web frameworks such as FastAPI implement server-side routing, validation, and serialization for JSON-based APIs, the framework behind the endpoint examples in the integration chapter. FastAPI builds on ASGI and integrates OpenAPI documentation generation automatically. A Pydantic model, introduced above, specifies the expected request body; FastAPI validates incoming JSON against it before the function body ever runs.

```
from fastapi import FastAPI
from pydantic import BaseModel
```

```
app = FastAPI()

class Item(BaseModel):
    name: str
    price: float

@app.post("/items")
async def create_item(item: Item):
    return item
```

A local test client executes requests against the application in the same process, without a network round trip, which is what makes an API endpoint straightforward to test alongside the rest of a codebase.

```
from fastapi.testclient import TestClient

client = TestClient(app)
resp = client.post("/items", json={"name": "widget", "price": 9.99})
resp.status_code, resp.json()
```

Exercise: Petstore API

Use the Petstore API hosted at `petstore.swagger.io` to practice `GET` and `POST` requests. Retrieve the pet with ID `1`, then create a new pet by sending a JSON body to the `/v2/pet` endpoint and retrieve the created resource again.

C. Deep learning

Deep learning refers to machine learning models built from compositions of differentiable functions that are trained by gradient-based optimization. The practical relevance for agentic systems lies in two roles: deep learning as the foundation for Large Language Models (LLMs), and deep learning components as learned modules inside larger pipelines that combine probabilistic inference with deterministic tools. This appendix starts one level below that: the vector and matrix operations that every layer, loss, and attention computation in this book is built from, but that the models chapter uses without pausing to define.

C.1. Vectors, matrices, and softmax

A vector is simply an ordered list of numbers. An embedding, as used throughout the models chapter, is a vector; so is a single training example's set of numeric features. Vectors of the same length can be combined element by element, most importantly through the dot product: multiply corresponding entries and sum the results.

```
import numpy as np

a = np.array([1.0, 2.0, 3.0])
b = np.array([4.0, 5.0, 6.0])
np.dot(a, b)
```

```
np.float64(32.0)
```

The dot product is the computational core of cosine similarity, defined and used in the models chapter to compare embedding vectors — that formula divides a dot product by the two vectors' magnitudes, but the dot product itself is exactly the sum computed above.

A matrix is a rectangular array of numbers, and a matrix-vector product generalizes the dot product: each row of the matrix is dotted with the input vector, producing one output number per row. This is precisely what a neural network layer computes. `nn.Linear(10, 64)`, used below, stores a 64×10 weight matrix W and a bias vector b, and maps an input vector x to $Wx + b$ — sixty-four dot products, one per output dimension, computed in a single operation. Figure C.1 shows this for a smaller, concrete example: the first row of W is highlighted and traced through to the first output entry, since that entry is nothing more than that row's dot product with x — the same operation as the cosine-similarity numerator above, repeated once per row.

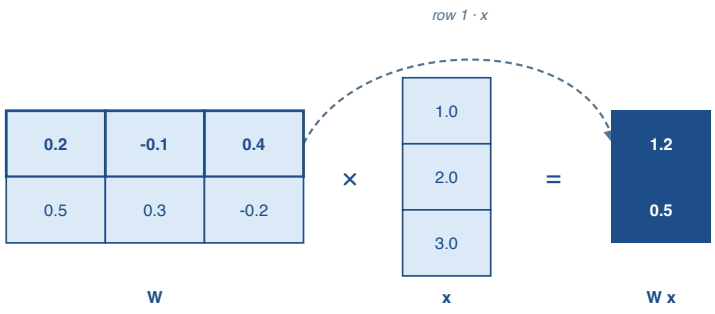

Figure C.1.: Matrix-vector product. Each output entry is the dot product of one row of the matrix with the input vector; the highlighted first row is traced explicitly.

```
W = np.array([[0.2, -0.1, 0.4], [0.5, 0.3, -0.2]])   # 2x3 weight matrix
x = np.array([1.0, 2.0, 3.0])                         # 3-dim input
W @ x                                                 # 2-dim output
```

```
array([1.2, 0.5])
```

Real training never processes one input at a time. Stacking several input vectors as the rows of a matrix X and computing $XW^\top$ applies the same matrix-vector product to every row at once — a matrix-matrix product is simply a batch of matrix-vector products, computed together. The training loop later in this appendix creates `x = torch.randn(32, 10)`, a batch of 32 examples, for exactly this reason: the `32` is the batch dimension this operation processes in parallel.

```
X = np.array([[1.0, 2.0, 3.0], [0.0, 1.0, -1.0]])   # a batch of 2 inputs
X @ W.T                                              # 2x2 output: one row per input
```

```
array([[ 1.2,  0.5],
       [-0.5,  0.5]])
```

The first row of `X` is exactly the `x` used above, and its output row reproduces the earlier result — batching changes nothing about what gets computed per example, only how many examples are computed at once.

Softmax turns a vector of arbitrary real-valued scores into a probability distribution: it exponentiates every entry, then divides by the sum, so the results are all positive and add up to 1. The models chapter uses exactly this operation twice — to turn a language model's output scores into a distribution over the next token, and to turn attention scores into weights over which positions to attend to.

```
def softmax(scores):
    exp_scores = np.exp(scores - np.max(scores))
    return exp_scores / exp_scores.sum()

scores = np.array([2.0, 1.0, 0.1])
probs = softmax(scores)
probs, probs.sum()
```

```
(array([0.65900114, 0.24243297, 0.09856589]), np.float64(1.0))
```

Subtracting the maximum score before exponentiating is a numerical safeguard: it keeps every exponent at or below zero, which avoids floating-point overflow, without changing the result, since the same constant appears in every term of the numerator and denominator. This is a different fix for a related problem to the scaling used in the models chapter's attention computation, which divides dot products by $\sqrt{d_k}$ *before* the softmax to keep the scores themselves from growing too large as embedding dimensionality increases — but both exist because softmax output is sensitive to the scale of its input.

Once scores become probabilities, cross-entropy loss (introduced below) measures how far that distribution is from the true answer: for a true class c, cross-entropy is simply $-\log(\text{probs}[c])$ — a small value when the model assigned high probability to the correct class, and a large one when it did not.

```
true_class = 0
-np.log(probs[true_class])
```

```
np.float64(0.4170300162778335)
```

Exercise: softmax by hand

Compute `softmax(np.array([1.0, 1.0, 1.0]))` and confirm all three probabilities are equal. Then compute `softmax(np.array([10.0, 1.0, 1.0]))` and observe how concentrated the distribution becomes on the largest score. What does this predict about a language model that is very confident in its next token compared to one that is uncertain among several candidates?

C.2. Machine learning problem setup

Machine Learning problems are commonly framed as the minimization of an expected loss. A dataset provides examples (x, y) drawn from an unknown distribution, and a model f_θ parameterized by θ predicts $\hat{y} = f_\theta(x)$. Training selects parameters that minimize an empirical objective, typically an average loss over a training set.

Supervised learning uses labeled targets y. Unsupervised learning uses objectives that do not require explicit targets, such as reconstruction or contrastive learning. Reinforcement Learning optimizes policies based on reward signals and environmental interaction.

Generalization denotes performance on unseen data. Overfitting occurs when a model fits idiosyncrasies of the training set and fails to generalize, while underfitting occurs when the model class cannot capture relevant structure.

C.3. Artificial neural networks

An artificial neural network is a parameterized function that maps inputs to outputs through layers. A layer applies an affine transformation — the

matrix-vector product $Wx + b$ introduced above — followed by a nonlinear activation. A multilayer perceptron (MLP) composes such layers to produce increasingly abstract representations. Figure C.2 shows a single unit performing exactly this computation: each input is multiplied by its own weight, the weighted inputs and a bias are summed — one row of W dotted with x, as in Figure C.1 — and the sum passes through a nonlinear activation, here a ReLU, which simply replaces any negative value with zero.

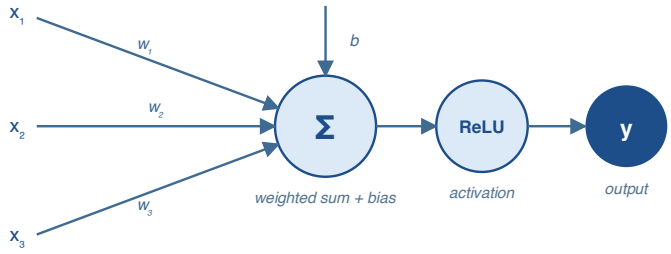

Figure C.2.: A single artificial neuron: a weighted sum of its inputs, plus a bias, passed through a nonlinear activation.

```
Sequential(
  (0): Linear(in_features=10, out_features=64, bias=True)
  (1): ReLU()
  (2): Linear(in_features=64, out_features=1, bias=True)
)
```

Listing C.1 stacks many such units into two fully connected layers: an input of 10 values maps to a hidden layer of 64 units with a ReLU activation each, which maps to a single output unit. Figure C.3 draws this topology directly — a representative subset of the input and hidden units,

Listing C.1 A minimal multilayer perceptron defined with PyTorch's `nn.Sequential`

```
import torch
from torch import nn

model = nn.Sequential(
    nn.Linear(10, 64),
    nn.ReLU(),
    nn.Linear(64, 1),
)
model
```

since drawing all 10 and all 64 would be unreadable, but every connection shown is a real weight the network learns.

Forward computation and parameter updates form a repeated cycle, shown in Figure C.4.

C.3.1. Loss functions and outputs

The loss function defines the learning signal. Mean squared error is common for regression. Cross-entropy is common for classification, and pairs naturally with the softmax introduced above: cross-entropy compares a predicted probability distribution to the true one, so a classifier's output layer typically produces logits that a softmax (often fused into the loss for numerical stability) turns into that distribution.

task	typical output	typical loss
regression	linear output	mean squared error
binary classification	logit	logistic loss
multi-class classification	logits	cross-entropy

task	typical output	typical loss

C.3.2. Backpropagation and gradient computation

Backpropagation computes gradients of the loss with respect to parameters efficiently by applying the chain rule to the computational graph — the backward arc in Figure C.4. Modern deep learning frameworks implement automatic differentiation, which provides gradients for arbitrary compositions of differentiable operations, including the matrix multiplications and softmax introduced above. The method is the standard training mechanism for deep networks in practice, with early formulations and analyses described in Rumelhart et al. (1986).

The chain rule itself is nothing more than multiplying local derivatives along a computational path. Take the simplest possible unit — a single weight and bias, $\hat{y} = wx + b$ — with a squared-error loss $L = (\hat{y} - y)^2$. The chain rule gives the gradient of L with respect to each parameter by differentiating through $\hat{y}$:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w} = 2(\hat{y} - y) \cdot x, \qquad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = 2(\hat{y} - y).$$

With $w = 1$, $b = 0$, $x = 2$, and a target $y = 5$: $\hat{y} = 2$, $\partial L/\partial w = 2(2 - 5)(2) = -12$, and $\partial L/\partial b = 2(2 - 5) = -6$. `loss.backward()` computes exactly this, via automatic differentiation rather than by hand, and should agree with the hand calculation to the last digit. Listing C.2 runs the comparison.

```
(tensor(-12.), tensor(-6.))
```

Every parameter in a network as large as an LLM is updated by the same mechanism: a longer chain of local derivatives, multiplied together, computed automatically instead of by hand — nothing about the underlying rule changes with scale.

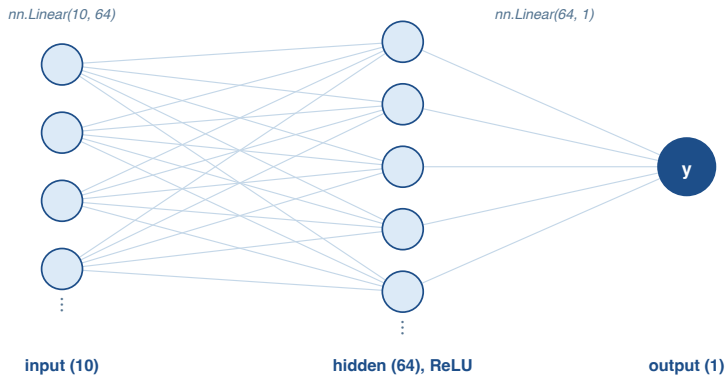

Figure C.3.: The topology of the multilayer perceptron defined in Listing C.1. Only a representative subset of the 10 input units and 64 hidden units is drawn; the output unit is the single value the network predicts.

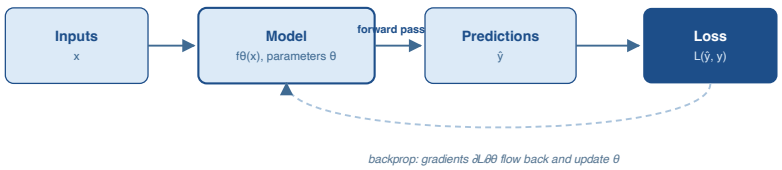

Figure C.4.: Forward and backward pass in gradient-based training. Inputs flow forward through the model to a prediction and a loss; backpropagation then flows the loss's gradient backward to update the parameters.

Listing C.2 Checking a hand-derived gradient against PyTorch's autograd

```
w = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(0.0, requires_grad=True)
x_ = torch.tensor(2.0)
target = torch.tensor(5.0)

y_hat = w * x_ + b
loss = (y_hat - target) ** 2
loss.backward()
w.grad, b.grad
```

C.4. Optimization and training dynamics

Gradient descent updates parameters by moving opposite to the gradient of the loss:

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta),$$

where η is the learning rate and $\nabla_\theta L(\theta)$ is the gradient computed by backpropagation — for the toy neuron above, this is literally `w = w - lr * w.grad` applied to every parameter at once. Stochastic gradient descent (SGD) approximates the gradient with minibatches of samples — the batches introduced earlier in this appendix — which reduces computation per update and introduces noise that can affect convergence and generalization.

The learning rate controls the update magnitude. Adaptive optimizers such as Adam maintain per-parameter step sizes based on moment estimates and are widely used in practice Kingma and Ba (2015). Listing C.3 runs the update cycle from Figure C.4 for real, eight times in a row, on a fixed batch of synthetic data — the loss printed after every step should decrease.

Listing C.3 A minimal training loop: repeated gradient updates on synthetic data

```
torch.manual_seed(0)
x = torch.randn(32, 10)
y = torch.randn(32, 1)

model = nn.Sequential(
    nn.Linear(10, 64),
    nn.ReLU(),
    nn.Linear(64, 1),
)
loss_fn = nn.MSELoss()
opt = torch.optim.Adam(model.parameters(), lr=1e-2)

for step in range(8):
    opt.zero_grad()
    pred = model(x)
    loss = loss_fn(pred, y)
    loss.backward()
    opt.step()
    print(f"step {step}: loss = {loss.item():.4f}")
```

```
step 0: loss = 1.9334
step 1: loss = 1.7661
step 2: loss = 1.6382
step 3: loss = 1.5352
step 4: loss = 1.4425
step 5: loss = 1.3545
step 6: loss = 1.2699
step 7: loss = 1.1895
```

Exercise: breaking the training loop

Rerun Listing C.3 with `lr=5.0` instead of `1e-2`, keeping everything else the same. What happens to the printed loss? Then try a very small learning rate such as `1e-5` over more steps. Relate what you see to why the learning rate is often called the single most consequential hyperparameter in training.

C.5. Regularization and generalization controls

Regularization constrains the effective capacity of the model or stabilizes learning. Weight decay penalizes large parameters and is often implemented as L_2 regularization. Dropout randomly masks activations during training to reduce co-adaptation Srivastava et al. (2014). Batch normalization normalizes intermediate activations to reduce internal covariate shift and improve optimization Ioffe and Szegedy (2015).

Data augmentation is a complementary approach that expands the training distribution by applying label-preserving transformations, particularly in vision.

Note

Evaluation requires a strict separation between training, validation, and test sets. Data leakage can produce optimistic performance estimates that do not transfer to deployment settings.

C.6. Neural network architectures

Architecture choices encode inductive biases that match input structure. The models chapter traces the historical progression from n-gram models

through recurrent networks to Transformers; this section stays with the training-mechanics angle — what each family asks of the gradient-descent machinery introduced above.

Convolutional neural networks (CNNs) exploit spatial locality in grid-structured data such as images, through convolution and pooling layers; they became dominant for image classification after large-scale empirical results such as Krizhevsky et al. (2012), with deep residual networks addressing optimization issues in very deep CNN architectures through skip connections He et al. (2016). Neither vision nor CNNs are otherwise a topic in this book.

Recurrent neural networks (RNNs) model sequential structure by carrying a hidden state forward across positions, updating it one token at a time. Backpropagation through this recurrence tends to make gradients vanish or explode over long sequences; Long Short-Term Memory (LSTM) units use learned gates to mitigate the problem and support longer-range dependencies Hochreiter and Schmidhuber (1997). RNNs and LSTMs are trained with the exact same loss-gradient-update cycle as the MLP above, just unrolled across sequence positions, and were the dominant architecture for language modeling before attention.

Transformers replace recurrent state with self-attention, and self-attention is built entirely from the operations introduced at the start of this appendix: queries and keys are compared with dot products, the results are scaled and passed through softmax to obtain attention weights, and those weights combine the values — the mechanism worked through in full in the models chapter, including causal masking and multi-head attention. What matters for this appendix is that none of that changes the training mechanics: a Transformer's parameters are still updated by backpropagation and gradient descent exactly as in Listing C.3, at a much larger scale. The self-attention formulation was introduced in Vaswani et al. (2017) and underpins essentially every model discussed in this book.

C.7. Practical training workflow

Training is typically organized around a data pipeline, a model, an objective, an optimizer, and an evaluation protocol. Several choices materially affect reproducibility and comparability across runs:

- dataset splits and preprocessing
- parameter initialization and random seeds
- optimizer and learning-rate schedule
- early stopping criteria
- metric definitions and reporting

Checkpointing stores model parameters periodically. It supports failure recovery and enables model selection based on validation metrics. Mixed precision training can reduce memory usage and increase throughput on modern accelerators, but it may require numerical safeguards such as loss scaling.

C.8. Common failure modes

Training instabilities include exploding gradients, divergent loss due to an overly large learning rate — as seen in the exercise above — and sensitivity to initialization. Non-stationary data distributions, mislabeled data, and distribution shift between training and deployment often dominate model performance limitations in applied settings.

The general treatment of deep learning foundations, architectures, and optimization is provided in Goodfellow et al. (2016), while statistical learning perspectives on generalization and evaluation appear in Bishop (2006).

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning*. Springer. https://link.springer.com/book/9780387310732.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. https://mitpress.mit.edu/9780262035613/deep-learning/.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. https://doi.org/10.1109/CVPR.2016.90.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *Proceedings of the 32nd International Conference on Machine Learning (ICML)*. https://proceedings.mlr.press/v37/ioffe15.

Kingma, Diederik P., and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." *International Conference on Learning Representations (ICLR)*. https://doi.org/10.48550/arXiv.1412.6980.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36. https://doi.org/10.1038/323533a0.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent

Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (56): 1929–58. https://jmlr.org/papers/v15/srivastava14a.html.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Red Hook, NY, USA), NIPS'17, 6000–6010. https://doi.org/10.48550/arXiv.1706.03762.