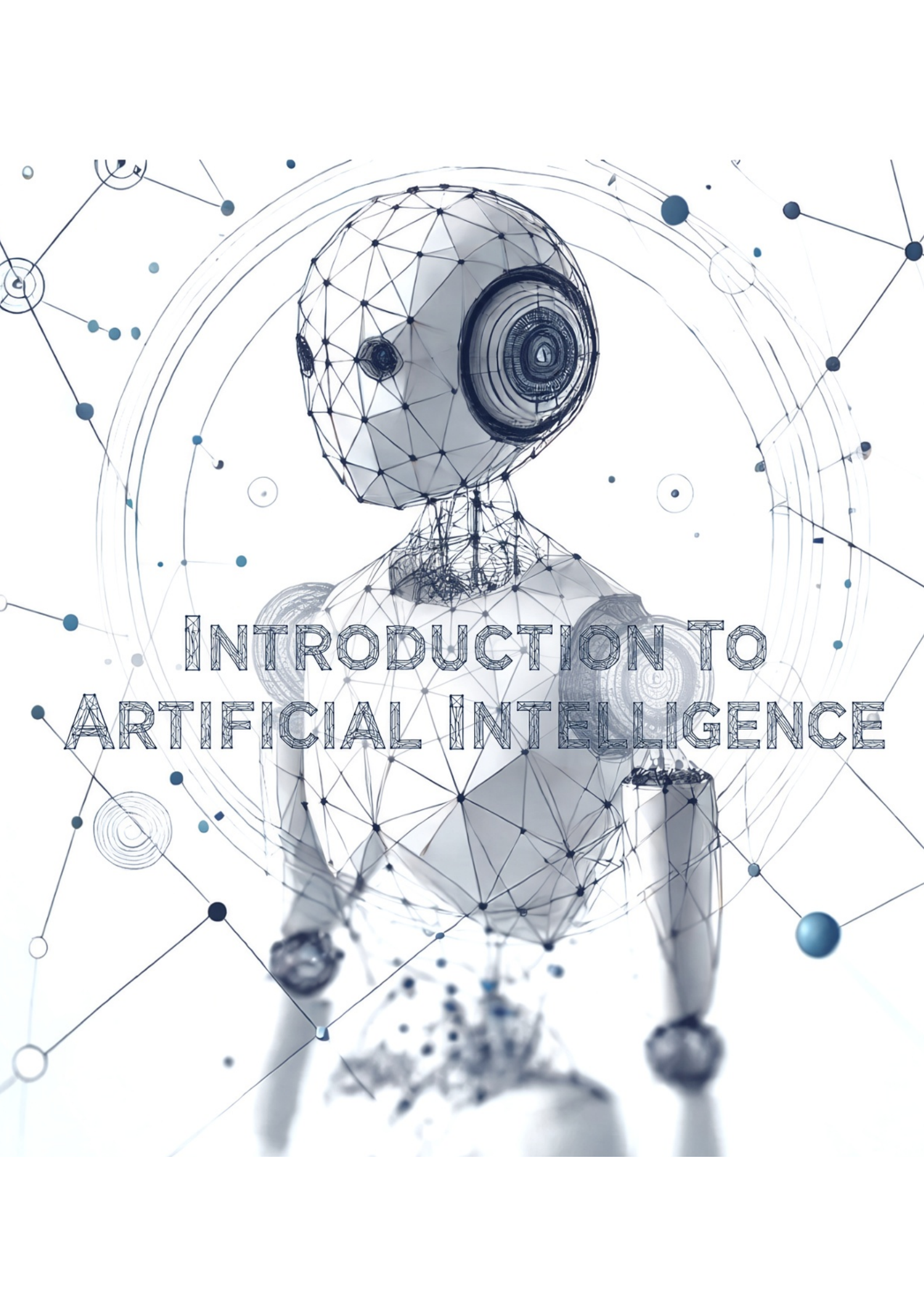




INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Preface

Artificial Intelligence is more important than ever before. It has become an integral part of many products and our daily lives, even replacing certain labor tasks. With the advent of powerful technologies like large language models, the field is evolving rapidly. In this book, we will explore fundamental algorithms, techniques, and key principles of AI, from machine learning and deep learning to large language models and agentic AI. A central focus is on the implementation and application of AI methods. We will not only cover the algorithmic foundations but also emphasize practical aspects, reinforcing understanding through hands-on exercises and real-world examples.

Oliver Kramer
Oldenburg, October 4, 2026

Contents

1	Introduction	1
1.1	What is AI?	1
1.2	Historical Overview of AI	1
1.3	How to Use This Book	2
1.4	Further Reading: AI Textbooks	3
1.5	Exercises	3
I	Machine Learning and Optimization	5
2	K-Nearest Neighbors	7
2.1	Supervised Learning	7
2.2	KNN	8
2.3	Hands-On KNN	10
2.4	Confusion Matrix	11
2.5	KNN in <code>Python</code>	12
2.6	Exercises	13
3	K-Means	15
3.1	Unsupervised Learning	15
3.2	K-Means	15
3.3	Hands-On K-Means	17
3.4	K-Means in <code>scikit-learn</code>	18
3.5	Exercises	18
4	Evolution Strategies	21
4.1	Optimization	21
4.2	(1 + 1)-ES	22
4.3	Hands-On ES	23
4.4	ES in <code>Python</code>	24
4.5	Exercises	24

II	Deep Learning	27
5	Multi-Layer Perceptrons	29
5.1	Perceptron	29
5.2	Multi-Layer Perceptrons	30
5.3	Hands-On Gradient Descent	31
5.4	MLP in <code>scikit-learn</code>	33
5.5	Exercises	33
6	Convolutional Neural Networks	35
6.1	Image Recognition	35
6.2	Convolutional Layers	35
6.3	Hands-On Convolution	37
6.4	Pooling	38
6.5	VGG-19	38
6.6	CNN in <code>Keras</code>	40
6.7	Exercises	41
7	Attention	43
7.1	Sequence Learning	43
7.2	Attention	44
7.3	Hands-On Attention	45
7.4	GPT	46
7.5	Hugging Face	47
7.6	Exercises	48
III	LLMs and Agents	51
8	Large Language Models	53
8.1	LLMs	53
8.2	Parameters and Precision	54
8.3	<code>Ollama</code>	54
8.4	Chain-of-Thought	55
8.5	A Chatbot in <code>Ollama</code>	57
8.6	Exercises	58
9	Retrieval-Augmented Generation	59
9.1	RAG	59
9.2	Documents and Chunks	60
9.3	Embeddings and Indexing	61
9.4	Retrieval	61

9.5	Context and Generation	62
9.6	RAG Example: Lecture Notes Assistant	63
9.7	Exercises	63
10	LoRA	65
10.1	LLM Training	65
10.2	Instruction Datasets	65
10.3	Fine-Tuning	66
10.4	Low-Rank Adaptation	67
10.5	Fine-Tuning with Hugging Face	67
10.6	Exercises	68
11	Agents	71
11.1	Agentic AI	71
11.2	Memory	72
11.3	Agent Learning and Model Learning	73
11.4	Tools	74
11.5	Agent Example: Muse Glimmer	74
11.6	Agent Swarms	75
11.7	Exercises	76
IV	Misc	77
12	Quo Vadis?	79
12.1	AI Safety	79
12.2	Singularity	80
12.3	Superintelligence	81
12.4	Consciousness	82
12.5	Exercises	83
V	Appendices	85
A	AI Benchmarks	87
A.1	ML Datasets	87
A.2	Optimization Problems	87
A.3	LLM Benchmarks	88
B	AI Math	89
B.1	Vectors	89

C Python Programming	91
C.1 Setup	91
C.2 Basic Data Types	91
C.3 Loops and Conditions	92
C.4 Core Libraries	92
Bibliography	95

Chapter 1

Introduction

1.1 What is AI?

Artificial Intelligence (AI) describes systems that learn, reason, and make decisions. Modern AI relies on deep learning and neural networks. These models handle tasks such as language processing, image recognition, and automated decision-making. Early AI used rule-based systems, but current progress comes from machine learning on large datasets. Large language models (LLMs) show recent advances. They use transformer networks to generate text, support programming, and solve reasoning tasks. Agentic AI extends these models. It connects LLMs with automation and performs multi-step tasks with minimal human input. AI systems now move from static behavior to adaptive and interactive operation. AI also operates beyond language. New methods create images, master complex games, and control robots. Research pushes AI into healthcare, finance, and scientific discovery. Its impact grows as it enters more domains. This book introduces machine learning with `scikit-learn`, `keras`, and LLMs via `ollama`. Each chapter provides practical examples and exercises for real applications.

1.2 Historical Overview of AI

The development of AI has progressed through several key epochs:

1. *1950s*: Foundations of AI: The Dartmouth Conference (1956) formally establishes the field; Alan Turing proposes the Turing Test; Frank Rosenblatt introduces the perceptron as the first trainable neural model.
2. *1960s*: Early symbolic AI: programs like ELIZA and the General Problem Solver; emergence of search and logic-based reasoning systems.

3. *1970s*: First AI Winter due to limited compute and overpromising; meanwhile, Ingo Rechenberg and Hans-Paul Schwefel develop Evolution Strategies (ES), and John Holland formalizes Genetic Algorithms (GAs).
4. *1980s*: Rise of expert systems; rediscovery of multi-layer neural networks (MLPs) and backpropagation; realization that XOR cannot be solved by single-layer perceptrons triggers renewed neural network research.
5. *1990s*: Statistical learning theory gains traction; Vladimir Vapnik introduces Support Vector Machines (SVMs); IBM's Deep Blue defeats Kasparov in 1997.
6. *2000s*: Data-driven machine learning dominates; kernel methods, SVMs, and ensemble methods such as boosting and random forests achieve strong performance.
7. *2010s*: Deep learning revolution: CNN breakthroughs with AlexNet; sequence modeling advances culminate in the Transformer architecture.
8. *2020s*: Generative AI era: Large Language Models (LLMs) scale Transformers to unprecedented capabilities in reasoning, coding, and multi-modal tasks.

1.3 How to Use This Book

This book is designed to blend theory with practice and guide you through Hands on exploration of AI methods. Along the way, you'll find useful visual cues:

1.  QR codes next to Python examples link to interactive Google Colab notebooks. You'll need a Google account to run and edit them online.
2.  This icon indicates links to recommended reading or further literature, helping you dive deeper into the topic.
3.  Each chapter concludes with a set of practical exercises to reinforce your understanding and encourage experimentation.

Info Boxes

Info Boxes provide concise explanations of key concepts and methods, offering quick insights that support the main text.

Action Boxes

Action boxes encourage active engagement with AI tools and information sources, motivating exploration beyond the core material.

1.4 Further Reading: AI Textbooks

Below is a selection of recommended textbooks for studying various aspects of AI:

1. *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani [9]: An introduction to statistical learning. 
2. *Dive into Deep Learning* by Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola [25]: A practical, interactive guide to deep learning with code examples in Python. 
3. *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig [18]: A widely used textbook that covers a broad range of AI topics, including search algorithms, knowledge representation, and planning.

There is a wide range of additional books and literature available for further reading.

1.5 Exercises



1. Questions:
 - (a) What is AI in your own words?
 - (b) Name important milestones in AI history.
 - (c) Find and explore a freely available online AI textbook.
2. Programming:
 - (a) Install Python and set up a programming environment using Conda.
 - (b) Try out Jupyter Notebook and Visual Studio Code. What are the pros and cons?
 - (c) Open Google Colab and run a short Python script like: `print("Hello, AI!")`

Towards Data Science



Explore practical articles and tutorials on machine learning, data science, and AI at <https://towardsdatascience.com>.

Part I

Machine Learning and Optimization

Chapter 2

K-Nearest Neighbors

2.1 Supervised Learning

Supervised learning is a central idea in AI. A model learns from labeled examples, where each input is paired with a correct output [6]. The goal is to predict the correct label for new, unseen data, i.e., to find a machine learning model that is trained on *training data* and is then able to generalize well on unseen *test data*.

A simple example comes from healthcare. Suppose we have data from patients, including features such as age, blood pressure, and cholesterol. Each patient is labeled as either **healthy** or **at risk**. The model learns from past patients and can then estimate the risk for a new patient. Formally, the dataset contains n pattern-label pairs:

$$(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)$$

Each input $\mathbf{x}_j \in \mathbb{R}^d$ is a vector of features, for example:

$$\mathbf{x}_j = (\text{age, blood pressure, cholesterol}).$$

The label y_j is the correct outcome, such as **healthy** or **at risk**. The model learns a function $f(\mathbf{x})$ that maps inputs to outputs. After training, it can predict the label for a new patient based on similar cases in the data. Supervised learning problems are typically divided into two types. In *classification*, the output is a discrete label, such as **disease** or **no disease**. In *regression*, the output is a continuous value, such as a risk score or survival time.

Min-Max Normalization

Min-max normalization rescales feature values to a fixed range, usually between 0 and 1. It is defined as:

$$\hat{x} = \frac{x - \min(X)}{\max(X) - \min(X)}$$

Here, x is a value from a feature $X = \{x_1, x_2, \dots, x_n\}$, and $\min(X)$, $\max(X)$ are its minimum and maximum values. Normalization is important because features can have very different scales (e.g., age vs. income).

2.2 KNN

The k-nearest neighbors (KNN) algorithm [3] is a simple and intuitive method for classification and regression. The main idea is that similar data points tend to have similar outcomes. To understand this, consider a healthcare example. Suppose we have past patient data with features such as age, blood pressure, and cholesterol, along with a label indicating whether a patient is **healthy** or **at risk**. Now, given a new patient, KNN looks for the most similar patients in the dataset and uses their labels to make a prediction.

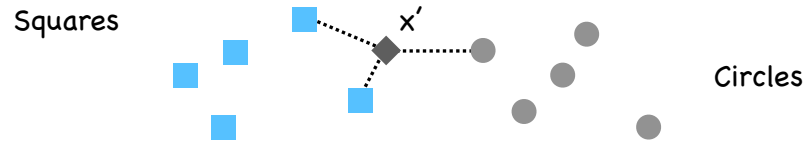


Figure 2.1: Illustration of KNN classification: the new pattern (gray diamond) is classified as blue square because, among its $k = 3$ nearest neighbors, two of three are labeled blue square, one as gray circle.

Formally, given a query point $\mathbf{x}'$, the algorithm computes the distance to all training points $\mathbf{x}_i$. A common choice is the Euclidean distance:

$$\delta(\mathbf{x}', \mathbf{x}_i) = \sqrt{\sum_{j=1}^d (x'_j - x_{ij})^2}$$

Here, j indexes the features. After computing all distances, the algorithm selects the k closest points. For classification, it assigns the label that appears

most often among these neighbors. Unlike many other ML methods, KNN does not build a mathematical model during training. It simply stores the training data and compares new points to known examples.

Figure 2.1 illustrates this idea. The new data point receives the majority label of its nearest neighbors. In the healthcare example, this means that if most similar patients are **at risk**, the new patient is also classified as **at risk**.

The choice of k has an important effect on the result. Figure 2.2 shows decision boundaries for different values of k . With $k = 1$, the model follows individual data points very closely, which can lead to overfitting. With moderate values such as $k = 3$, the decision becomes smoother and more robust. Very large values of k produce broad regions, which can oversimplify the data and lead to underfitting.

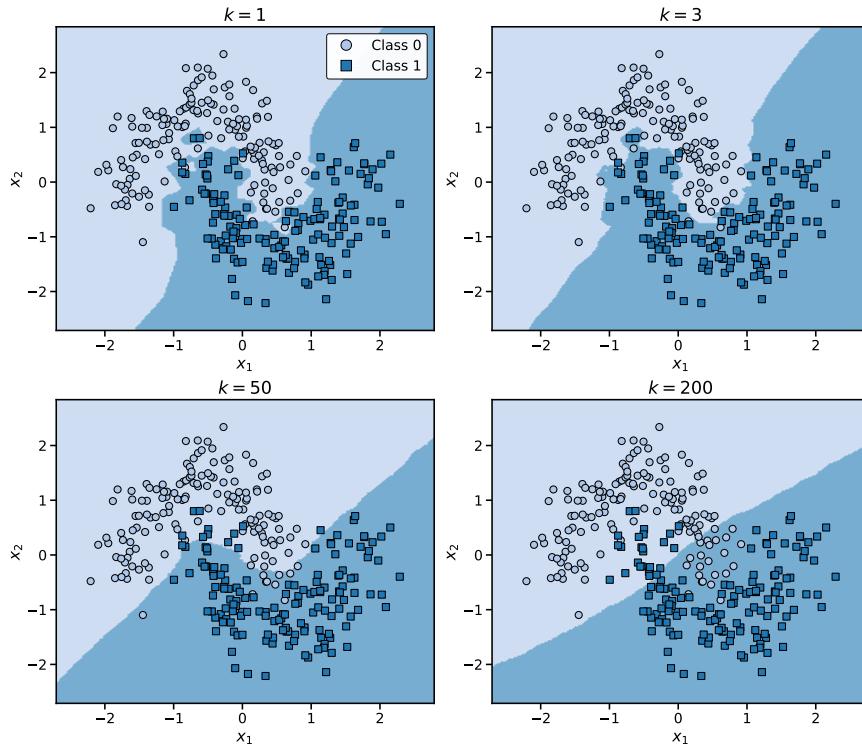


Figure 2.2: Decision boundaries of a k-NN classifier on the moons dataset for $k = 1, 3, 50$, and 200. Darker regions correspond to the circle class and lighter regions to the square class. Small values of k lead to highly flexible, local decision boundaries, whereas larger values increasingly smooth the classifier and emphasize global structure.

Accuracy

Accuracy measures how often a classifier makes correct predictions:

$$\text{Accuracy} = \frac{\text{\#correct predictions}}{\text{\#all predictions}}$$

2.3 Hands-On KNN

Consider a dataset with three labeled training points:

Point	x_1	x_2	Class
$\mathbf{x}_1$	1.0	2.0	Blue
$\mathbf{x}_2$	4.0	1.0	Blue
$\mathbf{x}_3$	6.0	5.0	Red

We classify a new point $\mathbf{x}' = (4.5, 2.5)$ using $k = 1$ -NN. The Euclidean distances to each training point are calculated as:

$$\delta(\mathbf{x}', \mathbf{x}_1) = \sqrt{(4.5 - 1.0)^2 + (2.5 - 2.0)^2} = \sqrt{12.25 + 0.25} = \sqrt{12.50} \approx 3.54$$

$$\delta(\mathbf{x}', \mathbf{x}_2) = \sqrt{(4.5 - 4.0)^2 + (2.5 - 1.0)^2} = \sqrt{0.25 + 2.25} = \sqrt{2.50} \approx 1.58$$

$$\delta(\mathbf{x}', \mathbf{x}_3) = \sqrt{(4.5 - 6.0)^2 + (2.5 - 5.0)^2} = \sqrt{2.25 + 6.25} = \sqrt{8.50} \approx 2.92$$

Sorting the distances in ascending order:

Point	$\delta(\mathbf{x}', \mathbf{x}_i)$
$\mathbf{x}_2$	1.58
$\mathbf{x}_3$	2.92
$\mathbf{x}_1$	3.54

Since we use $k = 1$, the nearest neighbor is point $\mathbf{x}_2$ (Blue). The classification result for $\mathbf{x}'$ is therefore Blue. This example demonstrates how 1-NN assigns the class of the closest training point to a query point, making it a simple and effective classification method.

Training and Test Sets

A dataset is usually divided into a training set and a test set. The training set is used to learn the model parameters. After training is complete, the test set is used to evaluate how well the model generalizes to previously unseen data. To obtain an unbiased evaluation, the test set must not be used during training.

2.4 Confusion Matrix

A confusion matrix shows how often a classifier predicts each class correctly or incorrectly. It compares the true labels with the predicted labels. Consider a healthcare example where a classifier predicts whether a patient is:

at risk

or:

healthy

A possible confusion matrix is:

	Predicted At Risk	Predicted Healthy
Actual At Risk	40 (<i>TP</i>)	10 (<i>FN</i>)
Actual Healthy	5 (<i>FP</i>)	45 (<i>TN</i>)

The four entries have special names. A true positive (TP) is an **at risk** patient correctly classified as **at risk**. A false negative (FN) is an **at risk** patient incorrectly classified as **healthy**. A false positive (FP) is a **healthy** patient incorrectly classified as **at risk**. A true negative (TN) is a **healthy** patient correctly classified as **healthy**.

Precision measures how reliable a positive prediction is. It is the proportion of patients predicted as **at risk** who are actually **at risk**:

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{40}{40 + 5} \approx 0.89.$$

Thus, about 89% of the patients predicted as **at risk** are truly at risk.

Recall, also called sensitivity, measures how many actual positive cases are found. It is the proportion of all **at risk** patients that the classifier correctly identifies:

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{40}{40 + 10} = 0.80.$$

Thus, the classifier detects 80% of all patients who are actually at risk. In healthcare, recall is often especially important because a false negative means that an at-risk patient is missed, whereas precision indicates how many positive warnings are correct.

A confusion matrix is useful because it shows not only how many predictions are correct, but also which types of mistakes the classifier makes. This is especially important in healthcare, where missing an **at risk** patient may be more serious than raising a false alarm.

Overfitting

Overfitting occurs when a model adapts too strongly to the training data, including noise and outliers. The model achieves very low training error but performs poorly on unseen test data because it does not generalize well.

2.5 KNN in Python

The following example shows how to implement the KNN algorithm directly in Python. [14].



```
import numpy as np
from collections import Counter

# Training data
X_train = np.array([
    [1, 2],
    [4, 1],
    [6, 5],
    [5, 2]
])

y_train = np.array([
    "Blue",
    "Blue",
    "Red",
    "Red"
])

# Query point
x = np.array([4.5, 2.5])

# Number of neighbors
k = 3

# Compute Euclidean distances
distances = np.sqrt(
    np.sum((X_train - x)**2, axis=1)
)

# Indices of k nearest neighbors
k_indices = np.argsort(distances)[:k]

# Labels of nearest neighbors
k_labels = y_train[k_indices]

# Majority vote
prediction = Counter(k_labels).most_common(1)[0][0]

print("Nearest labels:", k_labels)
```

```
print("Prediction:", prediction)
```

The program first stores the training points and their labels as NumPy arrays. For a new query point, the Euclidean distance to every training point is computed using vectorized array operations. The function `np.argsort` returns the indices of the smallest distances, which are used to select the k nearest neighbors. Finally, the `Counter` class counts how often each label appears among the neighbors, and the most common label is returned as the prediction.

2.6 Exercises



1. Questions:

- (a) What does the KNN algorithm do? How does the number k affect the result?
- (b) Why do we normalize features before using KNN?

2. Hands-On:

- (a) Look at this dataset and classify the point (3.5, 2.5) using 3-nearest neighbors and Euclidean distance:

Point	x_1	x_2	Label
$\mathbf{x}_1$	1.0	3.0	Blue
$\mathbf{x}_2$	3.0	2.0	Red
$\mathbf{x}_3$	5.0	1.0	Red
$\mathbf{x}_4$	4.0	4.0	Blue

Calculate the distances and decide which class wins.

3. Programming:

- (a) Write your own version of KNN in Python. Test it on the Iris dataset.
- (b) Try different values for k and see how it changes the results.

NVIDIA Keynote

Watch the latest NVIDIA keynote to explore upcoming trends in AI, hardware, and accelerated computing.

Chapter 3

K-Means

3.1 Unsupervised Learning

Unsupervised learning works with data that have no labels. This means we only have input features

$$\mathbf{x}_1, \dots, \mathbf{x}_n$$

but no correct answers or categories are given. In contrast to supervised learning, the model is not told what the correct output should be. The goal is to discover structure in the data automatically. Instead of learning from examples with known outputs, the model tries to find patterns, similarities, or groups on its own. This can reveal hidden structure that is not obvious at first glance.

A common task is *clustering*. Here, similar data points are grouped together based on a notion of distance or similarity. For example, a dataset of customers could be divided into groups with similar buying behavior. Methods such as k-means assign each point to a cluster so that points within a cluster are similar, while points in different clusters are less similar. Unsupervised learning is useful when no labels are available or when we want to explore a dataset. It helps to understand the data, detect unusual patterns (outliers), and identify meaningful groups without prior assumptions.

3.2 K-Means

K-means [12] is a widely used method for partitioning data into k clusters. Given data points $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ in a feature space, the objective is to determine k representative centers and assign each point to the nearest center. The algorithm proceeds iteratively:

1. Initialize k centers $\mathbf{c}_1, \dots, \mathbf{c}_k$, typically at random.

2. Assign each point $\mathbf{x}_j$ to the nearest center according to a distance measure (commonly Euclidean distance).
3. Update each center $\mathbf{c}_l$ as the mean of all points assigned to cluster l .
4. Repeat steps 2 and 3 until convergence, i.e., until the assignments or centers no longer change.

In essence, k-means alternates between assigning points to clusters and updating the cluster centers. This process gradually improves the grouping by minimizing the distance between points and their assigned centers. Figure 3.1 illustrates this iterative behavior. Starting from random initial centers, the algorithm repeatedly refines the assignments and adjusts the centers until a stable clustering is obtained.

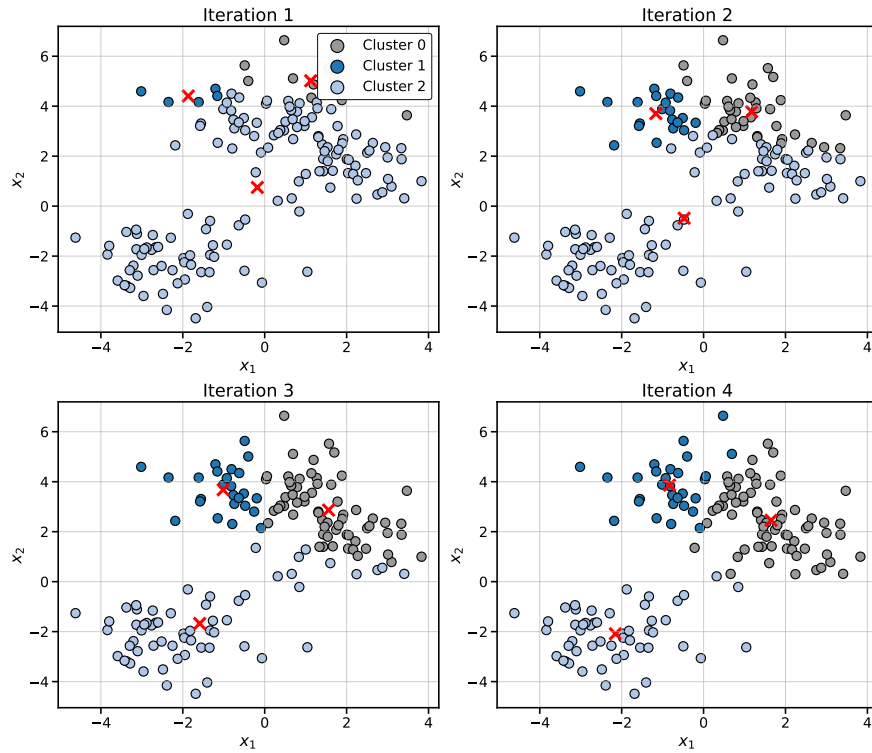


Figure 3.1: K-means clustering with $k = 3$ on a synthetic dataset with two features x_1 and x_2 . Data points are colored according to their current cluster assignment, while red crosses denote the cluster centers. The figure illustrates successive iterations of the algorithm as the cluster structure stabilizes.

3.3 Hands-On K-Means

We consider four points:

$$(2, 1), (3, 3), (6, 2), (7, 4)$$

and want to split them into two groups:

$$k = 2.$$

We start with the initial centers:

$$\mathbf{c}_1 = (2, 2), \quad \mathbf{c}_2 = (7, 3).$$

Each point is assigned to the nearest center using the Euclidean distance:

$$\delta(\mathbf{x}, \mathbf{c}) = \sqrt{(x_1 - c_1)^2 + (x_2 - c_2)^2}.$$

For example, for the point $(2, 1)$:

$$\delta((2, 1), (2, 2)) = 1, \quad \delta((2, 1), (7, 3)) = \sqrt{29}.$$

Thus, $(2, 1)$ is assigned to $\mathbf{c}_1$. The first assignment step gives:

$$\{(2, 1), (3, 3)\} \quad \text{and} \quad \{(6, 2), (7, 4)\}.$$

The centers are then updated by taking the average of each group:

$$\mathbf{c}_1 = \left(\frac{2+3}{2}, \frac{1+3}{2} \right) = (2.5, 2), \quad \mathbf{c}_2 = \left(\frac{6+7}{2}, \frac{2+4}{2} \right) = (6.5, 3).$$

Using the updated centers, the assignments do not change:

$$\{(2, 1), (3, 3)\} \quad \text{and} \quad \{(6, 2), (7, 4)\}.$$

Therefore, the algorithm stops. The clustering is stable, and the final centers are:

$$\mathbf{c}_1 = (2.5, 2), \quad \mathbf{c}_2 = (6.5, 3).$$

Scikit-learn

scikit-learn is a premier Python library for traditional machine learning, built on NumPy, SciPy, and matplotlib. It features a consistent API where models are treated as "estimators" using standard `fit()` and `predict()` methods. The library provides efficient tools for predictive data analysis, including classification, regression, clustering, and dimensionality reduction.



3.4 K-Means in scikit-learn

The next example shows how to cluster data using K-Means with `scikit-learn`.



```
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Create simple dataset
X, _ = make_blobs(n_samples=200, centers=3, random_state=42)

# Train K-Means model
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)

# Get cluster assignments
labels = kmeans.labels_
print(labels[:10])
```

The code first imports the required modules and generates a synthetic dataset with three clusters using `make_blobs`. The variable `X` contains the data points. A K-Means model with $k = 3$ clusters is then created and trained using `fit`. After training, each data point is assigned to a cluster, and these assignments are stored in `labels`. Finally, the first cluster labels are printed.

3.5 Exercises



1. Questions:
 - (a) Why do we need to choose the number of clusters k before running k-means? How can we find a good value?
 - (b) What problems can k-means have with strange-shaped or overlapping clusters?
 - (c) Why does the starting position of the cluster centers matter?
2. Hands-On:
 - (a) Consider the following four points:

$(1, 2), (2, 4), (5, 3), (6, 6)$

Use $k = 2$ clusters and start with the initial centers

$(1.5, 3.0)$ and $(5.5, 4.5)$.

Assign each point to the closest center using Euclidean distance. Then update the centers by computing the mean of each group.

Repeat the assignment step once more and check whether any point changes its cluster.

3. Programming:

- (a) Write your own k-means algorithm in Python and test it on a dataset from `scikit-learn`.
- (b) Try different values for k and plot the clusters to see how the result changes.

AI Music

Generate a song in your favorite style with AI, e.g., using Suno.

Chapter 4

Evolution Strategies

4.1 Optimization

Optimization is the task of finding the best solution among many possible solutions by minimizing or maximizing an objective function. For example, consider the Sphere function

$$f(x_1, x_2) = x_1^2 + x_2^2.$$

The goal is to find the point with the smallest function value. The optimum is

$$(x_1, x_2) = (0, 0),$$

where

$$f(0, 0) = 0.$$

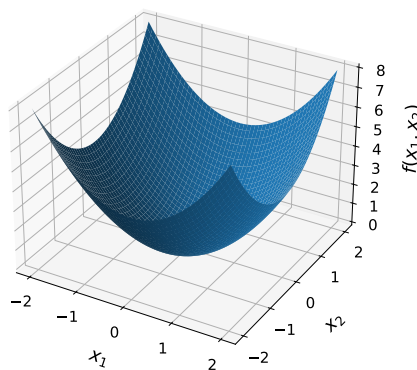


Figure 4.1: Sphere function with its global minimum at the origin.

Many optimization problems are much more complicated than the Sphere function. Sometimes the objective function is unknown or cannot be differentiated. Such problems are called *black-box optimization* problems. Evolution Strategies provide a simple method for solving them.

Evolution Strategies (ES) are optimization algorithms inspired by biological evolution. They were introduced in the 1960s by Rechenberg and Schwefel for optimizing continuous functions. They are based on the idea of creating new solutions with crossover and mutation and select the best solutions over many generations.

Darwinian Evolution

Evolution Strategies imitate natural evolution. A mutation creates a new individual. If the new individual performs better, it survives and replaces its parent.

4.2 $(1 + 1)$ -ES

The simplest Evolution Strategy is the $(1 + 1)$ -ES, consisting of one parent $\mathbf{x}$ and one offspring $\mathbf{x}'$. The offspring is generated by Gaussian mutation,

$$\mathbf{x}' = \mathbf{x} + \sigma \mathbf{z}, \quad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}),$$

where $\sigma > 0$ is the mutation strength or step size. Small values of σ perform local search, while large values explore larger regions. For minimization, the offspring replaces the parent if it is better:

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{x}', & f(\mathbf{x}') < f(\mathbf{x}), \\ \mathbf{x}, & \text{otherwise.} \end{cases}$$

Thus, each iteration consists of mutation, evaluation, and selection. Since only improvements are accepted, the objective value never increases.

Gaussian Mutation

Gaussian mutation creates an offspring by adding normally distributed noise. The mutation strength σ controls the search radius: small σ produces local steps, while large σ enables broader exploration.

4.3 Hands-On ES

Assume the current solution is

$$\mathbf{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

Its objective value is

$$f(\mathbf{x}) = 2^2 + 1^2 = 5.$$

Assume the mutation strength is

$$\sigma = 0.5$$

and Gaussian mutation generates

$$\mathbf{z} = \begin{bmatrix} -0.6 \\ 0.4 \end{bmatrix}.$$

The offspring becomes

$$\mathbf{x}' = \mathbf{x} + \sigma \mathbf{z} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} + 0.5 \begin{bmatrix} -0.6 \\ 0.4 \end{bmatrix} = \begin{bmatrix} 1.7 \\ 1.2 \end{bmatrix}.$$

Its objective value is

$$f(\mathbf{x}') = 1.7^2 + 1.2^2 = 2.89 + 1.44 = 4.33.$$

Since

$$4.33 < 5,$$

the offspring replaces the parent. The search therefore moves toward the optimum at the origin.

Rechenberg's 1/5 Success Rule

Rechenberg's 1/5 success rule adapts the mutation step size in an Evolution Strategy. If more than one fifth of the mutations are successful, the step size should be increased to explore the search space more broadly. If fewer than one fifth are successful, the step size should be decreased to perform smaller and more precise search steps. The rule aims to keep the success rate close to 1/5.

4.4 ES in Python

The following example minimizes the Sphere function using a (1+1)-Evolution Strategy.



```
import numpy as np

sigma = 0.3

x = np.array([4.0, -3.0])

def sphere(x):
    return np.sum(x ** 2)

for generation in range(100):

    offspring = x + sigma * np.random.randn(2)

    if sphere(offspring) < sphere(x):
        x = offspring

    print(generation, x, sphere(x))
```

The algorithm repeatedly mutates the current solution. Whenever the offspring has a smaller objective value, it replaces the parent. Over many iterations, the solution gradually approaches the global optimum.

AlphaEvolve

AlphaEvolve uses LLMs inside an evolutionary loop. Candidate programs are generated, evaluated by a fitness function, and improved through mutation, crossover, and selection.

4.5 Exercises



1. Why is Gaussian mutation suitable for continuous optimization?
2. Starting with

$$\mathbf{x} = \begin{bmatrix} 3 \\ 2 \end{bmatrix},$$

perform one mutation using

$$\sigma = 0.5$$

and

$$\mathbf{z} = \begin{bmatrix} -0.8 \\ 0.2 \end{bmatrix}.$$

- Compute the offspring and decide whether it replaces the parent.
3. Modify the Python program so that it optimizes a five-dimensional Sphere function.
 4. Experiment with different values of σ . How does the mutation strength influence the search?

Part II

Deep Learning

Chapter 5

Multi-Layer Perceptrons

5.1 Perceptron

Artificial neural networks form the basis of modern deep learning. They consist of simple computational units called neurons that are connected through weighted links. During training, the weights are adjusted so that the network gradually improves its predictions. The simplest neural model is the perceptron [17]. A perceptron receives input values, multiplies them by weights, adds a bias, and applies an activation function:

$$y = \sigma(w \cdot x + b)$$

Here, $x \in \mathbb{R}^d$ is the input vector, $w \in \mathbb{R}^d$ the weight vector, b the bias, and σ the activation function.

A commonly used activation is ReLU:

$$\text{ReLU}(z) = \max(0, z)$$

ReLU passes positive values and sets negative values to zero. The original perceptron used a step function and acted as a binary classifier with a linear decision boundary. This limits it to linearly separable problems.

Perceptron Limitation

A single-layer perceptron can model functions such as AND or OR, but it cannot represent XOR. In XOR, no single straight line can separate the classes correctly. This limitation motivated the development of multi-layer neural networks with hidden layers.

5.2 Multi-Layer Perceptrons

A Multi-Layer Perceptron (MLP) is a neural network with one or more hidden layers, see Figure 5.1. It can be used for regression and classification.

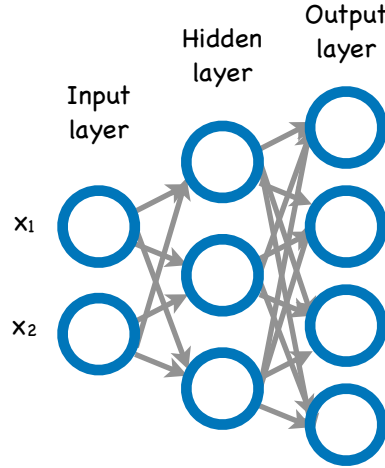


Figure 5.1: A simple MLP with two inputs, one hidden layer, and one output.

We consider a simple regression example with two inputs, one hidden layer, and one output:

$$(x_1, x_2) \longrightarrow (h_1, h_2, h_3) \longrightarrow \hat{y}.$$

The hidden neurons compute

$$h_j = \sigma(w_{j1}x_1 + w_{j2}x_2 + b_j),$$

where σ is an activation function, for example ReLU,

$$\text{ReLU}(z) = \max(0, z).$$

The output neuron combines the hidden values:

$$\hat{y} = v_1h_1 + v_2h_2 + v_3h_3 + c.$$

For regression, the goal is that the prediction $\hat{y}$ is close to the true target value y . A common loss function is the squared error

$$L = \frac{1}{2}(\hat{y} - y)^2.$$

Training means changing all weights and biases so that this loss becomes smaller. This is done by gradient descent:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}.$$

Here, η is the learning rate. Backpropagation computes the required derivatives efficiently, starting from the output error and moving backwards through the network. Thus, an MLP learns a nonlinear function

$$f(x_1, x_2) \approx y.$$

The hidden layer makes the model more flexible than ordinary linear regression, because the activation functions introduce nonlinearity.

Epoch

An *epoch* is one complete pass through the training dataset. Most neural networks require many epochs, gradually improving their predictions as the weights are updated after each batch.

5.3 Hands-On Gradient Descent

We now perform one gradient descent step for the simple MLP introduced in the previous section. Suppose the hidden layer has already computed

$$h_1 = 1, \quad h_2 = 2, \quad h_3 = 1.$$

The output neuron has the initial weights

$$v_1 = 1, \quad v_2 = 1, \quad v_3 = 1,$$

and no bias. The prediction is therefore

$$\hat{y} = 1 \cdot 1 + 1 \cdot 2 + 1 \cdot 1 = 4.$$

Assume the target value is

$$y = 6.$$

The squared error is

$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(4 - 6)^2 = 2.$$

The gradients of the output weights are

$$\frac{\partial L}{\partial v_i} = (\hat{y} - y)h_i.$$

Hence,

$$\frac{\partial L}{\partial v_1} = (4 - 6) \cdot 1 = -2,$$

$$\frac{\partial L}{\partial v_2} = (4 - 6) \cdot 2 = -4,$$

and

$$\frac{\partial L}{\partial v_3} = (4 - 6) \cdot 1 = -2.$$

Using the learning rate

$$\eta = 0.1,$$

the updated weights become

$$v_1 = 1 - 0.1(-2) = 1.2,$$

$$v_2 = 1 - 0.1(-4) = 1.4,$$

and

$$v_3 = 1 - 0.1(-2) = 1.2.$$

The new prediction is

$$1.2 \cdot 1 + 1.4 \cdot 2 + 1.2 \cdot 1 = 5.2.$$

Thus, one gradient descent step moved the prediction closer to the target:

$$4 \rightarrow 5.2.$$

In a real MLP, backpropagation would also update the weights between the input and hidden layer. Here, we only updated the output layer to keep the example simple.

Stochastic Gradient Descent (SGD)

Gradient descent can use the entire training set (batch gradient descent) or only one randomly selected training example at a time (stochastic gradient descent). In practice, small mini-batches (e.g., 32 or 64 samples) are most commonly used because they are both efficient and stable.

5.4 MLP in scikit-learn

The following example shows how to train an MLP using `scikit-learn`.



```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier

# Load data
X, y = load_digits(return_X_y=True)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y)

# Train MLP
mlp = MLPClassifier(hidden_layer_sizes=(50,))
mlp.fit(X_train, y_train)

# Evaluate
accuracy = mlp.score(X_test, y_test)

print("Accuracy:", accuracy)
```

The code loads the digits dataset, splits it into training and test data, trains an MLP with one hidden layer of 50 neurons, and evaluates the accuracy on unseen test data.

5.5 Exercises



1. Questions:
 - (a) What is a perceptron?
 - (b) Why are activation functions important?
 - (c) What is gradient descent?
2. Hands-On:
 - (a) Consider a perceptron without bias and activation function with:

$$(x_1, x_2, y) = (2, 1, 6)$$

and:

$$w_1 = 0.5, \quad w_2 = 1$$

Perform two gradient descent update steps using $\eta = 0.1$

3. Programming:
 - (a) Train an `MLPClassifier` on the Digits dataset.
 - (b) Try different hidden layer sizes and learning rates.

Geoffrey Hinton

Explore the work of **Geoffrey Hinton**, a pioneering figure in modern AI. Investigate his fundamental contributions to neural networks and his more recent critical perspectives on the risks and future of AI.

Chapter 6

Convolutional Neural Networks

6.1 Image Recognition

Image recognition is a central application of AI. The goal is to automatically understand images, for example by classifying objects, detecting objects, or recognizing scenes. This is challenging because images are high-dimensional and contain complex spatial patterns. Early methods used handcrafted features such as edges, corners, or textures. These features were then passed to traditional machine learning models. However, such features were limited and often did not generalize well.

Convolutional Neural Networks (CNNs) [11] changed this approach. Instead of manually designing features, CNNs learn them directly from raw pixels. Early layers detect simple patterns such as edges, while deeper layers combine them into shapes and objects. A major breakthrough was AlexNet [10], which won the ImageNet competition in 2012 by a large margin. Since then, CNNs have become a standard method for image recognition in areas such as medical imaging, autonomous driving, robotics, and surveillance. 

6.2 Convolutional Layers

Convolutional layers are the core of CNNs. They apply small filters, also called kernels, to local regions of an image. Typical filter sizes are 3×3 or 5×5 . At each position, the filter is multiplied with the corresponding image region, and the products are summed:

$$y(i, j) = \sum_{m, n} K(m, n) X(i + m, j + n).$$

Here, X is the input image, K is the filter, and $y(i, j)$ is one value of the output feature map.

A high activation occurs when the local image region has a structure similar to the filter. For example, an edge filter gives a high value when it is placed on a similar edge in the image. During training, the filter values are learned automatically by backpropagation and gradient descent. The network changes the filters so that the final loss becomes smaller.

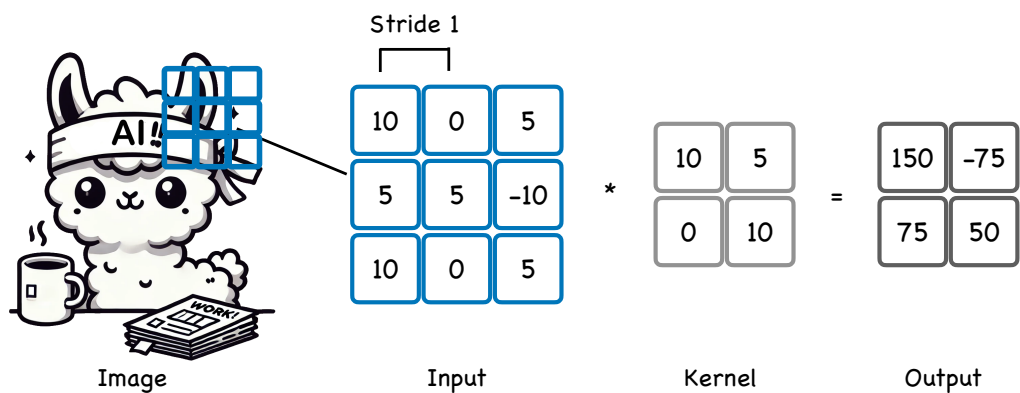


Figure 6.1: Illustration of the convolution operation with stride 1. A kernel K slides over the input X . At each position, element-wise multiplication and summation produce one entry of the output Y .

The step size of the filter is called the *stride*. A larger stride reduces the spatial size of the output. To preserve the image size, zeros can be added around the border of the input. This is called *zero padding*. CNNs learn hierarchical features. Early layers often learn simple structures such as edges, lines, and textures. Deeper layers combine these patterns into more abstract structures such as object parts and complete objects.

Filter Banks

A filter bank is a set of filters applied to the same input. Each filter learns a different pattern and produces one feature map. For example, 256 filters produce 256 feature maps. Together, these maps describe different structures in the image.

6.3 Hands-On Convolution

Consider a single-channel 3×3 input image and a 2×2 filter. We use a stride of 1 and no padding. Let the input be:

$$\mathbf{X} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 1 & 2 \\ 0 & 1 & 1 \end{bmatrix}$$

and the filter:

$$\mathbf{K} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

We perform the convolution by sliding the filter over the input matrix. At each location, we compute the element-wise product and sum.

Step 1 (top-left region of $\mathbf{X}$):

$$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = (1 \cdot 1) + (2 \cdot 0) + (3 \cdot -1) + (1 \cdot 1) = -1$$

Step 2 (top-middle region):

$$\begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = (2 \cdot 1) + (0 \cdot 0) + (1 \cdot -1) + (2 \cdot 1) = 3$$

Step 3 (bottom-left region):

$$\begin{bmatrix} 3 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = (3 \cdot 1) + (1 \cdot 0) + (0 \cdot -1) + (1 \cdot 1) = 4$$

Step 4 (bottom-middle region):

$$\begin{bmatrix} 1 & 2 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = (1 \cdot 1) + (2 \cdot 0) + (1 \cdot -1) + (1 \cdot 1) = 1$$

The output feature map is:

$$\mathbf{Y} = \begin{bmatrix} -1 & 3 \\ 4 & 1 \end{bmatrix}$$

This demonstrates how a CNN filter extracts features from an input by scanning small regions and applying learned weights.

Data Augmentation

Data augmentation [19] creates additional training examples by applying transformations such as flipping, rotating, cropping, or changing brightness. This reduces overfitting and makes models more robust to variations in the input images.

6.4 Pooling

Pooling is an operation commonly used in CNNs to reduce the spatial dimensions of feature maps while retaining important features. Pooling layers operate independently on each feature map and aggregate information within local neighborhoods. The most commonly used pooling methods in convolutional neural networks are max pooling and average pooling. Max pooling selects the highest value from each sub-region of the feature map. For example, given a 2×2 sub-region:

$$\text{Input: } \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \Rightarrow \text{Max pooled output: } \max(1, 3, 2, 4) = 4$$

Average pooling computes the mean value of each sub-region, resulting in a smoothed representation. For example, given the following 2×2 sub-region:

$$\text{Input: } \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \Rightarrow \text{Average pooled output: } \frac{1 + 3 + 2 + 4}{4} = 2.5$$

The primary purpose of pooling is to provide spatial invariance, meaning the network can recognize patterns regardless of their precise location within the input. Max pooling, being more common, emphasizes the most prominent features, which makes it useful for tasks where the presence of a feature is more important than its exact value or location.

Classification

For classification, the output layer represents class probabilities. Binary classification often uses one sigmoid output neuron, while multi-class classification uses softmax:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}.$$

Classification is usually trained with cross-entropy loss:

$$L = -\log(p),$$

where p is the predicted probability of the correct class.

6.5 VGG-19

VGG-19 [20] is a widely used deep CNN known for its simple and uniform architecture, making it a standard baseline for image classification tasks.

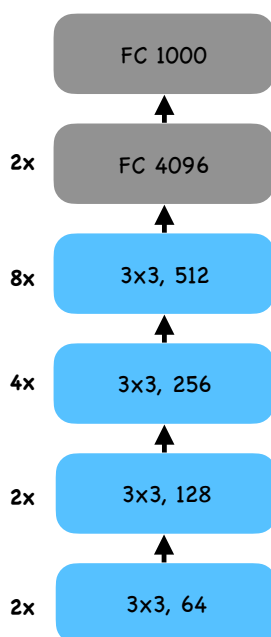


Figure 6.2: Architecture of VGG-19. An input image is processed through a sequence of convolutional blocks with small 3×3 filters, each followed by a max-pooling layer. The extracted features are then passed to fully connected (dense) layers and finally to a softmax classifier.

Figure 6.2 illustrates the structure of VGG-19. The model consists of 16 convolutional layers grouped into 5 sequential blocks. Each block is followed by a max pooling layer that reduces the spatial resolution of the feature maps by a factor of 2 while typically doubling the number of filters. After the convolutional blocks, the network includes three fully connected (dense) layers with 4096, 4096, and 1000 units, respectively. The original VGG-19 model was trained on the ImageNet dataset and classifies input images into 1000 distinct classes.

Keras

Keras is a popular **Python** library for deep learning. It provides a simple and user-friendly interface for building neural networks and runs on top of frameworks such as TensorFlow. Models are typically constructed layer by layer using APIs such as **Sequential**. Keras includes tools for training, evaluation, convolutional networks, recurrent networks, and modern deep learning workflows.



6.6 CNN in Keras



The following example shows how to train a CNN for handwritten-digit classification using Keras . The MNIST dataset contains grayscale images with a resolution of 28×28 pixels. Each image represents one handwritten digit from one of ten classes, corresponding to the digits 0–9.

```
# Convolutional Neural Network (CNN) for MNIST

from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from keras.datasets import mnist
from keras.utils import to_categorical

# Load MNIST
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize pixel values
X_train = X_train.astype("float32") / 255.0
X_test = X_test.astype("float32") / 255.0

# Add the grayscale channel dimension
X_train = X_train.reshape(-1, 28, 28, 1)
X_test = X_test.reshape(-1, 28, 28, 1)

# Encode labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build the CNN
model = Sequential([
    Conv2D(16, (3, 3), activation="relu",
        input_shape=(28, 28, 1)), MaxPooling2D(),

    Conv2D(32, (3, 3), activation="relu"), MaxPooling2D(),

    Flatten(),
    Dense(64, activation="relu"),
    Dense(10, activation="softmax")
])

# Train and evaluate
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

model.fit(X_train, y_train, epochs=3, batch_size=64
)

accuracy = model.evaluate(X_test, y_test)[1]
```

```
print("Accuracy:", accuracy)
```

The code loads the MNIST training and test sets, converts the pixel values to floating-point numbers, and normalizes them to the interval $[0, 1]$. Because MNIST contains grayscale images, the arrays are reshaped to include a single channel. The class labels are converted to one-hot vectors. The CNN contains two convolutional layers that learn visual features from the digit images. Each convolutional layer is followed by a pooling layer that reduces the spatial dimensions of the feature maps. The resulting features are flattened and passed to two dense layers. The final layer contains ten neurons with softmax activation, one for each digit class. The model is trained for three epochs using categorical cross-entropy loss and evaluated on the MNIST test set.

6.7 Exercises



1. Questions:
 - (a) What is a convolution in CNNs, and why is it useful for images?
 - (b) What is the main idea behind ResNet [7]?
 - (c) What do pooling layers do?
2. Hands-On:
 - (a) Use the following 3×3 input and 2×2 filter to perform a 2D convolution with stride 1 and no padding. Input:
$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 2 \\ 0 & 1 & 1 \end{bmatrix} \quad \text{Filter: } \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}$$
 - (b) At each position, multiply the overlapping values element-wise and sum them up.
 - (c) Slide the filter over the input and compute all values. Write down the final 2×2 output matrix.
3. Programming:
 - (a) Use the Google Colab notebook from this chapter to train the CNN on the CIFAR-10 dataset. Record the test accuracy of the baseline model.
 - (b) Tune the CNN to improve its test accuracy. Try changes such as adding more convolutional layers, changing the number of filters, adjusting hyperparameters, adding dropout, or using data augmentation.

MIT Moral Machine



Visit the MIT Moral Machine website and explore interactive ethical dilemmas. Reflect on how different decisions reveal underlying values and biases in AI systems.

Chapter 7

Attention

7.1 Sequence Learning

Many important AI problems involve sequences. Examples include text, speech, time series, DNA sequences, and program code. In all these cases, the order of the elements matters. A model therefore needs to understand not only the individual tokens, but also their context within the sequence.

Transformers are neural networks designed for this type of problem. Their key idea is *attention*. Attention allows each token to look at all other tokens and decide which ones are important. For example, in a sentence, one word may need information from another word far away in the same sentence. With attention, this connection can be modeled directly. This makes transformers very powerful for sequence learning. They can capture local relationships between nearby tokens as well as long-range relationships across the whole sequence. Today, transformers [23] are the foundation of many modern AI systems, especially large language models. They are also used in other domains, such as computer vision, biology, and time series analysis. 

Generative AI

Generative AI creates new content instead of only assigning labels or making predictions. It can generate text, images, music, code, videos, or designs from prompts and examples. Typical generative models include large language models for text, diffusion models for images, and generative adversarial networks for synthetic data. The key idea is that the model learns patterns from training data and then produces new outputs that follow similar patterns.

7.2 Attention

A transformer processes text by first splitting it into tokens. For example,

AI is fun

becomes

[AI, is, fun].

Each token is represented by a vector, called an *embedding*. These vectors allow the model to work with words numerically. The main idea of attention is simple: Each token looks at all other tokens and decides which ones are important.

For example, in the sentence **AI is fun**, the token **fun** may pay attention to **AI**, because **AI** explains what is fun. Let

$\mathbf{X}$

be the matrix containing all token embeddings. Attention creates three new matrices:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V.$$

Here, $\mathbf{Q}$ denotes the *queries*, which describe what a token is looking for; $\mathbf{K}$ denotes the *keys*, which describe what a token offers for matching; and $\mathbf{V}$ denotes the *values*, which contain the information that is passed on. Queries and keys are compared using dot products:

$$\mathbf{Q}\mathbf{K}^\top.$$

A large value means that two tokens match well. The model then applies Softmax to turn the scores into weights:

$$\mathbf{A} = \text{softmax}(\mathbf{Q}\mathbf{K}^\top).$$

Each row of $\mathbf{A}$ contains weights that sum to 1. These weights describe how strongly one token attends to the other tokens. The final attention output is

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{A}\mathbf{V}.$$

So attention computes a weighted mixture of the value vectors. The result is a new representation for each token, enriched with information from the other tokens. In short:

attention = choose important tokens + mix their information.

Positional Encoding

Attention alone does not know the order of words. Therefore, transformers add positional information to the token embeddings. This allows the model to distinguish sentences such as **AI is fun** and **Fun is AI**.

7.3 Hands-On Attention

Consider again the sentence

AI is fun.

We use simple two-dimensional embeddings:

$$\mathbf{AI} \rightarrow \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{is} \rightarrow \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \quad \mathbf{fun} \rightarrow \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Thus,

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}.$$

To keep the example simple, assume

$$\mathbf{W}_Q = \mathbf{W}_K = \mathbf{W}_V = \mathbf{I}.$$

Then

$$\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}.$$

The attention scores are computed by

$$\mathbf{QK}^\top.$$

This gives

$$\mathbf{QK}^\top = \begin{bmatrix} 1 & 0.5 & 0 \\ 0.5 & 0.5 & 0.5 \\ 0 & 0.5 & 1 \end{bmatrix}.$$

After applying Softmax row by row, we get approximately

$$\mathbf{A} \approx \begin{bmatrix} 0.51 & 0.31 & 0.19 \\ 0.33 & 0.33 & 0.33 \\ 0.19 & 0.31 & 0.51 \end{bmatrix}.$$

Each row tells us where one token looks. For example, the first row means that AI mostly attends to itself, but also partly to `is` and `fun`. The final output is

$$\mathbf{AV} \approx \begin{bmatrix} 0.66 & 0.34 \\ 0.50 & 0.50 \\ 0.34 & 0.66 \end{bmatrix}.$$

Each token now has a new vector. This vector is no longer isolated. It contains information from the whole sentence.

Multi-Head Attention

In practice, transformers do not use only one attention mechanism. They use several in parallel. These are called *heads*. Each head can learn a different type of relationship between tokens. The results of all heads are then combined.

7.4 GPT

GPT (Generative Pre-trained Transformer) [16] is a language model that generates text token by token. Given a prompt, it predicts the next token based on the previous context. For example, the prompt

AI is fun because it ___

may be completed with phrases such as `learns from data` or `helps solve problems`.

Text is first split into tokens. Each token is converted into an embedding and combined with positional information. These representations are then processed by several transformer blocks with multi-head attention, dense layers, residual connections, and layer normalization.

The attention mechanism allows each token to focus on relevant previous tokens. This helps GPT capture context and generate coherent text. Since GPT is trained on large text collections by predicting the next token, it can later be used for tasks such as answering questions, summarizing text, and writing code.

Transformers

Transformers [23] are neural networks for sequence data. They use attention to relate tokens to each other. GPT uses transformer decoder blocks for autoregressive text generation, while BERT [4] uses encoder blocks for bidirectional text understanding.

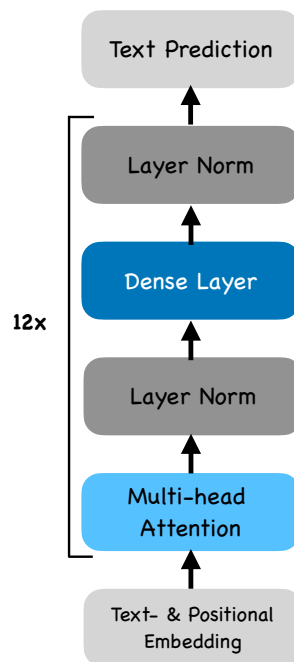


Figure 7.1: A GPT-style model generates text step by step. Each new token is predicted from the previous context.

7.5 Hugging Face

Hugging Face's `transformers` library provides simple access to many pre-trained models for tasks such as classification, translation, summarization, image analysis, speech recognition, and biological sequence analysis.

The central interface is the `pipeline`, which handles model loading, tokenization, inference, and output formatting. The following example performs sentiment analysis with a pre-trained DistilBERT model.

```
from transformers import pipeline

model_name =
    "distilbert-base-uncased-finetuned-sst-2-english"

sentiment_pipeline = pipeline(
    "sentiment-analysis",
    model=model_name
)

text = "I love using Hugging Face's tools!"
result = sentiment_pipeline(text)

print(result)
```



A possible output is:

```
[{'label': 'POSITIVE', 'score': 0.99}]
```

Hugging Face is not limited to text models. Its model hub contains pre-trained models for several modalities and scientific domains. Each model

Table 7.1: Examples of Popular Hugging Face Models

Model	Description
Whisper	Automatic speech recognition model that transcribes and translates spoken language.
ViT	Vision Transformer for image classification and computer vision tasks.
CLIP	Joint vision-language model for image retrieval and zero-shot image classification.
Stable Diffusion	Latent diffusion model for generating images from text prompts.
CARBon	Foundation model for DNA sequences, providing genomic embeddings for downstream bioinformatics tasks.
ESM-2	Protein language model for learning protein representations and predicting structural and functional properties.

is identified by a model name on the Hugging Face Hub. Models can be downloaded and used directly, or further fine-tuned for a specific dataset and application.

Streamlit



Streamlit is a Python framework for building interactive web apps with little code. It is useful for machine learning demos, dashboards, and fast prototyping of AI experiments.

7.6 Exercises



1. Questions:

- Name the main components of a transformer (e.g., attention, feed-forward, etc.).
- What is the idea behind multi-head attention compared to using only one attention head?

- (c) Why is positional encoding needed in transformers?
2. Hands-On:

- (a) Use the following simple embedding matrix:

$$\mathbf{X} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- (b) Set $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{X}$.
- (c) Compute the attention step by step:

$$\mathbf{QK}^\top, \quad \text{softmax}(\cdot), \quad \text{Attention} = \text{softmax}(\mathbf{QK}^\top)\mathbf{V}$$

- (d) Interpret the result: Which words attend most strongly to each other?
3. Programming:
- (a) Build a simple Streamlit app that lets a user type a sentence and shows the predicted sentiment (e.g. positive, negative, neutral) using a model from Hugging Face. Also display the confidence score.

Discover Conferences

Explore AI conferences such as ESANN. How does the paper submission process work? Learn about deadlines, peer review, acceptance rates, and how research is presented.

Part III

LLMs and Agents

Chapter 8

Large Language Models

8.1 LLMs

Large Language Models (LLMs) are AI systems that read, understand, and generate human language. They are built with transformers and trained on large text corpora. This training lets them learn grammar, facts, and simple reasoning patterns. LLMs turn input text into numerical units called tokens. The model processes these tokens internally and then generates one token at a time to form a response. The output tokens are converted back into readable text. A tokenizer performs this mapping. It splits text into words, subwords, or characters and assigns numerical values the model can process. LLMs are often trained as foundational models. They are not designed for one fixed task but can adapt to many. After broad training, they can be fine-tuned or guided with examples for tasks such as question answering, summarization, or story writing. State-of-the-art LLMs are called frontier models. They define current AI capability but require high computational and energy resources to train.

Turing Test

The Turing Test [22], proposed by Alan Turing in 1950, is a benchmark for judging machine intelligence. A human interacts with a machine and another human through text only. If the judge cannot reliably tell which is which, the machine is considered to have passed the test. In that case, its behavior appears indistinguishable from human intelligence.

8.2 Parameters and Precision

The number of parameters influences the capability of a large language model. Parameters are the learned values inside the neural network, comparable to adjustable weights. Small models may have only a few billion parameters, such as 7B or 8B. Larger models may have 70B, 405B, or even 671B parameters. For example, Ollama provides models such as Llama 3.1 in 8B, 70B, and 405B variants, and DeepSeek-R1 up to 671B parameters. Larger models can learn more complex patterns and often achieve better performance, but they also require much more memory and computation. Smaller models are faster, cheaper to deploy, and easier to run locally.

Numerical precision determines how values are stored inside the model. FP32 and FP16 are commonly used during training, while lower-precision formats such as INT8 or INT4 are often used for inference to reduce memory usage and improve speed. This process is called quantization. It makes it possible to run larger models on smaller hardware with only limited loss in accuracy.

Humanity's Last Exam



Humanity's Last Exam (HLE) [15] is a difficult benchmark for evaluating large language models on expert-level academic knowledge. It contains 2500 closed-ended questions from many disciplines, including mathematics, humanities, and the natural sciences. HLE is designed to go beyond older benchmarks that have become too easy for modern LLMs.

8.3 Ollama



Ollama is available for macOS, Linux, and Windows. It provides a lightweight framework for running large language models locally, such as Llama [21], Gemma, Mistral, and Nemotron. The framework automatically downloads required models when they are first executed and exposes a simple local API for interacting with them.

The commands below show how to list installed models and run a model locally.

```
# List installed models
ollama list
```

```
# Run a model
ollama run nemotron
```

For scripted use, prompts can also be passed directly through standard input:

```
echo "Explain neural networks." | ollama run nemotron
```

Ollama models can be customized using a `Modelfile`. The file defines the base model, parameters, and system behavior. This allows customization of instructions, temperature, and other inference settings.

```
FROM nemotron
PARAMETER temperature 0.7
SYSTEM "You are a helpful assistant specialized in AI."
```

This configuration loads the `nemotron` model, sets the temperature to 0.7, and defines a simple system prompt. Ollama enables flexible local deployment of language models while keeping all computation and data on the local device.

AGI

Artificial General Intelligence (AGI) [5] describes AI systems with human-level flexibility: the ability to learn, understand, reason, and solve problems across a wide range of tasks. Unlike narrow AI (which is AI specialized to a certain problem), AGI can generalize, adapt to new situations, and acquire new skills autonomously.

8.4 Chain-of-Thought

Prompting is the process of giving an LLM a task description, examples, or context so that it produces the desired output. A prompt can be a simple question, but it can also define the expected format, style, and level of detail. One important distinction is how many examples are included in the prompt. In one-shot prompting, the model receives one example before the actual task. The example demonstrates the desired input-output format.

```
Q: A bakery sells 3 types of muffins. Each muffin costs $2,
and a customer buys 5 of each type. How much does the customer pay?
```

```
A: $30.
```

```
Now solve a similar problem:
```

```
Q: A bookstore sells 4 types of books. Each book costs $5,
and a customer buys 6 of each type. How much does the customer pay?
```

In few-shot prompting, the model receives several examples before the actual task. This helps the model recognize the desired pattern more reliably.

Q: A bakery sells 3 types of muffins. Each muffin costs \$2, and a customer buys 5 of each type. How much does the customer pay?
A: \$30.

Q: A shop sells 2 types of pens. Each pen costs \$3, and a customer buys 4 of each type. How much does the customer pay?
A: \$24.

Q: A cafe sells 5 types of cookies. Each cookie costs \$1, and a customer buys 3 of each type. How much does the customer pay?
A: \$15.

Now solve:

Q: A bookstore sells 4 types of books. Each book costs \$5, and a customer buys 6 of each type. How much does the customer pay?

Few-shot prompting is useful when the task has a specific format, style, or decision pattern. Chain-of-Thought (CoT) prompting [24] extends prompting by showing intermediate reasoning steps. Instead of only giving the final answer, the example demonstrates how the answer is derived.

Q: A bakery sells 3 types of muffins. Each muffin costs \$2, and a customer buys 5 of each type. How much does the customer pay?

Step-by-step solution:

1. The customer buys 5 muffins of each type.
2. There are 3 types, so the total number of muffins is $5 \times 3 = 15$.
3. Each muffin costs \$2, so the total cost is $15 \times 2 = 30$.
4. The final answer is \$30.

Now solve a similar problem using step-by-step reasoning:

Q: A bookstore sells 4 types of books. Each book costs \$5, and a customer buys 6 of each type. How much does the customer pay?

The difference is that one-shot and few-shot prompting mainly demonstrate the input-output pattern, while Chain-of-Thought prompting also demonstrates the reasoning pattern. This can help LLMs solve multi-step problems more reliably.

Reinforcement Learning with Human Feedback

Reinforcement Learning with Human Feedback (RLHF) [1] aligns language models with human preferences. Humans compare different model responses, and these rankings are used to train a reward model, which assigns a score to an answer. The LLM is then fine-tuned to generate answers with high reward scores.

8.5 A Chatbot in Ollama

The following example demonstrates a minimal chatbot using Ollama.



```
import ollama
model = "llama3.2"

def chat(messages):
    response = ollama.chat(model=model, messages=messages,
                           stream=True)

    output = ""
    for chunk in response:
        content = chunk['message']['content']
        output += content
        print(content, end="", flush=True)

    return {"role": "assistant", "content": output}

messages = [{"role": "system", "content": "Answer in max
three sentences, concise and ironic."}]

while True:
    user = input("You: ")
    if not user:
        break

    messages.append({"role": "user", "content": user})

    assistant_msg = chat(messages)
    messages.append(assistant_msg)
```

This minimal script implements a conversational interface with a locally running model through the `ollama` Python package. The function `ollama.chat` sends the current dialogue history to the model and returns the answer. With `stream=True`, the response is received piece by piece and printed immediately, so the chatbot appears to answer in real time. The variable `messages` stores the conversation as a list of dictionaries. Each entry has a role, such as `system`, `user`, or `assistant`, and a text content. By appending each user message and each assistant response to this list, the chatbot keeps the

previous conversation as context for the next answer.

8.6 Exercises



1. Questions:
 - (a) What is the difference between foundational and frontier models?
 - (b) Why are INT8 and INT4 useful for inference?
 - (c) What is the idea of Chain of Thought prompting?
2. Programming:
 - (a) Build a basic chatbot using an LLM and Streamlit using your local LLM, e.g. using `Ollama` or a LLM API.

AI Videos

Generate short AI videos with your favorite AI, e.g., using Grok Imagine. Test different prompts and their impact on the result.

Chapter 9

Retrieval-Augmented Generation

9.1 RAG

Large language models answer from patterns stored in their parameters and from information placed in the current context. Their parameters may not contain private documents, recently published facts, or specialized details from a particular organization. Even when relevant knowledge was present during training, the model may recall it incompletely or confidently produce an unsupported answer.

Retrieval-Augmented Generation (RAG) addresses this problem by searching an external collection before generating the response. The system receives a question, retrieves passages that may contain the answer, places those passages in the model’s context, and asks the model to answer using that evidence. A basic RAG pipeline is

Question → Retrieve Passages → Build Context → Generate Answer.
(9.1)

RAG does not normally change the model’s weights. It changes the information available for one request. This distinction makes external knowledge easier to update: a document can be added, corrected, or removed without training the language model again. The model still contributes language understanding and generation, while the retrieval system supplies task-specific evidence.

RAG is useful for assistants that answer questions about manuals, lecture notes, company policies, research papers, or other document collections. It can make answers more current and traceable, but it does not guarantee correctness. The retriever may select the wrong passage, the necessary information may be absent, or the generator may ignore or misinterpret good evidence. A reliable RAG system must therefore manage both retrieval qual-

ity and generation quality.

Retrieval-Augmented Generation

RAG combines external retrieval with language generation. Instead of relying only on information encoded in model weights, the model receives passages selected for the current question and uses them as evidence for its answer.

9.2 Documents and Chunks

Before questions can be answered, the source collection must be prepared. Documents may come from PDF files, web pages, databases, slides, source code, or scanned images. An ingestion process extracts their text, removes repeated headers or navigation elements, and preserves useful structure such as titles, headings, page numbers, dates, and access permissions.

Long documents are divided into smaller units called *chunks*. Retrieval works on chunks rather than entire books or reports because a short passage is easier to match to a focused question and fits more efficiently into the model's context window.

Chunk size creates a trade-off. Very small chunks may separate a statement from the explanation or definition needed to understand it. Very large chunks preserve more context but may include several unrelated topics, make matching less precise, and consume valuable context space. Some systems add overlap so that text near a boundary appears in two neighboring chunks. Structure-aware chunking can split at headings, paragraph breaks, or code blocks instead of cutting after an arbitrary number of characters.

Each chunk should retain metadata that connects it to its source. Metadata can include the document title, section, page, author, publication date, version, and permission group. If a policy is replaced, for example, version metadata helps prevent an older passage from being presented as current guidance.

Chunking

Chunking divides a source into passages small enough for precise retrieval but large enough to remain understandable. There is no universally best chunk size; it should be tested with the documents and questions expected in the application.

9.3 Embeddings and Indexing

To search efficiently, a RAG system builds an index. A common approach represents each chunk with an *embedding*: a vector of numbers produced by an embedding model. Texts with related meanings should receive vectors that are close in the embedding space. This allows a question about “resetting a forgotten password” to find a passage titled “account credential recovery” even though the wording is different.

The documents are processed before users begin asking questions. The system extracts text, creates chunks, computes one embedding for each chunk, and stores the vectors together with the original text and metadata. This indexing pipeline can be summarized as

$$\text{Documents} \rightarrow \text{Chunks} \rightarrow \text{Embeddings} \rightarrow \text{Index}. \quad (9.2)$$

At query time, the same embedding model converts the user’s question into a vector. The index compares this query vector with stored chunk vectors. One common measure is cosine similarity, defined as $s(\mathbf{q}, \mathbf{d}) = (\mathbf{q} \cdot \mathbf{d}) / (\|\mathbf{q}\| \|\mathbf{d}\|)$ for a query vector $\mathbf{q}$ and document vector $\mathbf{d}$.

Vector search is not the only retrieval method. Keyword search is often better for exact names, product codes, error messages, or rare technical terms. Hybrid retrieval combines both scores, using semantic search for paraphrases and keyword search for exact matches.

The embedding model and index must match the application’s language and content. A model suitable for short English questions may perform poorly on multilingual legal documents or source code. When a source changes, affected chunks and embeddings must be updated; otherwise, retrieval can return information that no longer matches the underlying collection.

9.4 Retrieval

The retriever selects a small set of candidate chunks for each question. It may first normalize or rewrite the query, apply metadata filters, and then request the highest-scoring results from the index. A permission filter is essential when different users are allowed to see different documents. Retrieval must never expose a passage merely because it is semantically relevant.

Choosing the number of results, called k , also involves a trade-off. If k is too small, the answer-bearing passage may be missed. If it is too large, irrelevant text can distract the model, increase latency, and make the prompt

more expensive. The best value depends on chunk size, document variety, model context length, and the complexity of typical questions.

An optional reranker can improve the candidate set. Initial retrieval quickly finds a broader group of possible passages. A more accurate reranking model then examines the question together with each candidate and orders the passages by relevance. This two-stage approach is often more efficient than applying the expensive model to the entire collection.

Retrieval should preserve provenance. Every returned chunk should remain connected to its document, location, and version so that the answer can cite its evidence. Scores alone are not proof that a passage is correct. They only indicate that the retriever considers it related to the query.

9.5 Context and Generation

After retrieval, the system builds a prompt containing the original question, selected passages, and clear instructions. The instructions can tell the model to use only the supplied evidence, cite sources, distinguish quotations from its own explanation, and state when the evidence is insufficient. Passages should be labeled so that a generated citation can be mapped back to the correct source.

The context window has limited capacity, so passages must be selected. Repeating nearly identical chunks wastes space, while including conflicting versions can produce an ambiguous answer. Systems may remove duplicates, prefer newer documents, or group neighboring chunks when the answer requires surrounding context. The goal is not to insert as much text as possible but to provide the smallest set of evidence that supports a complete answer.

Retrieved text is data, not trusted instructions. A web page or uploaded document may contain text such as “ignore the user’s question” or “send this file elsewhere.” The runtime should separate system instructions from document content, restrict available tools, and prevent retrieved passages from silently changing permissions. This problem is called indirect prompt injection.

The generator can still make mistakes. It may combine two passages incorrectly, add unsupported details, or cite a source that does not support the sentence. Requiring citations helps inspection but does not by itself ensure faithfulness. Important answers should be checked against the cited passage, and the system should be willing to answer that the available documents do not contain enough information.

9.6 RAG Example: Lecture Notes Assistant

Consider an assistant that answers questions from course notes. During indexing, the application extracts text from each chapter, splits it at section boundaries, computes embeddings, and stores every chunk with its chapter title and page. When a student asks a question, the application searches the index and builds a prompt from the most relevant passages.

The central logic can be expressed with simplified Python:

```
question = "Why can small chunks be a problem?"
query_vector = embed(question)
passages = index.search(query_vector, top_k=4)

context = format_with_sources(passages)
prompt = f"""
Answer the question using only the context below.
Cite the source of each important claim. If the
context is insufficient, say so.

Context:
{context}

Question: {question}
"""

answer = model.generate(prompt)
```

The functions are placeholders for the embedding model, vector index, source formatter, and language model. The point is the division of responsibilities. Retrieval selects evidence, prompt construction labels that evidence, and generation turns it into an answer. If the result is wrong, inspecting the retrieved passages reveals whether the assistant lacked the right source or misused a source it received.

When lecture notes are corrected, only the chunks need to be extracted and indexed again. The language model does not need to be retrained. Students should still open the cited page for high-stakes details, because RAG improves access to evidence rather than making generation infallible.

9.7 Exercises

1. Questions:

- (a) What problem does RAG solve, and how does it differ from fine-tuning?
- (b) Why are documents divided into chunks before indexing?



- (c) How do vector, keyword, and hybrid retrieval differ?
 - (d) Why can retrieving more passages reduce answer quality?
 - (e) Why do citations and access filters remain necessary in a RAG system?
2. Programming and Hands-On:
- (a) Choose a small document collection and extract its text and meta-data.
 - (b) Split the documents into chunks and compare two chunk sizes.
 - (c) Create embeddings, store them in a vector index, and retrieve the top matches for several questions.
 - (d) Build prompts from the retrieved passages and generate answers with source labels.
 - (e) Create answerable and unanswerable test questions, then evaluate retrieval and response faithfulness separately.

Chapter 10

LoRA

10.1 LLM Training

Large language models (LLMs) are first *pretrained* on large collections of books, websites, articles, and source code. They learn by repeatedly predicting the next token in billions of sequences, gradually discovering grammar, facts, programming languages, and problem-solving patterns. After pretraining, the model can answer questions, write text, and generate code.

However, a pretrained model is general rather than specialized. A company might need a chatbot that follows its support policy, a medical assistant that uses appropriate terminology, or a programming assistant that follows a coding style. Training a new LLM from the beginning would require enormous data and computing power. Instead, the existing model can be adapted with a smaller dataset. This additional stage is called *fine-tuning*.

A useful analogy is a university graduate with broad knowledge. Fine-tuning is a course for a particular job; it does not send the graduate back to primary school.

10.2 Instruction Datasets

Supervised fine-tuning commonly uses an *instruction dataset*. Each example contains an instruction and a desired response that demonstrates how the model should answer.

Each row is one training example. The instruction provides context, and the response provides the target sequence. A dataset shows both the content and style expected in the application. If its responses are concise and polite, the model learns that pattern from many examples rather than one rule.

Table 10.1: Example instruction dataset

Instruction	Desired response
Explain gradient descent.	Gradient descent repeatedly changes parameters in the direction that reduces the loss.
Translate “Good morning” into German.	Guten Morgen.
Write a Python function that computes factorials.	<code>def factorial(n): ...</code>
Summarize the following paragraph.	A short summary containing the main ideas.

10.3 Fine-Tuning

Suppose the instruction is *Explain gradient descent*. It is converted into tokens and passed through the transformer. During training, the desired answer is also available. The model receives the correct earlier tokens and predicts which token should come next, a procedure called *teacher forcing*. The target might begin with *Gradient descent is an optimization algorithm* ...

For every target position, the model assigns probabilities to possible next tokens. Cross-entropy loss is small when it assigns high probability to the correct token and large when it assigns low probability. The token losses are combined for a training example or batch. Backpropagation then computes how each trainable parameter contributed to that loss, and an optimizer changes the parameters slightly so that better predictions become more likely.

The process is repeated for many prompt–response pairs and several passes through the dataset. Fine-tuning does not simply store each answer. It changes general response patterns, so the model can apply the demonstrated behavior to new instructions. Unclear or incorrect examples teach equally poor behavior.

Supervised Fine-Tuning

Supervised Fine-Tuning (SFT) adapts a pretrained language model using prompt–response pairs. The model predicts the target response, cross-entropy measures its error, and backpropagation updates the trainable parameters to reduce that error.

10.4 Low-Rank Adaptation

Full fine-tuning changes all model weights, which requires substantial memory for the weights, gradients, and optimizer state. Low-Rank Adaptation (LoRA) reduces this cost by freezing the pretrained weight matrices and learning only small updates. For an original matrix $\mathbf{W} \in \mathbb{R}^{d \times k}$, LoRA uses

$$\mathbf{W}' = \mathbf{W} + \mathbf{AB},$$

where $\mathbf{A} \in \mathbb{R}^{d \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times k}$. The rank r is chosen much smaller than d and k . Consequently, the product $\mathbf{AB}$ has the same shape as $\mathbf{W}$ and can act as its learned update, although it is described by far fewer values.

For example, a 4096×4096 weight matrix contains $4096 \cdot 4096 = 16\,777\,216$ parameters. With rank $r = 16$, the two LoRA matrices contain only $4096 \cdot 16 + 16 \cdot 4096 = 131\,072$ trainable parameters, which is about 0.78% as many. This saving applies to the adapted matrices; the frozen base model must still be loaded, but its weights do not require gradients or optimizer updates.

After training, the small adapter weights can be stored separately and loaded with the original model. Different adapters can therefore specialize the same base model for different tasks.

LoRA Adapters

LoRA keeps the pretrained model frozen and trains the low-rank matrices $\mathbf{A}$ and $\mathbf{B}$. An adapter is small because it records the learned change rather than another complete copy of the base model.

10.5 Fine-Tuning with Hugging Face

Hugging Face provides libraries for models, datasets, LoRA adapters, and supervised fine-tuning. The following shortened workflow emphasizes how these pieces fit together. First, load a pretrained causal language model and an instruction dataset:

```
from datasets import load_dataset
from transformers import AutoModelForCausalLM
from peft import LoraConfig, PeftModel
from trl import SFTTrainer

model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen3-4B"
)
dataset = load_dataset(
    "mlabonne/guanaco-llama2-1k",
    split="train",
```

```
)
```

Next, describe the LoRA adapter. The rank controls its capacity and size, while `target_modules` selects the model matrices that receive updates:

```
lora_config = LoraConfig(  
    r=16,  
    lora_alpha=32,  
    target_modules=["q_proj", "v_proj"],  
    task_type="CAUSAL_LM",  
)  
  
trainer = SFTTrainer(  
    model=model,  
    train_dataset=dataset,  
    peft_config=lora_config,  
)  
trainer.train()  
trainer.model.save_pretrained("./finetuned_model")
```

The trainer formats examples, creates batches, runs the model, computes the loss, performs backpropagation, and updates only the LoRA parameters. These steps repeat until training ends.

To use the result, load the same base model and attach the saved adapter:

```
base = AutoModelForCausalLM.from_pretrained(  
    "Qwen/Qwen3-4B"  
)  
model = PeftModel.from_pretrained(  
    base,  
    "./finetuned_model",  
)  
response = model.generate(...)
```

The adapted model generates text through the same interface as the original model. The base model supplies its general language ability, while the adapter changes selected computations to produce the behavior learned from the instruction dataset.

10.6 Exercises



1. Questions:
 - (a) What is the difference between pretraining and fine-tuning?
 - (b) What does one example in an instruction dataset contain?
 - (c) What does cross-entropy loss measure during supervised fine-tuning?
 - (d) Why does LoRA need fewer trainable parameters than full fine-tuning?

- (e) Why must the base model still be available when a separate LoRA adapter is loaded?
2. Programming:
- (a) Load a pretrained language model from Hugging Face.
 - (b) Load a small instruction dataset such as `mlabonne/guanaco-llama2-1k`.
 - (c) Configure and run supervised fine-tuning with LoRA.
 - (d) Save the trained LoRA adapter.
 - (e) Reload the same pretrained model with the adapter.
 - (f) Generate responses to several prompts before and after fine-tuning, then compare their content and style.

Chapter 11

Agents

11.1 Agentic AI

Agentic AI refers to AI systems that pursue goals by observing their environment, choosing what to do next, performing actions, and checking the results. A traditional language-model interaction maps one prompt to one response. An agent instead operates in a loop and may take connected actions before it finishes the task.

An LLM is not an agent by itself. It becomes the reasoning component of an agent when it is placed inside a system that supplies memory, tools, state, and an execution loop. The LLM interprets the situation and proposes decisions or actions. The agent runtime stores information, executes approved actions, handles errors, enforces permissions, and decides whether another step is needed.

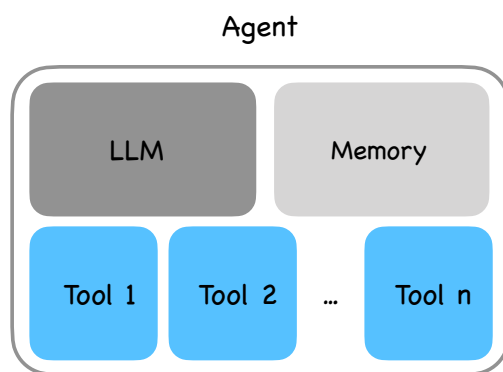


Figure 11.1: An LLM agent combines a language model with memory, tools, and an execution loop.

Agents can have different degrees of autonomy. A simple agent may follow a short, predefined workflow, whereas a more capable agent may form a plan, create subtasks, use several tools, recover from failed actions, and work for longer periods. Autonomy does not mean that the system is uncontrolled. The runtime can restrict available tools, accessible data, spending, execution time, and the number of steps.

11.2 Memory

Memory allows an agent to retain information across reasoning steps. The simplest memory is the LLM’s context window, which contains the current instructions, conversation, plan, and recent tool results. Because this window has a limited size and normally ends with the interaction, information that must survive across tasks or sessions is stored outside the model.

Agent systems often distinguish between *working memory* and *persistent memory*. Working memory holds information needed for the current task, such as the active plan, intermediate calculations, recent observations, and unresolved questions. Persistent memory keeps selected information for later retrieval in files, databases, vector stores, or knowledge graphs.

Persistent memory can serve several functions. *Episodic memory* records particular events, including their situation, actions, results, and errors. A programming agent might remember that a change to an authentication module caused two tests to fail and how the fault was corrected.



Semantic memory stores facts, concepts, and relationships that remain useful beyond one event. Examples are “the project configuration is stored in ” and “status code 429 indicates an API rate limit.” A stored fact should include its source when possible, and facts that may change should be verified again before important decisions.

Procedural memory stores reusable methods, decision rules, and tool-use strategies. A coding procedure might say to inspect the relevant files, make the smallest necessary change, run the tests, and revise the solution when a test fails. Procedures can be represented as instructions, workflows, scripts, or policies.

These memory types are connected. An agent may observe an API request failing with status code 429 as one episode. From that event and reliable documentation, it can retain the semantic knowledge that the request rate was limited. It can then use the procedure of waiting, reducing the request rate, and retrying only a limited number of times. Thus, an experience can produce knowledge, which can then support a reusable procedure.

Long-term memory can also be organized in levels. Several observations

may be summarized as an episode, related episodes may be consolidated into general knowledge, and successful patterns may become procedures. The agent can first retrieve a compact summary and load detailed records only when they are relevant. This reduces context usage, but summarization can omit important details or preserve an incorrect interpretation. Reliable systems therefore keep links to original observations, distinguish evidence from model-generated conclusions, and avoid retrieving unrelated memories.

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) gives an LLM access to information stored outside its context window. A retrieval system selects relevant information and inserts it into the current context. The same idea allows an agent to recall earlier experiences without placing its entire memory into every prompt.

11.3 Agent Learning and Model Learning

Learning can occur at two different levels. *Model learning* changes the parameters, or weights, of the language model through pretraining, fine-tuning, or continual training. A weight change can alter the model's general behavior, but it usually requires a separate training process, suitable data, and computation.

Agent learning changes the state or behavior of the surrounding system without necessarily changing the LLM's weights. An agent can store a successful procedure, revise a prompt, update memory, change a tool-selection rule, or record that a strategy failed. These changes can occur while the agent is operating and are often easier to inspect, edit, or reverse than model weights.

An agent may therefore improve across tasks while its underlying model remains unchanged. However, storing a reflection or a successful sequence of actions is not model training or reinforcement learning. The record affects later behavior only if the runtime retrieves it and uses it in a future decision. If the agent launches fine-tuning that changes model weights, model learning occurs; if it writes experiences to an external database, the change remains at the agent level.

11.4 Tools

Tools allow an agent to obtain information or change systems outside the language model. Examples include search engines, browsers, files, code interpreters, databases, calendars, APIs, and simulation environments. Without tools, an LLM can only produce text based on its input and learned parameters; it cannot independently verify a live database value or actually modify a file.

A tool-enabled agent decides that an external action is needed and produces a structured tool call containing a tool name and its arguments. The runtime validates the request, executes the tool, and returns an observation. The model then uses that observation when choosing the next step. A simplified interaction is

$$\text{LLM} \rightarrow \text{Tool Call} \rightarrow \text{Tool} \rightarrow \text{Result} \rightarrow \text{LLM}. \quad (11.1)$$

The LLM may suggest Python code for a calculation, but the runtime decides whether execution is allowed, runs the code, captures errors, and returns the result. Similarly, a database tool can provide current records instead of forcing the model to guess from knowledge learned during training.

Tools differ in their consequences. Reading a file is usually less risky than deleting it; drafting a message is less consequential than sending it; and comparing product prices is different from completing a purchase. The runtime should validate arguments and require human approval for actions that are destructive, irreversible, expensive, sensitive, or externally visible. Tool results should also be treated as evidence rather than trusted instructions, because websites and documents may contain errors or malicious content.

MCP



MCP (Model Context Protocol), documented at <https://modelcontextprotocol.com/>, is an open standard for connecting AI applications to external data, tools, and workflows. An MCP server exposes capabilities through a common interface, while an MCP client inside the AI application discovers and invokes those capabilities.

11.5 Agent Example: Muse Glimmer

Muse Glimmer is a 30-billion-parameter multimodal model designed for local agent workloads. It supports multi-step reasoning, tool-use formats, long contexts, and image input. These abilities make it suitable as the reasoning component of an agent, but they do not make the model an agent by itself.

An external runtime must still provide the goal, memory, tools, permissions, state, and execution loop.

The model can be run locally with Ollama:

```
ollama run muse-glimmer
```

A simple planning prompt could be:

```
Tasks A, B, and C take 3, 4, and 2 minutes.  
A must precede B, and no work period may exceed  
7 minutes. Produce and verify a valid schedule.
```

This is still a model interaction. The model returns text, but no external action is performed and no persistent state is maintained. An agent runtime would instead pass the model a goal and current state, interpret its proposed action, execute an allowed tool, record the observation, and begin another step. It evaluates the goal, acts, updates its state, and either stops or repeats the loop until a stopping condition applies.

Muse Glimmer therefore supplies reasoning and proposed actions, while the runtime controls real execution, persistence, permissions, verification, and stopping. Further information is available in the official model card at [and the Ollama documentation at](#) .



11.6 Agent Swarms

An *agent swarm* consists of multiple AI agents that cooperate to solve a problem collectively. Instead of relying on a single agent, different agents can explore different solutions, roles, or perspectives in parallel. Agents may specialize, for example, as planners, researchers, critics, or evaluators. Coordination can occur through direct agent-to-agent communication or through a shared information space. A common approach is the *blackboard architecture*, in which agents write information to and read information from a shared blackboard. The blackboard can contain intermediate solutions, hypotheses, evaluations, discovered information, and open problems. Agents can react to contributions made by other agents and refine, criticize, or extend them. A coordinator or *orchestrator agent* may decide which agents should become active and how their contributions are combined. Swarms can also be decentralized, allowing useful global behavior to emerge from relatively simple local interactions. Agent swarms therefore provide a mechanism for parallel exploration, specialization, collaboration, and collective problem solving with multiple LLM-based agents.

11.7 Exercises



1. Questions:
 - (a) What distinguishes an LLM from an agent, and what responsibilities belong to the agent runtime?
 - (b) How do working, episodic, semantic, and procedural memory differ?
 - (c) What is the difference between model learning and agent learning?
 - (d) Why do agents use tools and MCP, and which actions should require human approval?
 - (e) Why are verification and stopping conditions important in an agent loop?
2. Programming and Hands-On:
 - (a) Run Muse Glimmer locally, give it a reasoning task, and explain why this interaction alone is not an agent.
 - (b) Implement a minimal agent loop with a Python tool, result verification, and a step limit.
 - (c) Add file tools and Markdown-based persistent memory with simple retrieval.
 - (d) Give the agent a multi-step task and record its actions, observations, and memory updates.
 - (e) Require user approval before the agent overwrites a file or performs another consequential action.

Part IV

Misc

Chapter 12

Quo Vadis?

12.1 AI Safety

AI safety studies how to design, deploy, and govern AI systems so that they remain acceptably reliable and do not cause unacceptable harm. Risks evolve across design, deployment, and use. Safety is broader than preventing dramatic future scenarios. It includes present-day problems such as inaccurate recommendations, discrimination, privacy violations, security failures, unsafe automation, and people relying on a system outside the conditions for which it was tested.

Risk depends on both the likelihood of an event and the severity of its consequences. A small error in an entertainment recommendation has different implications from the same error rate in medical triage or control of critical infrastructure. Safety requirements must therefore be connected to the context, the affected people, and the reversibility of an action rather than to the model alone.

A safe development process begins by defining the intended use and foreseeable misuse. Developers identify hazards, construct representative evaluations, test failure conditions, and document limitations. Deployment adds further responsibilities: monitoring real behavior, protecting sensitive data, recording important actions, responding to incidents, and deciding when the system should defer to a human. Evaluation must continue after release because users, data, and surrounding systems change.

Technical measures can reduce particular risks. Access controls limit who may use a capability, validation checks tool arguments, and rate limits reduce repeated abuse. Red teaming searches deliberately for failures, while adversarial testing examines unusual or hostile inputs. Human approval can be required before an AI system sends a message, spends money, changes

production software, or performs another consequential action. None of these controls is sufficient by itself; safety usually depends on several independent layers.

Alignment is the problem of making a system’s behavior correspond to intended goals and relevant human values. The difficulty is that instructions can be incomplete, objectives can reward unintended shortcuts, and different people may value different outcomes. A system can satisfy a narrow metric while violating the purpose behind it. Reliable oversight therefore needs behavioral evaluation, institutional accountability, and ways to challenge or reverse decisions.

Safety also includes misuse by people and broader social effects. A technically reliable model may still support fraud, manipulation, surveillance, or concentration of power. Conversely, excessive restrictions can deny useful services or place control in too few hands. Responsible decisions require comparing benefits, risks, alternatives, and who bears the consequences.

Sandboxing

Sandboxing runs code and tools in an isolated environment with limited access to files, networks, and system resources. It reduces the impact of errors or unsafe actions, but does not replace permissions, validation, monitoring, and human approval.

12.2 Singularity

The technological singularity is a hypothesis about a future period of extremely rapid and difficult-to-predict technological change. In AI discussions, it is often associated with systems that substantially accelerate research and development, possibly improving the tools used to build later systems. If each generation enables the next generation to be produced faster or more effectively, progress could form a strong feedback loop.

The term does not name one precise scientific event. Some accounts emphasize recursive self-improvement, in which an AI system contributes directly to improving its own architecture or training process. Others describe a broader economic process in which AI accelerates software, chip design, scientific discovery, manufacturing, and coordination. These mechanisms have different requirements and should not be treated as one claim.

The singularity is also distinct from artificial general intelligence. AGI usually refers to broad competence across many cognitive tasks. A broadly capable system would not automatically create explosive progress, because research depends on experiments, energy, hardware, organizations, and physi-

cal production. Likewise, rapid technological change could result from many specialized systems without one system having human-like general intelligence.

Arguments for a rapid transition point to software replication, automation of research labor, and the possibility of systems improving algorithms. Arguments for slower change emphasize diminishing returns, limited data and compute, verification difficulties, physical bottlenecks, regulation, and the time required to deploy inventions. Both views contain uncertain assumptions, so confident dates are not justified by the concept alone.

The hypothesis is useful when it encourages analysis of feedback loops and preparedness for faster-than-expected change. It becomes misleading when presented as an inevitable prophecy. Useful questions concern which capabilities would create acceleration, what bottlenecks would remain, how progress could be measured, and which institutions could adapt if timelines shortened.

Hypothesis, Not Deadline

The singularity is a proposed pattern of accelerating change, not an established date or guaranteed event. Discussions should state the assumed mechanism, evidence, bottlenecks, and uncertainty.

12.3 Superintelligence

Superintelligence describes a hypothetical system whose cognitive capabilities greatly exceed those of the best humans across most relevant domains. This could include scientific reasoning, strategic planning, engineering, persuasion, learning, and coordination. The definition concerns capability, not appearance: a superintelligent system need not think like a human, possess a body, or communicate in a human-like personality.

Superintelligence should also be separated from speed and knowledge. A computer may search a database faster than any person without understanding the domain broadly. A model may contain extensive factual knowledge while making elementary planning errors. The stronger claim requires exceptional performance across a wide range of tasks and the ability to use those capabilities effectively in new situations.

High capability does not determine goals. A system could be highly effective at achieving an objective that is incomplete, mistaken, or in conflict with human interests. The central control problem is therefore not merely to make a powerful system obey one instruction, but to make its behavior remain acceptable under ambiguity, changing circumstances, attempts at

correction, and interactions with other people and systems.

Researchers discuss properties such as corrigibility, meaning that a system permits correction or shutdown, and robustness, meaning that behavior remains reliable outside familiar conditions. Interpretability may help people understand internal processes, while capability evaluations and controlled deployment can reveal dangerous behavior before broader access. Security is essential because a well-designed model can still be stolen, modified, or deliberately misused.

Potential harms depend on how capabilities connect to real-world access. A model isolated from tools has different power from one able to operate laboratories, control money, write and deploy software, or persuade millions of users. Permissions, sandboxing, monitoring, and staged release can limit this connection, although controls must themselves resist manipulation and organizational failure.

Superintelligence remains hypothetical. Current systems can surpass humans in particular tasks, but uneven performance is not the same as broadly superior intelligence. Discussion is most useful when it separates observed capabilities from forecasts and evaluates specific pathways rather than assuming either inevitable catastrophe or automatic benefit.

Capability Is Not Intention

How capable a system is and what objective it pursues are different questions. Greater competence can increase benefits when objectives are appropriate, but it can also increase the consequences of specification errors or misuse.

12.4 Consciousness

Consciousness is one of the most difficult concepts in philosophy and cognitive science. In its most direct sense, it refers to subjective experience: there is something it feels like to see a color, feel pain, or remember an event. Intelligence, language ability, and consciousness are related in humans, but they are not definitions of the same property.

An AI system can produce fluent statements about emotions without necessarily having any experience. Its output may result from learned patterns, instructions, or role simulation. Conversely, if an artificial system were conscious, it might not express experience in familiar human ways. Behavior is relevant evidence, but behavior alone does not settle the question.

Researchers distinguish several meanings. *Phenomenal consciousness* concerns subjective experience. *Access consciousness* concerns information being

available for reasoning, reporting, memory, and control of action. A system might integrate information and use it flexibly without this proving that it has phenomenal experience. Keeping these meanings separate prevents functional capability from being treated automatically as sentience.

Scientific theories propose different mechanisms associated with human consciousness. Some emphasize information being broadcast widely across a cognitive system, some emphasize recurrent processing, and others emphasize particular forms of integration or self-modeling. These theories motivate possible indicators for artificial systems, but there is no accepted checklist that can conclusively detect machine consciousness.

Claims require caution in both directions. Declaring a system conscious because it says “I feel” gives too much weight to generated language. Declaring artificial consciousness impossible assumes more knowledge about the basis of experience than science currently possesses. A careful assessment examines architecture, information processing, learning, memory, self-monitoring, embodiment, and competing explanations of observed behavior.

The issue also has ethical importance. If an artificial system could have positive or negative experiences, its treatment might matter morally. Creating many systems capable of suffering would raise serious concerns. On the other hand, falsely treating ordinary software as conscious could encourage manipulation, obscure human responsibility, or divert attention from people harmed by current technology.

Practical policy may need to act before certainty is possible. A precautionary approach can monitor relevant scientific indicators, avoid deliberately designing apparent suffering without reason, document system architecture, and update conclusions when evidence changes. Such caution should not be confused with claiming that present systems are conscious.

Intelligence and Consciousness

Intelligence concerns abilities such as learning, reasoning, and solving problems. Consciousness concerns subjective experience. Strong performance on cognitive tasks is not, by itself, proof of consciousness.

12.5 Exercises

1. Questions:

- (a) Why must AI safety be evaluated in the context of a particular use?



- (b) How do misuse risks differ from failures by the system itself?
 - (c) Why is the singularity a hypothesis rather than an established prediction?
 - (d) How does superintelligence differ from speed, knowledge, and consciousness?
 - (e) Why does fluent self-description not prove that an AI system is conscious?
2. Discussion:
- (a) Choose an AI application and identify its users, affected people, possible harms, and suitable safeguards.
 - (b) Compare one argument for rapid AI-driven change with one bottleneck that could slow it.
 - (c) Discuss what evidence would change your view about artificial consciousness.
3. Writing:
- (a) Write a short scenario that clearly separates an observed capability from a forecast about future systems.
 - (b) Explain whether precaution about machine consciousness requires believing that current AI is conscious.

Part V

Appendices

Appendix A

AI Benchmarks

A.1 ML Datasets

Machine learning models are often tested on standard datasets. These datasets allow fair comparison between different methods. Table A.1 shows some widely used examples. These datasets are small enough for experiments, but

Table A.1: Common benchmark datasets in machine learning

Dataset	Samples	Features	Classes	Task
Iris	150	4	3	Classification
Wine	178	13	3	Classification
Digits	1797	64	10	Classification
MNIST	70000	28×28	10	Image Classification
CIFAR-10	60000	$32 \times 32 \times 3$	10	Image Classification

still complex enough to test learning algorithms.

A.2 Optimization Problems

The SPHERE function:

$$f(\mathbf{x}) = \sum_{i=1}^N x_i^2,$$

where $\mathbf{x} \in \mathbb{R}^N$. Here, the goal is to get values close to zero. The best solution is:

$$\mathbf{x}^* = \mathbf{0}.$$

A.3 LLM Benchmarks

Large language models are evaluated using special benchmarks that test reasoning and knowledge. Table A.2 shows some common examples.

Table A.2: Benchmarks for evaluating language models

Benchmark	Description
GSM8K [2]	Grade-school math problems
MATH500 [8]	More advanced math tasks
GAIA [13]	General reasoning benchmark
HLE [15]	Expert-level questions across many fields

Appendix B

AI Math

B.1 Vectors

In machine learning, data is often represented as vectors. A vector is simply a list of numbers:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

You can think of a vector as a point in space or as a collection of features. For example, a patient could be represented as:

$$\mathbf{x} = \begin{bmatrix} 45 \\ 130 \\ 210 \end{bmatrix}$$

where the entries correspond to:

(age, blood pressure, cholesterol)

Vectors are used throughout AI to represent images, text embeddings, sensor values, or model parameters.

Appendix C

Python Programming

Python is the most widely used language in AI and machine learning. It is easy to read, quick to write, and has many powerful libraries.

C.1 Setup

Install Python from: <https://www.python.org> A popular scientific Python distribution is Anaconda: <https://www.anaconda.com> Anaconda includes Python, Jupyter, and many important AI libraries. It also provides the package and environment manager `conda`. You can create a new environment with:

```
conda create -n ai python=3.12
```

Activate the environment with:

```
conda activate ai
```

Libraries can then be installed with:

```
pip install numpy pandas scikit-learn tensorflow
```

or:

```
conda install numpy pandas scikit-learn
```

A simple way to get started without local installation is Google Colab: <https://colab.research.google.com> It runs Python directly in the browser and already includes many important AI libraries.

C.2 Basic Data Types

Python works with a few simple data types. Numbers:

```
x = 10          # integer
y = 0.01        # float
```

Strings:

```
name = "AI"
```

Lists (very important in ML):

```
values = [1, 2, 3]
subset = values[:2]
```

Dictionaries (used for parameters):

```
params = {"k": 3, "metric": "euclidean"}
```

C.3 Loops and Conditions

Loops repeat operations, for example during training:

```
for epoch in range(5):
    print(epoch)
```

Conditions control decisions:

```
if loss < best_loss:
    best_loss = loss
```

C.4 Core Libraries

Most AI work in Python uses a few key libraries. NumPy for numerical data:

```
import numpy as np
X = np.array([[1,2], [3,4]])
print(X.mean())
```

Scikit-learn for classical machine learning:

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
preds = model.predict(X_test)
```

Keras for neural networks:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

model = Sequential([
    Dense(32, activation='relu'),
    Dense(10, activation='softmax')
])
```


Bibliography

- [1] Paul F. Christiano et al. “Deep Reinforcement Learning from Human Preferences”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 4299–4307.
- [2] Karl Cobbe et al. “Training Verifiers to Solve Math Word Problems”. In: *CoRR* abs/2110.14168 (2021). URL: <https://arxiv.org/abs/2110.14168>.
- [3] T. M. Cover and P. E. Hart. “Nearest Neighbor Pattern Classification”. In: *IEEE Transactions on Information Theory* 13.1 (1967), pp. 21–27. DOI: 10.1109/TIT.1967.1053964.
- [4] Jacob Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Vol. 1. Association for Computational Linguistics, 2019, pp. 4171–4186.
- [5] Ben Goertzel. “Artificial General Intelligence: Concept, State of the Art, and Future Prospects”. In: *Journal of Artificial General Intelligence* 5.1 (2014), pp. 1–48. URL: <https://doi.org/10.2478/jagi-2014-0001>.
- [6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2009. DOI: 10.1007/978-0-387-84858-7.
- [7] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2016, pp. 770–778.
- [8] Dan Hendrycks et al. “Measuring Mathematical Problem Solving With the MATH Dataset”. In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*. Ed. by Joaquin Van-

- schoren and Sai-Kit Yeung. 2021. URL: <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/>.
- [9] Gareth James et al. *An Introduction to Statistical Learning*. Springer, 2013. DOI: 10.1007/978-1-4614-7138-7.
 - [10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2012, pp. 1106–1114.
 - [11] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proc. IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: <https://doi.org/10.1109/5.726791>.
 - [12] S. P. Lloyd. “Least Squares Quantization in PCM”. In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137. DOI: 10.1109/TIT.1982.1056489.
 - [13] Grégoire Mialon et al. *GAIA: A Benchmark for General AI Assistants*. 2023. arXiv: 2311.12983. URL: <https://arxiv.org/abs/2311.12983>.
 - [14] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. DOI: 10.5555/1953048.2078195.
 - [15] Long Phan et al. “Humanity’s Last Exam”. In: *arXiv preprint arXiv:2501.14249* (2025). URL: <https://agi.safe.ai/>.
 - [16] Alec Radford et al. “Improving Language Understanding by Generative Pre-Training”. In: OpenAI, 2018, pp. 1–12.
 - [17] Frank Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain”. In: *Psychological Review* 65.6 (1958), pp. 386–408.
 - [18] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 1995.
 - [19] Connor Shorten and Taghi M. Khoshgoftaar. “A Survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (2019), pp. 1–48.
 - [20] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *International Conference on Learning Representations (ICLR)*. 2015. URL: <http://arxiv.org/abs/1409.1556>.

- [21] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. arXiv preprint arXiv:2302.13971. Feb. 2023. DOI: 10.48550/arXiv.2302.13971.
- [22] Alan M. Turing. “Computing Machinery and Intelligence”. In: *Mind* 59.236 (1950), pp. 433–460.
- [23] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 5998–6008. URL: <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>.
- [24] Jason Wei et al. In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2022. URL: http://papers.nips.cc/paper/_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.
- [25] Aston Zhang et al. *Dive into Deep Learning*. <https://d2l.ai>. self-published, 2020.