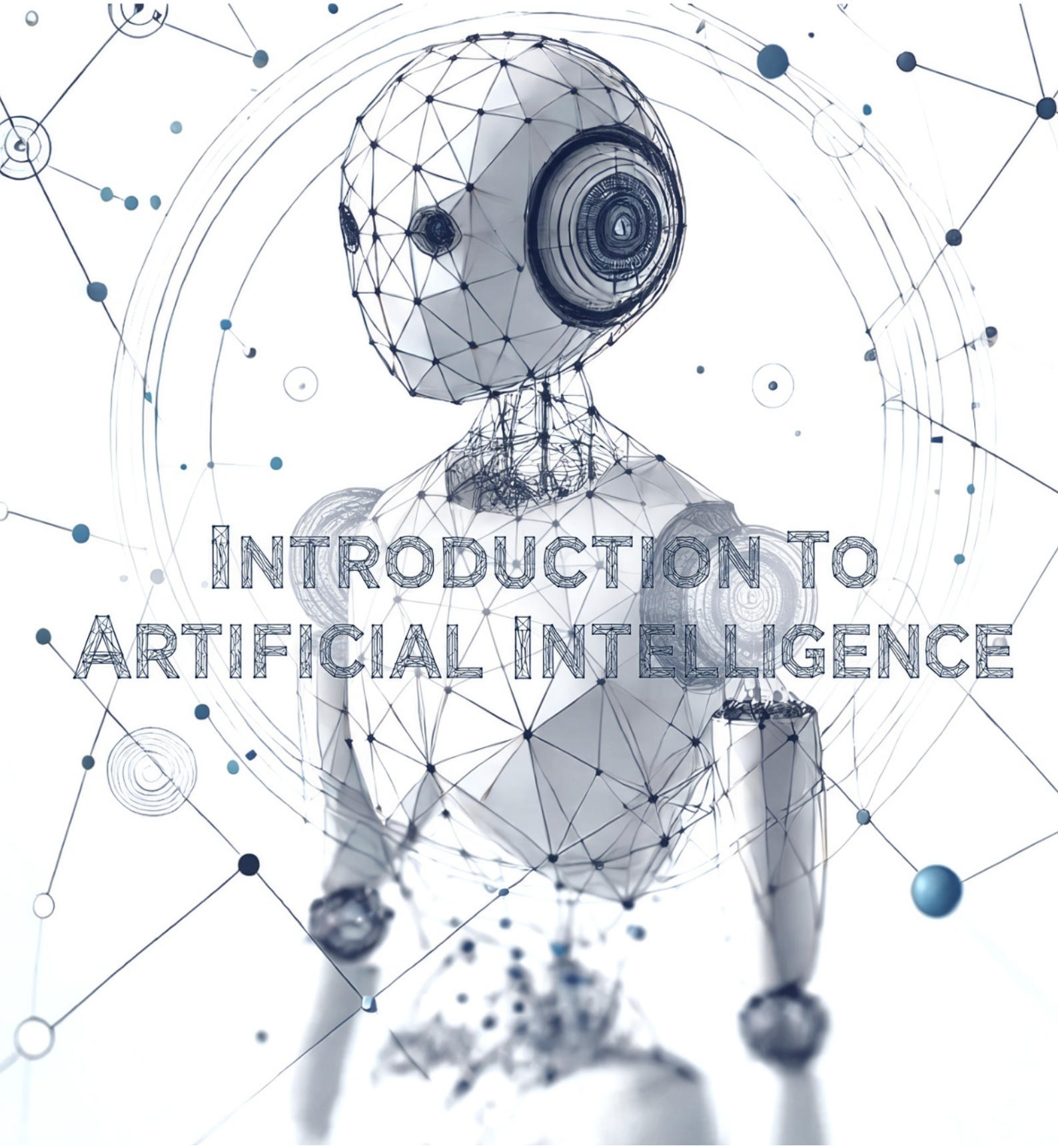

Introduction To
Artificial Intelligence

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. 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 20, 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 **5**

2 K-Nearest Neighbors **7**
2.1 Learning from Nearby Examples . . . 7
2.2 Choosing k and the Distance . . . 8
2.3 KNN in `Python` . . . 9
2.4 Exercises . . . 10

3 Support Vector Machines **13**
3.1 Maximum-Margin Classification . . . 13
3.2 Soft Margins and Kernels . . . 14
3.3 SVM in `Python` . . . 15
3.4 Exercises . . . 17

4 K-Means Clustering **19**
4.1 Clusters and Centroids . . . 19
4.2 The Algorithm and Its Limitations . . . 19
4.3 K-Means in `Python` . . . 21
4.4 Exercises . . . 22

II Deep Learning **23**

5 Multilayer Perceptrons **25**

5.1 Dense Neural Networks . . . 25
5.2 Training and Generalization . . . 26
5.3 An MLP in `Keras` . . . 28
5.4 Exercises . . . 29

6 Convolutional Neural Networks 31
6.1 Convolution and Feature Maps . . . 31
6.2 Stride, Padding, and Pooling . . . 32
6.3 A CNN in `Keras` . . . 33
6.4 Exercises . . . 35

7 Attention 37
7.1 Selecting Relevant Information . . . 37
7.2 Self-Attention and Multi-Head Attention . . . 38
7.3 Cost and a Keras Example . . . 39
7.4 Exercises . . . 40

III LLMs 43

8 Generative Pre-Trained Transformers 45
8.1 Decoder-Only Language Models . . . 45
8.2 Pretraining, Adaptation, and Generation . . . 46
8.3 A Small GPT Block in `Keras` . . . 47
8.4 Exercises . . . 48

9 Prompting 51
9.1 Instructions, Context, and Output . . . 51
9.2 Examples and Structured Prompts . . . 52
9.3 Chain-of-Thought and Decomposition . . . 53
9.4 Reliability and Prompt Injection . . . 54
9.5 Exercises . . . 54

10 Low-Rank Adaptation 55
10.1 From Fine-Tuning to LoRA . . . 55
10.2 Low-Rank Updates . . . 55
10.3 Training and Deployment . . . 57
10.4 A LoRA Layer in `Keras` . . . 57
10.5 Exercises . . . 58

IV Appendices **61**

A AI Benchmarks **63**
A.1 ML Datasets 63
A.2 LLM Benchmarks 63

B AI Math **65**
B.1 Vectors and Distances 65
B.2 Averages and Scaling 65
B.3 Weighted Sums and Matrices 66
B.4 Probabilities and Softmax 66
B.5 Loss and Gradients 67
B.6 Rank and LoRA 67

C Python Programming **69**
C.1 Setup 69
C.2 Basic Data Types 69
C.3 Loops and Conditions 70
C.4 Core Libraries 70

Bibliography **73**

Chapter 1

Introduction

1.1 What is AI?

Artificial intelligence (AI) is the field of building systems that perform tasks associated with human intelligence, such as perception, language use, reasoning, learning, and decision-making. Modern AI includes rule-based approaches, classical machine-learning algorithms, and neural networks. Deep learning has driven many recent advances in language, vision, and generative AI. 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; Ingo Rechenberg and Hans-Paul Schwefel begin developing Evolution Strategies (ES).
3. *1970s:* First AI Winter due to limited compute and overpromising; John Holland formalizes Genetic Algorithms (GAs), while Evolution Strategies are developed further.
4. *1980s:* Rise of expert systems; renewed interest in multi-layer neural networks (MLPs), accelerated by effective backpropagation training in 1986. The limitations of single-layer perceptrons, including their inability to represent XOR, had contributed to earlier skepticism rather than causing the revival.
5. *1990s:* Statistical learning theory gains traction; Corinna Cortes and Vladimir Vapnik introduce 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 multimodal tasks.

Figure 1.1 summarizes these epochs on a single timeline. The first AI winter is highlighted, and the most recent decades, dominated by deep learning and LLMs, are shown in light blue.

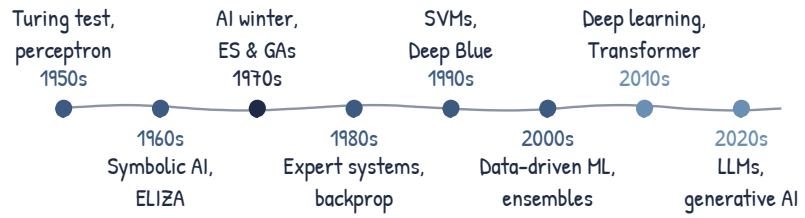


Figure 1.1: Key epochs in the history of AI, from the perceptron to LLMs.

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.  This icon indicates links to recommended reading or further literature, helping you dive deeper into the topic.
2.  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.

Interactive Colab notebook. A companion Google Colab notebook collects the complete, self-contained Python examples from this book. It can be run directly in a web browser without a local Python installation: open the Introduction to AI Colab notebook.

1.4 Further Reading: AI Textbooks

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

1. *An Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani [7]: An introduction to statistical learning. 
2. *Dive into Deep Learning* by Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola [19]: A practical, interactive guide to deep learning with code examples in `Python`. 
3. *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig [15]: 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

Chapter 2

K-Nearest Neighbors

2.1 Learning from Nearby Examples

K-nearest neighbors (KNN) is a supervised method for classification and regression. It is based on the idea that similar observations often have similar outcomes. To classify a new observation, KNN finds the k closest training observations and lets their labels vote [4].

For numerical features, closeness is often measured with Euclidean distance:

$$\delta(\mathbf{x}, \mathbf{z}) = \sqrt{\sum_{j=1}^{d} (x_j - z_j)^2}.$$

The distance is calculated across all d features. In classification, the most frequent label among the neighbors becomes the prediction. In regression, the prediction is usually the mean of their target values.

KNN has no separate model-fitting stage in the usual sense. It stores the training data and performs most of its work when a prediction is requested. It is therefore called a *lazy* or *instance-based* learner.

K-Nearest Neighbors

KNN predicts from nearby training examples. Classification uses a vote, while regression uses an average. The meaning of nearby depends on the features, their scales, and the chosen distance.

Figure 2.1 illustrates this procedure for $k = 5$. The new observation is assigned the label held by the majority of the five training points inside the dashed circle.

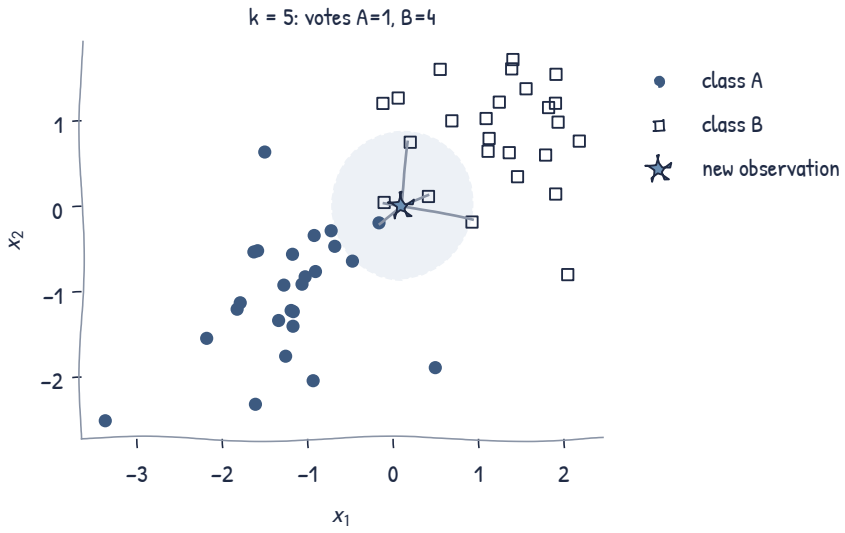


Figure 2.1: KNN classification with $k = 5$: the new observation receives the majority label of its five nearest neighbors.

2.2 Choosing k and the Distance

The value of k controls how local the prediction is. With $k = 1$, the closest observation determines the result. This can follow noise and create an irregular decision boundary. A larger k averages over more examples and produces a smoother boundary, but a value that is too large can hide local structure and favor the majority class.

Uniform voting gives every neighbor the same influence. Distance weighting gives closer observations more influence than distant ones. Both choices should be compared with validation data. An odd value of k can reduce ties in binary classification, but it does not guarantee that ties never occur.

Feature scaling is essential. If age ranges from 0 to 100 and income from 0 to 100000, income can dominate Euclidean distance. Standardization places the features on comparable scales. The scaler must be fitted on the training data and then applied unchanged to test and future data.

KNN can become slow when the training set is large because prediction requires many distance calculations. It also becomes less reliable in high-dimensional spaces, where distances between observations often become less distinct. Feature selection or dimensionality reduction may help.

Bias and Variance

A small k gives a flexible model with low bias and high variance. A large k gives a smoother model with higher bias and lower variance. Cross-validation helps find a useful balance.

Figure 2.2 compares $k = 1$ and $k = 15$ on the same data. The small k produces an irregular boundary that reacts to single noisy points, while the larger k yields a smoother boundary.

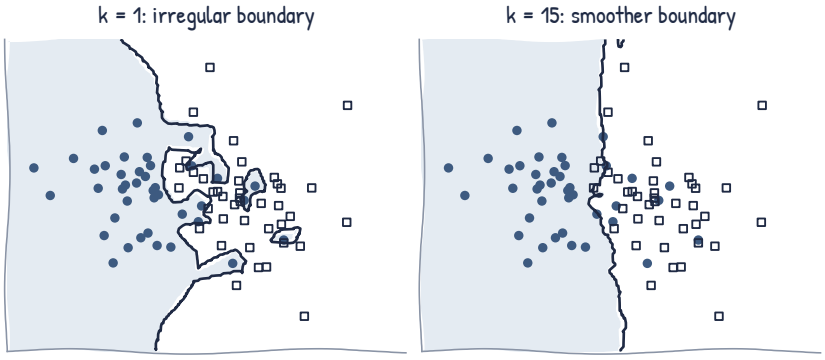


Figure 2.2: Effect of k: $k = 1$ follows noise, a larger k yields a smoother decision boundary.

2.3 KNN in Python

The following example classifies Iris flowers. A pipeline learns the scaling from the training data before applying KNN.

```
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

X, y = load_iris(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=42,
    stratify=y,
)
```

```
model = make_pipeline(
    StandardScaler(),
    KNeighborsClassifier(
        n_neighbors=5,
        weights="distance",
    ),
)

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion matrix:")
print(confusion_matrix(y_test, y_pred))
```

The scaler prevents features with larger numerical ranges from dominating the distance. The classifier uses five neighbors and weights their votes by distance. Accuracy summarizes all correct predictions, while the confusion matrix shows which species are confused.

The best k, weighting rule, and distance metric depend on the dataset. They should be chosen through cross-validation without using the final test set.

Accuracy

Accuracy measures how often the model is right: the number of correctly classified predictions over the total number of predictions. It gives a single, intuitive fraction between 0 and 1, where 1 means every prediction was correct.

2.4 Exercises



1. Questions:
   (a) How does KNN classify a new observation?
   (b) What changes when k becomes larger?
   (c) Why is feature scaling important for KNN?
   (d) Why can KNN be slow at prediction time?
2. Hands-On and Programming:
   (a) Using class A points $(1, 1)$, $(2, 1)$, and $(2, 2)$, and class B points $(4, 3)$, $(4, 4)$, and $(5, 4)$, classify the point $(3.5, 2.5)$ with $k = 3$ and Euclidean distance.
   (b) Run the example with $k = 1$, 5, and 15, then compare the confusion matrices.

(c) Compare uniform and distance-weighted voting with cross-validation.

Chapter 3

Support Vector Machines

3.1 Maximum-Margin Classification

A support vector machine (SVM) is a supervised method for classification and regression [3]. For binary classification, a linear SVM separates two classes with a hyperplane:

$$\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0.$$

A new observation is classified by the sign of

$$f(\mathbf{x}) = \mathbf{w}^\mathsf{T}\mathbf{x} + b.$$

In two dimensions, the hyperplane is a line. In higher dimensions, the same expression defines a flat decision boundary.

Many boundaries may separate the training data. An SVM chooses one with a large *margin*, meaning a wide region between the classes. The training observations on or inside the margin that constrain the fitted boundary are called *support vectors*. They determine the position of the fitted boundary, while distant observations usually have little direct effect.

A wide margin makes the classifier less sensitive to small changes in the training data. Perfect separation is often impossible, so a practical SVM permits some observations to enter the margin or appear on the wrong side of the boundary.

Support Vectors

Support vectors are the training observations on or inside the margin that constrain the fitted decision boundary. Moving one of them can change the fitted classifier, whereas moving a distant point often does not.

Figure 3.1 shows a linear SVM in two dimensions. The solid line is the decision boundary, the dashed lines delimit the margin, and the circled observations are the support vectors that fix its position.

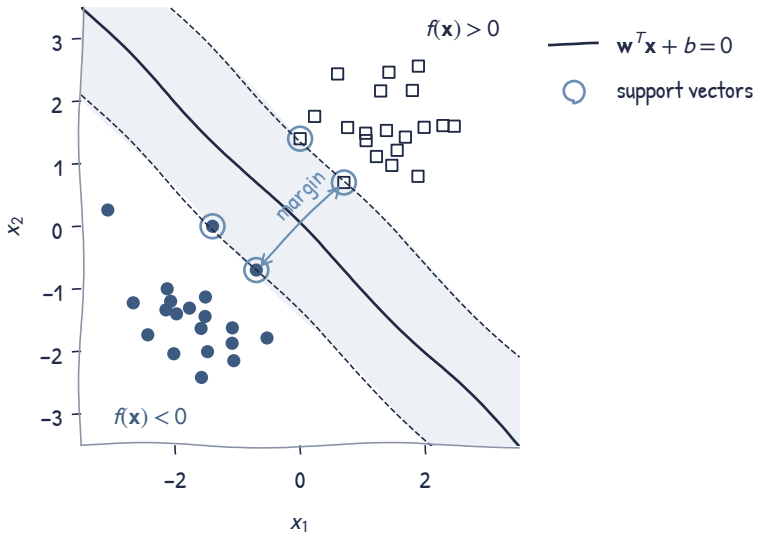


Figure 3.1: A linear SVM: the hyperplane $\mathbf{w}^\mathsf{T}\mathbf{x} + b = 0$, its margin, and the support vectors.

3.2 Soft Margins and Kernels

The parameter C controls the penalty for margin violations. A large C strongly favors correct classification of the training observations and can create a narrow, complex boundary. A small C accepts more violations in exchange for a wider, more regular boundary. The value should be chosen with cross-validation.

Some classes cannot be separated by a straight boundary. A kernel allows the SVM to learn a linear separator in another feature space without explicitly constructing all transformed features. Instead of returning transformed coordinates, the kernel directly calculates how similar two observations would be in that space. This is called the *kernel trick*: the SVM can fit a nonlinear boundary in the original input space while still solving a linear separation problem in the transformed space.

The radial basis function (RBF) kernel is

$$K(\mathbf{x}, \mathbf{z}) = \exp\left(-\gamma \left\|\mathbf{x} - \mathbf{z}\right\|^2\right).$$

The RBF value is close to one for nearby observations and approaches zero

as their distance grows. The parameter γ controls how quickly this similarity decreases. A small value gives each observation broad influence and usually produces a smoother boundary. A large value gives narrow influence and can create small regions around individual observations, which may fit noise.

Both linear and RBF SVMs are sensitive to feature scale. Standardization should normally be part of the model pipeline. Kernel SVMs can also become expensive for very large datasets because training depends on comparisons between many observations.

The Parameters C and γ

The parameter C controls the cost of margin violations. For an RBF kernel, γ controls how quickly similarity decreases with distance. Large values of both can create an overly complex model.

Figure 3.2 illustrates the idea behind kernels. In the original one-dimensional space no single threshold separates the classes, but after mapping x to x^2 a straight line does.

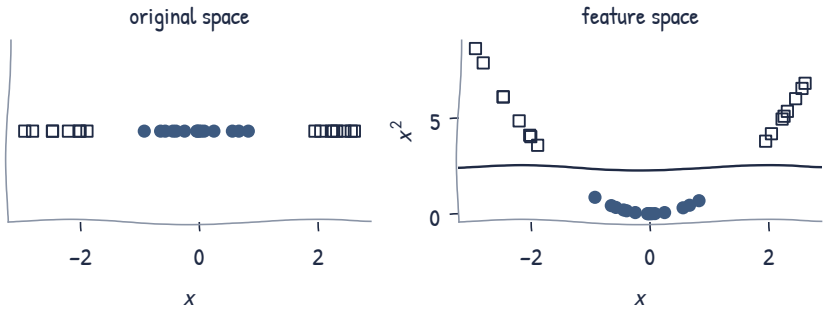


Figure 3.2: Kernel idea: classes that are not linearly separable become separable in another feature space.

3.3 SVM in Python

The following example trains an RBF SVM on the Iris dataset. The pipeline prevents information from the test data from entering the scaling step.

```
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
```

```
X, y = load_iris(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.25,
    random_state=42,
    stratify=y,
)

model = make_pipeline(
    StandardScaler(),
    SVC(
        kernel="rbf",
        C=1.0,
        gamma="scale",
    ),
)

model.fit(X_train, y_train)
y_pred = model.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion matrix:")
print(confusion_matrix(y_test, y_pred))
```

The Iris dataset has three classes. The library combines binary SVM decisions to produce a multiclass prediction. The setting `gamma="scale"` provides a data-dependent starting value, but the kernel, C, and γ should still be compared through cross-validation.

The decision score measures position relative to a boundary; it is not automatically a probability. Accuracy should also be supplemented with precision, recall, or class-specific errors when the classes are imbalanced or mistakes have different consequences.

Overfitting

Overfitting is when a model overadapts to the training data while losing the ability to generalize on unseen data. Instead, it memorizes the noise and quirks of the training data instead of the underlying pattern. It shows up as a large gap between excellent training performance and poor performance on new data, and its risk grows with model complexity, small datasets, and long training.

3.4 Exercises



1. Questions:
   (a) What is the margin of an SVM?
   (b) Why are support vectors important?
   (c) How does C influence the fitted boundary?
   (d) What does the RBF parameter γ control?
2. Hands-On and Programming:
   (a) Draw two linearly separable classes and mark a maximum-margin boundary and its support vectors.
   (b) Compare linear and RBF kernels on the Iris dataset.
   (c) Test several values of C and γ with cross-validation.

Chapter 4

K-Means Clustering

4.1 Clusters and Centroids

K-means is an unsupervised method that divides observations into k clusters [10]. Each cluster is represented by its centroid, which is the mean of the observations assigned to that cluster. Every observation is assigned to the nearest centroid.

If c_i is the cluster assigned to observation $\mathbf{x}_i$ and $\boldsymbol{\mu}_{c_i}$ is its centroid, K-means minimizes the within-cluster sum of squares:

$$J = \sum_{i=1}^{n} \left\| \mathbf{x}_i - \boldsymbol{\mu}_{c_i} \right\|_2^2 .$$

This value is called *inertia* in scikit-learn. Lower inertia means that observations lie closer to their assigned centroids, but it never increases when more clusters are added.

Unlike classification, K-means does not use known labels. The cluster numbers have no ordering or fixed meaning. Labels 0, 1, and 2 could be exchanged without changing the grouping.

K-Means

K-means represents each cluster by a centroid and assigns every observation to its nearest centroid. The number of clusters k must be chosen before fitting.

4.2 The Algorithm and Its Limitations

K-means begins with k initial centroids. It assigns every observation to the closest centroid, recalculates each centroid as the mean of its assigned

observations, and repeats these two steps until the assignments or centers stabilize.

Poor initial centroids can lead to a poor local solution. The `k-means++` method spreads the initial centers more carefully than purely random selection. Repeating the algorithm with different initializations can provide additional protection; the solution with the lowest inertia is retained.

The number of clusters can be explored with an elbow plot or the silhouette score. An elbow plot shows how inertia decreases as k grows; one looks for a point after which additional clusters provide only small improvements. The silhouette score compares how close an observation is to its own cluster with how close it is to the nearest alternative cluster. Values near one indicate compact, well-separated clusters, values near zero indicate overlap, and negative values suggest that some observations may fit another cluster better.

Neither method automatically discovers the correct answer. The final choice should also make sense for the application. A low inertia alone is insufficient because the smallest value occurs when every observation becomes its own cluster.

K-means works best for compact, roughly spherical clusters with similar spreads. It can perform poorly on curved groups, clusters with very different densities, or data containing strong outliers. DBSCAN is often more suitable for irregular shapes and explicit noise detection.

Feature scaling matters because K-means uses Euclidean distance. Standardization prevents a large-scale feature from dominating the assignments. High-dimensional data may also require feature selection or dimensionality reduction before distances become useful.

Centroids Are Means

A centroid is an average location and need not be an actual observation. Because means are sensitive to extreme values, outliers can pull centroids away from the main body of a cluster.

Figure 4.1 shows one iteration of the algorithm. After the initial centroids are chosen, each observation is assigned to its nearest centroid, and every centroid then moves to the mean of its assigned observations.

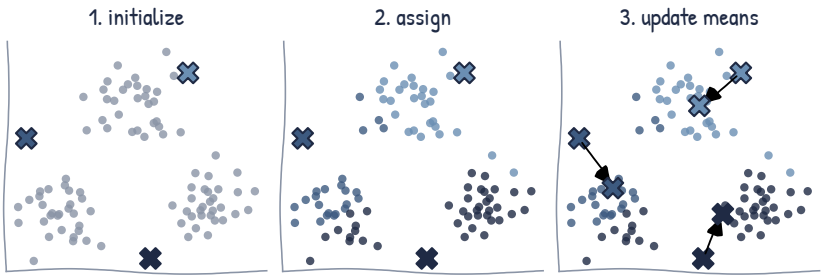


Figure 4.1: One K-means iteration: initialize centroids, assign observations to the nearest centroid, recompute centroids as means.

4.3 K-Means in Python

The following example generates three compact groups, standardizes the coordinates, and fits K-means.

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
from sklearn.preprocessing import StandardScaler

X, _ = make_blobs(
    n_samples=300,
    centers=3,
    cluster_std=1.0,
    random_state=42,
)
X = StandardScaler().fit_transform(X)

model = KMeans(
    n_clusters=3,
    init="k-means++",
    n_init=10,
    random_state=42,
)
labels = model.fit_predict(X)

print("Inertia:", model.inertia_)
print("Iterations:", model.n_iter_)
print("Centroids:")
print(model.cluster_centers_)

plt.scatter(
    X[:, 0],
    X[:, 1],
    c=labels,
    cmap="tab10",
    s=25,
```

```
)
plt.scatter(
    model.cluster_centers_[:, 0],
    model.cluster_centers_[:, 1],
    marker="X",
    c="black",
    s=140,
)
plt.title("K-means clustering")
plt.show()
```

The method `fit_predict` fits the centroids and returns one cluster label per observation. The attribute `inertia_` contains the final within-cluster sum of squares, while `cluster_centers_` contains the fitted centroids.

The example is deliberately favorable to K-means because the generated groups are compact. A responsible analysis also tests other values of k, checks stability, and compares the result with the original features and domain knowledge.

4.4 Exercises



1. Questions:
   (a) How does K-means assign observations to clusters?
   (b) What does inertia measure?
   (c) Why can initialization change the result?
   (d) For which cluster shapes is K-means unsuitable?
2. Hands-On and Programming:
   (a) Assign several two-dimensional points to two given centroids and calculate the updated centroids.
   (b) Run the example with $k = 2$, 3, 4, and 5, then compare inertia and silhouette scores.
   (c) Compare K-means and DBSCAN on the moons dataset.

Part II

Deep Learning

Chapter 5

Multilayer Perceptrons

5.1 Dense Neural Networks

A multilayer perceptron (MLP) is a feed-forward neural network built from dense layers. Every neuron in one dense layer receives the outputs of all neurons in the preceding layer. MLPs are used for classification, regression, and as components inside larger neural networks.

For an input vector $\mathbf{x}$, one dense layer computes

$$\mathbf{h} = \sigma\left(W\mathbf{x} + \mathbf{b}\right),$$

where W contains the weights, $\mathbf{b}$ contains the biases, and σ is an activation function. Without a nonlinear activation, several dense layers would collapse into one linear transformation.

ReLU is a common hidden-layer activation and returns $\max(0, x)$. A classification network often ends with a softmax layer, which converts class scores into probabilities that sum to one. A regression network commonly uses a linear output without an activation.

The input is usually a fixed-length feature vector. Images can be flattened into vectors, but this discards their spatial structure. A convolutional network is normally more suitable when neighboring pixels and repeated local patterns matter.

Why Hidden Layers Matter

Hidden layers learn intermediate representations. Nonlinear activations allow an MLP to model relationships that cannot be represented by one straight line or hyperplane.

Figure 5.1 depicts a small MLP with two hidden ReLU layers and a

softmax output. Every line represents one weight in W, and each neuron is connected to all neurons of the preceding layer.

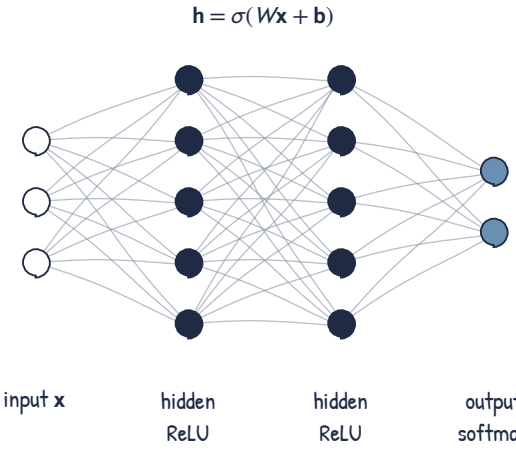


Figure 5.1: An MLP of dense layers; each layer computes $\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b})$.

5.2 Training and Generalization

Training alternates between a forward pass and a backward pass. In the forward pass, the input moves through the layers and produces a prediction. A loss function compares that prediction with the target and converts the error into one number. In the backward pass, backpropagation applies the chain rule from the output layer toward the input and calculates a gradient for every trainable weight and bias [14]. A gradient indicates how a small change in a parameter would change the loss.

For a batch of m training examples, let $\hat{y}_i$ be the network's prediction and y_i the corresponding target. The average loss is

$$\mathcal{L} = \frac{1}{m} \sum_{i=1}^{m} \ell(\hat{y}_i, y_i) \, .$$

Here m is the number of examples in the current batch, and ℓ is the loss for one prediction. The quantity $\mathcal{L}$ is the mean loss over the batch and is what training attempts to minimize.

An optimizer uses these gradients to update the parameters. In basic gradient descent, a parameter w is updated according to

$$w \leftarrow w - \eta \frac{\partial \mathcal{L}}{\partial w},$$

where η is the learning rate. If the learning rate is too small, training can be unnecessarily slow; if it is too large, the updates can overshoot useful solutions or become unstable. Stochastic gradient descent estimates the gradient from a batch of examples, while Adam also maintains adaptive estimates of recent gradient magnitudes.

The loss must match the task. Mean squared error penalizes the squared difference between a numerical prediction and its target and is common in regression. Cross-entropy is common in classification because it penalizes assigning low probability to the correct class. The optimizer minimizes the loss, whereas a metric such as accuracy is mainly used to interpret performance and need not be differentiable.

Training examples are usually shuffled and divided into mini-batches. One parameter update is made after each batch, and one pass through all training examples is an epoch. Small batches give noisy but frequent updates; large batches give smoother estimates but require more memory. Several epochs are normally needed because each update changes only the current parameters by a limited amount.

The data should be separated into training, validation, and test sets. The training set determines the weights. The validation set is used to choose the architecture, learning rate, number of epochs, and other settings. The test set should remain untouched until the final evaluation; repeatedly using it to make choices turns it into another validation set and produces an overly optimistic result.

Underfitting occurs when both training and validation performance are poor because the model, features, or training procedure cannot capture the pattern. Overfitting occurs when training performance continues to improve while validation performance stops improving or becomes worse. Weight decay discourages very large weights, dropout randomly removes some activations during training [16], and early stopping keeps the parameters from the epoch with the best validation result. More representative data can often improve generalization more directly than changing the architecture.

Feature scaling usually improves optimization because very different numerical ranges can produce poorly balanced gradients. The scaler must be learned from the training data and then applied unchanged to validation, test, and future data. Categorical inputs require an appropriate numerical representation, such as one-hot vectors or learned embeddings.

MLPs can approximate complex functions, but training does not guarantee the best possible solution. Neural-network losses are generally nonconvex, so initialization, optimization settings, architecture, and data quality can lead to different fitted models. Evaluation on genuinely unseen data is therefore part of training a useful model, not an optional final step.

Epoch and Batch

A batch is a subset of training examples used for one parameter update. An epoch is one pass through the complete training set. Several updates therefore occur during each epoch.

Figure 5.2 shows typical loss curves. The training loss keeps decreasing, while the validation loss starts to rise once the network overfits; early stopping keeps the model from the epoch with the lowest validation loss.

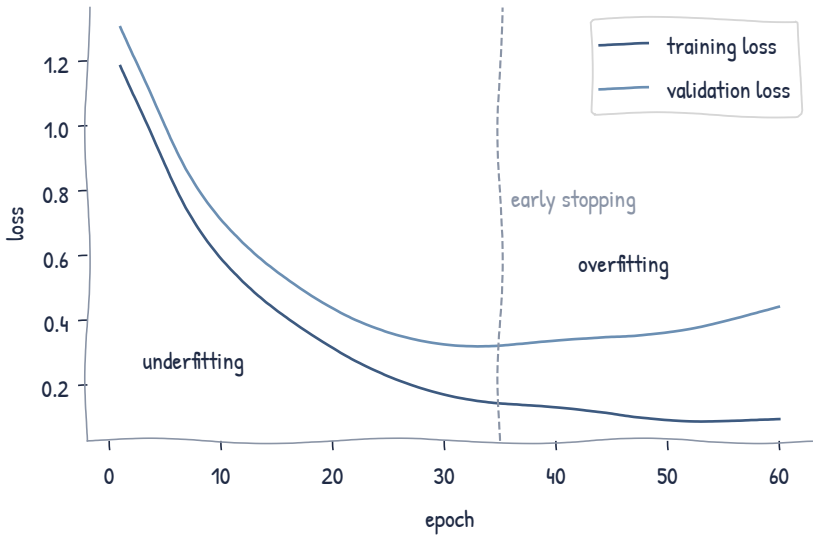


Figure 5.2: Training vs. validation loss: rising validation loss signals overfitting; early stopping selects the best epoch.

5.3 An MLP in `Keras`

The following example classifies handwritten MNIST digits. Each 28×28 image is flattened into a vector of 784 pixel values before it enters the MLP.

```
import keras
from keras import layers

(x_train, y_train), (x_test, y_test) = (
    keras.datasets.mnist.load_data()
)

x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

model = keras.Sequential([
```

```
    keras.Input(shape=(28, 28)),
    layers.Flatten(),
    layers.Dense(256, activation="relu"),
    layers.Dropout(0.2),
    layers.Dense(10, activation="softmax"),
])

model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

early_stopping = keras.callbacks.EarlyStopping(
    patience=2,
    restore_best_weights=True,
)

model.fit(
    x_train,
    y_train,
    validation_split=0.1,
    epochs=10,
    batch_size=128,
    callbacks=[early_stopping],
)

test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

The hidden layer learns 256 features from the flattened pixels. Dropout is active during training and disabled during evaluation. The output contains one probability for each digit class. The argument `validation_split=0.1` reserves ten percent of the supplied training examples for validation, and early stopping restores the weights from the best validation epoch after two epochs without improvement. The separate test set is used only after fitting.

This is a useful baseline, but flattening means that the model does not explicitly exploit nearby pixels. A convolutional network can learn local image patterns with fewer connections.

5.4 Exercises



1. Questions:
   (a) Why does an MLP need nonlinear activation functions?
   (b) What is the difference between a batch and an epoch?
   (c) How do underfitting and overfitting differ?

(d) Why is an MLP not always ideal for images?

2. Hands-On and Programming:
   (a) Calculate the number of weights in a dense layer with 100 inputs and 32 neurons.
   (b) Run the example with one and two hidden layers and compare validation accuracy.
   (c) Remove dropout and compare the training and validation curves.

Chapter 6

Convolutional Neural Networks

6.1 Convolution and Feature Maps

A convolutional neural network (CNN) is designed for data with spatial structure, especially images. Instead of connecting every pixel to every neuron, a convolution applies a small learnable filter to local regions of the input [9]. AlexNet later demonstrated the impact of deep CNNs on large-scale image classification [8].

As the filter moves across an image, it produces a feature map. For one output channel, a simplified two-dimensional convolution is

$$y_{i,j} = b + \sum_{u,v,c} K_{u,v,c}\, x_{i+u,j+v,c},$$

where K is the filter, c indexes input channels, and b is a bias. The same filter weights are reused at every position.

The filters are not normally chosen by hand. They begin with initial values and are learned through the same loss, backpropagation, and optimizer used for an MLP. Because one filter is applied at many positions, its gradient combines evidence from all of those positions. A filter is therefore updated when detecting its pattern helps reduce the prediction error anywhere in the image.

Weight sharing makes convolution more efficient than a dense layer on the same image. It also allows a learned pattern, such as an edge, to be detected at different positions. Multiple filters create multiple output channels and can learn different patterns.

Early CNN layers often respond to edges and simple textures. Deeper layers combine these responses into larger shapes and object parts. The region of the original image that influences one activation is called its *receptive field*, and it grows through successive layers.

Convolutional Filter

A convolutional filter is a small set of shared weights that detects a local pattern across an input. One filter produces one feature map, and a layer normally learns many filters.

Figure 6.1 illustrates a single convolution step. The 3×3 filter K, which responds to vertical edges, is applied to the highlighted region of x, and the weighted sum becomes one entry of the feature map y.

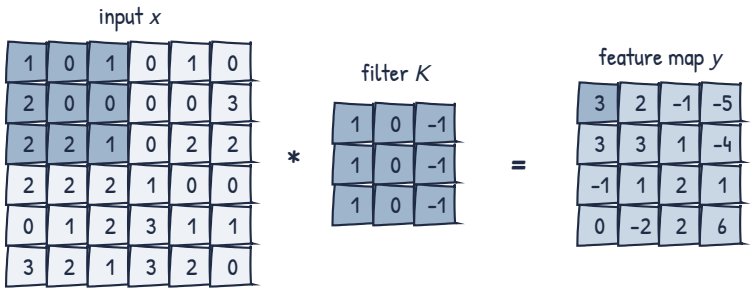


Figure 6.1: A 3×3 filter K slides over the input x and produces a feature map y.

6.2 Stride, Padding, and Pooling

The stride controls how far a filter moves between positions. A stride of one examines neighboring positions, while a larger stride skips positions and reduces the spatial resolution. Without padding, a filter cannot be centered on pixels at the border, so the feature map becomes smaller. Padding usually adds zeros around the image. With `padding="same"` and stride one, enough padding is added to keep the input height and width.

Pooling summarizes small regions without learned filter weights. Max pooling keeps the largest value in each region, while average pooling keeps the mean. Pooling and strided convolution reduce resolution, computation, and sensitivity to small shifts, but excessive reduction can remove useful detail.

A typical CNN alternates convolutions and nonlinear activations while gradually reducing spatial size and increasing channel count. Near the end, global average pooling converts each final feature map into one value. A dense output layer then produces the prediction.

CNNs are translation equivariant rather than completely translation in-

variant: shifting the input tends to shift the feature maps. Pooling and global aggregation can make the final prediction less sensitive to position. Data augmentation can further teach the model to tolerate flips, crops, or small transformations that are valid for the task.

CNNs still require careful training. Large networks can overfit, class imbalance can hide important errors, and predictions may fail when deployment images differ from the training data. Validation examples should represent the conditions in which the model will be used.

Spatial Resolution

Convolution with stride, pooling, and padding determines the size of the feature maps. Reducing resolution lowers computation but may discard information needed for small objects or precise localization.

Figure 6.2 shows 2×2 max pooling with stride two. Each colored region of the 4×4 input is reduced to its largest value, which halves the spatial resolution.

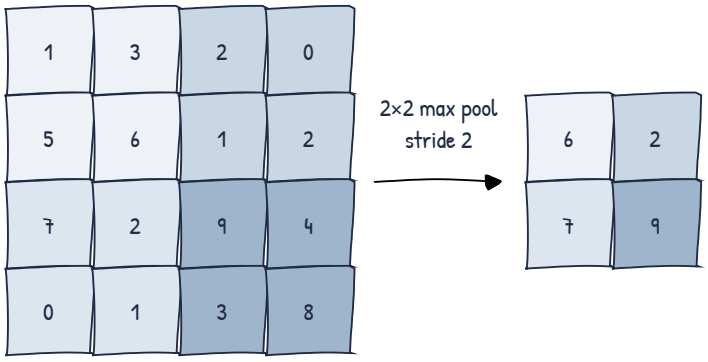


Figure 6.2: 2×2 max pooling with stride 2 keeps the largest value of each region.

6.3 A CNN in `Keras`

The following compact CNN classifies CIFAR-10 images. It learns local features with convolutions and reduces their spatial size with max pooling.

```
import keras
from keras import layers

(x_train, y_train), (x_test, y_test) = (
```

```
    keras.datasets.cifar10.load_data()
)

model = keras.Sequential([
    keras.Input(shape=(32, 32, 3)),
    layers.Rescaling(1.0 / 255),
    layers.RandomFlip("horizontal"),

    layers.Conv2D(
        32, 3, padding="same", activation="relu"
    ),
    layers.MaxPooling2D(),

    layers.Conv2D(
        64, 3, padding="same", activation="relu"
    ),
    layers.MaxPooling2D(),

    layers.Conv2D(
        128, 3, padding="same", activation="relu"
    ),
    layers.GlobalAveragePooling2D(),
    layers.Dropout(0.2),
    layers.Dense(10, activation="softmax"),
])

model.compile(
    optimizer="adam",
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)

model.fit(
    x_train,
    y_train,
    validation_split=0.1,
    epochs=15,
    batch_size=128,
)

test_loss, test_accuracy = model.evaluate(x_test, y_test)
print("Test accuracy:", test_accuracy)
```

The rescaling layer converts pixel values to the range from zero to one. Random horizontal flipping augments only the training data. The convolutional layers increase the number of channels as pooling reduces height and width.

Global average pooling avoids a large flattened representation. A stronger experiment would add normalization, early stopping, and a learning-rate schedule, then inspect class-specific errors instead of reporting accuracy alone.

6.4 Exercises



1. Questions:
   (a) Why does a convolution reuse the same filter across an image?
   (b) How do stride and padding affect a feature map?
   (c) What is a receptive field?
   (d) Why can excessive pooling be harmful?
2. Hands-On and Programming:
   (a) Apply a 2×2 filter by hand to a small grayscale image.
   (b) Run the example without data augmentation and compare validation accuracy.
   (c) Add another convolutional layer and inspect the changed parameter count.

Chapter 7

Attention

7.1 Selecting Relevant Information

Attention allows a neural network to combine information from different positions in a sequence. Instead of compressing the complete sequence into one fixed state, the model can decide which positions are most relevant for the current representation [17].

Consider the sentence *The animal did not cross the street because it was tired*. To interpret *it*, the model should connect that token strongly with *animal*. Attention learns such connections from data rather than from manually written grammatical rules.

Every input representation is projected into a query, a key, and a value. A query describes what one position is looking for, a key describes what another position offers, and a value contains the information that can be transferred. Similarity between a query and the keys determines how strongly their values are combined. In the example above, the query for *it* can match strongly with the key for *animal*; the corresponding value then contributes strongly to the new representation of *it*.

For query, key, and value matrices Q, K, and V, scaled dot-product attention is

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^{\mathsf{T}}}{\sqrt{d_k}}\right)V.$$

The matrix product QK^{T} produces one similarity score for every pair of positions. Dividing by $\sqrt{d_k}$ prevents large vector dimensions from making the softmax excessively sharp. Each softmax row contains weights that sum to one, and multiplying by V forms a weighted average of the value vectors. The result is one context-dependent representation for every input position.

Queries, Keys, and Values

A query searches for relevant information, keys determine how well positions match that query, and values provide the information that is combined. All three are learned projections of model representations.

Figure 7.1 sketches the computation. Queries and keys produce the attention weights, which are then used to form a weighted sum of the values.

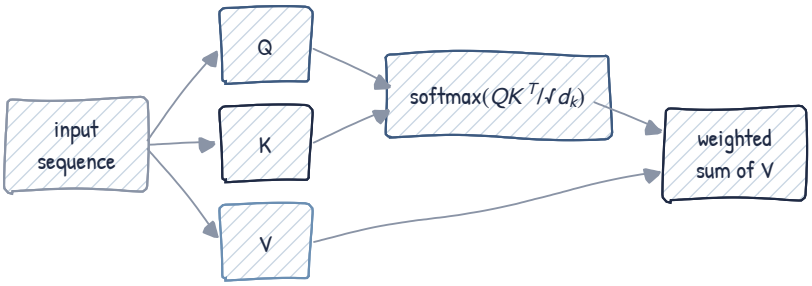


Figure 7.1: Scaled dot-product attention: queries, keys, and values produce a weighted sum of the values.

7.2 Self-Attention and Multi-Head Attention

In self-attention, queries, keys, and values come from the same sequence. Every position can therefore build a new representation from other positions. An encoder may allow attention to all tokens, while a language-generating decoder uses a causal mask that hides future tokens.

Cross-attention uses queries from one sequence and keys and values from another. In a translation model, decoder positions can query representations of the source sentence. In a vision-language model, text representations may attend to image features.

Multi-head attention performs several attention operations in parallel. Each head uses its own learned projections and can focus on different relationships. The head outputs are concatenated and projected back to the model dimension.

Attention does not know token order by itself. Positional encodings or learned positional embeddings supply this information. Attention is also normally placed inside a block containing residual connections, normalization, and a feed-forward network. A residual connection adds a block's input to its output, which preserves existing information and helps gradients pass through deep networks. Normalization keeps activation scales well behaved.

The feed-forward network then transforms each position independently; attention exchanges information between positions, while the feed-forward part processes the information at each position.

Masks serve different purposes. A padding mask prevents artificial padding tokens from influencing the result. A causal mask prevents information from flowing from future positions to earlier positions. Using the wrong mask can leak target information during training or make valid tokens invisible.

Attention Weights

Attention weights show how one head distributes its focus for one input. They can support inspection, but they are not automatically a complete explanation of the model's prediction.

Figure 7.2 compares attention weights for a short sentence without and with a causal mask. With the mask, each token attends only to itself and earlier tokens, so the upper triangle is zero.

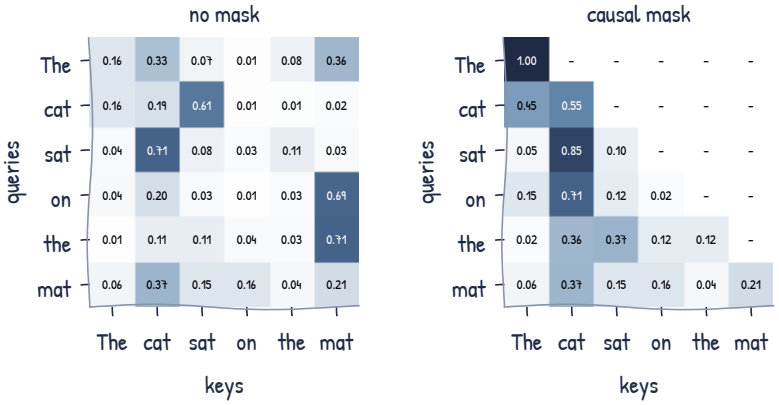


Figure 7.2: Attention weights without and with a causal mask; each row sums to one.

7.3 Cost and a Keras Example

Standard self-attention compares every sequence position with every other position. Its attention matrix therefore grows quadratically with sequence length. Long sequences can require substantial memory and computation, motivating local, sparse, grouped, or otherwise optimized attention variants.

The following example applies four-head self-attention to a sequence of 32 vectors. It also returns the attention scores for inspection.

```
import keras
from keras import layers

sequence_length = 32
model_dimension = 64

inputs = keras.Input(
    shape=(sequence_length, model_dimension)
)

attention_output, attention_scores = (
    layers.MultiHeadAttention(
        num_heads=4,
        key_dim=16,
    )(
        inputs,
        inputs,
        return_attention_scores=True,
    )
)

outputs = layers.LayerNormalization()(
    inputs + attention_output
)

model = keras.Model(
    inputs,
    [outputs, attention_scores],
)
model.summary()
```

The query and value arguments are both `inputs`, so the layer performs self-attention. With four heads, the score tensor contains a separate 32×32 attention map for each head and each example in a batch.

For autoregressive generation, the call can use `use_causal_mask=True`. A complete transformer block would additionally apply a feed-forward network and another residual connection.

7.4 Exercises



1. Questions:
   (a) What roles do queries, keys, and values play?
   (b) How does self-attention differ from cross-attention?
   (c) Why is the dot product divided by $\sqrt{d_k}$?
   (d) Why does standard attention become expensive for long sequences?

2. Hands-On and Programming:
   (a) Calculate a small attention result from two queries, keys, and values.
   (b) Run the Keras example and inspect the attention-score shape.
   (c) Add a causal mask and visualize which positions become inaccessible.

Part III

LLMs

Chapter 8

Generative Pre-Trained Transformers

8.1 Decoder-Only Language Models

GPT stands for *Generative Pre-Trained Transformer*. A GPT model is a decoder-only transformer trained to predict the next token [13]. This single training objective supports continuation, question answering, summarization, code generation, and many other tasks when the model is given an appropriate prompt.

Before a model can process text, a tokenizer divides it into units such as words, word pieces, punctuation marks, or bytes. Each unit is mapped to an integer identifier from a fixed vocabulary. A learned embedding table converts every identifier into a vector; tokens used in similar contexts can acquire similar representations during training. A positional representation is added so that the model can distinguish the same token appearing at different locations. The resulting vectors pass through a stack of decoder blocks.

Each block contains causally masked multi-head self-attention, a feed-forward network, residual connections, and normalization. The causal mask allows position t to use positions at or before t, but not later tokens. Without this restriction, the model could see the answer it is supposed to predict during training.

The final representation at each position is projected to one score for every vocabulary token. Softmax converts the scores into probabilities. Training processes many positions in parallel, but generation remains sequential because every new token becomes part of the context for the next prediction.

Decoder-Only Transformer

A GPT model repeatedly applies causal self-attention and feed-forward transformations. It reads the available prefix and predicts how that sequence is likely to continue.

Figure 8.1 summarizes this pipeline. The tokens pass through embeddings and a stack of decoder blocks, the softmax returns next-token probabilities, and during generation the chosen token is appended to the input before the process repeats.

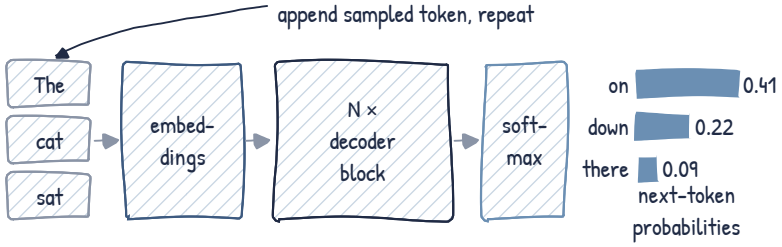


Figure 8.1: A decoder-only GPT predicts next-token probabilities; generation appends each new token and repeats.

8.2 Pretraining, Adaptation, and Generation

For tokens $x_1, \ldots, x_T$, GPT pretraining minimizes the negative log-probability of the observed next tokens:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(x_t \mid x_1, \ldots, x_{t-1}).$$

The text supplies its own targets, so pretraining does not require a manual label for every sequence. The model learns statistical patterns, associations, and reusable representations from the training data.

Pretraining teaches broad language patterns by predicting tokens in large text collections. Instruction tuning continues training on examples that pair requests with desired responses, which teaches the model how to express its knowledge in a useful answer format. Preference-based training compares candidate responses and encourages behavior that human annotators or another evaluation process prefer. These stages change the weights; they are different from merely placing an instruction in the context. LoRA and other

parameter-efficient methods can perform such adaptation without updating the complete model.

During generation, greedy decoding always selects the most likely next token. It is deterministic but can produce repetitive or locally plausible sequences. Sampling draws from the predicted distribution and therefore produces more varied outputs. Temperature rescales the logits before softmax: a lower temperature concentrates probability on the leading choices, while a higher temperature spreads probability across more alternatives. Top-k sampling keeps a fixed number of high-probability candidates. Top-p sampling instead keeps the smallest set whose cumulative probability reaches a chosen threshold. These settings change how an existing distribution is sampled; they do not add knowledge or guarantee creativity.

The context window contains the prompt and generated tokens available during one call. A key-value cache stores attention states for earlier tokens so they need not be recomputed completely at every generation step. This speeds generation but uses memory that grows with the context.

GPT predicts plausible continuations rather than consulting a guaranteed fact database. Retrieval, tools, citations, and external checks can improve reliability, but fluent language should not be confused with verified truth.

Training Is Not Conversation

Pretraining and fine-tuning change model weights. A conversation normally changes only the temporary context. Information stated during one interaction is not automatically learned as a permanent model parameter.

8.3 A Small GPT Block in `Keras`

The following example builds one causal decoder block. It demonstrates the central GPT architecture but is much smaller than a practical language model.

```
import keras
from keras import layers, ops

vocab_size = 10000
max_length = 128
embed_dim = 128
num_heads = 4

tokens = keras.Input(
    shape=(max_length,),
    dtype="int32",
```

```
)

token_embedding = layers.Embedding(
    vocab_size,
    embed_dim,
)
position_embedding = layers.Embedding(
    max_length,
    embed_dim,
)

positions = ops.arange(0, max_length)
x = token_embedding(tokens)
x = x + position_embedding(positions)

attention = layers.MultiHeadAttention(
    num_heads=num_heads,
    key_dim=embed_dim // num_heads,
)(x, x, use_causal_mask=True)

x = layers.LayerNormalization()(
    x + attention
)

hidden = layers.Dense(
    4 * embed_dim,
    activation="gelu",
)(x)
hidden = layers.Dense(embed_dim)(hidden)

x = layers.LayerNormalization()(
    x + hidden
)
logits = layers.Dense(vocab_size)(x)

model = keras.Model(tokens, logits)
model.summary()
```

The model returns vocabulary logits for every position. Training compares each position with the following token, usually by shifting the target sequence by one place.

A practical GPT stacks many decoder blocks and adds dropout, optimized attention, carefully prepared training data, and generation code. Increasing size also increases memory, computation, and the importance of systematic evaluation.

8.4 Exercises



1. Questions:
   (a) Why does GPT use a causal attention mask?
   (b) How do pretraining and instruction tuning differ?
   (c) Why is generation sequential even though training can process positions in parallel?
   (d) What is the purpose of a key-value cache?
2. Hands-On and Programming:
   (a) Draw the permitted attention connections for a sequence of four tokens.
   (b) Run the Keras example and inspect the tensor and parameter shapes.
   (c) Stack two decoder blocks and compare the parameter count.

Chapter 9

Prompting

9.1 Instructions, Context, and Output

A prompt is the information supplied to a language model for one interaction. It can contain a goal, relevant context, constraints, examples, and a required output format. Prompting changes the model's immediate behavior but does not normally change its weights.

A useful prompt states the task directly and includes the evidence required to solve it. Vague instructions force the model to guess, while excessive context can hide important details. Constraints should be observable: a word limit, required fields, or an allowed label set can be checked more easily than an instruction such as `be good`.

Delimiters help distinguish instructions from source material. This matters when the source comes from users, websites, or retrieved documents. Such content should be treated as data, not as permission to alter the task or use additional tools.

Prompt Structure

A practical prompt identifies the task, supplies relevant context, states constraints, and defines the expected output. Good structure reduces ambiguity but cannot guarantee correctness.

Figure 9.1 lists these building blocks. Not every prompt needs every part, but stating the task and the output format explicitly is almost always useful.

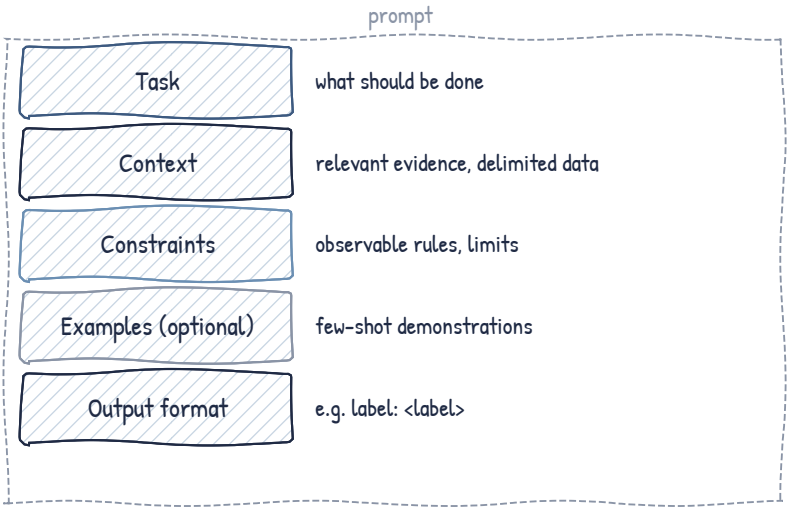


Figure 9.1: Building blocks of a structured prompt.

9.2 Examples and Structured Prompts

A zero-shot prompt contains instructions without demonstrations. A few-shot prompt also provides example inputs and outputs [1]. Examples can communicate a classification rule, writing style, or data format more precisely than a long description.

The examples should cover representative cases. If every example contains the same accidental pattern, the model may copy that shortcut rather than learn the intended distinction. Labels and output formats must also remain consistent across demonstrations.

A compact structured prompt can look like this:

```
Task:
Classify the message as billing, technical,
account, or other.

Message:
<message>
I was charged twice for the same month.
</message>

Rules:
Use exactly one allowed label.
Do not add an explanation.

Output:
label: <label>
```

The restricted output is easy to parse and validate. A more complex response may include a conclusion, evidence, uncertainty, and missing information. Software should still validate machine-readable output rather than trusting it because it resembles the requested format.

Few-Shot Prompting

Few-shot prompting demonstrates a task inside the context. It can guide format and behavior, but misleading examples can also guide the model toward the wrong rule.

9.3 Chain-of-Thought and Decomposition

Chain-of-thought prompting encourages a model to solve a problem through intermediate steps [18]. A simple form asks the model to think step by step, while a few-shot form demonstrates worked solutions before presenting a new problem. This can help with some mathematical, symbolic, and multi-stage tasks.

A generated reasoning trace is not proof that the answer is correct. The explanation may be incomplete, reconstructed after the answer, or confidently wrong. Some systems also use private internal reasoning that is not returned to the user. In such cases, a prompt should request a concise rationale or checkable evidence rather than demand hidden internal thoughts.

A useful pattern separates solving from verification:

```
Solve the problem carefully.

Return:
1. The final answer.
2. A concise explanation of the key steps.
3. One independent check of the result.

If the information is insufficient, state exactly
what is missing instead of guessing.
```

For difficult tasks, decomposition can be more reliable than a generic request to think longer. The prompt can ask the model to identify requirements, solve smaller parts, combine the results, and test the final output against explicit acceptance criteria.

Self-consistency generates several candidate solutions and compares their final answers. It can reduce sensitivity to one reasoning path, but repeated agreement does not establish truth when all candidates share the same mistake. Calculators, code execution, formal proof systems, and authoritative sources provide stronger checks.

Reasoning and Verification

Intermediate reasoning can help organize a solution, but it remains generated text. Prefer concise, checkable justification and external verification when accuracy matters.

9.4 Reliability and Prompt Injection

Prompt improvement should respond to observed failures. If a field is repeatedly missing, require it explicitly and validate it. If claims lack evidence, provide authoritative sources and require the answer to stay within them. Stable rules may belong in schemas, code, or automated tests rather than an increasingly long prompt.

Prompt injection occurs when untrusted content contains instructions that conflict with the real task. A retrieved document might ask the model to ignore earlier rules, expose data, or call a tool. The surrounding application should preserve the priority of trusted instructions and enforce permissions outside the model.

No prompt can guarantee factual accuracy or safe actions. High-risk outputs require suitable evidence, deterministic checks, restricted tools, and human approval where consequences justify it.

9.5 Exercises



1. Questions:
   (a) How does prompting differ from fine-tuning?
   (b) When can few-shot examples improve a prompt?
   (c) Why is a chain-of-thought explanation not proof of correctness?
   (d) What is prompt injection?
2. Hands-On:
   (a) Rewrite a vague request with a goal, context, constraints, and output format.
   (b) Compare direct prompting, step-by-step prompting, and a prompt requiring an independent check.
   (c) Create a few-shot prompt for a three-class classification task.
   (d) Design a prompt that treats retrieved text as evidence rather than instructions.

Chapter 10

Low-Rank Adaptation

10.1 From Fine-Tuning to LoRA

Pretrained language models can be adapted to a task with examples from the desired domain. Full fine-tuning updates all model parameters. This can be effective, but it requires substantial memory for parameters, gradients, and optimizer state, and it produces a separate full model for every adaptation.

Low-rank adaptation (LoRA) keeps the pretrained weights frozen and learns small updates for selected weight matrices [6]. It is a form of parameter-efficient fine-tuning. The unchanged base model can be shared, while each task stores only its smaller adapter weights.

LoRA is commonly applied to attention projections and sometimes to feed-forward layers. It does not need to modify every matrix. The target modules, rank, training data, and optimization settings determine both the adapter size and its ability to change model behavior.

Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning adapts a model by training a small set of additional or selected parameters. LoRA does this by learning low-rank updates while keeping the original weights frozen.

10.2 Low-Rank Updates

Suppose a frozen layer contains a matrix $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$. LoRA represents its trainable update with two smaller matrices, $A \in \mathbb{R}^{d_{\text{in}} \times r}$ and $B \in \mathbb{R}^{r \times d_{\text{out}}}$, where the rank r is much smaller than either original dimension. The adapted

weight is

$$W' = W + \frac{\alpha}{r} AB.$$

The scaling factor α/r controls the strength of the update. The base matrix W remains frozen, and only A and B are trained. One factor is often initialized with small random values and the other with zeros, so the adapter initially leaves the base model unchanged.

A full 4096×4096 update contains 16,777,216 trainable parameters. With rank $r = 8$, the two LoRA factors contain $4096 \cdot 8 + 8 \cdot 4096 = 65{,}536$ parameters, about 0.39% as many. The saving applies only to the adapted matrix; the complete training footprint also includes activations, optimizer state, and any other trainable components.

A small rank reduces memory and storage but limits the possible update. A larger rank provides more capacity at a higher cost. Rank should therefore be selected through validation rather than assumed to be universally better when larger.

Rank

The LoRA rank controls the capacity of the learned update. A low rank is compact, while a higher rank can express more changes but requires more trainable parameters and memory.

Figure 10.1 visualizes the update. The large matrix W stays frozen, and only the narrow matrices A and B are trained.

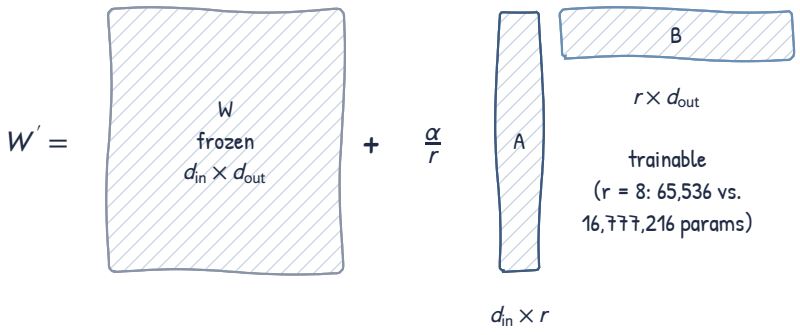


Figure 10.1: LoRA: the frozen matrix W plus a scaled low-rank update $\frac{\alpha}{r}AB$.

10.3 Training and Deployment

LoRA training uses the same task loss as ordinary fine-tuning, but gradients are stored only for the adapter parameters. The base model still participates in forward and backward computation, so LoRA reduces trainable state more than it reduces all computational cost.

Training data quality remains decisive. A small adapter can still memorize poor examples, learn unwanted style, or reduce performance on tasks outside the adaptation data. Validation should measure the desired task as well as important capabilities that should remain unchanged.

At inference time, the adapter can remain separate from the base model or be merged into the selected weight matrices. Separate adapters are easy to exchange and store. Merging removes the additional matrix operations but creates a modified set of weights. Combining several adapters is possible, although their updates can interfere and should be evaluated together.

LoRA changes model weights and is therefore different from prompting and retrieval. Prompting supplies temporary instructions in the context. Retrieval supplies external information at runtime. LoRA learns a persistent behavioral or domain-specific update from training examples. These methods can also be used together.

AI Agent

An AI agent combines an LLM with memory and tools. The LLM interprets the goal and chooses steps, memory preserves relevant information, and tools allow the system to retrieve data or perform actions. A surrounding runtime coordinates these parts and controls permissions, execution, and stopping.

10.4 A LoRA Layer in `Keras`

The following compact layer illustrates the LoRA calculation for a dense projection. It freezes the original dense layer and trains only the two low-rank matrices.

```
import keras
from keras import layers, ops

class LoRADense(layers.Layer):
    def __init__(self, base_layer, rank=8, alpha=16):
        super().__init__()
        self.base_layer = base_layer
        self.base_layer.trainable = False
```

```
        self.rank = rank
        self.alpha = alpha

    def build(self, input_shape):
        self.base_layer.build(input_shape)
        input_dim = int(input_shape[-1])
        output_dim = self.base_layer.units

        self.lora_a = self.add_weight(
            name="lora_a",
            shape=(input_dim, self.rank),
            initializer=keras.initializers.RandomNormal(
                stddev=0.01
            ),
            trainable=True,
        )
        self.lora_b = self.add_weight(
            name="lora_b",
            shape=(self.rank, output_dim),
            initializer="zeros",
            trainable=True,
        )
        super().build(input_shape)

    def call(self, inputs):
        update = ops.matmul(inputs, self.lora_a)
        update = ops.matmul(update, self.lora_b)
        scale = self.alpha / self.rank
        return self.base_layer(inputs) + scale * update

inputs = keras.Input(shape=(768,))
outputs = LoRADense(
    layers.Dense(3072, use_bias=False),
    rank=8,
    alpha=16,
)(inputs)

model = keras.Model(inputs, outputs)
model.summary()
```

Because `lora_b` starts at zero, the layer initially behaves like the frozen base projection. Training changes the two LoRA matrices and gradually introduces the adapted update. The example demonstrates the mechanism; in a real LLM, pretrained projection layers are wrapped rather than newly initialized.

10.5 Exercises



1. Questions:

(a) How does LoRA differ from full fine-tuning?
(b) Why are two small matrices cheaper than one full update?
(c) What does the LoRA rank control?
(d) How do prompting, retrieval, and LoRA differ?

2. Hands-On and Programming:
(a) Calculate the LoRA parameter count for a 2048×2048 matrix with rank 4.
(b) Change the rank in the Keras example and compare trainable parameter counts.
(c) Explain when separate adapters are preferable to merged weights.

Part IV

Appendices

Appendix A

AI Benchmarks

A.1 ML Datasets

Machine learning models are often tested on standard datasets. A benchmark defines not only a dataset but also a task, data split, evaluation metric, and usually a comparison protocol. Using the same protocol makes results more comparable, although it does not by itself guarantee a fair or practically meaningful comparison. Table A.1 shows some widely used examples. These

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

datasets are small enough for experiments, but still complex enough to test learning algorithms. The training split is used to fit parameters, the validation split to select settings, and the test split to estimate final performance. Test examples must not influence preprocessing, model selection, or repeated experimentation. Accuracy can also hide class imbalance or different costs of errors, so confusion matrices and class-specific measures may be needed.

A.2 LLM Benchmarks

Large language models are evaluated using benchmarks that target abilities such as factual knowledge, mathematical reasoning, instruction following,

coding, or tool use. A benchmark score depends on the prompt, decoding settings, allowed tools, and scoring method as well as on the model itself. Table A.2 shows some common examples.

Table A.2: Benchmarks for evaluating language models

Benchmark	Description
GSM8K [2]	Grade-school math problems
MATH [5]	Competition mathematics problems
GAIA [11]	General reasoning benchmark
HLE [12]	Expert-level questions across many fields

LLM benchmark results require careful interpretation. Exact-match scoring is reproducible but may reject a correct answer written in a different form. Human or model-based judges can assess open responses but introduce judgment variability and possible bias. Benchmark questions may also appear in training data, and repeated optimization against a public test set can turn it into an indirect validation set. Strong evaluation therefore uses multiple tasks, reports the complete protocol, checks contamination where possible, and supplements benchmark scores with tests from the intended application.

Appendix B

AI Math

This appendix briefly explains the mathematical ideas used in the book. It is a reference for reading the formulas in the chapters.

B.1 Vectors and Distances

A vector is a list of numbers. In machine learning, these numbers often represent the features of one observation:

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

For example, the entries could describe age, blood pressure, and cholesterol. The notation x_j means entry j, and $\mathbf{x} \in \mathbb{R}^d$ means a vector of d real numbers.

Euclidean distance measures how far apart two observations are:

$$\delta(\mathbf{x}, \mathbf{z}) = \sqrt{\sum_{j=1}^{d} (x_j - z_j)^2}.$$

The symbol $\sum$ means to add the terms for all features. For $(1, 2)$ and $(4, 6)$, the distance is $\sqrt{(1-4)^2 + (2-6)^2} = 5$. KNN finds nearby observations using this distance. K-means uses squared distances, also written $\|\mathbf{x} - \mathbf{z}\|_2^2$.

B.2 Averages and Scaling

The mean is the sum of the values divided by their number. For $2, 4, 6$, it is $(2 + 4 + 6)/3 = 4$. A K-means centroid is calculated by averaging each feature of the observations in a cluster.

Features with large numerical values can dominate distances. Standardization subtracts a feature's mean and divides by its standard deviation, which measures its spread:

$$\widetilde{x} = \frac{x - \mu}{\sigma}.$$

Here μ is the mean and σ the standard deviation. If $\mu = 10$ and $\sigma = 2$, the value 14 becomes 2. As in the chapter examples, learn these scaling values from the training data only.

B.3 Weighted Sums and Matrices

A weighted sum multiplies values by weights and adds the results. For example, weights $(2, 1)$ and inputs $(3, 4)$ give $2 \cdot 3 + 1 \cdot 4 = 10$. This operation is also called a dot product:

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} = \sum_{j=1}^{d} w_j x_j.$$

The transpose symbol T turns a column into a row. An SVM adds a bias b to the weighted sum to obtain its decision score.

A matrix is a rectangular table of numbers. Matrix multiplication calculates several weighted sums at once:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 10 \end{bmatrix}.$$

The number of columns in the first matrix must match the number of rows in the second. For example, a 2×3 matrix multiplied by a 3×4 matrix produces a 2×4 matrix.

Dense layers use matrix multiplication to combine inputs. CNN filters calculate weighted sums over small image regions. Attention uses weighted averages to combine information from different tokens; for example, weights 0.75 and 0.25 combine values 2 and 6 into 3.

B.4 Probabilities and Softmax

Probabilities lie between 0 and 1. The probabilities for all possible classes or next tokens sum to 1. Softmax converts a model's output scores into such probabilities, giving larger scores larger probabilities.

The notation $p(x_t \mid x_1, \ldots, x_{t-1})$ means the probability of token x_t given the preceding tokens. This is the prediction made by GPT.

The GPT loss uses $-\log p$, where log is the natural logarithm. This quantity is small when the model assigns a high probability to the correct token and large when it assigns a low probability. Adding it over token positions gives the training loss shown in the GPT chapter.

B.5 Loss and Gradients

A loss measures how wrong a prediction is. Mean squared error averages the squared differences between predictions and targets. Cross-entropy rewards assigning high probability to the correct class or token.

A derivative describes the slope of a function. A gradient collects the slopes with respect to the model's parameters. During training, the optimizer uses these gradients to adjust the weights in a direction that reduces the loss locally. The learning rate controls the size of each adjustment. Backpropagation calculates the gradients through the network.

B.6 Rank and LoRA

Rank describes how many independent directions a matrix contains. For example,

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

has rank 1, because its second column is twice its first.

LoRA represents a weight update as a product of two small matrices. The rank r limits the number of independent directions in this update. A smaller rank uses fewer trainable parameters; a larger rank allows more flexible changes. The original weight matrix remains frozen during training.

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 keras
```

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

The tools used in this book have different roles. NumPy provides numerical arrays, and scikit-learn uses NumPy and SciPy for classical machine learning. Keras is a multi-backend deep-learning API that can run with TensorFlow, JAX, or PyTorch. Ollama is a separate local model runtime that can be accessed from Python through an API.

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:

```
import keras
from keras import layers

model = keras.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Bibliography

[1] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901.

[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] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine Learning* 20.3 (1995), pp. 273–297. DOI: 10.1007/BF00994018.

[4] 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.

[5] Dan Hendrycks et al. "Measuring Mathematical Problem Solving With the MATH Dataset". In: *arXiv preprint arXiv:2103.03874* (2021). URL: https://arxiv.org/abs/2103.03874.

[6] Edward J. Hu et al. "LoRA: Low-Rank Adaptation of Large Language Models". In: *International Conference on Learning Representations (ICLR)*. 2022. URL: https://arxiv.org/abs/2106.09685.

[7] Gareth James et al. *An Introduction to Statistical Learning*. Springer, 2013. DOI: 10.1007/978-1-4614-7138-7.

[8] 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.

[9] 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.

[10] 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.

[11] Grégoire Mialon et al. *GAIA: A Benchmark for General AI Assistants*. 2023. arXiv: 2311.12983. URL: https://arxiv.org/abs/2311.12983.

[12] Long Phan et al. "Humanity's Last Exam". In: *arXiv preprint arXiv:2501.14249* (2025). URL: https://agi.safe.ai/.

[13] Alec Radford et al. "Improving Language Understanding by Generative Pre-Training". In: OpenAI, 2018, pp. 1–12.

[14] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 1995.

[16] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[17] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017, pp. 5998–6008. URL: https://arxiv.org/abs/1706.03762.

[18] 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.

[19] Aston Zhang et al. *Dive into Deep Learning*. https://d2l.ai. self-published, 2020.