Methods of AI

A Complete Study Companion

Local Search · CSP · Planning · Knowledge Representation
Uncertainty · Machine Learning · SVMs · Neural Nets · Transformers

Cognitive Science — Universität Osnabrück
WiSe 2024/25 & SoSe 2026

Lecture content, formulas, worked examples, figures,
and an extensive exam-style quiz section.

Contents

1 Introduction: What Is AI and How This Book Is Organised **1**
1.1 What Is This Course About? 1
1.2 The Agent and Rationality Framing 2
1.3 From Classical Search to the Broader Landscape 2
1.4 Major Paradigms Covered in This Book 3
1.5 How This Book Is Structured 4

2 Local Search **6**
2.1 Optimisation Problems and the State-Space Landscape 6
2.1.1 Features of the Landscape 7
2.2 Hill Climbing 8
2.2.1 Steepest-Ascent Hill Climbing 8
2.2.2 The 8-Queens Example 9
2.2.3 Neighbourhood Size Tradeoff 9
2.2.4 Variants of Hill Climbing 10
2.3 Simulated Annealing 11
2.3.1 Physical Intuition 11
2.3.2 The Algorithm 12
2.3.3 How Temperature Governs Acceptance 12
2.3.4 The Cooling Schedule 13
2.3.5 Worked Example: SA Acceptance Calculation 14
2.4 Local Beam Search 14
2.4.1 Algorithm 14
2.4.2 Local Beam Search vs. k Parallel Hill Climbers 15
2.4.3 Stochastic Beam Search 15
2.5 Genetic Algorithms 16
2.5.1 Representation 16

2.5.2 Selection 16
2.5.3 Crossover 16
2.5.4 Mutation 17
2.5.5 The Basic Algorithm 17
2.5.6 Permutation Problems and Crossover Repair 17
2.5.7 Hybrid (Memetic) Algorithms 18
2.6 Algorithm Comparison 18
2.7 Exam Traps and Common Misconceptions 19
2.8 Going Further: Oral-Exam Deep Cuts 21

3 Constraint Satisfaction Problems 24
3.1 The CSP Formalism 24
3.1.1 Variables, Domains, and Constraints 24
3.1.2 Constraint Graphs 25
3.2 Classic Examples 26
3.2.1 Map Colouring 26
3.2.2 Sudoku 26
3.2.3 Frequency Assignment and Timetabling 26
3.3 Consistency Concepts 27
3.3.1 Node Consistency 27
3.3.2 Arc Consistency 27
3.3.3 The AC-3 Algorithm 28
3.3.4 Limits of Arc Consistency: Path Consistency and k-Consistency 29
3.4 Worked Example: Applying AC-3 to a Three-Node Graph-Colouring Problem 29
3.5 Backtracking Search 30
3.6 Ordering Heuristics 31
3.6.1 MRV: Minimum Remaining Values (Variable Ordering) . 31
3.6.2 Degree Heuristic (Variable Ordering, Tie-Breaker) 31
3.6.3 LCV: Least Constraining Value (Value Ordering) 31
3.6.4 Forward Checking 32
3.7 Spatial Reasoning: RCC-8 33
3.8 Putting It All Together: A Small Worked Example with MRV . . 34
3.9 Common Exam Traps 34
3.10 Going Further: Oral-Exam Deep Cuts 35

4 Classical Planning **38**
4.1 What Is Planning? . . . 38
4.1.1 Planning vs. Classical Search . . . 39
4.1.2 Components of a Planning System . . . 39
4.2 Three Classic Challenges . . . 39
4.2.1 The Frame Problem . . . 40
4.2.2 The Qualification Problem . . . 40
4.2.3 The Ramification Problem . . . 41
4.2.4 Side-by-Side Comparison . . . 41
4.3 Deductive Planning: The Situation Calculus . . . 42
4.3.1 Situations and Fluents . . . 42
4.3.2 Effect Axioms . . . 43
4.3.3 Frame Axioms . . . 43
4.3.4 Planning as Theorem Proving . . . 44
4.3.5 Strengths and Weaknesses . . . 44
4.4 State-Based Planning: STRIPS . . . 44
4.4.1 States as Sets of Atoms . . . 45
4.4.2 Operator Schemas: PRE / ADD / DEL . . . 45
4.4.3 STRIPS and the Three Classic Problems . . . 45
4.5 Worked Example: Blocksworld Operators . . . 46
4.6 Planning as Search . . . 47
4.6.1 Forward Search (Progression) . . . 47
4.6.2 Backward Search (Regression) . . . 48
4.6.3 Planning Heuristics . . . 48
4.7 PDDL: The Modern Extension of STRIPS . . . 49
4.7.1 Situation Calculus vs. STRIPS: A Direct Comparison . . . 50
4.8 Summary . . . 50

5 Probabilistic Planning: Markov Decision Processes **52**
5.1 Why Probabilistic Planning? . . . 52
5.2 The Markov Property . . . 53
5.3 Formal Definition of an MDP . . . 54
5.3.1 The Grid World Example . . . 55
5.4 Policies and the Objective . . . 55
5.4.1 Policies . . . 55
5.4.2 Discounted Cumulative Reward . . . 56
5.5 The Value Function and Bellman Equations . . . 57

5.5.1 State-Value Function 57
5.5.2 The Bellman Expectation Equation 57
5.5.3 The Bellman Optimality Equation 58
5.6 Algorithms: Value Iteration and Policy Iteration 58
5.6.1 Value Iteration 59
5.6.2 Policy Iteration 59
5.6.3 Comparison Table 60
5.7 Worked Example: A Bellman Backup 61
5.8 Exam Traps and Common Pitfalls 62

6 Knowledge Representation I: Ontologies and Description Logics 65
6.1 Why Represent Knowledge Explicitly? 65
6.1.1 Semantic Networks and Frames 66
6.2 Ontologies 66
6.2.1 What Is an Ontology? 66
6.2.2 Taxonomy versus Ontology 67
6.2.3 Upper Ontologies and Domain Ontologies 68
6.3 Description Logics 68
6.3.1 Motivation: From FOL to a Decidable Fragment 68
6.3.2 The Basic Language $\mathcal{ALC}$ 69
6.3.3 Dialects and the Expressivity Ladder 70
6.4 TBox and ABox: The Two Pillars of a DL Knowledge Base . . . 70
6.4.1 The TBox — Terminological Knowledge 71
6.4.2 The ABox — Assertional Knowledge 71
6.4.3 Comparison Table 72
6.4.4 The Knowledge Base as an Ontology 73
6.5 OWL and Its Relation to Description Logics 73
6.6 Reasoning Tasks 74
6.6.1 The Tableaux Algorithm 74
6.7 A Worked Example 75
6.8 Expressivity, Complexity, and the Decidability Trade-off 76

7 Knowledge Representation II: RDF, SPARQL, and the Semantic Web 78
7.1 The Resource Description Framework (RDF) 78
7.1.1 Motivation 78
7.1.2 The Triple: Subject, Predicate, Object 79

7.1.3 RDF Graphs 80
7.1.4 RDFS: Adding Schemas to RDF 80
7.2 Querying RDF: SPARQL 81
7.2.1 Triple Patterns and Variables 81
7.2.2 SELECT Queries 81
7.2.3 FILTER 82
7.3 Worked Example: RDF Triples and a SPARQL Query 82
7.4 The Open World Assumption 83
7.4.1 Two Philosophical Stances Toward Missing Information . 83
7.4.2 Consequences of the OWA 84
7.4.3 Comparison Table: OWA vs. CWA 85
7.5 Linked Data and Knowledge Graphs 85
7.6 Common Exam Traps 86

8 Vagueness and Uncertainty 89
8.1 Two Distinct Problems That Look the Same 89
8.1.1 The Sorites (Heap) Paradox as a Motivation 90
8.2 Fuzzy Set Theory 91
8.2.1 From Characteristic Functions to Membership Functions . 91
8.2.2 Membership Function Shapes 91
8.2.3 Fuzzy Set Operations: The Zadeh Definitions 92
8.2.4 Fuzzy Logic: Extending Propositions 94
8.2.5 t-Norms and s-Norms: Generalising AND and OR 94
8.2.6 Worked Example: Fuzzy Set Operations 95
8.2.7 Brief Note on Linguistic Variables and Defuzzification . . 96
8.3 Probabilistic Logic and Uncertainty 97
8.3.1 Probability Spaces 97
8.3.2 Propositional Probabilistic Logic 97
8.4 Conditional Probability and Reasoning 98
8.4.1 A Note on Bayes' Theorem 99
8.5 Comparing the Three Frameworks 100
8.6 Dempster–Shafer Theory (Supplementary) 101
8.6.1 The Mass Function 101
8.6.2 Belief and Plausibility 101
8.6.3 Worked Example: Bel and Pl from a Mass Function . . . 102
8.6.4 DS Theory vs. Probability vs. Fuzzy Logic 103
8.7 Common Exam Traps Revisited 104

9 Machine Learning I: Foundations and Methodology 107
9.1 What Is Machine Learning? . . . 107
9.1.1 Inductive Bias . . . 108
9.2 Types of Learning . . . 109
9.2.1 Supervised Learning . . . 109
9.2.2 Unsupervised Learning . . . 109
9.2.3 Reinforcement Learning . . . 110
9.3 Features and the ML Pipeline . . . 111
9.3.1 Feature Spaces . . . 111
9.3.2 Train, Validation, and Test Splits . . . 111
9.3.3 Generalisation . . . 112
9.4 Overfitting, Underfitting, and the Bias-Variance Tradeoff . . . 112
9.4.1 Overfitting and Underfitting . . . 112
9.4.2 The Bias-Variance Decomposition . . . 113
9.4.3 Avoiding Overfitting . . . 115
9.5 Model Selection and Cross-Validation . . . 116
9.5.1 k-Fold Cross-Validation . . . 116
9.5.2 Leave-One-Out Cross-Validation . . . 117
9.6 Evaluation Metrics . . . 117
9.6.1 The Confusion Matrix . . . 117
9.6.2 Accuracy, Precision, and Recall . . . 118
9.7 Worked Example: Hypothesis Evaluation . . . 119
9.8 Putting It All Together: The Standard ML Workflow . . . 119
9.9 Going Further: Oral-Exam Deep Cuts . . . 121

10 Machine Learning II: Decision Trees and Ensembles 123
10.1 Decision Trees . . . 123
10.1.1 Axis-Aligned Splits and Recursive Partitioning . . . 123
10.1.2 Splitting Criteria . . . 125
10.2 A Worked Information-Gain Computation . . . 126
10.3 Overfitting and Pruning . . . 127
10.4 Ensemble Methods . . . 128
10.4.1 Bagging: Bootstrap Aggregating . . . 128
10.4.2 Random Forests . . . 129
10.4.3 Out-of-Bag Error . . . 130
10.4.4 Feature Importance . . . 130
10.5 Comparison: Single Tree vs. Bagging vs. Random Forest . . . 131

10.6 Exam Traps and Common Pitfalls 132
10.7 Going Further: Oral-Exam Deep Cuts 134

11 Machine Learning III: Support Vector Machines and Kernels 135
11.1 The Perceptron 135
11.1.1 Biological Motivation and the Basic Model 135
11.1.2 Perceptron Learning Rule 136
11.1.3 Linear Separability, Convergence, and XOR 136
11.2 Support Vector Machines: The Core Idea 137
11.2.1 From Linear Separation to Maximum Margin 137
11.2.2 The Primal Optimisation Problem 139
11.2.3 Lagrangian Formulation and Support Vectors 139
11.2.4 Soft-Margin SVMs 140
11.3 The Dual Formulation and the Kernel Trick 140
11.3.1 Dot Products Are All You Need 140
11.3.2 Lifting Data into Feature Space 141
11.3.3 The Kernel Trick 142
11.4 Important Kernels 142
11.4.1 Linear Kernel 142
11.4.2 Polynomial Kernel 143
11.4.3 Gaussian (Radial Basis Function) Kernel 144
11.4.4 Tree Kernels and Structured Data 145
11.5 Kernel Comparison 146
11.6 Worked Examples 147
11.6.1 Computing the Margin from $\vec{w}$ 147
11.6.2 Evaluating the RBF Kernel 147
11.7 Multi-Class SVMs 148
11.8 Exam Traps: Summary 148
11.9 Going Further: Oral-Exam Deep Cuts 150

12 Neural Networks 152
12.1 From the Perceptron to the Multilayer Perceptron 152
12.2 Activation Functions 154
12.2.1 Sigmoid 154
12.2.2 Hyperbolic Tangent 154
12.2.3 ReLU 155
12.2.4 Leaky ReLU 155

12.2.5 Comparison Table 155
12.3 Feedforward Computation: The Forward Pass 156
12.3.1 Worked Example: One Forward Pass 156
12.4 The Loss Function 157
12.4.1 Mean Squared Error 158
12.4.2 Categorical Cross-Entropy 158
12.5 Training by Gradient Descent 159
12.5.1 The Learning Rate 159
12.5.2 Stochastic, Mini-Batch, and Batch Gradient Descent . . . 160
12.6 Backpropagation 160
12.6.1 The Chain Rule 161
12.6.2 Forward Pass and Backward Pass 161
12.6.3 Vanishing Gradients Revisited 162
12.7 Universal Approximation 162
12.8 Deep Networks: Benefits and Challenges 163
12.8.1 Vanishing and Exploding Gradients 163
12.8.2 Weight Initialisation 164
12.8.3 Overfitting and Regularisation 164
12.8.4 The Lottery Ticket Hypothesis 165
12.9 MLP for Classification 165
12.10Common Exam Traps 166

13 Deep Learning: CNNs and Transformers 168
13.1 What Makes Learning “Deep”? 168
13.1.1 Challenges of Depth 169
13.2 Convolutional Neural Networks 170
13.2.1 Convolution, Filters, and Weight Sharing 170
13.2.2 Feature Maps and Depth 171
13.2.3 Pooling and Translation Equivariance 171
13.3 From CNNs to Transformers: The Need for Global Context . . . 172
13.4 Self-Attention and the Transformer 172
13.4.1 Scaled Dot-Product Attention 172
13.4.2 Why Divide by $\sqrt{d_k}$? 174
13.4.3 Queries, Keys, and Values — An Analogy 174
13.4.4 Multi-Head Attention 174
13.4.5 Positional Encoding 175
13.5 Encoder, Decoder, and the Full Transformer 175

13.6 BERT and GPT: Two Paradigms 176
13.6.1 BERT — Bidirectional Encoder Representations from Transformers 176
13.6.2 GPT — Generative Pre-trained Transformer 177
13.6.3 Comparison Table 177
13.7 A Worked Attention Example 178
13.8 Transfer Learning and Foundation Models 179
13.9 Exam Traps: Common Mistakes and How to Avoid Them 180

14 The Exam Paper: *Transformers Struggle to Learn to Search* 183
14.1 What the Paper Does 183
14.2 The Algorithm the Transformer Learns 184
14.3 Key Results 185
14.4 Connection to Classical Search Algorithms 186
14.5 Critical Evaluation 187
14.6 Placing It in a Larger Context 188
14.7 Tricky Oral-Exam Questions 189

I Exam-Style Quiz 191

15 Practice Questions 192
15.1 Local Search 192
15.2 Constraint Satisfaction 193
15.3 Planning — Classical and Probabilistic 194
15.4 Knowledge Representation 195
15.5 Vagueness and Uncertainty 196
15.6 Machine Learning Foundations 197
15.7 Decision Trees and Ensembles 197
15.8 Support Vector Machines 198
15.9 Neural Networks 199
15.10 Deep Learning and Transformers 200

16 Answer Key 201
16.1 Local Search 201
16.2 Constraint Satisfaction 203
16.3 Planning — Classical and Probabilistic 205

16.4 Knowledge Representation 207
16.5 Vagueness and Uncertainty 209
16.6 Machine Learning Foundations 210
16.7 Decision Trees and Ensembles 212
16.8 Support Vector Machines 213
16.9 Neural Networks 214
16.10Deep Learning and Transformers 216

Chapter 1

Introduction: What Is AI and How This Book Is Organised

1.1 What Is This Course About?

Artificial Intelligence is not a single idea but a broad and evolving collection of methods for building systems that perceive, reason, plan, and learn. This book accompanies the course *Methods of Artificial Intelligence* (WS 2024/25) taught at the University of Osnabrück, primarily aimed at Bachelor and Master students in Cognitive Science. Rather than surveying AI from a purely philosophical angle, the course takes a hands-on, methods-first perspective: every topic is grounded in concrete algorithms, formal definitions, and worked examples drawn from real problem domains.

The course builds on the foundational ideas usually encountered in an introductory AI module — graph search, agents, basic logic — and goes considerably further. By the end of the book you will have a working understanding of seven major paradigms: local search and optimisation, constraint satisfaction, automated planning, knowledge representation, reasoning under uncertainty, classical machine learning, and modern deep learning architectures including transformers. These are not isolated islands; you will see recurring themes — state spaces, objective functions, trade-offs between expressiveness and tractability — reappear across all of them.

Key Idea

The central question of AI, as framed throughout this course, is how to build an agent that behaves rationally: one that consistently selects actions likely to achieve its goals given its beliefs about the world.

1.2 The Agent and Rationality Framing

A natural starting point is the notion of a *problem-solving agent*: a system that can look ahead, simulate sequences of actions inside a model of the world, and identify a course of action that eventually achieves a specified goal. This four-phase loop — goal formulation, problem formulation, search, execution — already appears in the very first session of the course and will resurface in every subsequent topic.

Goal formulation focuses the agent's attention by limiting which objectives and actions are worth considering. Problem formulation produces a precise, abstract description of the relevant states and the transitions between them. Search then systematically or heuristically explores possible action sequences within that model. Finally, execution applies the discovered plan to the real environment. Although different chapters of this book emphasise different phases — planning chapters focus on the first two, machine learning chapters transform the notion of search altogether — the agent perspective ties everything together.

Formally, a *search problem* is defined as a 4-tuple $\langle S, R, s_0, G \rangle$, where S is the state space, $R \subseteq S \times S$ is the successor (transition) relation, $s_0 \in S$ is the initial state, and $G : S \to \{\text{true}, \text{false}\}$ is the goal predicate. A *solution* is a sequence of states $(s_0, \ldots, s_n)$ such that each consecutive pair satisfies the transition relation and the final state satisfies the goal: $G(s_n) = \text{true}$. This clean definition accommodates routing on a map, solving sliding-tile puzzles, and planning robot motions with equal generality.

1.3 From Classical Search to the Broader Landscape

Classical search algorithms — breadth-first search, depth-first search, and heuristic methods such as A — work by constructing a *search tree* over the state-space graph and systematically exploring paths from the initial state to a goal state. They assume that the sequence of actions taken matters, not just the destination.

Many real-world problems, however, do not fit this mould. Constraint satisfaction, production scheduling, and machine learning all share a common structure: we care about finding a *good configuration* (an assignment of values to variables) rather than a particular path that led there. This motivates *local search*, the first major topic of this course. Local search starts from some initial assignment, repeatedly moves to a "promising" neighbour in the space of assignments, and continues until no further improvement can be found. The approach scales to problem spaces far too large for exhaustive enumeration, which is why variants of local search underlie both combinatorial optimisation and, ultimately, the training of neural networks.

(!) Exam Trap

Local search methods are not guaranteed to find a globally optimal solution; they can become stuck in local optima. Understanding *when* and *why* this happens — and what algorithmic remedies exist — is one of the core skills developed in Chapter 2.

1.4 Major Paradigms Covered in This Book

The following paragraphs give a brief preview of each paradigm, in the order they appear throughout the book. Taken together, they form a representative cross-section of the methods that practitioners and researchers rely on today.

Local search and optimisation. Hill-climbing, simulated annealing, local beam search, and genetic algorithms all share the neighbourhood-search structure described above. They differ in how they escape local optima and in what guarantees (if any) they provide on solution quality.

Constraint satisfaction. Many combinatorial problems are most naturally expressed as a set of variables, each with a domain of possible values, together with constraints that legal solutions must satisfy. Dedicated constraint propagation and backtracking algorithms exploit this structure far more efficiently than brute-force search.

Planning. Automated planning goes beyond finding a path in a fixed graph: it requires the agent to reason about actions whose effects change the world, sequences of which must be discovered to reach a goal state. Classical planning formalisms such as STRIPS provide a precise language for expressing this, and dedicated planners exploit the structure of action descriptions to search efficiently.

Knowledge representation. Intelligent behaviour often requires storing and querying large bodies of structured knowledge. Logic-based representations — propositional and first-order — allow facts and rules to be encoded in a form amenable to automated inference, though expressiveness and computational cost must always be balanced.

Vagueness and uncertainty. Real environments are rarely fully observable or perfectly modelled. Probability theory and, more specifically, Bayesian reasoning provide the tools for representing degrees of belief and updating them rationally as new evidence arrives.

Machine learning. Rather than hand-crafting rules, machine learning systems *learn* their behaviour from data. This part of the course covers the methodology of supervised learning, model evaluation, random forests, and support vector machines — methods that remain highly relevant both in isolation and as components of larger systems.

Deep learning and transformers. Neural networks with many layers have transformed the practical capabilities of AI systems in perception, natural language, and beyond. The course concludes by examining transformer architectures, which underpin today's most powerful language models, and situating them within the broader landscape of the methods covered earlier.

1.5 How This Book Is Structured

The chapters follow the lecture sequence closely, progressing from foundational algorithmic ideas toward increasingly data-driven methods. Chapter 2 introduces local search algorithms in detail, motivating the need to move beyond classical tree search. Chapter 3 develops constraint satisfaction problems and the propagation and backtracking techniques that solve them efficiently. Chapters 4 and 5 cover automated planning, first introducing the planning problem and its representations, then examining more advanced planning algorithms. Chapters 6 and 7 address knowledge representation, moving from propositional to richer first-order formalisms. Chapter 8 introduces probability-based reasoning and tackles the challenge of vagueness and uncertainty that pervades real-world AI. Chapter 9 shifts to machine learning methodology, establishing the statistical framework for supervised learning and model evaluation. Chapters 10 and 11 examine two powerful classical ML models — random forests and support vector machines — in depth. Chapter 12 closes the technical content with transformer architectures,

connecting the deep learning revolution back to the reasoning and representation ideas introduced earlier.

Throughout, the presentation follows the main reference of the course: Russell and Norvig's *Artificial Intelligence: A Modern Approach* (4th edition, Pearson, 2021), supplemented where appropriate by additional sources. Each chapter provides formal definitions, illustrative examples, key insights, and common pitfalls, with the aim of preparing you both for the course examination and for independent engagement with the field.

In a Nutshell

Chapter Summary

- *Methods of AI* is a methods-first course covering seven major paradigms: local search, constraint satisfaction, planning, knowledge representation, reasoning under uncertainty, classical machine learning, and deep learning.
- The unifying perspective is the **rational agent**: a system that formulates goals, models the world, searches for solutions, and executes them.
- A search problem is formally a 4-tuple $\langle S, R, s_0, G \rangle$; a solution is a transition-consistent path ending in a goal state.
- Classical search finds solution *paths*; local search finds good *configurations* — a distinction that motivates the remainder of the book.
- The book follows the course lecture sequence, progressing from algorithmic search and constraint reasoning through planning and logic to probabilistic inference and modern neural architectures.

Chapter 2

Local Search

Local search is a family of algorithms that occupies a conceptually distinct niche from the classical search methods covered in most AI textbooks. Whereas algorithms like breadth-first search or A* maintain an explicit frontier and a record of how each state was reached, local search deliberately discards all of that bookkeeping. It keeps track of only a *single current state* (or, in population-based variants, a small set of states) and moves through the state space by repeatedly stepping to a neighbouring state that looks more promising. This makes local search extraordinarily memory-efficient and often surprisingly effective — even in search spaces so vast that systematic exploration is completely infeasible.

The price paid is that local search cannot guarantee finding an *optimal* solution, and it cannot even certify that the solution it does return is globally optimal. What it can do — and does very well in practice — is find *good* solutions quickly. This makes it the method of choice for a wide class of real-world optimisation problems: scheduling, layout design, the travelling salesman problem, constraint satisfaction, machine learning hyperparameter tuning, and more.

2.1 Optimisation Problems and the State-Space Landscape

Every local search problem has three ingredients that the designer must specify:

1. **A state space $\mathcal{S}$** — the set of all complete configurations. In the 8-queens problem, for instance, one natural choice is to let every state be a tuple $(q_1, q_2, \ldots, q_8)$ where $q_i \in \{1, \ldots, 8\}$ gives the row of the queen in column i.

This encodes all $8^8 = 16\,777\,216$ board positions.

2. **A neighbourhood function** $N : \mathcal{S} \to 2^{\mathcal{S}}$ — the set of states reachable from the current state in a single step. The design of this function is crucial: a neighbourhood that is too small leads to many local optima, while one that is too large makes each step expensive to evaluate.
3. **An objective (fitness) function** $f : \mathcal{S} \to \mathbb{R}$ — the quantity to be maximised. (Minimisation problems are handled by maximising $-f$.)

Key Idea

In local search the *solution is the state itself*, not the path to it. This distinguishes it fundamentally from path-based search: we do not care *how* we arrived at a state — only how good that state is.

It is illuminating to think of the state space as a landscape in which the height of every point represents the value of the objective function f. Local search then becomes a walk across this landscape, always trying to climb higher.

2.1.1 Features of the Landscape

Several terrain features are important for understanding why local search can fail:

- **Global maximum.** The highest point in the entire landscape — the state $s^* = \arg\max_{s \in \mathcal{S}} f(s)$. Finding it is the ultimate goal.
- **Local maximum.** A state s such that $f(s) \geq f(s')$ for all $s' \in N(s)$, yet $f(s) < f(s^*)$. A climber following local improvements will stop here, mistaking it for the summit.
- **Plateau.** A flat region in which many neighbouring states have the same value. A pure hill climber stalls on a plateau because no neighbour is strictly better.
- **Shoulder.** A slope that levels off temporarily before rising again — a gentle plateau on the side of a hill. A hill climber may get stuck here if it requires a strict improvement at every step.
- **Ridge.** A long narrow elevated region whose flanks drop away steeply on both sides. Moving along a ridge is possible, but the path must be found carefully; naive hill climbing often slides off the sides.

The key insight is that all variants of local search are, at bottom, strategies

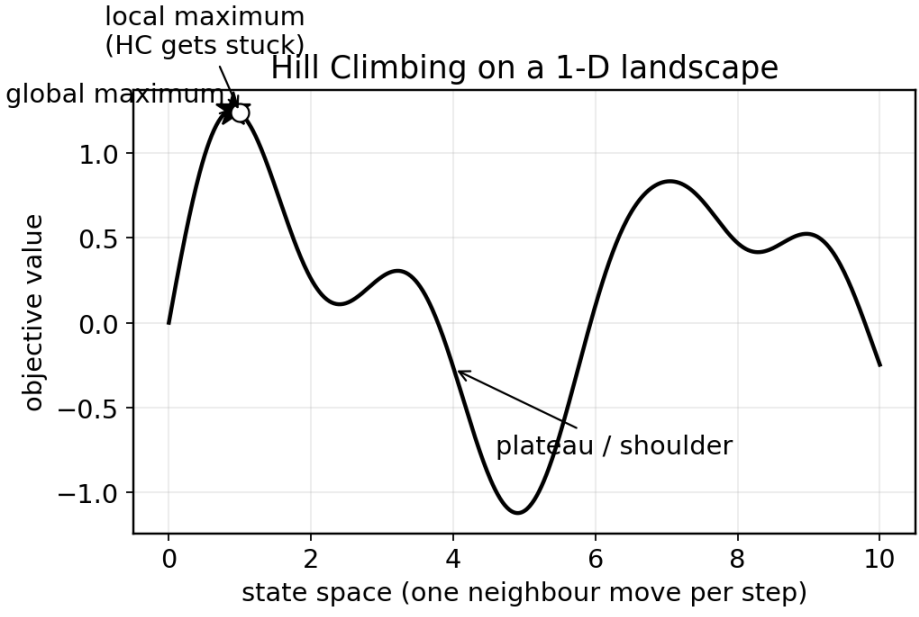

Figure 2.1: Hill climbing on a 1-D landscape: it follows the local gradient and can get trapped at a local maximum, a plateau, or a shoulder.

for navigating this landscape — they differ chiefly in how aggressively they accept moves that do *not* immediately increase f.

2.2 Hill Climbing

Hill climbing is the simplest local search algorithm and serves as the baseline against which all other variants are measured. The idea is exactly what the name suggests: always move to the best available neighbouring state, and stop when no improvement is possible.

2.2.1 Steepest-Ascent Hill Climbing

The standard (steepest-ascent) version evaluates *all* neighbours at each step and moves to the one with the highest value. If no neighbour improves upon the current state, the algorithm terminates and returns the current state as its answer.

```
current = select_random_initial_state()
```

```
while not termination_condition():
    neighbor = argmax(neighbors(current), value)
    if value(neighbor) <= value(current):
        return current
    current = neighbor
```

The algorithm is simple, fast per iteration (given enumerable neighbours), and completely memoryless — it stores only `current`. Its fatal flaw is equally simple: it stops at the first local maximum it encounters, which need not be the global maximum.

2.2.2 The 8-Queens Example

The 8-queens problem asks for a placement of eight queens on a chessboard such that no two queens attack each other. Using the tuple representation above, the neighbourhood is defined as all states reachable by moving a single queen to any other square in its column, giving $8 \times 7 = 56$ neighbours per state. The objective function to *minimise* is the number of attacking pairs; equivalently, we *maximise* $f(s) = -h(s)$ where $h(s)$ counts attacking pairs.

Hill climbing can go from a state with $h = 17$ attacking pairs to one with $h = 1$ in as few as five steps — a dramatic improvement. Yet that final state is typically only a *local* optimum: it has $h = 1$ (one remaining attacking pair, say queens in columns 4 and 7) and no single-column move reduces h further. The algorithm is stuck.

This illustrates a crucial point: local search is often extremely fast at making large gains, but the last mile — escaping a near-solution that happens to be a local optimum — can be surprisingly hard.

2.2.3 Neighbourhood Size Tradeoff

The choice of neighbourhood has a direct effect on the quality of local optima:

	Small neighbourhood	**Large neighbourhood**
Cost per step	Low	High
Local optima	More (many)	Fewer
Depth of basin	Shallow	Deeper

A very small neighbourhood (e.g. moving each queen down by exactly one row on a toroidal board, giving only 8 neighbours) makes each step cheap but vastly increases the number of local optima. A larger neighbourhood reduces the density of local optima at the cost of evaluating more states per step.

2.2.4 Variants of Hill Climbing

Because steepest-ascent hill climbing gets stuck so easily, several variants have been developed, each offering a different tradeoff between computational cost and robustness.

Stochastic Hill Climbing

Instead of always selecting the *best* improving neighbour, stochastic hill climbing selects a *random* neighbour that improves the current value:

```
neighbor = random.choice(n for n in neighbors(current)
                         if value(n) > value(current))
```

If no improving neighbour exists, the algorithm terminates. The selection probability can optionally be weighted to favour neighbours with higher values, providing a smooth interpolation between purely random search and greedy hill climbing. Stochastic hill climbing is particularly useful when the neighbourhood is large and evaluating all neighbours is expensive; it trades the guarantee of finding the best step for the benefit of acting on the first promising signal encountered.

(!) Exam Trap

Stochastic hill climbing still requires a **strictly improving** neighbour — it will *not* accept a move to a state of equal value. It therefore does **not** escape plateaus, and it does **not** accept downhill moves. Only simulated annealing and genetic algorithms genuinely escape local optima by accepting worse states.

First-Choice Hill Climbing

First-choice hill climbing terminates the neighbour search as soon as it finds *any* neighbour better than the current state, rather than exhaustively scanning all neighbours. This is especially useful when the neighbourhood is too large to

enumerate completely: the algorithm generates neighbours randomly (or in some order) and commits to the first one that shows improvement.

Random-Restart Hill Climbing

The most direct solution to the local-optima problem is simply to run hill climbing many times from independently and uniformly drawn random starting states. Each run produces a local optimum; the algorithm returns the best one found across all runs.

Key Idea

As the number of restarts $n \to \infty$, the probability that at least one run finds the global optimum approaches 1. For any finite n, however, there is no guarantee.

Random-restart hill climbing is embarrassingly parallel (each run is independent), extremely simple to implement, and often highly effective in practice. Its efficiency depends heavily on the fraction of the state space that consists of global (or near-global) optima: if that fraction is even moderately large, a small number of restarts suffices.

2.3 Simulated Annealing

The fundamental limitation shared by all hill climbing variants is that they *never* accept a move that makes the objective function worse. Yet, as the landscape picture makes clear, descending from a local optimum is sometimes necessary in order to find the global one. Simulated annealing (SA) addresses this by occasionally accepting downhill moves — but with a probability that decreases over time, so the algorithm gradually shifts from exploration to exploitation.

2.3.1 Physical Intuition

The name comes from metallurgy. When a metal is heated to high temperature and then cooled slowly, its atoms have time to settle into a low-energy crystalline structure — the global energy minimum. If cooled too quickly (quenching), the atoms freeze into a disordered, high-energy configuration (a local minimum). SA

mimics this process: a high "temperature" T early in the search allows large, seemingly bad moves; as T falls, only small improvements are accepted.

2.3.2 The Algorithm

```
current = select_random_initial_state()
for time in range(1, infinity):
    T = schedule(time)
    if T == 0:
        return current
    neighbor = random.choice(neighbors(current))
    delta = value(neighbor) - value(current)
    if delta > 0 or random.bernoulli(exp(delta / T)):
        current = neighbor
```

Two observations distinguish SA from hill climbing:

1. SA picks a *random* neighbour (not the best one).
2. If that neighbour is worse ($\Delta < 0$), it is accepted with probability

$$P(\text{accept}) = e^{\,\Delta E/T}, \qquad \Delta E = f(s') - f(s) < 0.$$

Because $\Delta E < 0$ and $T > 0$, the exponent is negative, so $P \in (0, 1)$. Uphill moves ($\Delta E > 0$) are always accepted.

2.3.3 How Temperature Governs Acceptance

The Boltzmann-inspired acceptance probability has an elegant structure. Fix $\Delta E = -2$ (a move that loses 2 units of value) and vary T:

Temperature T	$\boldsymbol{e^{-2/T}}$
∞	1.000
100	0.980
10	0.819
1	0.135
0.1	0.002
0	0

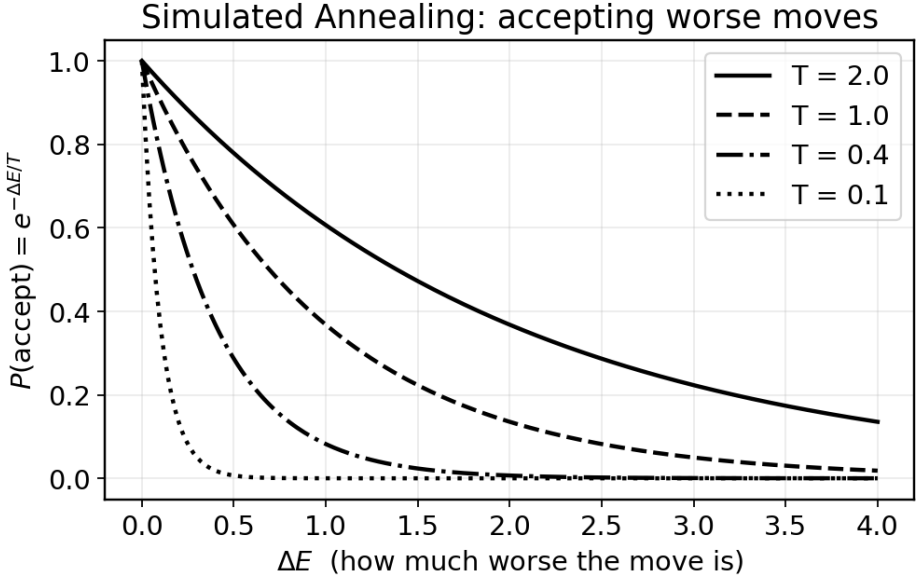

Figure 2.2: Simulated annealing accepts a worse move with probability $e^{-\Delta E/T}$. As T falls the algorithm becomes greedier.

At high temperature the algorithm accepts almost any move and behaves like a random walk. As $T \to 0$, only moves with $\Delta E \geq 0$ are accepted, and SA converges to ordinary first-choice hill climbing. The schedule thus traces a continuum between exploration and exploitation.

(!) Exam Trap

When $T \to 0$, simulated annealing becomes equivalent to **first-choice hill climbing** — *not* steepest-ascent. This is because SA always picks a *random* neighbour: at $T = 0$ it accepts that neighbour only if it is an improvement, which is exactly first-choice hill climbing.

2.3.4 The Cooling Schedule

The function `schedule(time)` determines how T decreases. The lecture presents two straightforward schemes:

- **Stepwise Linear.** Start with an initial temperature T_0 and subtract a

constant c at every step: $T(t) = T_0 - ct$.

- **Delayed Stepwise Linear.** Decrease T by c every k-th step rather than every step, giving the algorithm more time to explore at each temperature level.

A slower cooling schedule increases the probability of finding a high-quality solution but also increases runtime. The slides do *not* claim a guarantee of reaching the global optimum for any finite schedule.

2.3.5 Worked Example: SA Acceptance Calculation

Worked Example

Suppose the current state has value $f(s) = 40$ and a randomly chosen neighbour has value $f(s') = 37$, so $\Delta E = 37 - 40 = -3$. The current temperature is $T = 5$.

$$P(\text{accept}) = e^{-3/5} = e^{-0.6} \approx 0.549$$

SA would therefore accept this worse move roughly 55% of the time. If instead $T = 1$:

$$P(\text{accept}) = e^{-3/1} = e^{-3} \approx 0.050$$

At the lower temperature the same move is accepted only 5% of the time. As the schedule drives T toward zero, the probability that any downhill move is accepted approaches zero, and the algorithm locks in on improvements.

2.4 Local Beam Search

All the variants of hill climbing considered so far follow a single trajectory through the state space. Local beam search maintains k *states simultaneously* and uses information from across the entire beam to guide the next step.

2.4.1 Algorithm

```
k_current = set(select_random_initial_state() for _ in range(k))
while not termination_condition():
    all_neighbors = union(neighbors(c) for c in k_current)
```

```
k_best = argmax(all_neighbors, value, count=k)
if max(value(n) for n in k_best) <= max(value(c) for c in k_current):
    return argmax(k_current, value)
k_current = k_best
```

At each step, the algorithm generates *all* successors of *all* k current states, then selects the k best from that combined pool. Only k states are ever stored at one time, so memory usage is $O(k)$ — still linear, not exponential.

2.4.2 Local Beam Search vs. k Parallel Hill Climbers

This distinction is one of the most common exam traps in the course, and it is worth dwelling on carefully.

(!) Exam Trap

Local Beam Search is not the same as running k independent hill climbers. A parallel hill climber maintains k separate searches that never interact: each one looks only at its own neighbours and picks its own best step. Local Beam Search, by contrast, pools the successors of *all* k states and selects the k globally best. This means that a promising region discovered by one thread can "attract" the entire beam: states from that region will disproportionately make it into the top k, and the beam concentrates its search there.

This sharing of information is precisely what gives local beam search an edge over independent restarts: the k threads are not wasted by searching regions that are already known to be poor. The downside is that the beam can converge prematurely, with all k states clustered in the same local basin.

2.4.3 Stochastic Beam Search

To counteract premature convergence, stochastic beam search replaces the deterministic top-k selection with a randomised selection biased toward high-value states:

```
k_best = random.choice(all_neighbors, count=k,
                       weights=proportional_to_value)
```

This maintains diversity in the beam while still favouring better states — a direct analogue of the exploration–exploitation tradeoff seen in stochastic hill climbing. Stochastic beam search also happens to be the direct precursor to genetic algorithms.

2.5 Genetic Algorithms

Genetic algorithms (GAs) carry the beam search analogy one step further by explicitly modelling the biological mechanisms of selection, reproduction, and mutation. They maintain a *population* of states (individuals) and iterate through generations, each time producing offspring by combining pairs of parents and occasionally mutating the result.

2.5.1 Representation

Every state in a GA is encoded as a *chromosome* — a string over some alphabet of *genes*. For the 8-queens problem, a natural chromosome is a string of 8 digits, where the i-th digit is the row position of the queen in column i:

`83742516`

Each digit is a gene; the alphabet is $\{1, \ldots, 8\}$. The *fitness function* is the GA's name for the objective function: for 8-queens we can use $f(s) = 28 - h(s)$, where $h(s)$ is the number of attacking pairs (28 is the maximum possible, so fitness is always non-negative and equals 28 for a solution).

2.5.2 Selection

At each generation, pairs of parents are chosen for reproduction. The probability of selecting a given individual is proportional to its fitness — a scheme called *fitness-proportionate selection* (also known as roulette-wheel selection). More fit individuals are more likely to become parents and contribute their genetic material to the next generation.

2.5.3 Crossover

Two parent chromosomes are combined by a *crossover* operation. In the simplest case (1-point crossover), a random *crossover point* is chosen along the chromosome.

The child inherits the genes to the left of that point from one parent and the genes to the right from the other:

$$\underbrace{8\,3\,7\,4}_{\text{parent 1}} \mid \underbrace{2\,5\,1\,6}_{\text{parent 2}} \longrightarrow \text{child: } 8\,3\,7\,4\,2\,5\,1\,6$$

The intuition is that if two reasonably fit parents each encode useful partial solutions, their offspring may combine those solutions and achieve even higher fitness.

2.5.4 Mutation

After crossover, each gene in the child chromosome is independently subjected to a small random perturbation with probability p_m (typically $p_m \ll 1$). For the 8-queens encoding, a mutation might replace the row value of one queen with a randomly chosen alternative. Mutation prevents the population from converging prematurely to a local optimum by continuously injecting diversity.

2.5.5 The Basic Algorithm

```
population = {select_random_chromosome() for _ in range(k)}
while not termination_condition(population):
    for i in range(k):
        x, y = random.choice(population, count=2, weights=fitness)
        child = reproduce(x, y)     # crossover
        if random() < p_mutation:
            child = mutate(child)
        population.add(child)
    remove_random_chromosome(population, weights=1/fitness)
return argmax(population, fitness)
```

2.5.6 Permutation Problems and Crossover Repair

When chromosomes encode *permutations* — as they do in the travelling salesman problem (TSP) — naive 1-point crossover can produce invalid offspring. Consider two parents that each encode a valid tour of four cities:

Parent 1: 1 2 3 4 Parent 2: 3 1 4 2

Cutting after position 2 and combining gives 1 2 4 2 — city 2 appears twice and city 3 is missing. This is not a valid tour.

The standard fix is *uniform-order crossover*: a random binary template is generated; the child inherits the genes of one parent at positions where the template is 1, and fills the remaining positions in the order they appear in the other parent. This guarantees that every gene appears exactly once in the offspring. Alternatively, a *repair* operation can be applied after crossover to restore validity.

Worked Example

TSP example (4 cities, 0 fixed as start). Genes: $\{1, 2, 3, 4\}$. Chromosome length: 4 (order of remaining cities after city 0).

- Parent 1: `1 2 3 4`, tour $0 \to 1 \to 2 \to 3 \to 4 \to 0$.
- Parent 2: `3 1 4 2`, tour $0 \to 3 \to 1 \to 4 \to 2 \to 0$.
- Template: `1 0 1 0` (inherit positions 1 and 3 from Parent 1).
- Child (positions 1,3 from P1): `1 _ 3 _`.
- Fill blanks in order of P2 $(3, 1, 4, 2)$, skipping already-used genes 1 and 3: remaining order is $4, 2$.
- Child: `1 4 3 2`, tour $0 \to 1 \to 4 \to 3 \to 2 \to 0$. Valid!

2.5.7 Hybrid (Memetic) Algorithms

GAs and local search are complementary: GAs are good at exploring the landscape globally but slow at fine-tuning near a local optimum; hill climbing is the opposite. *Memetic algorithms* combine both: after each crossover-and-mutation step, a brief round of hill climbing is applied to the new child before it enters the population. This polishes each individual quickly, exploiting the gains from crossover without the overhead of many generations of pure evolution.

2.6 Algorithm Comparison

The following table summarises the key properties of all algorithms covered in this chapter.

Algorithm	Downhill moves?	Memory	Escapes local optima?	Guarantee
Steepest-ascent HC	No	1 state	No	Local optimum
Stochastic HC	No	1 state	No	Local optimum
First-choice HC	No	1 state	No	Local optimum
Random-restart HC	No	1 state	Via restarts	$P \to 1$ as $n \to \infty$
Simulated Annealing	Yes (prob.)	1 state + T	Yes	Heuristic
Local Beam Search	No	k states	Limited	Best of beam
Stochastic Beam Srch	No	k states	Better	Heuristic
Genetic Algorithm	Indirectly	Population	Yes	Heuristic

A few annotations deserve explanation. "Escapes local optima via restarts" means that random-restart hill climbing can escape a local optimum *between* runs, not within a single run — each individual run terminates at whatever local optimum it first encounters. "Indirectly" for genetic algorithms refers to the fact that crossover can move a child far from either parent, effectively teleporting across the landscape and landing outside the basin of the parents' local optima.

2.7 Exam Traps and Common Misconceptions

(!) Exam Trap

Local Beam Search is not k parallel hill climbers. Parallel hill climbers are entirely independent: each maintains its own state and never influences the others. Local Beam Search pools *all* successors of *all* k states and picks the global top-k. The consequence is that the beam can concentrate resources on promising regions — but can also suffer from premature convergence if all k states cluster in one basin.

(!) Exam Trap

What does SA do as $T \to 0$? It does *not* become steepest-ascent hill climbing. SA always picks a *random* single neighbour; at $T = 0$ it accepts that neighbour only if it is an improvement. This matches the behaviour of **first-choice hill climbing**, not steepest-ascent.

(!) Exam Trap

Stochastic HC does not escape plateaus. It requires a strictly improving neighbour. If the current state is on a flat plateau (all neighbours have the same value), stochastic hill climbing terminates just like steepest-ascent hill climbing. Only SA accepts moves that are not strictly improving (via the Boltzmann probability).

(!) Exam Trap

The sign of ΔE in SA. For a downhill move, $\Delta E = f(s') - f(s) < 0$. The exponent $\Delta E/T$ is therefore *negative*, giving $e^{\Delta E/T} \in (0, 1)$. Students sometimes write $e^{|\Delta E|/T}$ (positive exponent > 1), which would be a probability greater than 1 — nonsensical. Always keep the sign: $P = e^{\Delta E/T}$ with $\Delta E < 0$.

(!) Exam Trap

Cooling schedule scope. The lecture presents only **Stepwise Linear** and **Delayed Stepwise Linear** cooling. Exponential cooling (common in textbooks) is not part of the slides. Slower cooling increases the *probability* of a high-quality solution but does not guarantee the global optimum for any finite schedule.

(!) Exam Trap

Permutation crossover. Applying naive 1-point crossover to permutation-encoded chromosomes (e.g. TSP) yields invalid solutions. The fix is uniform-order crossover, a domain-specific repair operation, or a different representation altogether.

2.8 Going Further: Oral-Exam Deep Cuts

These are the "why / compare / what-if" questions that an examiner asks *after* you have recited the definitions. Each comes with a model answer.

Q: When is first-choice hill climbing actually *better* than steepest-ascent, even though steepest-ascent makes "the best" move?
A: When the neighbourhood is huge. Steepest-ascent must evaluate *every* neighbour before each step, which is wasteful if there are thousands of them. First-choice commits to the first improving neighbour it samples, so it makes progress far faster per unit of computation. "Best move" is not the same as "best use of compute."

Q: Is hill climbing complete? Is it optimal?
A: Neither. It is incomplete (it can terminate at a local optimum without finding a solution) and not optimal (the local optimum it returns need not be global). Random-restart makes it *probabilistically* complete: $P(\text{find global}) \to 1$ as restarts $\to \infty$, but never with a finite guarantee.

Q: Simulated annealing is sometimes said to "guarantee" the global optimum. True?
A: Only as a theoretical limit: with a sufficiently *slow* (logarithmic) cooling schedule and infinite time, SA converges to the global optimum with probability 1. For any *practical, finite* schedule there is no such guarantee — and the lecture's stepwise-linear schedules certainly do not provide one.

Q: Why is genetic crossover more than just "a big mutation"?
A: Mutation perturbs one individual locally. Crossover *combines* useful partial solutions ("building blocks") from two fit parents, letting the offspring inherit good substructures from both — effectively teleporting across the landscape, outside the basins of either parent. This is why GAs can escape local optima that mutation alone could not.

Q: When do genetic algorithms fail badly?
A: On *deceptive* problems, where good building blocks combine into bad offspring (the fitness landscape misleads crossover), and on tightly-coupled encodings where crossover routinely destroys structure (e.g. naive crossover on permutations — hence uniform-order crossover). They also suffer premature convergence if diversity is lost.

Q: How does local beam search differ from k parallel hill climbers, and what is its failure mode?
A: It pools the successors of all k states and keeps the global top-k, so a promising region attracts the whole beam (information sharing). Its failure mode is *premature convergence*: all k states cluster in one basin. Stochastic beam search fixes this by selecting successors probabilistically in proportion to value, preserving diversity.

Q: What is the relationship between hill climbing and gradient descent?
A: Hill climbing is the *discrete* analogue of gradient ascent: both follow the local direction of improvement. In continuous, differentiable spaces the gradient gives that direction analytically; in discrete spaces we must enumerate or sample neighbours. Both share the same weakness — getting trapped at local optima — and the same family of fixes (restarts, momentum/temperature, stochasticity).

Q: Why does the 8-queens hill climb so often stall at exactly one attacking pair?
A: Because the last conflict often sits in a *narrow basin*: every single-queen move either keeps the conflict or introduces a new one, so no neighbour strictly improves h. The algorithm is at a local optimum with $h = 1$. Escaping requires a temporarily worse move (SA) or a fresh start (random restart).

In a Nutshell

Local search solves optimisation problems by iteratively improving a current state, using only local information.
Hill Climbing follows the gradient of the objective function; it is fast but stops at local optima. Variants (stochastic, first-choice, random-restart) reduce this risk to varying degrees.
Simulated Annealing introduces a temperature parameter T controlling the probability $e^{\Delta E/T}$ of accepting a downhill move. High T encourages exploration; as $T \to 0$ the algorithm converges to first-choice hill climbing.
Local Beam Search maintains k states simultaneously, pooling their successors and retaining the k globally best at each step. This is fundamentally different from running k independent hill climbers, because states share information.
Genetic Algorithms maintain a population, selecting fit individuals for crossover and mutation. They explore the landscape broadly and combine partial solutions, at the cost of more complex implementation (especially

for permutation-encoded problems).
All methods trade *exploration* (escaping local optima) against *exploitation* (improving within a promising region). No method in this family guarantees a globally optimal solution in polynomial time for arbitrary landscapes.

Chapter 3

Constraint Satisfaction Problems

Constraint satisfaction problems occupy a central position in AI because they provide a clean, declarative language for modelling a huge variety of real-world tasks — from timetable scheduling and frequency assignment to map colouring, Sudoku, and spatial reasoning. Unlike a plain state-space search, where a state is treated as a black box, a CSP exposes the *internal structure* of every state as a set of variable–value pairs. Algorithms can exploit this structure to prune enormous portions of the search space before ever testing a complete assignment.

3.1 The CSP Formalism

3.1.1 Variables, Domains, and Constraints

A **constraint satisfaction problem** (CSP) is formally a triple

$$(\{X_1, \ldots, X_n\},\ \{D_1, \ldots, D_n\},\ R),$$

where:

- $X_1, \ldots, X_n$ are the **variables** — the slots to be filled.
- Each D_i is the **domain** of X_i, i.e. the set of values that X_i may legally take. Domains can be finite (e.g. $\{\text{red}, \text{blue}, \text{green}\}$, $\{1, \ldots, 9\}$) or infinite (e.g. $\mathbb{N}$, $\mathbb{R}$).

- R is a **set of constraints**. Each constraint is a relation — a predicate $P(X_{i_1}, \ldots, X_{i_k})$ — that restricts which combinations of values the listed variables may simultaneously take.

Constraints come in several arities. A *unary* constraint involves a single variable, e.g. $X_7 \in \{\text{red}, \text{blue}, \text{green}\}$. A *binary* constraint involves two variables, e.g. $X_1 \neq X_2$ or $X_1 \leq X_2$. Higher-arity constraints are also possible, e.g. $X_3 + X_4 \geq 4X_5 + 2X_6$ (a 4-ary constraint). For algorithmic purposes it is convenient to restrict attention to *normalised CSPs*, which contain only unary and binary constraints; any higher-arity CSP can be transformed into an equivalent normalised one in polynomial time.

Constraints may be represented *explicitly* — by listing every satisfying tuple — or *implicitly* — by a function that returns `true` or `false` for any given tuple. If the criterion is easy to compute (e.g. a simple inequality), the implicit representation is far more efficient: for two variables with domain $\{1, \ldots, 20\}$ the constraint $X_1 < X_2$ has $\frac{20 \cdot 19}{2} = 190$ explicit tuples, but can be tested with a single comparison implicitly.

Key Idea

A **solution** is a *complete* assignment (every variable has a value) that is also *consistent* (no constraint is violated). A CSP is called **consistent** if at least one solution exists, and **inconsistent** otherwise.
Note the two uses of the word "consistent": it applies both to an assignment (no current violation, possibly partial) and to the CSP as a whole (a solution exists). Context always disambiguates.

3.1.2 Constraint Graphs

For normalised CSPs a compact visual representation is the **constraint graph**:

- Each **node** represents a variable X_i, and its unary constraint P_i can be thought of as a self-loop or simply absorbed into the domain.
- Each **edge** (X_i, X_j) represents the binary constraint P_{ij} between those two variables.

The topology of the constraint graph immediately reveals how tightly the variables are coupled. Variables connected by many edges are highly constrained; isolated nodes (or nodes connected only to themselves) are essentially independent.

Efficient algorithms exploit structural properties such as tree-shaped constraint graphs, which can be solved in linear time.

3.2 Classic Examples

3.2.1 Map Colouring

In the **map colouring** problem, regions of a map must be assigned colours so that no two adjacent regions share the same colour. For a map with regions A, B, C, D, E, F:

- **Variables:** A, B, C, D, E, F.
- **Domains:** $\{\text{red}, \text{blue}, \text{green}\}$ for every variable.
- **Constraints:** $A \neq B$, $A \neq C$, $A \neq E$, $B \neq E$, $B \neq F$, $C \neq E$, $C \neq F$, $D \neq F$, $E \neq F$.

Each constraint corresponds to an edge in the constraint graph. Solving the problem means finding a colouring consistent with all nine constraints — a concrete task that highlights every major CSP algorithm.

3.2.2 Sudoku

A 9×9 Sudoku puzzle is modelled as a CSP with 81 variables (one per cell), each with domain $\{1, \ldots, 9\}$. The constraints enforce that every row, every column, and every 3×3 box contains each digit exactly once. Pre-filled cells add unary constraints that immediately fix certain domain values. Because Sudoku has a very dense constraint graph, it showcases why arc-consistency preprocessing is so valuable: propagating constraints after each assignment can often reduce domains to singletons, effectively solving large parts of the puzzle without any backtracking.

3.2.3 Frequency Assignment and Timetabling

Practical applications abound. In **frequency assignment**, radio stations are variables, frequencies are domain values, and constraints prevent stations with overlapping coverage areas from sharing a frequency. In **timetable scheduling**, courses are variables, (time-slot, room) pairs are domain values, and constraints forbid double-booking rooms, respect lecturer availability, and avoid clashes for

students in the same programme and semester. Both problems have exactly the CSP structure above; only the domain and constraint details change.

3.3 Consistency Concepts

Rather than jumping straight into search, a powerful preprocessing strategy is to *reduce domains* by enforcing consistency. The basic idea: if a value in D_i cannot participate in any solution consistent with the current domains, delete it. This can never remove a solution, but it can shrink the search space dramatically.

3.3.1 Node Consistency

Variable X_i is **node consistent** if every value in D_i satisfies all unary constraints on X_i. Making the entire CSP node consistent is trivial: for each variable, restrict its domain to only those values that pass its unary constraint. This is always the first step of preprocessing.

Formally: X_i is node consistent iff $\forall x \in D_i : P_i(x)$ holds.

3.3.2 Arc Consistency

Arc consistency is the more powerful and widely-used concept. Variable X_i is **arc consistent with** X_j if for every value $x \in D_i$ there exists at least one value $y \in D_j$ such that the binary constraint $P_{ij}(x, y)$ is satisfied. Formally:

$$\forall x \in D_i \;\; \exists y \in D_j : P_{ij}(x, y).$$

Crucially, arc consistency is **directional**: X_i being arc consistent with X_j does not imply that X_j is arc consistent with X_i. To make the whole CSP arc consistent, both directions of every edge must be made consistent. If, after removing a value from D_i, a previously arc-consistent arc pointing *into* X_i may be violated — so those arcs must be re-examined.

(!) Exam Trap

Arc consistency is directional. The arc (X_i, X_j) and the arc (X_j, X_i) are *separate*. Revising one does not automatically revise the other. AC-3 must explicitly add both directions to the queue and must re-add arcs whenever a domain shrinks (see below).

3.3.3 The AC-3 Algorithm

AC-3 is the standard, efficient algorithm for establishing arc consistency across an entire CSP. It maintains a **worklist** (queue) of arcs to check, initialised with all directed arcs. It then processes arcs one by one, revising each: removing from D_i any value that has no support in D_j. If any value is removed, all arcs pointing *into* X_i (from its other neighbours) are re-added to the queue, because those arcs may now be violated.

AC-3 in pseudocode:

1. Initialise queue $Q \leftarrow \{(X_i, X_j) \mid P_{ij} \in R\} \cup \{(X_j, X_i) \mid P_{ij} \in R\}$.
2. While Q is not empty:
   a. Remove arc (X_i, X_j) from Q.
   b. **Revise**(X_i, X_j): for each $x \in D_i$, if there is no $y \in D_j$ with $P_{ij}(x, y)$, delete x from D_i.
   c. If D_i was reduced:
      - If $D_i = \emptyset$: return INCONSISTENT.
      - For every $X_k \neq X_j$ with constraint P_{ki}: add (X_k, X_i) to Q.
3. Return CONSISTENT (with reduced domains).

Complexity of AC-3. Let c be the total number of *directed* arcs (equivalently, twice the number of binary constraints, since each undirected edge yields two directed arcs), and let $d = \max_i |D_i|$ be the maximum domain size. Each arc (X_i, X_j) can be added to the queue at most d times (once per value removed from D_i, and no more values can be removed than the domain has). Each revision scans D_j, taking $O(d)$ time. Therefore the overall complexity is:

$$O(c \cdot d^3).$$

Here c counts arcs (not constraints) and d is the *maximum* domain size across all variables. This bound is tight in the worst case.

Key Idea

Complexity of AC-3 is $O(c \cdot d^3)$, where c = number of directed arcs and d = maximum domain size. Each arc is re-queued at most d times; each revision costs $O(d)$.

(!) Exam Trap

AC-3 does not solve the CSP — it only reduces domains. If all domains remain non-empty after AC-3, the CSP may still be inconsistent. Arc consistency (2-consistency) is strictly weaker than full consistency. Only if a domain becomes *empty* can AC-3 certify inconsistency.

3.3.4 Limits of Arc Consistency: Path Consistency and k-Consistency

Arc consistency considers pairs of variables. **Path consistency** extends this: nodes X_i and X_j are path consistent with respect to X_m if, for every arc-consistent assignment (x, z) to (X_i, X_j), there exists a value $y \in D_m$ consistent with both x and z. Formally:

$$\forall x \in D_i,\ z \in D_j : P_{ij}(x, z) \ \Rightarrow\ \exists y \in D_m : P_{im}(x, y) \wedge P_{mj}(y, z).$$

These concepts generalise to **k-consistency**: every consistent assignment of $k-1$ variables can be extended to a consistent assignment of k variables.

- 1-consistency = node consistency.
- 2-consistency = arc consistency.
- 3-consistency = path consistency (given node consistency).

Strong k-consistency means j-consistent for all $j \leq k$. If a CSP is strongly n-consistent (where n is the number of variables), it can be solved without any backtracking. However, achieving high k-consistency is exponentially expensive in k, so in practice one stops at arc consistency ($k = 2$) and relies on search for the rest.

3.4 Worked Example: Applying AC-3 to a Three-Node Graph-Colouring Problem

Worked Example

Consider three variables x, y, z with:

$$D_x = \{\text{red}, \text{blue}\}, \quad D_y = \{\text{red}\}, \quad D_z = \{\text{red}, \text{blue}\}.$$

Constraints: $x \neq y$, $x \neq z$, $y \neq z$ (a triangle; all pairs must differ).

Step 1 – initialise queue: $Q = \{(x,y),(y,x),(x,z),(z,x),(y,z),(z,y)\}$.

Step 2 – revise (x,y): For $x = \text{red}$: is there a value in $D_y = \{\text{red}\}$ with red $\neq$ red? No. Delete **red** from D_x.
For $x = \text{blue}$: is there a value in D_y with blue $\neq$ red? Yes. Keep **blue**.
D_x shrinks to $\{\text{blue}\}$. Re-add (z,x) (since z has a constraint into x).

Step 3 – revise (y,x): For $y = \text{red}$: is there a value in $D_x = \{\text{blue}\}$ with red $\neq$ blue? Yes. No change.

Step 4 – revise (x,z): For $x = \text{blue}$: is there a value in $D_z = \{\text{red}, \text{blue}\}$ with blue $\neq v$? Yes (red). No change.

Step 5 – revise (z,x) (re-added at Step 2): For $z = \text{red}$: is there a value in $D_x = \{\text{blue}\}$ with red $\neq$ blue? Yes. Keep.
For $z = \text{blue}$: is there a value in $D_x = \{\text{blue}\}$ with blue $\neq$ blue? No. Delete **blue** from D_z.
$D_z = \{\text{red}\}$. Re-add (y,z).

Step 6 – revise (y,z) (re-added at Step 5): For $y = \text{red}$: is there a value in $D_z = \{\text{red}\}$ with red $\neq$ red? No. Delete **red** from D_y.
$D_y = \emptyset$. **Return Inconsistent.**

Conclusion: The triangle with only two colours is indeed inconsistent. AC-3 found this *without* enumerating any complete assignment. Note how Step 2 caused a cascade: shrinking D_x forced re-examination of (z,x), which then forced re-examination of (y,z), eventually emptying D_y.

3.5 Backtracking Search

Consistency preprocessing alone is usually insufficient to find a solution; search is still needed. **Backtracking search** is the standard complete algorithm for CSPs. It assigns values to variables one at a time, checking consistency after each assignment. If a partial assignment violates any constraint, it *backtracks* — reverting the most recent assignment and trying a different value. The naive generate-and-test approach, by contrast, checks consistency only after a full assignment, which is far less efficient.

The performance of backtracking search depends critically on two choices made at each step:

1. **Which variable to assign next?** (variable ordering)
2. **Which value to try first?** (value ordering)

These choices are governed by *heuristics*.

3.6 Ordering Heuristics

3.6.1 MRV: Minimum Remaining Values (Variable Ordering)

The **Minimum Remaining Values** (MRV) heuristic — also called the *fail-first* or first-fail heuristic — selects the unassigned variable with the *smallest* current domain. The intuition is that the most constrained variable is most likely to cause a failure soon; by choosing it early, the algorithm detects dead ends quickly and avoids wasting time on subtrees that will fail anyway.

Formally: choose $X_i = \arg\min_{i \text{ unassigned}} |D_i|$.

3.6.2 Degree Heuristic (Variable Ordering, Tie-Breaker)

When two or more variables have the same domain size, MRV alone does not distinguish them. The **degree heuristic** breaks ties by selecting the variable involved in the largest number of constraints with *other unassigned* variables. The variable with the highest degree is chosen because assigning it will impose the most constraints going forward, again tending to prune the search space early.

(!) Exam Trap

MRV and the degree heuristic serve different roles. MRV is the primary variable-ordering heuristic; the degree heuristic is the *tie-breaker*. They are not interchangeable, and the degree heuristic is rarely useful on its own.

3.6.3 LCV: Least Constraining Value (Value Ordering)

Once a variable has been selected, the **Least Constraining Value** (LCV) heuristic determines the order in which its values are tried. It selects the value that *rules out the fewest choices* for the remaining unassigned neighbours — the value that imposes the least burden on the rest of the problem. Unlike MRV (which tries to fail fast on the variable side), LCV tries to *succeed* by keeping as many options open as possible.

To compute LCV for a value v of variable X_i: count how many values in the domains of X_i's unassigned neighbours would be eliminated if $X_i = v$. Assign v with the smallest such count first.

(!) Exam Trap

MRV selects variables; LCV selects values. Their goals are opposite: MRV *fails early* (picks the most constrained variable); LCV *succeeds gently* (picks the value with least collateral damage). They are complementary — both should be applied simultaneously.

3.6.4 Forward Checking

Forward checking is a simple but powerful inference technique integrated into backtracking search. After assigning $X_i = v$, it immediately removes v (or any other value now inconsistent) from the domains of all unassigned neighbours of X_i. If any neighbour's domain becomes empty, the current partial assignment is immediately abandoned (no need to explore deeper). Forward checking is essentially a lightweight, one-step version of arc consistency applied incrementally during search.

The table below summarises the three heuristics and forward checking.

Technique	Applies to	Goal	Key idea
MRV	Variable ordering	Fail fast	Pick variable with smallest domain
Degree heuristic	Variable ordering (tie-break)	Fail fast	Pick variable with most constraints on unassigned vars
LCV	Value ordering	Succeed gently	Pick value eliminating fewest options for neighbours
Forward checking	Inference during search	Detect failures early	Remove inconsistent values from neighbours after each assignment

3.7 Spatial Reasoning: RCC-8

CSP techniques are not limited to discrete assignment problems. Spatial reasoning over regions of space is a rich application domain, and the **Region Connection Calculus** (RCC-8) provides a qualitative constraint language for describing how two-dimensional regions relate to each other. RCC-8 defines exactly **eight mutually exclusive and jointly exhaustive** base relations between any two non-empty, connected regions A and B:

Relation	**Name**	**Meaning**
DC	Disconnected	A and B share no points
EC	Externally Connected	A and B share only a boundary point
PO	Partial Overlap	A and B overlap but neither contains the other
TPP	Tangential Proper Part	A is inside B, touching B's boundary
NTPP	Non-Tangential Proper Part	A is strictly inside B, not touching B's boundary
TPPi	TPP inverse	B is a tangential proper part of A
NTPPi	NTPP inverse	B is a non-tangential proper part of A
EQ	Equal	A and B are identical

In a spatial CSP, variables represent regions, domains are subsets of these eight relations, and constraints (derived from maps, descriptions, or common sense) restrict which relation(s) can hold between each pair of regions. Algorithms like AC-3 can propagate spatial constraints using a *composition table* that encodes which relations can result from combining two known relations. This makes RCC-8 a direct instance of the general CSP framework.

3.8 Putting It All Together: A Small Worked Example with MRV

Worked Example

We return to the six-region map colouring problem with regions A, B, C, D, E, F, domains $\{\mathrm{R}, \mathrm{G}, \mathrm{B}\}$, and the nine $\neq$ constraints listed in Section 1.2. We apply backtracking with MRV and forward checking.

Step 1: All domains have size 3. MRV breaks ties arbitrarily; say we pick A. Assign $A = \mathrm{R}$. Forward checking removes R from all neighbours of A (namely B, C, E): $D_B = \{G, B\}$, $D_C = \{G, B\}$, $D_E = \{G, B\}$.

Step 2: Remaining domain sizes: B, C, E all have size 2; D, F still have size 3. MRV picks one of B, C, E, say B. Assign $B = \mathrm{G}$. Forward checking: remove G from neighbours of B that are unassigned (E, F): $D_E = \{B\}$, $D_F = \{R, B\}$.

Step 3: MRV now picks E (domain size 1). Assign $E = \mathrm{B}$. Forward checking: remove B from unassigned neighbours of E (namely C, F): $D_C = \{G\}$, $D_F = \{R\}$.

Step 4: MRV picks either C or F (both have domain size 1). Assign $C = \mathrm{G}$ and $F = \mathrm{R}$ (no conflicts; C's only constraint with F is $C \neq F$, and $G \neq R$). Assign D last; D's only constraint is $D \neq F$, and $D_F = \{R\}$, so any of $\{G, B\}$ works.

Result: $A = \mathrm{R}$, $B = \mathrm{G}$, $C = \mathrm{G}$, $D = \mathrm{G}$, $E = \mathrm{B}$, $F = \mathrm{R}$ — a valid 3-colouring, reached without any backtracking thanks to the combination of MRV and forward checking.

3.9 Common Exam Traps

(!) Exam Trap

Arc consistency is directional — revise both ways and re-add arcs. The arc (X_i, X_j) is separate from (X_j, X_i). When AC-3 removes a value from D_i, every arc (X_k, X_i) for all neighbours $X_k \neq X_j$ must be re-added to the queue. Forgetting this step leaves the algorithm incomplete.

(!) Exam Trap

AC-3 complexity is $O(c \cdot d^3)$, where c counts directed arcs. Each undirected constraint contributes two directed arcs, so $c = 2 \times$ (number of binary constraints) in the worst case. Confusing c with the number of constraints (rather than arcs) gives the wrong factor of 2 — small but tested.

(!) Exam Trap

Non-empty domains after AC-3 do not certify consistency. AC-3 can only *prove* inconsistency (via an empty domain); it cannot prove that a solution exists. A classic counterexample: three variables, two colours, all pairwise $\neq$. Arc consistency leaves all domains intact ($\{\mathrm{R}, \mathrm{B}\}$), yet the CSP is clearly inconsistent.

(!) Exam Trap

MRV is a variable-ordering heuristic; LCV is a value-ordering heuristic. Do not conflate them. MRV picks the *variable* most likely to fail (smallest domain). LCV picks the *value* least likely to cause future failure (fewest eliminations). The degree heuristic acts only as a tie-breaker for MRV, not as a standalone replacement.

(!) Exam Trap

Backtracking search is not local search. Backtracking works on *partial* assignments and is *complete* (it will find a solution if one exists, or certify inconsistency). Local search works on *complete* (but possibly invalid) assignments and is *incomplete* — a locally optimal solution with violations tells us nothing about whether the CSP is consistent.

3.10 Going Further: Oral-Exam Deep Cuts

Q: In the AC-3 complexity $O(c\,d^3)$, where does the *third* factor of d come from? Why is it not $O(c\,d^2)$?
A: A single `Revise` of an arc costs $O(d^2)$: for each of the d values of X_i, scan

up to d values of X_j for support. The extra factor of d is because each arc can be *re-queued up to d times* — it is re-added every time the neighbour's domain shrinks, and a domain can lose at most d values. So c arcs $\times$ d re-queues $\times$ d^2 per revise $= O(c\,d^3)$. $O(c\,d^2)$ would be a single pass over all arcs with no re-queuing.

Q: Arc consistency leaves all domains non-empty. Has it proven the CSP is solvable?

A: No. Arc consistency (2-consistency) can only *disprove* solvability (by emptying a domain); it can never *prove* it. Counterexample: three variables, two colours, all pairwise $\neq$. Every arc is consistent, every domain stays $\{R, B\}$, yet no solution exists. What is missing is higher-order (k-) consistency: you would need strong n-consistency to skip search entirely, which is exponentially expensive.

Q: Why apply "fail-first" (MRV) to variables but "succeed" (LCV) to values — isn't that contradictory?

A: No, because the two decisions have opposite leverage. You are forced to assign *some* variable, so choose the one most likely to fail *now* (smallest domain) to prune dead subtrees early. But once the variable is fixed, you want the assignment most likely to *lead to a solution*, so pick the value that removes the fewest options from neighbours. "Fail fast on the variable, succeed gently on the value."

Q: What is the difference between forward checking and maintaining arc consistency (MAC) during search?

A: Forward checking only prunes the *direct* neighbours of the just-assigned variable (one step of propagation). MAC runs full AC-3 after each assignment, propagating *transitively* through the whole constraint graph. MAC prunes more (fewer dead-end descents) but costs more per node; forward checking is the cheaper, shallower version.

Q: For which constraint-graph structure is a CSP solvable in *linear* time, and why?

A: A *tree*-structured constraint graph. Order the variables topologically from a root, make the graph directed-arc-consistent in one backward pass, then assign in one forward pass — no backtracking is ever needed. This runs in $O(n\,d^2)$. General graphs can sometimes be reduced to trees via cutset conditioning or tree decomposition.

Q: Can local search (Chapter 2) solve CSPs? What is the trade-off versus backtracking?

A: Yes — the *min-conflicts* heuristic: start from a complete (possibly invalid)

assignment and repeatedly reassign a conflicted variable to the value minimising conflicts. It is fast and scales to huge problems (famously n-queens for large n), but it is *incomplete*: it cannot certify inconsistency, and a local optimum with violations tells you nothing. Backtracking is complete but can be exponential.

Q: How do you turn an n-ary constraint into binary ones?
A: Introduce a fresh "meta-variable" whose domain is the set of satisfying tuples of the n-ary constraint, then add binary constraints linking each original variable to the corresponding component of the meta-variable (the *dual* / hidden-variable encoding). This lets arc-consistency machinery handle constraints of any arity.

In a Nutshell

Constraint Satisfaction Problems — In a Nutshell

- A CSP is a triple $(\{X_i\}, \{D_i\}, R)$: variables, domains, constraints. A *solution* is a complete, consistent assignment.
- **Constraint graphs** represent normalised (unary/binary) CSPs: nodes are variables, edges are binary constraints.
- **Node consistency**: every domain value satisfies the variable's unary constraint. Trivial to achieve.
- **Arc consistency**: for every value of X_i, there exists a compatible value for X_j. Directional — must be enforced in both directions.
- **AC-3** establishes arc consistency in $O(c \cdot d^3)$ time (c = directed arcs, d = max domain size). When a domain shrinks, re-add incoming arcs to the queue.
- Arc consistency is necessary but *not sufficient* for a solution to exist.
- **Backtracking search** with **MRV** (fail-first variable selection), the **degree heuristic** (tie-breaker), **LCV** (value ordering), and **forward checking** (lightweight propagation) dramatically reduces the search space in practice.
- **RCC-8** applies the CSP framework to spatial reasoning via eight base relations: DC, EC, PO, TPP, NTPP, TPPi, NTPPi, EQ.

Chapter 4

Classical Planning

4.1 What Is Planning?

Planning is one of the most fundamental capacities attributed to intelligent agents. In everyday language, a plan is a sequence of steps that gets you from where you are to where you want to be. AI formalises this intuition precisely: given a description of the world's current state, a set of available actions, and a description of a desired goal state, a *planning system* must find a sequence of actions that, when executed, transforms the initial state into one that satisfies the goal. The classical view assumes a **fully observable, deterministic, and static** environment—the agent always knows the complete current state, actions have guaranteed outcomes, and the world does not change unless the agent acts.

Key Idea

A **plan** (classical definition) is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$ such that applying a_1 through a_n to the initial state s_0 produces a state that satisfies the goal G.

An alternative view is to assign an action to every possible state, producing a *policy* rather than a fixed sequence. This view is the natural stepping stone towards probabilistic planning (Markov Decision Processes, covered in the next chapter). The classical chapters focus on the sequence-based view.

4.1.1 Planning vs. Classical Search

Students often ask why planning is a separate discipline at all, given that search algorithms were introduced in earlier sessions. The distinction lies in what the algorithm *exploits*:

- **Local search** is not interested in the sequence of actions—only the final configuration matters.
- **Classical search** (BFS, DFS, A*) treats states as opaque objects. A state can be an arbitrarily complex data structure, but the search algorithm never inspects its internal components; states are compared for equality and successors are generated by applying transitions, but the algorithm has no vocabulary for talking about *parts* of a state.
- **Planning** is interested in *both* the sequence of actions and the *internal structure* of states. States have a structured, compositional representation—they are collections of properties (atoms, fluents)—and the planner actively exploits this structure to reason about which actions are applicable and what effects they have.

This structural insight is what allows planners to define operators generically ("move block X onto block Y" rather than one transition per pair of blocks) and to apply powerful heuristics that estimate goal distance by inspecting how many goal conditions remain unsatisfied.

4.1.2 Components of a Planning System

A planning system has two major components: a **planning language** and a **planning algorithm**. The language specifies how states, goals, and actions are represented. Most planning languages are subsets of first-order logic; states and goals can typically be expressed as conjunctions of ground literals, and operators are described as rules with a structured action schema. The algorithm then uses this representation to construct a plan, either by search, logical inference, or specialised techniques.

4.2 Three Classic Challenges

Before diving into formalisms, it is essential to understand the three canonical challenges that any planning representation must address. These problems recur

throughout the literature and are a perennial source of exam errors when students conflate them.

4.2.1 The Frame Problem

When an agent acts, most of the world *does not change*. A robot that picks up a box in a warehouse leaves hundreds of other boxes exactly where they were. Yet in a logical representation, the planner must somehow "know" that all those other facts persist. The **frame problem** is the challenge of efficiently representing and inferring what *stays unchanged* after an action.

The difficulty is not merely philosophical. In a warehouse with 500 portable storage units (PSUs) and 50 robots, moving one PSU changes only a handful of facts; everything else—all other PSU locations, all other robot states—remains the same. A naive representation would require explicit "nothing else changed" axioms for every invariant fact, leading to an explosion in the number of axioms. Different formalisms address this problem differently, as we will see.

Key Idea

Frame problem: How do we represent, without enumerating every non-change, that most of the world stays the same after an action?

Discriminator: it is about the things that ***do NOT change****.*

4.2.2 The Qualification Problem

Real-world actions rarely succeed unconditionally. A robot's motion might fail because the battery is depleted, because a second robot is blocking the path, because a wheel has a mechanical fault, because the warehouse management system has issued a freeze command, or because of a dozen other reasons that could in principle be enumerated but can never be exhaustively listed. The **qualification problem** is the observation that the list of preconditions for an action to succeed is potentially infinite—any attempt to enumerate all preconditions will be incomplete.

In the blocksworld, consider moving block A from on top of block B to the table. A minimal precondition is that A is clear (nothing on top of it). But we might also want to add that A is not glued down, that the table is not full, that the agent has a gripper, that the gripper is not broken, and so on indefinitely.

Key Idea

Qualification problem: We can never enumerate *all* preconditions for an action to succeed; the list is potentially infinite.

*Discriminator: it is about the things that must be **true before** the action—and there are too many of them.*

4.2.3 The Ramification Problem

Actions often have *indirect* or *implicit* effects beyond those directly described. If a robot picks up a PSU, it not only moves the PSU—it also moves every item stored inside the PSU. A formal description of the pick action that only mentions the PSU's location will miss these downstream consequences. The **ramification problem** is the problem of specifying all the indirect, implicit side-effects of an action.

Key Idea

Ramification problem: Actions can have unintended or implicit side-effects that are not directly described in the action's definition.

*Discriminator: it is about the things that **change indirectly** as a consequence of the action.*

4.2.4 Side-by-Side Comparison

The table below gives a structured comparison of all three problems. Memorise one concrete example for each; in an exam, the three are the most commonly confused concepts in this chapter.

Problem	What it is	Example (warehouse)
Frame	How to represent that *unchanged* facts persist across an action without listing every one explicitly.	Moving PSU-17 leaves PSU-1 through PSU-16 and PSU-18 onwards in exactly the same locations.
Ramification	How to represent *indirect / implicit effects*—consequences that follow from the action but are not directly stated.	Moving PSU-17 also moves every item stored inside PSU-17, even though the action only mentions the PSU.
Qualification	The impossibility of listing *all preconditions* for an action to succeed.	The robot may fail to move PSU-17 because the battery is dead, the path is blocked, a motor has failed, the system issued a freeze, ... (list never ends).

(!) Exam Trap

Do **not** confuse the three problems. A one-line discriminator for each:

- **Frame** = what *doesn't change* (persistence).
- **Ramification** = what *changes indirectly* (implicit effects).
- **Qualification** = what must be *true beforehand* (infinite prerequisites).

Frame and ramification are both about effects, but in opposite directions: frame is about non-effects, ramification is about uncaptured effects.

4.3 Deductive Planning: The Situation Calculus

The oldest systematic approach to AI planning is **deductive planning**, exemplified by the *Situation Calculus* introduced by McCarthy (1963) and used for automated plan construction by Green (1969) through logical resolution.

4.3.1 Situations and Fluents

The Situation Calculus operates within first-order logic. Its central innovation is to make time—or more precisely, the history of actions taken—an explicit

argument in every time-varying predicate. A **fluent** is a predicate that can change from situation to situation; it carries an extra argument representing the current *situation*.

> In the blocksworld, the fact that block A is on block B in situation S_1 is written $on(A, B, S_1)$, and the fact that block C is on the table is $onTable(C, S_1)$.

Situations are terms. They are either *atomic* constants (like S_1, a named starting point) or *complex* situations formed by applying an action operator to an existing situation. For example, $put_on_table(A, S_1)$ denotes the situation that results from putting block A on the table starting from S_1. Crucially, the action name itself becomes a term—there is no separate "state" object; the situation *encodes its entire history*.

4.3.2 Effect Axioms

Changes are described by **effect axioms**: logical rules that specify which fluents become true in a new situation. For the $put(X, Y, S)$ operator (place block X on block Y in situation S):

$$on(X, Y, put(X, Y, S)) \leftarrow clear(X, S) \wedge clear(Y, S)$$
$$clear(Z, put(X, Y, S)) \leftarrow on(X, Z, S) \wedge clear(X, S) \wedge clear(Y, S)$$

The first axiom says: after putting X on Y, it will be the case that X is on Y—provided X and Y were both clear beforehand. The second says: Z will become clear after placing X on Y, if X was on Z before (so Z is now uncovered).

4.3.3 Frame Axioms

Effect axioms only state what *becomes* true. Situation Calculus, unlike STRIPS (see below), does **not** adopt the closed-world assumption. Therefore, without additional axioms, a theorem prover cannot infer that facts which are not mentioned in an effect axiom remain unchanged. The Situation Calculus addresses the frame problem by requiring **frame axioms**—explicit logical statements that certain fluents persist across certain actions. For $put(X, Y, S)$:

$$on(Y, Z, put(X, Y, S)) \leftarrow on(Y, Z, S)$$

This says: after putting X on Y, Y is still on Z if it was before. A full treatment requires one such axiom for every (fluent, action) pair where the action does not affect the fluent, resulting in $O(\text{actions} \times \text{fluents})$ frame axioms—a significant cost.

4.3.4 Planning as Theorem Proving

A planning problem in the Situation Calculus is solved in three steps:

1. Express the initial state as a set of ground fluent facts (logical axioms).
2. Express the goal as a theorem: find a situation term G such that certain fluents hold in G.
3. Run a theorem prover (e.g., resolution) to derive the sequence of action terms embedded in G.

The situation term that satisfies the goal literally encodes the plan:

$$put(B,\ A,\ put_on_table(A,\ S_1))$$

reveals that you first put A on the table, then put B on A.

4.3.5 Strengths and Weaknesses

The Situation Calculus offers full first-order expressivity: the "deductive reasoning ability comes for free" from the logical framework. It can in principle handle the qualification problem (by adding more precondition conjuncts to effect axioms), the ramification problem (by adding additional axioms that propagate indirect effects), and the frame problem (via frame axioms). In practice, however, guaranteeing completeness for all three is extremely difficult, and theorem proving is computationally expensive for large domains.

4.4 State-Based Planning: STRIPS

The second major formalism is **STRIPS** (Stanford Research Institute Problem Solver), developed by Fikes and Nilsson in 1971. STRIPS is the foundation of all modern classical planners and the language from which PDDL descended.

4.4.1 States as Sets of Atoms

Where the Situation Calculus embeds time into predicate arguments, STRIPS takes a cleaner state-space view. A **state** is simply a *set of ground atoms*—positive facts that are true right now. Everything not in the set is assumed false (**closed-world assumption**, CWA). A goal G is also a set of ground atoms; a state s satisfies the goal if and only if $G \subseteq s$.

For the blocksworld with blocks A, B, C, D:

$$s = \{onTable(A), onTable(C), onTable(D), clear(A), clear(B), clear(D), on(B, C)\}$$

4.4.2 Operator Schemas: PRE / ADD / DEL

Actions are described by **operator schemas** with three lists:

- **PRE** (preconditions): the set of atoms that must be true in the current state for the action to be applicable.
- **ADD** (add list): the set of atoms that become true after the action executes.
- **DEL** (delete list): the set of atoms that become false after the action executes.

Schemas contain *variables* (written in upper case) that act as placeholders for domain objects. Applying an operator requires substituting concrete objects for the variables (instantiation) and checking that the resulting PRE is a subset of the current state.

The state transition is:

$$s' = (s \setminus DEL(o)) \cup ADD(o)$$

(!) Exam Trap

The **DEL list contains positive atoms** that will be *removed*—not negative literals. You should *never* write $\neg clear(B)$ in the DEL list; you write $clear(B)$.

4.4.3 STRIPS and the Three Classic Problems

- **Frame problem**: STRIPS solves this elegantly by convention. Anything not mentioned in DEL remains in the state unchanged. There are no frame axioms to write. This "implicit persistence" is the defining advantage of STRIPS over Situation Calculus.

- **Qualification problem**: Handled by the PRE list—any precondition the designer deems important can be listed. Of course, no finite PRE list can capture all real-world qualifications, so the problem is only partially addressed.
- **Ramification problem**: Indirect effects must be explicitly included in ADD and DEL. STRIPS does not propagate implicit consequences automatically; the designer must anticipate and encode them.

4.5 Worked Example: Blocksworld Operators

Worked Example

Domain: Blocks A, B, C, D on a table. Predicates: $on(X, Y)$, $onTable(X)$, $clear(X)$.

Operator 1: Place block X (from the table) onto block Y.

put(X, Y)	
PRE:	$\{onTable(X),\ clear(X),\ clear(Y)\}$
ADD:	$\{on(X, Y)\}$
DEL:	$\{onTable(X),\ clear(Y)\}$

Interpretation: X must be on the table and clear, and Y must be clear. After the move, X is on Y; X is no longer on the table; Y is no longer clear (it now supports X).

Operator 2: Move block X (currently on block Z) onto block Y.

stack(X, Y)	(X on top of some Z)
PRE:	$\{on(X, Z),\ clear(X),\ clear(Y)\}$
ADD:	$\{on(X, Y),\ clear(Z)\}$
DEL:	$\{on(X, Z),\ clear(Y)\}$

Interpretation: After unstacking X from Z and placing it on Y: X is now on Y, Z is now clear; the old $on(X, Z)$ and $clear(Y)$ are removed.

Operator 3: Put block X (from on top of Y) onto the table.

put_on_table(X)	
PRE:	$\{clear(X),\ on(X,Y)\}$
ADD:	$\{onTable(X),\ clear(Y)\}$
DEL:	$\{on(X,Y)\}$

Applying the operators to a concrete problem:

Initial state s_0: $\{on(D,C),\ on(C,A),\ clear(D),\ clear(B),\ onTable(A),\ onTable(B)\}$

Goal G: $\{on(A,B),\ on(B,C)\}$

A plan:

1. $put_on_table(D)$: remove D from C; C becomes clear.
2. $put_on_table(C)$: remove C from A; A becomes clear.
3. $put(B,C)$: place B onto C (both clear); B no longer on table.
4. $put(A,B)$: place A onto B (both clear); A no longer on table.

After step 4, the state contains $on(A,B)$ and $on(B,C)$, so $G \subseteq s_4$. □

Frame problem illustration: Note that we never had to state "D is still on the table after step 2" or "B is still clear after step 1." STRIPS's implicit-persistence convention handles this automatically through the DEL-only state update rule.

4.6 Planning as Search

STRIPS planning reduces naturally to classical search. The state space is the graph whose nodes are sets of atoms and whose edges are operator applications. From any state s, the applicable operators are those whose PRE is a subset of s; applying one yields a successor state via the $(s \setminus DEL) \cup ADD$ update. A plan is a path from the initial state to a goal state.

4.6.1 Forward Search (Progression)

The straightforward approach is **forward search** (also called *progression*): start from s_0 and expand states by applying applicable operators, in the manner of standard BFS or DFS. Uninformed variants are typically depth-first (for memory reasons). Informed variants use A* with domain-specific heuristics.

The difficulty with forward search is *relevance*: many operators may be applicable in any given state, most of which make no progress toward the goal. Good heuristics are therefore critical.

4.6.2 Backward Search (Regression)

An alternative is **backward search** (also called *relevant-state search* or *regression*): start from the goal description and work backwards. At each step, we ask: "which operator, applied just before the current state, could have produced the goal conditions?" We find operators whose ADD list intersects the current goal and whose DEL list does not contradict it; we then "un-apply" the operator by removing the ADD atoms and adding the PRE atoms to the goal set.

Backward search has the advantage of only considering *relevant* actions—those that actually contribute to the goal. This can dramatically prune the search space in domains where many actions are applicable but few are useful.

4.6.3 Planning Heuristics

Both forward and backward search benefit from informative heuristics. A heuristic $h(s)$ estimates the cost (number of actions) from state s to a goal. Two standard relaxation-based heuristics from the lecture:

- **Ignore-PRE-DEL heuristic**: Remove both the PRE and DEL lists from all operators. Each operator now freely satisfies whichever goal atoms appear in its ADD list. Finding the minimum number of operators to cover all goal atoms is then a *set-cover* problem (NP-hard), but a greedy approximation runs in polynomial time. The resulting heuristic is **not admissible** (it may overestimate), so A* with this heuristic does not guarantee optimality.
- **Ignore-DEL heuristic**: Remove only the DEL lists. No operator can undo a previously achieved goal; once a goal atom is added, it stays. The relaxed problem is still NP-hard in general, but again a polynomial-time approximation exists. Also **not admissible**.

(!) Exam Trap

Both standard planning heuristics—Ignore-PRE-DEL and Ignore-DEL—are **not admissible**. They can overestimate the true cost. Using them with A* loses the guarantee of finding an optimal plan. This is a common

exam error.

4.7 PDDL: The Modern Extension of STRIPS

The **Planning Domain Definition Language** (PDDL) was developed starting in 1998 to provide a standardised input language for the International Planning Competition (IPC). It extends STRIPS with several features that make it suitable for richer real-world domains.

PDDL syntax uses a LISP-like prefix notation. A PDDL model consists of two files: a *domain* definition (types, predicates, actions) and a *problem* definition (objects, initial state, goal). A minimal blocksworld domain looks like this:

```
(define (domain blocksworld)
  (:requirements :strips)
  (:action move
    :parameters (?b ?t1 ?t2)
    :precondition (and (block ?b) (table ?t1) (table ?t2)
                       (on ?b ?t1) (not (on ?b ?t2)))
    :effect (and (on ?b ?t2) (not (on ?b ?t1)))))
```

The `:effect` clause combines the ADD list (positive literals) and DEL list (negated literals) in a single expression, which is more readable than the separate lists of STRIPS.

PDDL adds several capabilities beyond basic STRIPS:

- **Typing**: parameters can be annotated with types (e.g., `?p - player`), reducing the grounding search space.
- **Numeric fluents**: predicates can carry numeric values, e.g., $\mathit{Contains}(P, I, 100)$ means PSU P contains 100 units of item I.
- **Derived predicates**: facts that are implied by other facts can be stated as rules, e.g., $\mathit{ItemAt}(I, x, y) \leftarrow \mathit{Contains}(P, I, N) \wedge N > 0 \wedge \mathit{PSUAt}(P, x, y)$.

(!) Exam Trap

Only `:strips` and `:typing` (plus numeric fluents and derived predicates) are explicitly covered in the lecture slides. Features such as action costs, durative actions, conditional effects, and `:adl` are **not** lecture material.

Do not write about them in an exam unless specifically asked.

4.7.1 Situation Calculus vs. STRIPS: A Direct Comparison

Dimension	**Situation Calculus**	**STRIPS / PDDL**
State representation	Sets of fluents (FOL atoms with situation argument)	Set of ground atoms (no time argument)
Frame problem	Explicit frame axioms required; $O(\text{actions} \times \text{fluents})$	Implicit: only DEL changes the state
Closed-world assumption	**No**	**Yes**
Planning algorithm	Theorem prover (resolution)	Classical state-space search
Expressivity	Full FOL	Restricted (subset of FOL)
Computational cost	Typically high	More tractable in practice

The most important row is the CWA. In Situation Calculus, a fact not derivable from the axioms is simply *unknown*—it might be true or false. In STRIPS, a ground atom absent from the state is definitively *false*. This is what allows STRIPS to avoid frame axioms: the DEL list is the only mechanism that removes atoms, so the planner knows without any additional axiom that everything not deleted persists.

4.8 Summary

In a Nutshell

Classical Planning deals with finding action sequences that transform an initial state into a goal state in fully observable, deterministic, static environments. Unlike search, planning exploits the *internal structure* of states.

Three classic challenges — remember them by their distinct signatures:

- **Frame**: what *doesn't* change.
- **Ramification**: what changes *indirectly*.
- **Qualification**: preconditions you can never fully enumerate.

Situation Calculus (McCarthy 1963): deductive planning using FOL fluents. Actions described by effect axioms; frame problem handled by explicit frame axioms. No CWA. Plans found by theorem proving. Pro: full logical expressivity. Con: expensive, frame axiom proliferation.

STRIPS (Fikes & Nilsson 1971): state-based planning. States are sets of ground atoms (CWA). Operators have PRE / ADD / DEL lists. Frame problem solved implicitly: only DEL changes anything. Planning reduces to classical search. State update: $s' = (s \setminus DEL) \cup ADD$.

PDDL: modern STRIPS extension with typing, numeric fluents, and derived predicates. LISP-like syntax; separates domain and problem files.

Forward search (progression): start from s_0, apply operators forward. **Backward search** (regression): start from goal, un-apply operators. Both benefit from heuristics; Ignore-PRE-DEL and Ignore-DEL heuristics are *not* admissible.

Chapter 5

Probabilistic Planning: Markov Decision Processes

Classical planning, as introduced in the previous chapter, rests on a comfortable assumption: actions are deterministic. When a robot arm executes *pick up the cup*, the cup is picked up — every time, without exception. This idealization allows us to represent plans as simple action sequences and to reason about them with logical certainty. Real environments, however, are stubborn in refusing to behave this cleanly. Sensors are noisy, actuators slip, traffic jams appear without warning, and supply chains fail in unpredictable ways. A planning framework that cannot accommodate this uncertainty is fragile by design.

Markov Decision Processes (MDPs) were developed precisely to fill this gap. They extend classical planning in two fundamental directions: they model *stochastic action outcomes* using probability distributions, and they account for *long-term consequences* by evaluating policies not just on their immediate reward but on the discounted sum of rewards they accumulate over time. The result is a powerful and mathematically principled framework that underpins much of modern AI, from robot navigation to game-playing agents.

5.1 Why Probabilistic Planning?

Two recurring phenomena in real planning problems motivate the move to probabilistic models.

Stochastic action outcomes. When a mobile robot attempts to move one cell to the left, it usually succeeds — but not always. A worn motor, an uneven floor, or a sensor delay may cause it to drift sideways instead. When a delivery service dispatches a package, it usually arrives within two days — but labour strikes and logistical errors occasionally delay it. These are not edge cases to be handled by exception; they are the normal operating conditions of most interesting AI systems. A planning framework must therefore represent the *distribution* over possible next states that an action produces, not merely a single guaranteed successor.

Long-term consequences. Even when individual actions are deterministic, optimizing for immediate reward can lead to poor long-term behaviour. Skipping a robot's battery recharge saves a few seconds now but risks a shutdown mid-task later. Canceling insurance frees up budget today but exposes one to catastrophic loss tomorrow. An intelligent agent must reason about the cumulative effect of its decisions over an extended horizon, weighing immediate gains against future costs. MDPs provide exactly the tools needed to formalize and solve this kind of temporally extended reasoning.

5.2 The Markov Property

The name "Markov Decision Process" honours the Russian mathematician Andrey Markov (1856–1922), whose work on stochastic processes established the key independence assumption that makes these models tractable.

Key Idea

The Markov Property. The future is conditionally independent of the past, given the present. Formally, the probability of transitioning to state s' at time $t+1$ depends only on the current state s_t and the action a_t taken, not on any earlier states or actions:

$$\begin{aligned} &P(S_{t+1} = s' \mid S_t = s_t,\, A_t = a_t,\, S_{t-1} = s_{t-1},\, \ldots,\, S_0 = s_0) \\ &\quad = P(S_{t+1} = s' \mid S_t = s_t,\, A_t = a_t). \end{aligned}$$

This is not a restriction on how rich the state representation may be. Rather, it is a statement about what the state must encode: *everything relevant for*

predicting the future. If the robot's battery level matters for what happens next, then battery level must be part of the state. Once the state is defined to be sufficiently comprehensive, the Markov property holds by construction. Readers familiar with state-based search will recognize this as exactly the assumption that makes successor functions well-defined — MDPs generalize it to the probabilistic setting.

5.3 Formal Definition of an MDP

Key Idea

A **Markov Decision Process** is a tuple (S, A, p, r), where:

- S is a (finite) set of **states**;
- A is a (finite) set of **actions**;
- $p(s' \mid s, a)$ is the **transition probability function**: the probability of reaching state s' when action a is executed in state s, satisfying $\sum_{s' \in S} p(s' \mid s, a) = 1$ for all $s \in S$, $a \in A$;
- $r(s, a)$ is the **reward function**: the immediate scalar reward received for executing action a in state s.

Each component earns its place: S and A define the combinatorial structure of the problem; p encodes all uncertainty about how the world changes; r encodes the designer's preferences — what counts as good or bad.

Note that the reward function is sometimes written as $r(s, a, s')$ to make the dependence on the resulting state explicit. In the Grid World example discussed below, the reward for entering a goal state is assigned by convention to all actions taken *from* the goal state, but the substance is the same: reward quantifies the desirability of state-action pairs.

(!) Exam Trap

The discount factor γ is NOT part of the MDP tuple. The MDP is (S, A, p, r) — four components, no γ. The discount factor belongs to the *solution objective*: it specifies how future rewards are weighted when evaluating a policy. The same MDP can be solved under different values of γ, yielding different optimal policies. Conflating γ with the MDP itself is one of the most common errors on exams. When asked "define an

MDP," write (S, A, p, r) and explain γ separately as a parameter of the optimization criterion.

5.3.1 The Grid World Example

To make the abstract definition concrete, consider the canonical Grid World: a 4×3 grid in which an agent begins at position $(1, 1)$ and must navigate to one of two terminal states — a positive goal at $(4, 3)$ and a negative goal (trap) at $(4, 2)$.

- **States:** $S = \{(1, 1), (2, 1), \ldots, (4, 3)\}$, eleven non-terminal cells plus the two terminal states (twelve total, with one obstacle cell blocked).
- **Actions:** $A = \{\text{up}, \text{down}, \text{left}, \text{right}\}$.
- **Transition probabilities:** Actions are unreliable. When the agent attempts to move in direction d, it moves as intended with probability 0.8, drifts 90ř clockwise with probability 0.1, and drifts 90ř counter-clockwise with probability 0.1. If the intended move would leave the grid or hit an obstacle, the agent stays in place; that probability mass is added to the self-transition. For example:

  $$p\big((1, 2) \mid (1, 1), \text{ up}\big) = 0.8, \quad p\big((2, 1) \mid (1, 1), \text{ up}\big) = 0.1, \quad p\big((1, 1) \mid (1, 1), \text{ up}\big) = 0.1.$$

- **Rewards:** $r((4, 3), a) = +1$ (positive goal); $r((4, 2), a) = -1$ (negative goal); $r(s, a) = -0.04$ for all other states (a small step penalty encouraging efficiency).

The step penalty is crucial: without it, the agent might wander indefinitely. With it, the agent is nudged to finish the task, but not so severely that it would risk the trap to save a step. Changing the step penalty to -2 shifts the optimal policy dramatically: now every step is so expensive that the agent rushes toward the nearest terminal state, even at the cost of occasionally entering the trap. At -0.4, an intermediate regime appears. This sensitivity illustrates that *reward design is a consequential modelling choice*, not a technical afterthought.

5.4 Policies and the Objective

5.4.1 Policies

A **policy** π specifies the agent's behaviour: for every state, it prescribes which action to take. A *deterministic* policy is a function $\pi : S \to A$. A *stochastic* policy

generalizes this to a conditional distribution $\pi(a \mid s)$, giving the probability of taking each action in each state.

It is important to appreciate that a policy covers *every* state in S, not just the states reachable from a particular start state. This distinguishes it fundamentally from a classical plan, which is a sequence of actions anchored to a single initial state. A policy is more like a standing instruction set: "wherever you find yourself, here is what to do." This completeness is necessary precisely because stochastic transitions mean the agent may end up in unexpected states and must still know how to act.

5.4.2 Discounted Cumulative Reward

Following policy π from state s generates a (possibly infinite) sequence of rewards $r_1, r_2, r_3, \ldots$ The naive sum $\sum_k r_k$ may diverge if the MDP has no terminal states. Two further motivations — the economic observation that a reward now is worth more than the same reward later, and the mathematical need for convergence — lead to **discounting**.

The γ-**discounted return** from time step 1 onward is:

$$G = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \cdots = \sum_{k=1}^{\infty} \gamma^{k-1} r_k,$$

where $\gamma \in [0, 1]$ is the **discount factor**. When $\gamma = 0$, the agent is myopic and cares only about the immediate reward. As $\gamma \to 1$, the agent becomes increasingly far-sighted. For $\gamma < 1$ and bounded rewards, the series converges absolutely. The case $\gamma = 1$ is mathematically admissible only when the MDP is guaranteed to terminate.

For $\gamma = 0.5$, the discounted weights of the first four steps are $1,\ 0.5,\ 0.25,\ 0.125$ — each step into the future is worth half the previous one. This geometric decay is both mathematically convenient and intuitively plausible for many real-world domains.

5.5 The Value Function and Bellman Equations

5.5.1 State-Value Function

The **state-value function** $V^\pi : S \to \mathbb{R}$ assigns to each state s the expected discounted return when the agent starts in s and follows policy π thereafter:

$$V^\pi(s) := \mathbb{E}_\pi\Big[r_1 + \gamma r_2 + \gamma^2 r_3 + \cdots \mid S_0 = s\Big].$$

Here $\mathbb{E}_\pi[\cdot]$ denotes the expectation over the distribution of trajectories induced by the stochastic transition function p and the policy π. The value function compresses an infinite-horizon prospect into a single number, giving a meaningful ranking of states under a given policy.

(!) Exam Trap

V^π versus V^*: These are fundamentally different objects. $V^\pi(s)$ is the value of state s *when a specific, possibly suboptimal policy π is followed.* $V^*(s)$ is the value of state s under the *best possible policy.* We have $V^*(s) \geq V^\pi(s)$ for all s and all π, with equality when π is itself optimal. Computing V^π is *policy evaluation*; finding V^* is the *control problem.*

5.5.2 The Bellman Expectation Equation

The value function satisfies a recursive self-consistency condition known as the **Bellman expectation equation**. Intuitively, the value of being in state s under π equals the expected immediate reward plus the discounted value of the next state:

$$V^\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a)\Big[r(s, a, s') + \gamma\, V^\pi(s')\Big]. \tag{5.1}$$

Let us unpack every symbol:

- $\pi(a \mid s)$: probability of choosing action a in state s under policy π (for a deterministic policy, this is 1 for $a = \pi(s)$ and 0 otherwise);
- $p(s' \mid s, a)$: transition probability of landing in s';
- $r(s, a, s')$: immediate reward received on the transition from s to s' via a (often abbreviated $r(s, a)$ when the dependence on s' is suppressed);
- γ: discount factor;
- $V^\pi(s')$: value of the successor state under the same policy π.

The equation says: average over all actions the policy might take, and for each action average over all possible next states, collecting the immediate reward plus the discounted future value. Because the same V^π appears on both sides, equation (5.1) is a linear system in the unknowns $\{V^\pi(s)\}_{s \in S}$ — it has a unique solution that can be computed by iterative methods or, for small state spaces, by direct matrix inversion.

5.5.3 The Bellman Optimality Equation

Rather than evaluating a fixed policy, we often want the *optimal* value function V^*, defined as the highest achievable value in each state:

$$V^*(s) := \max_\pi V^\pi(s).$$

V^* satisfies the **Bellman optimality equation**:

$$V^*(s) = \max_{a \in A} \sum_{s'} p(s' \mid s, a) \Big[r(s, a, s') + \gamma \, V^*(s') \Big]. \tag{5.2}$$

The structure is identical to equation (5.1) except that the weighted average over the policy's action distribution is replaced by a maximization over actions. This reflects the insight that an optimal agent should, in every state, take the action that maximizes the sum of immediate reward and discounted future value.

Unlike the expectation equation, the optimality equation (5.2) is *non-linear* (due to the max), so it cannot be solved by matrix inversion. Iterative algorithms are required.

Given V^*, extracting an optimal policy is straightforward: for each state, take the action that achieves the maximum in (5.2):

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s'} p(s' \mid s, a) \Big[r(s, a, s') + \gamma \, V^*(s') \Big].$$

5.6 Algorithms: Value Iteration and Policy Iteration

Two main algorithms compute the optimal value function (and thereby the optimal policy) for a finite MDP. Both are guaranteed to converge under mild assumptions and run in time polynomial in $|S|$ and $|A|$.

5.6.1 Value Iteration

Value Iteration (VI) directly computes V^* by repeatedly applying the Bellman optimality equation as an update rule. Starting from an arbitrary initialization — typically $V_0(s) = 0$ for all s — each iteration performs one **Bellman backup** for every state:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \sum_{s'} p(s' \mid s, a) \Big[r(s, a, s') + \gamma \, V_k(s') \Big], \quad \forall s \in S.$$

The iteration continues until the values stabilize: $\max_s |V_{k+1}(s) - V_k(s)| < \varepsilon$ for some small tolerance $\varepsilon > 0$.

Key Idea

Value Iteration converges to V^* for any $\gamma < 1$. Each sweep over all states costs $O(|S|^2 \cdot |A|)$ in the worst case. Convergence is guaranteed but may require many iterations; the contraction factor γ controls the convergence rate.

5.6.2 Policy Iteration

Policy Iteration (PI) maintains an explicit policy π and alternates between two phases:

1. **Policy Evaluation.** Given the current policy π, compute V^π exactly (or approximately) by solving the Bellman expectation equation (5.1) iteratively until convergence. Starting from $V_0(s) = 0$, the update is:

   $$V_{k+1}(s) \leftarrow \sum_{s'} p(s' \mid s, \pi(s)) \Big[r(s, \pi(s), s') + \gamma \, V_k(s') \Big].$$

   This inner loop runs until V^π has converged (or for a fixed number of steps in *Modified Policy Iteration*).
2. **Policy Improvement.** Update the policy greedily with respect to the current value estimate:

   $$\pi'(s) \leftarrow \arg\max_{a \in A} \sum_{s'} p(s' \mid s, a) \Big[r(s, a, s') + \gamma \, V^\pi(s') \Big].$$

These two steps repeat until the policy no longer changes ($\pi' = \pi$). At that point, by the **Policy Improvement Theorem**, the policy is optimal. Because

the number of deterministic policies is finite ($|A|^{|S|}$), and each iteration strictly improves the policy or leaves it unchanged, PI must terminate in a finite number of outer iterations — typically far fewer than VI requires to achieve the same accuracy.

Key Idea

The Policy Improvement Theorem guarantees: if π' is the greedy policy with respect to V^π, then $V^{\pi'}(s) \geq V^\pi(s)$ for all s, with equality only when π is already optimal (i.e., satisfies the Bellman optimality equation).

A practical variant, **Modified Policy Iteration**, performs only a fixed number of evaluation sweeps before the improvement step. This trades evaluation accuracy for faster outer iterations, often converging faster in practice. It also sidesteps the divergence issue that arises when $\gamma = 1$ in non-terminating MDPs.

5.6.3 Comparison Table

Aspect	**Value Iteration**	**Policy Iteration**
Core update	Bellman optimality backup	Policy eval + greedy impr
Per-iteration cost	Cheap (one sweep)	Heavy (full inner eval loo
Number of iterations	Many (geometric convergence)	Few (policy space is finite
What is stored	Value function V_k	Policy π and value V^π
Stopping criterion	$\|V_{k+1} - V_k\|_\infty < \varepsilon$	π unchanged
Convergence guarantee	Yes (for $\gamma < 1$)	Yes (for $\gamma \leq 1$ under term
Optimal for	Large state spaces, coarse tolerance	Small-to-medium problem

In practice, if computing V^π is cheap (e.g., via matrix inversion for small S), Policy Iteration often wins due to its very small number of outer iterations. For large state spaces where even a single full evaluation sweep is expensive, Value Iteration's simpler per-step cost is advantageous. Both algorithms are polynomial in $|S|$ and $|A|$, a striking contrast to the exponential worst cases of classical planning.

5.7 Worked Example: A Bellman Backup

Worked Example

Setup. Consider a minimal MDP with three states: $S = \{A, B, \text{Goal}\}$. State `Goal` is absorbing (terminal) with $V^*(\texttt{Goal}) = 0$. From state A and state B, only one action (*move*) is available, with the following transition probabilities and rewards:

From	Action	To	$p(s' \mid s, a)$	$r(s, a, s')$
A	move	B	0.7	-1
A	move	Goal	0.3	$+10$
B	move	Goal	1.0	$+10$

Set $\gamma = 0.9$.

Iteration 0. Initialize $V_0(A) = 0$, $V_0(B) = 0$, $V_0(\texttt{Goal}) = 0$.

Iteration 1 — update $V_1(B)$:

$$V_1(B) = \max_a \sum_{s'} p(s' \mid B, a)\big[r(B, a, s') + 0.9\, V_0(s')\big].$$

With only one action:

$$V_1(B) = 1.0 \times (10 + 0.9 \times 0) = 10.$$

Iteration 1 — update $V_1(A)$:

$$V_1(A) = 0.7 \times (-1 + 0.9 \times V_0(B)) + 0.3 \times (10 + 0.9 \times V_0(\text{Goal})).$$

$$= 0.7 \times (-1 + 0) + 0.3 \times (10 + 0) = -0.7 + 3 = 2.3.$$

Iteration 2 — update using V_1:

$$V_2(B) = 1.0 \times (10 + 0.9 \times 0) = 10 \quad \text{(unchanged, Goal is absorbing)}.$$

$$V_2(A) = 0.7 \times (-1 + 0.9 \times 10) + 0.3 \times (10 + 0).$$

$$= 0.7 \times (-1 + 9) + 3 = 0.7 \times 8 + 3 = 5.6 + 3 = 8.6.$$

After just two iterations, the value of A has jumped from 0 to 8.6, reflecting the propagation of the high reward at `Goal` backward through state B. The algorithm will continue until values stabilize.

Takeaway. Each Bellman backup takes the current value estimates of successor states and uses them to compute an improved estimate for the current state. Information propagates from high-reward states outward with each iteration, one step at a time.

5.8 Exam Traps and Common Pitfalls

(!) Exam Trap

Trap 1: γ in the MDP tuple. The MDP is defined as (S, A, p, r) — four elements. If an exam question asks you to define an MDP formally, do not include γ. The discount factor parametrizes the *objective function*, not the environment model. Different values of γ on the same MDP can yield qualitatively different optimal policies.

(!) Exam Trap

Trap 2: Confusing V^π and V^*. $V^\pi(s)$ is the expected discounted return when starting in s and following the *specific* policy π — which may be far from optimal. $V^*(s)$ is the maximum achievable value in s over *all* policies. Policy evaluation computes V^π; value iteration and policy iteration both converge to V^*. Writing $V^\pi = V^*$ without qualification is incorrect.

(!) Exam Trap

Trap 3: $\gamma = 1$ and non-termination. The lectures note explicitly (Slide 53 and Slide 67) that for $\gamma = 1$, values may be unbounded in non-terminating MDPs. Policy evaluation may not converge, and even policy iteration may cycle. The safe default is $\gamma < 1$ for infinite-horizon problems. $\gamma = 1$ is valid only when termination is guaranteed (episodic tasks).

(!) Exam Trap

Trap 4: Policy vs. plan. A policy $\pi : S \to A$ specifies an action for every state, including states never visited during a run. A classical plan

is a finite action sequence from a single initial state. Policies are more general: they are reactive and robust to unexpected states produced by stochastic transitions.

(!) Exam Trap

Trap 5: When to use Value Iteration vs. Policy Iteration. There is no universally superior algorithm. Value Iteration performs cheap sweeps but needs many of them. Policy Iteration performs expensive evaluation sub-loops but converges in very few outer iterations. For large $|S|$ where full evaluation is prohibitive, Value Iteration (or Modified Policy Iteration with a small number of inner steps) is preferred. For small-to-medium problems requiring an exact optimal policy, Policy Iteration is often faster.

In a Nutshell

Chapter Summary.
Markov Decision Processes extend classical planning to stochastic environments. An MDP is the tuple (S, A, p, r): states, actions, transition probabilities, and rewards. The discount factor $\gamma \in [0, 1]$ is a *separate* parameter of the optimization objective, not an MDP component.
The Markov property states that action outcomes depend only on the current state, not the history. A policy $\pi : S \to A$ maps every state to an action. The state-value function $V^\pi(s)$ measures the expected discounted return under π; the optimal value function $V^*(s)$ is the best achievable value in s.
The **Bellman expectation equation** characterizes V^π as a linear fixed point; the **Bellman optimality equation** characterizes V^* as a non-linear fixed point involving a max over actions.
Value Iteration applies the Bellman optimality backup repeatedly until convergence ($\|V_{k+1} - V_k\|_\infty < \varepsilon$). **Policy Iteration** alternates between full policy evaluation and greedy policy improvement, converging in fewer (but costlier) outer iterations. Both are polynomial in $|S|$ and $|A|$ and guaranteed to find the optimal policy for $\gamma < 1$.
Key exam traps: (1) $\gamma \notin (S, A, p, r)$; (2) $V^\pi \neq V^*$ unless π is optimal; (3) $\gamma < 1$ required for convergence in non-terminating MDPs; (4) a policy

covers all states, a plan covers only a trajectory.

Chapter 6

Knowledge Representation I: Ontologies and Description Logics

6.1 Why Represent Knowledge Explicitly?

Intelligent behaviour presupposes knowledge. A chess-playing program needs to know the rules of the game; a medical diagnosis system needs to know which symptoms are associated with which diseases; a robot navigating a hospital corridor needs to know the layout of the building and the categories of objects it might encounter. The question is therefore not *whether* to represent knowledge but *how* — and the choice of representation has profound consequences for what can be inferred, how efficiently, and how transparently.

The field of **Knowledge Representation (KR)** is the branch of AI dedicated to encoding information in a formal, structured way so that automated reasoning becomes possible. The lecture formulation is deliberately broad: "probably every application for which you can use a computer needs a knowledge representation formalism" (Krumnack, Session 6). In symbolic approaches the representation is explicit and transparent — you can read it, debug it, and extend it. In sub-symbolic approaches such as deep neural networks the representation is distributed across weight matrices and attention heads, making it powerful but opaque.

Several KR formalisms are in use: propositional and predicate logic, Prolog, probabilistic graphical models, neural networks, and structured formats such as

semantic networks and ontologies. The central engineering tension runs through all of them:

Key Idea

Expressivity vs. Decidability. The more expressive a KR language is — the more distinctions it can draw, the more complex the sentences it can form — the harder (or even impossible) automated reasoning over it becomes. Full First-Order Logic (FOL) is highly expressive but only *semi-decidable*: a sound and complete theorem prover will always find a proof if one exists, but may run forever if none does. Good KR design is about finding the sweet spot for a given application.

6.1.1 Semantic Networks and Frames

Historically, before logic-based formalisms became dominant, AI researchers used *semantic networks* — graph structures in which nodes represent concepts or individuals and labelled edges represent relations between them. A node `Dog` might be connected to `Animal` via an `is-a` edge, and to `Fido` via a `instance-of` edge. Inheritance of properties along `is-a` chains allowed compact encodings of commonsense knowledge.

Frames (Minsky, 1975) extended this idea by packaging a concept together with its typical attributes ("slots") and default values. A `Car` frame might have slots for `numberOfWheels` (default: 4) and `fuelType` (default: gasoline). Both formalisms are intuitive and easy to visualise, but they lack a precise formal semantics: the meaning of an edge or a slot is defined only by convention, which makes it difficult to give a rigorous account of what follows from a given network. Description Logics, introduced below, can be seen as a direct response to this shortcoming — they preserve the graph-based intuition while grounding it in formal logic.

6.2 Ontologies

6.2.1 What Is an Ontology?

The word "ontology" has roots in ancient philosophy (the study of what exists), but in AI it has acquired a precise technical meaning. Several definitions are in circulation:

- "A representation of the categories, properties and relations between concepts that can refer to one, many or all domains of discourse." (Wikipedia, adapted)
- "An ontology is an explicit specification of a conceptualisation." (Gruber, 1993)
- "An ontology is a shared understanding of some domain of interest." (Uschold & Gruninger, 1996)

Formally, Stumme & Maedche (2001) define an ontology as a tuple $\mathcal{O} = \langle C, is_a, R, \sigma \rangle$, where C is a set of *concepts*, $is_a \subseteq C \times C$ is a partial order (the subsumption hierarchy), R is a set of *relations*, and σ assigns each relation an arity. Three features distinguish an ontology from a mere vocabulary list: it is *explicit* (the conceptualisation is written down formally), *shared* (it represents a community consensus rather than one person's idiosyncratic view), and *formal* (a machine can process and reason over it).

6.2.2 Taxonomy versus Ontology

Students often conflate a *taxonomy* with an *ontology*. The distinction matters.

A **taxonomy** is a pure `is-a` (subsumption) hierarchy — a tree or directed acyclic graph of concepts arranged from most general (root) to most specific (leaves). The Linnaean classification of living organisms is the canonical example: *Animalia* → *Chordata* → *Mammalia* → *Canidae* → *Canis lupus familiaris*. A taxonomy answers the question "what kind of thing is this?" but nothing more.

An **ontology** goes further: it also encodes *properties* of concepts (e.g., a `Dog` has a `furColour`), *relations* between concepts (e.g., `owns`, `livesIn`), *constraints* (e.g., every `Parent` must have at least one `hasChild` relation), and *axioms* that govern how these pieces interact. An ontology can answer not only "what kind of thing is this?" but also "what can be inferred about it?"

(!) Exam Trap

Ontology ≠ Taxonomy. A taxonomy is only the subsumption skeleton of an ontology. An ontology additionally specifies roles (binary relations), constraints on their fillers, and logical axioms. Confusing the two on an exam suggests a misunderstanding of the formalism.

6.2.3 Upper Ontologies and Domain Ontologies

A *domain ontology* describes a specific area of knowledge — biology, medicine, e-commerce, linguistics. Domain ontologies are easier to build because the scope is limited, and many successful ones exist (Gene Ontology, SNOMED CT, Dublin Core).

An *upper ontology* (also called a foundational or top-level ontology) attempts to provide a general framework applicable across all domains. It focuses on highly abstract categories that appear in any domain: `Event`, `PhysicalObject`, `TimeInterval`, `Belief`, `Agent`. Details of particular object types (`Television`, `Book`) are added in domain layers below. The Russell & Norvig textbook illustrates a compact upper ontology in Chapter 10; the DOLCE and Cyc projects are real-world attempts at a comprehensive upper ontology.

Ontologies have been created through four main routes: by trained ontologists (e.g. the CYC system), by importing structure from existing databases (e.g. DBpedia from Wikipedia), by open information extraction from text (TextRunner, DeepEx), and by crowd-sourcing commonsense knowledge (Open Mind Common Sense).

6.3 Description Logics

6.3.1 Motivation: From FOL to a Decidable Fragment

First-Order Logic is a powerful but blunt instrument for ontology work. Consider the FOL definition of a teacher:

$$\forall x \big(\mathit{Teacher}(x) \;\leftrightarrow\; \mathit{Person}(x) \;\wedge\; \exists y \, (\mathit{teaches}(x, y) \wedge \mathit{Course}(y)) \big).$$

This is correct but verbose, and reasoning over arbitrary FOL is only semi-decidable. *Description Logics* (DLs) are a family of KR languages that deliberately restrict the expressiveness of FOL in order to recover decidability while remaining expressive enough for ontology engineering. The same definition in DL notation reads:

$$\mathit{Teacher} \equiv \mathit{Person} \sqcap \exists \mathit{teaches}.\mathit{Course}.$$

DLs make it easy to describe properties of *categories* (classes of objects) using three kinds of building block:

- **Concepts** (also called classes or concept names) denote sets of individuals. Examples: *Person*, *Student*, *Course*.

- **Roles** (also called properties or role names) denote binary relations between individuals. Examples: *teaches*, *hasChild*, *employedBy*.
- **Individuals** (also called individual names) are named objects in the domain. Examples: *Alice*, *John*, *CS101*.

6.3.2 The Basic Language $\mathcal{ALC}$

The most studied base language is $\mathcal{ALC}$ — the **Attributive Concept Language with Complements**. It is considered the minimal DL of practical interest. Given a set of concept names N_C and role names N_R, the set of $\mathcal{ALC}$ concept descriptions is defined inductively:

Constructor	**Syntax**	**Reading**
Atomic concept	$A \in N_C$	a named class
Top concept	$\top$	the universal class (everything)
Bottom concept	$\bot$	the empty class (nothing)
Negation	$\neg C$	complement of C
Conjunction	$C \sqcap D$	intersection (both C and D)
Disjunction	$C \sqcup D$	union (either C or D)
Value restriction	$\forall R.C$	all R-fillers are in C
Existential restriction	$\exists R.C$	some R-filler is in C

The formal semantics is given by an *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty domain and $\cdot^{\mathcal{I}}$ maps every concept name to a subset of $\Delta^{\mathcal{I}}$ and every role name to a binary relation over $\Delta^{\mathcal{I}}$. Complex concepts are interpreted inductively:

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}, \qquad \bot^{\mathcal{I}} = \emptyset, \qquad (\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}},$$

$$(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}, \qquad (C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}},$$

$$(\forall R.C)^{\mathcal{I}} = \left\{ a \in \Delta^{\mathcal{I}} \mid \forall b : (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}} \right\},$$

$$(\exists R.C)^{\mathcal{I}} = \left\{ a \in \Delta^{\mathcal{I}} \mid \exists b : (a,b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}} \right\}.$$

(!) Exam Trap

The vacuous truth of $\forall R.C$. An individual a belongs to $(\forall R.C)^{\mathcal{I}}$ automatically if a has *no* R-successors at all, because the universal quantifier is vacuously satisfied. Concretely, $\forall \mathit{hasChild}.\bot$ correctly describes a childless person: since no child exists, the condition "every child is $\bot$" holds vacuously. This surprises many students who expect the formula to denote something impossible.

6.3.3 Dialects and the Expressivity Ladder

$\mathcal{ALC}$ is the starting point of a family of languages. Extensions are named by appending letters to the base:

- $\mathcal{U}$ — concept union (already in $\mathcal{ALC}$, but sometimes counted separately)
- $\mathcal{C}$ — complex concept negation (also in $\mathcal{ALC}$)
- $\mathcal{H}$ — role hierarchy ($R \sqsubseteq S$, meaning every R-pair is also an S-pair)
- $\mathcal{O}$ — nominals (enumerated classes: $\{a\} \sqcup \{b\}$)
- $\mathcal{E}$ — full existential qualification (fillers other than $\top$; again in $\mathcal{ALC}$)
- $\mathcal{N}$ — cardinality restrictions ($\geq n\, R.C$, $\leq n\, R.C$)
- $\mathcal{Q}$ — qualified cardinality restrictions

Adding more constructors increases expressiveness but pushes up computational complexity. For $\mathcal{ALC}$, satisfiability is decidable in ExpTime. The full $\mathcal{SROIQ}$ language (the logical basis of OWL 2) is also decidable but only in N2ExpTime. Full FOL, at the other extreme, is undecidable.

6.4 TBox and ABox: The Two Pillars of a DL Knowledge Base

A DL knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of two complementary parts introduced by Brachman & Levesque (1985) in their system KL-ONE. The distinction between them is fundamental and is the single most important source of exam errors.

6.4.1 The TBox — Terminological Knowledge

The **TBox** (Terminological Box) encodes *schema-level* knowledge: general statements about classes and their relationships, analogous to a database schema. It contains two kinds of axiom.

A **General Concept Inclusion (GCI)** has the form:

$$C \sqsubseteq D$$

read "every C is a D" or "C is subsumed by D". An interpretation $\mathcal{I}$ satisfies this axiom iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Example:

$$\mathit{Flu} \sqsubseteq \mathit{Illness}$$

states that every flu case is an illness.

A **Concept Equivalence** combines two inclusions:

$$C \equiv D \quad \text{iff} \quad C \sqsubseteq D \text{ and } D \sqsubseteq C,$$

which holds iff $C^{\mathcal{I}} = D^{\mathcal{I}}$. This is used to give *definitions*:

$$\mathit{Parent} \equiv \mathit{Person} \sqcap \exists \mathit{hasChild}.\top$$

defines a Parent as precisely a Person who has at least one child.

A finite set of GCIs constitutes a TBox. Any interpretation satisfying all GCIs in $\mathcal{T}$ is called a *model of* $\mathcal{T}$. The TBox is used to constrain the space of admissible interpretations — "the schema determines what kinds of worlds are possible."

6.4.2 The ABox — Assertional Knowledge

The **ABox** (Assertion Box) encodes *instance-level* knowledge: facts about named individuals, analogous to the rows of a database table.

A **concept assertion** states that individual a is an instance of concept C:

$$a : C$$

satisfied by $\mathcal{I}$ iff $a^{\mathcal{I}} \in C^{\mathcal{I}}$. Example:

$$\mathit{Alice} : \mathit{Student} \sqcap \neg \exists \mathit{Pays}.\mathit{Tax}$$

asserts that Alice is a student who pays no tax.

A **role assertion** states that individuals a and b stand in relation r:

$$(a, b) : r$$

satisfied by $\mathcal{I}$ iff $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in r^{\mathcal{I}}$. Example:

$$(\mathit{Mary}, \mathit{CS600}) : \mathit{teaches}$$

asserts that Mary teaches the course CS600.

An interpretation satisfying all assertions in $\mathcal{A}$ is a *model of* $\mathcal{A}$. Note that the set of individual names N_I is disjoint from both N_C and N_R, so individuals can never be confused with concepts or roles.

6.4.3 Comparison Table

Aspect	TBox	ABox	Analogy
Content	Terminological axioms (GCIs, definitions) about *classes*	Assertions about *named individuals*	Schema vs. data
Example axiom	$\mathit{Parent} \sqsubseteq \mathit{Person}$	$\mathit{alice} : \mathit{Parent}$	Table definition vs. row
Another example	$\mathit{Course} \sqsubseteq \neg \mathit{Person}$	$(\mathit{alice}, \mathit{bob}) : \mathit{hasChild}$	Integrity constraint vs. fact
Reasoning task	Subsumption, classification, concept satisfiability	Instance checking, ABox consistency	Type queries vs. lookup
Question answered	"Is every C also a D?"	"Is individual a an instance of C?"	

(!) Exam Trap

TBox vs. ABox confusion. A GCI $\mathit{Parent} \sqsubseteq \mathit{Person}$ is a *TBox* axiom: it makes a universal claim about all parents, regardless of which individuals exist. The assertion $\mathit{alice} : \mathit{Parent}$ is an *ABox* fact about the specific individual Alice. Mixing the two levels — writing an individual name in

what is supposed to be a GCI, or writing a class-level claim in an ABox — is the most common exam mistake in this chapter.

6.4.4 The Knowledge Base as an Ontology

Putting the two parts together, a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ constitutes a formal ontology: the TBox provides the conceptual skeleton (the schema of what kinds of things can exist and how they are related), while the ABox populates that skeleton with facts about actual individuals. An interpretation is a *model of* $\mathcal{K}$ iff it is simultaneously a model of $\mathcal{T}$ and of $\mathcal{A}$.

6.5 OWL and Its Relation to Description Logics

The **Web Ontology Language (OWL)**, standardised by the W3C, is built directly on top of Description Logics. OWL 2 DL corresponds to the $\mathcal{SROIQ}$ description logic, making it the most expressive decidable DL in widespread use. OWL provides a concrete XML/RDF syntax for authoring ontologies that tools such as Protégé can process.

The key point is that every OWL ontology *is* a DL knowledge base at its logical core: OWL classes correspond to DL concepts, OWL object properties correspond to DL roles, OWL individuals correspond to DL individuals, OWL subclass axioms ($A \sqsubseteq B$) are TBox GCIs, and OWL individual assertions are ABox facts. The theoretical results about DL reasoning — decidability, complexity bounds, soundness and completeness of tableaux — transfer directly to OWL reasoning.

An important semantic commitment underlying both DL and OWL is the **Open World Assumption (OWA)**: if a fact is not stated in the knowledge base, it is treated as *unknown*, not false. This contrasts sharply with the *Closed World Assumption* used in databases and classical planning (STRIPS), where an unstated fact is assumed to be false. The OWA means, for instance, that if a person's phone number is absent from an OWL knowledge base, we cannot conclude they have no phone number — it is simply unknown.

6.6 Reasoning Tasks

Four standard reasoning tasks are defined over DL knowledge bases. All of them can be reduced to the problem of checking *satisfiability*, which is the fundamental computational task.

Concept satisfiability. Is there any interpretation in which $C^{\mathcal{I}} \neq \emptyset$? A concept is unsatisfiable (equivalent to $\bot$) if no possible world can contain an instance of it. Example: $\mathit{Male} \sqcap \mathit{Female}$ is unsatisfiable if the ontology declares them disjoint.

Subsumption. Does $C \sqsubseteq D$ follow from the TBox, i.e. does every model of $\mathcal{T}$ satisfy $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$? Subsumption is the backbone of the class hierarchy and is computed by *classification*. The key reduction is:

$$\mathcal{T} \models C \sqsubseteq D \quad \text{iff} \quad C \sqcap \neg D \text{ is unsatisfiable w.r.t. } \mathcal{T}.$$

Instance checking. Does individual a belong to concept C in every model of $\mathcal{K}$? This asks whether $a^{\mathcal{I}} \in C^{\mathcal{I}}$ is forced by the combined TBox and ABox. The reduction is:

$$\mathcal{K} \models a : C \quad \text{iff} \quad (\mathcal{T}, \mathcal{A} \cup \{a : \neg C\}) \text{ is inconsistent.}$$

Classification. Compute the complete subsumption hierarchy for all named concepts in $\mathcal{T}$. Classification calls subsumption for each pair of concepts and organises the results into the full taxonomy. It is the most expensive standard task but is done once when an ontology is loaded.

6.6.1 The Tableaux Algorithm

All of these tasks are solved in practice by **tableaux algorithms**, which attempt to construct a finite model or detect a contradiction. The algorithm works as follows:

1. Translate the concept(s) under test into *Negation Normal Form* (NNF), pushing all negations inward until they rest on atomic concepts.
2. Attempt to build an ABox for an individual b asserted to be an instance of the concept under test.
3. Apply *expansion rules*: decompose conjunctions (splitting into both parts), disjunctions (branching), existential restrictions (introducing fresh individuals), and value restrictions (propagating constraints to existing successors).

4. A branch *closes* (is unsatisfiable) when it contains a *clash*: $A(x)$ and $\neg A(x)$ for some individual x and atomic concept A.
5. The whole concept is unsatisfiable iff every branch closes.

For example, to check whether $(\exists R.A) \sqcap (\exists R.B) \sqsubseteq \exists R.(A \sqcap B)$ holds, we test whether $(\exists R.A) \sqcap (\exists R.B) \sqcap \neg(\exists R.(A \sqcap B))$ is unsatisfiable. In NNF this becomes $(\exists R.A) \sqcap (\exists R.B) \sqcap (\forall R.(\neg A \sqcup \neg B))$. The tableaux finds a model $\Delta^{\mathcal{I}} = \{b, c, d\}$ with $R^{\mathcal{I}} = \{(b, c), (b, d)\}$, $A^{\mathcal{I}} = \{c\}$, $B^{\mathcal{I}} = \{d\}$, demonstrating that the subsumption does *not* hold in general.

6.7 A Worked Example

Worked Example

A small family ontology.

TBox $\mathcal{T}$:

$$\begin{aligned} \mathit{Parent} &\sqsubseteq \mathit{Person} \\ \mathit{Mother} &\equiv \mathit{Parent} \sqcap \mathit{Female} \\ \mathit{Father} &\equiv \mathit{Parent} \sqcap \mathit{Male} \\ \mathit{Male} &\sqsubseteq \neg \mathit{Female} \\ \mathit{Parent} &\equiv \mathit{Person} \sqcap \exists \mathit{hasChild}.\mathit{Person} \end{aligned}$$

ABox $\mathcal{A}$:

$$\begin{aligned} &\mathit{anna} : \mathit{Female} \\ &\mathit{anna} : \mathit{Person} \\ &\mathit{tom} : \mathit{Male} \\ &\mathit{tom} : \mathit{Person} \\ &(\mathit{anna}, \mathit{tom}) : \mathit{hasChild} \end{aligned}$$

Reasoning:

1. **Is *anna* a Parent?** The ABox states $(\mathit{anna}, \mathit{tom}) : \mathit{hasChild}$, and tom is a Person, so anna has a Person-child. By the last TBox axiom, $\mathit{Parent} \equiv \mathit{Person} \sqcap \exists \mathit{hasChild}.\mathit{Person}$, and since anna is a Person with a Person-child, instance checking concludes $\mathit{anna} : \mathit{Parent}$.

2. **Is *anna* a Mother?** Anna is a Parent (step 1) and Female (ABox). By the TBox definition $\mathit{Mother} \equiv \mathit{Parent} \sqcap \mathit{Female}$, instance checking concludes $\mathit{anna} : \mathit{Mother}$.
3. **Subsumption: $\mathit{Mother} \sqsubseteq \mathit{Person}$?** Mother $\equiv$ Parent $\sqcap$ Female, and Parent $\sqsubseteq$ Person. So every Mother is a Parent, and every Parent is a Person: the subsumption $\mathit{Mother} \sqsubseteq \mathit{Person}$ holds.
4. **Is $\mathit{Male} \sqcap \mathit{Female}$ satisfiable?** The TBox contains $\mathit{Male} \sqsubseteq \neg \mathit{Female}$, i.e. $\mathit{Male} \sqcap \mathit{Female} \sqsubseteq \bot$. Any individual in $\mathit{Male} \sqcap \mathit{Female}$ must be in *Male* and in *Female*, but also in $\neg \mathit{Female}$ by the GCI — a clash. The concept is *unsatisfiable*.

6.8 Expressivity, Complexity, and the Decidability Trade-off

The most important lesson of this chapter is that the expressivity–decidability trade-off is not a nuisance but a *design principle*. DLs are deliberately crafted subsets of FOL that retain just enough power for ontology engineering while guaranteeing that reasoning terminates.

Logic	**Expressivity**	**Decidability / Complexity**
Propositional logic	Very low	PTime (SAT: NP-complete)
$\mathcal{ALC}$	Moderate (concepts + roles)	ExpTime-complete
$\mathcal{SROIQ}$ (OWL 2 DL)	High	N2ExpTime
Full FOL	Very high	Undecidable

Each step up the expressivity ladder adds a useful constructor but raises the worst-case complexity of reasoning. In practice, most ontologies used in medicine, biology, and the semantic web stay within $\mathcal{ALC}$ or mild extensions, because their reasoning problems are manageable there.

(!) Exam Trap

More expressivity $\Rightarrow$ reasoning may become undecidable. It is tempting to model everything as precisely as possible. But adding role

chains, concrete domains, or certain number restrictions can push a DL beyond the decidability boundary. OWL Full (which removes the separation between individuals and classes) is undecidable for precisely this reason. Always check whether the language you are using remains in a decidable fragment.

In a Nutshell

Chapter Summary

- **Knowledge Representation** encodes information formally so that automated reasoning is possible; the central trade-off is *expressivity vs. decidability.*
- An **ontology** is an explicit, shared, formal conceptualisation of a domain: concepts, roles, and axioms. It extends a mere *taxonomy* (which is only a subsumption hierarchy) by adding relations and logical constraints.
- **Description Logics** are decidable fragments of FOL built around *concepts* (classes), *roles* (binary relations), and *individuals.* The base language $\mathcal{ALC}$ provides $\neg$, $\sqcap$, $\sqcup$, $\forall R.C$, and $\exists R.C$.
- The **TBox** is the terminological (schema) layer: GCIs of the form $C \sqsubseteq D$ and definitions $C \equiv D$. It makes universal claims about classes. The **ABox** is the assertional (data) layer: concept assertions $a : C$ and role assertions $(a, b) : r$ about named individuals. Confusing the two is the most common exam error.
- **OWL** is the W3C standard ontology language grounded in DL; OWL 2 DL corresponds to $\mathcal{SROIQ}$. DL and OWL adopt the *Open World Assumption*: what is not stated is unknown, not false.
- Standard reasoning tasks are **satisfiability**, **subsumption** ($C \sqsubseteq D$ iff $C \sqcap \neg D$ is unsatisfiable), **instance checking**, and **classification**. All reduce to satisfiability and are solved by **tableaux algorithms**.
- Adding expressivity (role hierarchies, nominals, cardinality restrictions) increases complexity and can push reasoning into undecidability. Good ontology design respects these limits.

Chapter 7

Knowledge Representation II: RDF, SPARQL, and the Semantic Web

The previous chapter established the logical foundations of knowledge representation — propositional and first-order logic, ontologies, and the Description Logic family. This chapter extends that picture to the *Semantic Web* stack: the data model that the World Wide Web Consortium (W3C) standardised for exchanging structured knowledge on the internet, the query language for interrogating it, and the philosophical assumption about incomplete knowledge that distinguishes it sharply from classical databases.

7.1 The Resource Description Framework (RDF)

7.1.1 Motivation

A relational database stores information in tables with a fixed schema: you decide up front how many columns a table has, what types they hold, and which tables link to one another. That rigidity works well inside a single organisation but breaks down when independent parties on the web want to *combine* their data. Who decides the schema? What happens when two schemas disagree on a field name?

The Resource Description Framework, first published by the W3C in 1999 and revised in 2004 and 2014, sidesteps the problem entirely by adopting a radically

flat data model: every piece of information is expressed as a single **triple**.

7.1.2 The Triple: Subject, Predicate, Object

An RDF triple has the form

```
(subject, predicate, object)
```

and is read as a simple sentence: the subject is related to the object by the predicate. For example, the fact "Marie Curie was born in Warsaw" becomes

```
(Marie_Curie, bornIn, Warsaw)
```

The power of this model comes from three constraints on what subjects, predicates, and objects may be.

Key Idea

In RDF, every **resource** (thing worth naming) is identified by a **URI** — a Uniform Resource Identifier. URIs are globally unique, so two organisations independently publishing data about the same city can use the same URI and their facts will automatically merge without ambiguity.

Concretely:

- The **subject** must be a URI (or a blank node, an anonymous resource).
- The **predicate** must always be a URI.
- The **object** may be a URI *or* a **literal** — a plain data value such as a string, integer, date, or boolean.

Literals allow RDF to record raw data, not just relationships. The triple

```
(ex:MarieCurie, ex:birthYear, "1867"^^xsd:integer)
```

gives Marie Curie a typed integer birth year, where `xsd:integer` is the XML Schema datatype URI. URIs are often abbreviated using *namespace prefixes*: writing `ex:MarieCurie` is shorthand for the full URI `http://example.org/MarieCurie`.

7.1.3 RDF Graphs

A set of RDF triples forms an **RDF graph**. The graph interpretation is immediate: each URI becomes a node, each triple becomes a directed, labelled edge from subject to object, with the predicate as the edge label.

$$\texttt{ex:MarieCurie} \xrightarrow{\texttt{ex:bornIn}} \texttt{ex:Warsaw}$$

$$\texttt{ex:MarieCurie} \xrightarrow{\texttt{ex:birthYear}} \texttt{"1867"}$$

$$\texttt{ex:MarieCurie} \xrightarrow{\texttt{rdf:type}} \texttt{ex:Scientist}$$

The special predicate `rdf:type` (usually written `a` in shorthand) connects a resource to its class — directly parallel to the ABox concept assertion `MarieCurie : Scientist` from Description Logics. This symmetry is not coincidental: the Semantic Web technology stack is grounded in DL theory, as we will see when we reach RDFS and OWL.

A key structural insight is that RDF graphs are *schema-free*: any triple about any subject can appear at any time without modifying a central schema. New properties can be added for some resources but not others. This *open-ended* extensibility is one reason RDF became the lingua franca for Linked Data.

7.1.4 RDFS: Adding Schemas to RDF

Plain RDF can record facts but cannot express what those facts mean at the class level. **RDF Schema** (RDFS) is a small vocabulary of built-in predicates that layers terminological knowledge on top of RDF, closely mirroring the TBox of a Description Logic knowledge base.

The most important RDFS constructs are:

- **`rdfs:Class`** — marks a resource as a class.
- **`rdfs:subClassOf`** — states a subsumption relationship: `ex:Physicist rdfs:subClassOf ex:Scientist` means every physicist is a scientist. An RDFS-aware reasoner can then infer that any resource typed as `ex:Physicist` is also of type `ex:Scientist`.
- **`rdfs:subPropertyOf`** — analogous hierarchy for predicates.
- **`rdfs:domain`** — declares the class of subjects for a predicate. `ex:bornIn rdfs:domain ex:Person` means anything with a `bornIn` triple is inferred to be a person.

- `rdfs:range` — declares the class of objects for a predicate. `ex:bornIn rdfs:range ex:Place` means the object of every `bornIn` triple is inferred to be a place.

Key Idea

RDFS turns a flat bag of triples into a typed, hierarchical knowledge structure. `rdfs:subClassOf` gives inheritance; `rdfs:domain` and `rdfs:range` give type propagation. Together they enable simple but practically important inferences without the full machinery of Description Logics.

Beyond RDFS lies **OWL** (the Web Ontology Language), a W3C standard directly grounded in the $\mathcal{SROIQ}$ family of Description Logics. OWL adds the full ALC concept constructors — existential and value restrictions, negation, disjunction, cardinality constraints — and supports automated reasoning via tableaux-based OWL reasoners such as HermiT and Pellet. For the purpose of this chapter we focus on the RDF/RDFS layer, since it illustrates the core ideas without the full complexity of OWL.

7.2 Querying RDF: SPARQL

Having a large RDF graph is only useful if you can retrieve information from it. **SPARQL** (SPARQL Protocol and RDF Query Language, pronounced "sparkle") is the W3C standard query language for RDF, analogous to SQL for relational databases.

7.2.1 Triple Patterns and Variables

The fundamental unit of a SPARQL query is a **triple pattern** — a triple in which any component (subject, predicate, or object) may be replaced by a **variable**. By convention, variables start with a question mark: `?x`, `?name`, `?city`.

A triple pattern `(?x, ex:bornIn, ex:Warsaw)` matches every triple in the graph whose predicate is `ex:bornIn` and whose object is `ex:Warsaw`, binding `?x` to each matching subject in turn.

7.2.2 SELECT Queries

A SPARQL `SELECT` query has the form:

```
SELECT <variables>
WHERE { <graph pattern> }
```

The `WHERE` clause contains a **basic graph pattern** (BGP): one or more triple patterns that must all be satisfied simultaneously by the same variable bindings. The query engine finds every mapping from variables to RDF terms that makes all patterns hold in the graph, then returns the requested variables.

This is essentially *subgraph matching*: the engine searches for a subgraph of the data graph that is isomorphic to the pattern graph specified in the query. Each solution corresponds to one such embedding, and the `SELECT` clause projects out the variable columns of interest.

7.2.3 FILTER

Within a `WHERE` clause, a `FILTER` expression prunes bindings that do not satisfy a Boolean condition. Conditions may compare literals (`?year > 1850`), test string patterns, or check datatype membership. For example:

```
FILTER (?birthYear > 1850)
```

restricts results to those where the bound integer value of `?birthYear` exceeds 1850. `FILTER` can use standard arithmetic operators, string functions (`CONTAINS`, `STRSTARTS`), and logical connectives (`&&`, `||`, `!`).

7.3 Worked Example: RDF Triples and a SPARQL Query

Worked Example

Dataset. Consider the following three RDF triples (written in Turtle notation, the most common human-readable RDF serialisation):

```
ex:MarieCurie   ex:bornIn    ex:Warsaw .
ex:MarieCurie   ex:birthYear "1867"^^xsd:integer .
ex:MarieCurie   rdf:type     ex:Physicist .
```

Query. Retrieve the name and birth year of all physicists born after 1800.

```
PREFIX ex:  <http://example.org/>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>

SELECT ?person ?year
WHERE {
  ?person rdf:type    ex:Physicist .
  ?person ex:birthYear ?year       .
  FILTER (?year > 1800)
}
```

Evaluation. The engine tries to find bindings for `?person` and `?year` such that *both* triple patterns are satisfied simultaneously. Checking the data graph:

1. `?person rdf:type ex:Physicist`: the only match is `?person = ex:MarieCurie`.
2. With that binding, `ex:MarieCurie ex:birthYear ?year` gives `?year = 1867`.
3. `FILTER (1867 > 1800)` evaluates to true.

Result: one row, `?person = ex:MarieCurie`, `?year = 1867`.

Notice that adding RDFS axioms (e.g. `ex:Physicist rdfs:subClassOf ex:Scientist`) would allow an RDFS-aware engine to also return physicists when querying for `ex:Scientist` — a direct analogue of subsumption inference in Description Logics.

7.4 The Open World Assumption

7.4.1 Two Philosophical Stances Toward Missing Information

Every knowledge representation system must decide how to treat facts that are not explicitly stored. There are two diametrically opposite answers to the question "Is what is not stated false?"

Key Idea

Closed World Assumption (CWA): If a fact is not entailed by the knowledge base, it is assumed to be *false*. The world is entirely captured by what is stated.

Open World Assumption (OWA): If a fact is not entailed by the knowledge base, its truth value is *unknown* — it may be true, it may be false; we simply do not know. The knowledge base is treated as a partial, potentially incomplete description of the world.

The CWA is the default in classical relational databases and in STRIPS-style planning. If the database does not contain a row stating that a flight exists, the booking system infers the flight does not exist — and stops looking. This is computationally convenient and usually correct when the database is *meant* to be complete (every relevant fact has been entered).

The Semantic Web, Description Logics, and OWL all adopt the OWA. The reasoning is simple: no single ontology on the web can claim to know everything. Data about Marie Curie may reside in ten different RDF graphs maintained by ten different institutions. Just because one graph does not state her nationality does not mean she has none — some other graph may contain that information, or the information may simply not have been published yet.

7.4.2 Consequences of the OWA

The Open World Assumption has several important practical consequences.

Absence of a fact is not evidence of falsity. Under CWA, querying for a fact that is not in the database returns "false" or an empty result that is interpreted as "does not exist." Under OWA, the same empty result means only "we do not currently know" — the reasoner neither confirms nor denies the fact.

Negation requires explicit assertion. If you need to express that Marie Curie is definitely *not* a citizen of Germany, you must *assert* this explicitly (e.g. using OWL's `owl:complementOf` or a negation axiom). The absence of the triple `(ex:MarieCurie, ex:citizenOf, ex:Germany)` does not, by itself, establish the negation.

Unique Name Assumption does not hold by default. Under CWA databases, two distinct keys always name two distinct entities. Under OWA, two different URIs might refer to the same real-world object unless an `owl:differentFrom`

(or `owl:AllDifferent`) assertion explicitly separates them. OWL provides `owl:sameAs` to merge resources and `owl:differentFrom` to separate them.

Reasoning is monotonic. Adding new facts to an OWA knowledge base can only increase the set of derivable conclusions; it can never retract them. This contrasts with some CWA systems (e.g. logic programming with negation-as-failure) where adding a fact can eliminate a previously valid conclusion.

7.4.3 Comparison Table: OWA vs. CWA

Aspect	**OWA**	**CWA**
Definition	Unknown fact = *unknown*	Unknown fact = *false*
Treatment of gaps	Incomplete knowledge is normal	Database assumed to be comple
Negation	Must be explicitly stated	Implicit: absence = false
Unique Name	Two names may co-refer	Two names ⇒ two objects
Reasoning	Monotonic	May be non-monotonic
Typical systems	OWL, RDF/RDFS, DL reasoners	SQL databases, STRIPS, Datalo
Use case	Open-ended, distributed data	Closed, controlled datasets

7.5 Linked Data and Knowledge Graphs

The combination of URIs for global naming, RDF for a common data model, and RDFS/OWL for shared vocabularies enabled an initiative called **Linked Data**, coined by Tim Berners-Lee in 2006. The core principles are:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look them up.
3. When a URI is dereferenced, return useful RDF data.
4. Include links to other URIs so that agents can discover more data.

Following these principles, the **Linked Open Data cloud** has grown to hundreds of interlinked datasets — DBpedia (structured facts from Wikipedia), Wikidata (the collaborative knowledge graph), GeoNames (geographic URIs), and many domain-specific sources in life sciences, government, and culture.

A **knowledge graph** is a large RDF graph (or a graph database inspired by RDF) that stores facts about real-world entities, their types, and their relationships.

Google's Knowledge Graph, which powers fact panels in search results, follows this model, as does the Wikidata knowledge base. These systems demonstrate the practical value of the Semantic Web vision: a web of machine-readable data that different applications can query, combine, and reason over using common standards.

7.6 Common Exam Traps

(!) Exam Trap

OWA vs. CWA — the classic question. The most frequently tested distinction in this topic area is which assumption the Semantic Web uses and what it implies. Remember:

- Semantic Web / OWL / DL: **OWA** — absence $\neq$ false.
- Databases (SQL) / STRIPS planning: **CWA** — absence = false.

A question like "In an OWL ontology, if the fact `hasDegree(Alice, PhD)` is not present, can we conclude Alice has no PhD?" has the answer **No**: under OWA we simply do not know. Under CWA (SQL), the answer would be yes.

(!) Exam Trap

RDF triple direction matters. An RDF triple is a *directed* edge from subject to object. The triple `(ex:Paris, ex:capitalOf, ex:France)` is a different assertion from `(ex:France, ex:capitalOf, ex:Paris)`. The predicate is always read left to right. Swapping subject and object changes the meaning entirely. This is the same directional sensitivity as the DL subsumption relation $C \sqsubseteq D$ ("C is a subclass of D", not the reverse).

(!) Exam Trap

Negation under OWA requires explicit statements. A common mistake is to treat an empty SPARQL result as evidence that something does not exist. Under OWA, empty results mean "not known to be true in this graph", not "false in the world." If you need to query for resources

that are *definitely not* of a certain type, you must either use `FILTER NOT EXISTS` (which tests the local graph, not the world) or ensure that the ontology contains explicit negation axioms.

(!) Exam Trap

RDFS `domain` and `range` infer types — they do not constrain them. In a database, declaring a foreign key constraint prevents invalid data from being inserted. In RDFS, declaring `ex:bornIn rdfs:domain ex:Person` does *not* raise an error if a non-person has a `bornIn` triple. Instead, the reasoner *infers* that whatever has a `bornIn` triple must be a person. This is a subtle but important difference: RDFS statements are inference rules, not integrity constraints.

(!) Exam Trap

SPARQL variables match within a single query solution. If the same variable appears in two triple patterns in the same `WHERE` clause, it must be bound to the *same* RDF term in both. A query pattern such as

```
?x ex:bornIn ?city .  ?x ex:nationality ?nat .
```

will only return results where *the same* resource fills `?x` in both triples. This join-on-variable semantics is how SPARQL performs the equivalent of a relational `JOIN`.

In a Nutshell

Chapter Summary

- **RDF** represents knowledge as subject–predicate–object **triples**; every resource is named by a **URI**; objects may also be typed **literals**. A set of triples forms a directed, labelled **RDF graph**.
- **RDFS** adds schema-level vocabulary: `rdfs:subClassOf`, `rdfs:domain`, and `rdfs:range` enable class hierarchies and type inference over RDF data.
- **SPARQL** queries RDF graphs via **triple patterns** with **variables** (`?x`); a `SELECT/WHERE` query performs subgraph matching; `FILTER` prunes

results by Boolean conditions.

- The Semantic Web adopts the **Open World Assumption**: the absence of a fact means it is *unknown*, not false. Databases and STRIPS use the **Closed World Assumption**: the absence of a fact means it is false.
- Under OWA, **negation must be explicitly asserted**; empty query results do not imply non-existence; and two URIs may refer to the same entity unless explicitly separated.
- **Linked Data** and **knowledge graphs** (DBpedia, Wikidata, Google KG) apply these standards at web scale, enabling machine-readable, interlinked data across independent organisations.

Chapter 8

Vagueness and Uncertainty

8.1 Two Distinct Problems That Look the Same

Consider two questions: "Is Alice tall?" and "Will it rain tomorrow?" At first glance both seem to call for the same hedged answer — "probably," or "sort of." Yet the underlying challenge in each case is fundamentally different, and conflating them is one of the most common conceptual errors in reasoning under incomplete information.

The first question is a question about **vagueness**. The predicate *tall* does not have a sharp boundary. There is no precise height threshold below which a person is unambiguously not tall and above which they definitively are. This is not a problem of missing information: even a god who knew Alice's exact height to the millimetre would still face the question of whether 178 cm counts as tall. The difficulty is located in the *predicate itself*, not in the knower. Vagueness is a semantic phenomenon: predicates like *tall*, *warm*, *heap*, *reddish*, and *middle-aged* all lack sharp extensions. Fuzzy logic is designed precisely to model this kind of partial truth.

The second question is a question about **uncertainty**. Whether it will rain tomorrow is a perfectly determinate fact — it either will or it will not. The difficulty is that we do not yet know which. Our belief is graded not because the proposition lacks a sharp truth value, but because our *knowledge* is incomplete. Probabilistic logic is the tool for quantifying and propagating such degrees of belief.

Key Idea

Vagueness is about predicates: a predicate lacks a sharp boundary, so an element can belong to it *to a degree*. Uncertainty is about knowledge: a fact is determinate but we do not know it, so we hold it with some *probability*. Both frameworks assign numbers in $[0, 1]$, but those numbers mean entirely different things.

Both frameworks arise because classical logic — where every statement is either true or false — cannot adequately represent these situations. Classical logic cannot say that Alice is "0.7 tall"; it is forced to pick a side. And while classical logic can *assert* that there is a 70 % chance of rain, it needs a separate probability layer to give that assertion formal meaning. Sessions 6 and 7 built the machinery of classical and description logics for precise, deterministic knowledge. This chapter builds the two extensions needed when precision gives way to degrees and probabilities.

8.1.1 The Sorites (Heap) Paradox as a Motivation

The motivation for fuzzy logic is most vividly conveyed by the *sorites* or *heap* paradox, known since antiquity. It proceeds as follows:

1. One million grains of sand form a heap. (Premise 1)
2. A heap minus one grain is still a heap. (Premise 2)
3. Therefore, one million minus one grains form a heap.
4. Iterating Premise 2 a million times: one grain of sand forms a heap.

The conclusion is absurd, yet both premises seem entirely reasonable. Classical logic, which demands that *heap* be either true or false for any given pile, forces us to accept one of two uncomfortable positions: either deny Premise 2 (and name the exact threshold), or accept the absurd conclusion. Fuzzy logic dissolves the paradox by allowing *heap* to be true *to a degree*. A pile of one million grains is a heap to degree 1; a pile of one grain is a heap to degree 0; piles in between carry intermediate membership values. No single grain crossing causes a discontinuity, because truth varies continuously.

The analogous motivation for probabilistic logic is simpler: many real decisions depend on degrees of belief in uncertain propositions, and we need a calculus that aggregates and updates such beliefs coherently. Sections 8.3 and 8.4 develop that

calculus.

8.2 Fuzzy Set Theory

8.2.1 From Characteristic Functions to Membership Functions

A classical (crisp) set A over a universe U can equivalently be described by its *characteristic function*:

$$\mathbf{1}_A : U \to \{0, 1\}, \quad \mathbf{1}_A(x) = 1 \text{ iff } x \in A.$$

Every element either belongs or does not; there is no middle ground. Fuzzy set theory, introduced by Lotfi Zadeh, relaxes the codomain from the two-element set $\{0, 1\}$ to the full unit interval $[0, 1]$.

Key Idea

A **fuzzy set** $A = (U, \mu_A)$ is defined by a universe U and a **membership function**

$$\mu_A : U \to [0, 1].$$

The value $\mu_A(x) = d$ is interpreted as "x belongs to A to degree d." Crisp sets are the special case $\mu_A : U \to \{0, 1\}$.

For example, one might define a fuzzy set *warm* over the universe of temperatures (in °C) such that $\mu_{\text{warm}}(15) = 0.1$, $\mu_{\text{warm}}(22) = 0.8$, and $\mu_{\text{warm}}(30) = 1.0$. The lecture gives the analogous example for colours: $\mu_{\text{red}}(\text{ball}) = 0.6$ captures the intuition that the ball is "reddish" rather than purely red.

8.2.2 Membership Function Shapes

Membership functions can take many forms. The two most commonly encountered in practice are the *triangular* and the *trapezoidal* functions.

A **triangular** function is defined by three parameters (a, b, c) with $a < b < c$:

$$\mu(x) = \max\left(0,\ \min\left(\frac{x - a}{b - a},\ \frac{c - x}{c - b}\right)\right).$$

It rises linearly from 0 at $x = a$ to 1 at the peak $x = b$, then falls linearly back to 0 at $x = c$.

A **trapezoidal** function extends this with a flat top over a core interval $[b, c]$, so that membership equals 1 for all $x \in [b, c]$ and falls linearly to 0 outside $[a, d]$:

$$\mu(x) = \max\left(0,\ \min\left(\frac{x-a}{b-a},\ 1,\ \frac{d-x}{d-c}\right)\right).$$

These shapes are used as *linguistic variables*: one might partition the temperature axis into overlapping fuzzy sets labelled *cold*, *cool*, *warm*, and *hot*, each with its own triangular or trapezoidal membership function. A specific temperature then has partial membership in several sets simultaneously, which captures the intuition that 22 °C is fairly warm while still being somewhat cool.

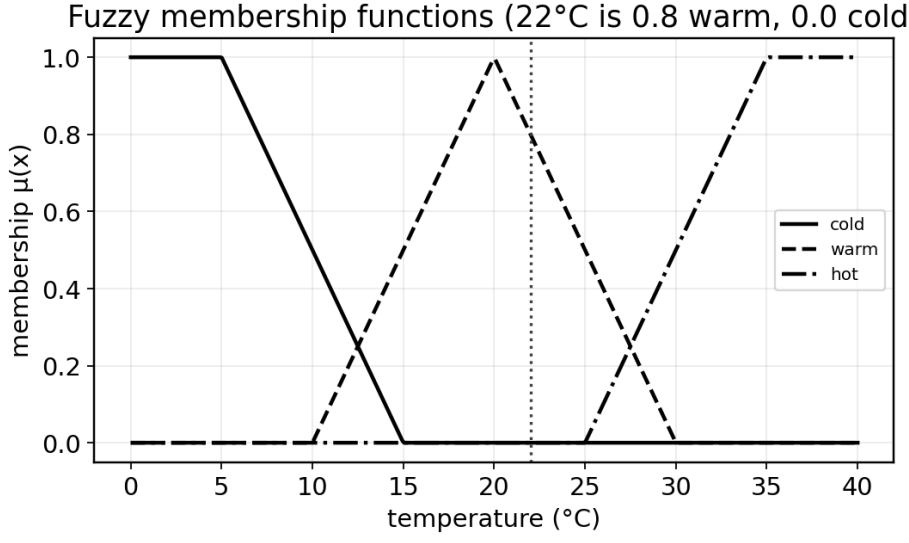

Figure 8.1: Fuzzy membership functions: a single temperature can belong partially to several fuzzy sets (e.g. 22°C is strongly “warm” and not “cold”).

8.2.3 Fuzzy Set Operations: The Zadeh Definitions

Given fuzzy sets A and B over the same universe U, the standard (Zadeh) generalisation of the classical set operations is:

Operation	Notation	Membership formula
Intersection (AND)	$A \cap B$	$\mu_{A\cap B}(x) = \min(\mu_A(x),\, \mu_B(x))$
Union (OR)	$A \cup B$	$\mu_{A\cup B}(x) = \max(\mu_A(x),\, \mu_B(x))$
Complement (NOT)	A^C	$\mu_{A^C}(x) = 1 - \mu_A(x)$

These definitions recover the classical behaviour at the boundary: if $\mu_A(x) \in \{0, 1\}$ for all x, then min, max, and $1 - \cdot$ coincide with the crisp intersection, union, and complement respectively.

Many classical set identities remain valid. Neutral elements: $\min(1, \mu_A(x)) = \mu_A(x)$ (corresponding to $U \cap A = A$) and $\max(0, \mu_A(x)) = \mu_A(x)$ (corresponding to $\emptyset \cup A = A$). Commutativity, associativity, distributivity, and De Morgan's laws all carry over.

However, two classical laws *fail* for fuzzy sets with non-crisp memberships, and this is entirely intentional:

$$A \cap A^C \neq \emptyset \quad \text{(law of non-contradiction fails)},$$
$$A \cup A^C \neq U \quad \text{(law of excluded middle fails)}.$$

For a ball with $\mu_{\text{red}}(\text{ball}) = 0.6$:

$$\mu_{\text{red}\cap\text{red}^C}(\text{ball}) = \min(0.6,\, 1 - 0.6) = \min(0.6, 0.4) = 0.4,$$

$$\mu_{\text{red}\cup\text{red}^C}(\text{ball}) = \max(0.6,\, 0.4) = 0.6.$$

The intersection is not empty (degree 0.4), and the union is not the whole universe (degree 0.6, not 1.0). For vague predicates this is perfectly reasonable: something can be, to a small positive degree, both red and not-red, and yet not fully either red or not-red.

(!) Exam Trap

In fuzzy logic, $A \vee \neg A$ is **not** guaranteed to have truth value 1. $\max(\mu, 1 - \mu) = 1$ only if $\mu \in \{0, 1\}$. This is the intentional departure from classical logic that makes fuzzy logic appropriate for vague predicates — but it means tautologies lose their absolute status.

8.2.4 Fuzzy Logic: Extending Propositions

Fuzzy logic applies the same membership-based semantics to propositional formulas. An interpretation $\mathcal{I}$ assigns a value in $[0, 1]$ to each propositional atom. For compound formulas:

$$\mathcal{I}(F \wedge G) = \min(\mathcal{I}(F), \mathcal{I}(G)),$$
$$\mathcal{I}(F \vee G) = \max(\mathcal{I}(F), \mathcal{I}(G)),$$
$$\mathcal{I}(\neg F) = 1 - \mathcal{I}(F).$$

8.2.5 t-Norms and s-Norms: Generalising AND and OR

The Zadeh min/max operators are not the only valid choice. The course introduces the general algebraic framework of *t-norms* and *s-norms*.

A **t-norm** $t : [0, 1]^2 \to [0, 1]$ generalises conjunction and must satisfy:

1. **Neutral element:** $t(1, x) = x$ for all x.
2. **Commutativity:** $t(x, y) = t(y, x)$.
3. **Associativity:** $t(x, t(y, z)) = t(t(x, y), z)$.
4. **Monotonicity:** $x \leq x'$ and $y \leq y'$ implies $t(x, y) \leq t(x', y')$.

An **s-norm** (or t-conorm) $s : [0, 1]^2 \to [0, 1]$ generalises disjunction with neutral element 0 and the same commutativity, associativity, and monotonicity conditions.

The lecture presents three important families:

Name	t-norm (AND)	s-norm (OR)
Zadeh (min/max)	$t(x, y) = \min(x, y)$	$s(x, y) = \max(x, y)$
Algebraic	$\text{alg}_t(x, y) = x \cdot y$	$\text{alg}_s(x, y) = x + y - xy$
Quotient	$\text{quo}_t(x, y) = \dfrac{xy}{x + y - xy}$	$\text{quo}_s(x, y) = \dfrac{x + y - 2xy}{1 - xy}$

Importantly, min is the *largest* possible t-norm in the sense that $t(x, y) \leq \min(x, y)$ for every t-norm t. The algebraic t-norm $x \cdot y$ is strictly less than $\min(x, y)$ whenever both $x, y \in (0, 1)$. There are infinitely many valid t-norms and s-norms; the lecture uses min/max (Zadeh) as the default, but the algebraic family is also explicitly introduced.

(!) Exam Trap

Fuzzy AND is **not** simply multiplication unless you are using the algebraic t-norm. The default in these slides is min. Conversely, never mistake the algebraic t-norm $x \cdot y$ for ordinary classical conjunction: it yields a value strictly below $\min(x, y)$ for non-crisp memberships. Know which t-norm is in use before computing.

8.2.6 Worked Example: Fuzzy Set Operations

Worked Example

Suppose we have two fuzzy sets *red* and *round* defined over a single object, a ball:

$$\mu_{\text{red}}(\text{ball}) = 0.6, \qquad \mu_{\text{round}}(\text{ball}) = 1.0.$$

Intersection (AND):

$$\mu_{\text{red}\cap\text{round}}(\text{ball}) = \min(0.6, 1.0) = 0.6.$$

The ball belongs to the fuzzy set *red and round* to degree 0.6.
Union (OR):

$$\mu_{\text{red}\cup\text{round}}(\text{ball}) = \max(0.6, 1.0) = 1.0.$$

The ball is fully *red or round*.
Complement (NOT red):

$$\mu_{\text{red}^C}(\text{ball}) = 1 - 0.6 = 0.4.$$

Law of excluded middle:

$$\mu_{\text{red}\cup\text{red}^C}(\text{ball}) = \max(0.6, 0.4) = 0.6 \neq 1.$$

The excluded middle does *not* hold. This is an intended feature for vague predicates.
Additional compound formulas:

$$\mathcal{I}(\text{red} \wedge \neg\text{red}) = \min(0.6, 0.4) = 0.4,$$

$$\mathcal{I}(\text{red} \vee \neg\text{round}) = \max(0.6,\ 1 - 1.0) = \max(0.6, 0) = 0.6.$$

8.2.7 Brief Note on Linguistic Variables and Defuzzification

A *linguistic variable* is a variable whose values are words or phrases from a natural language, each modelled as a fuzzy set. For example, the variable *Temperature* might take values *cold*, *cool*, *warm*, and *hot*, each defined by its own membership function over the real-valued temperature axis. This allows a fuzzy inference system to reason with human-interpretable rules of the form: "If temperature is warm *and* humidity is high, then fan speed is medium."

After combining multiple fuzzy rules through min/max operations, the result-

ing aggregated fuzzy set must often be converted back into a single crisp number for use in a real controller. This process is called *defuzzification*. The two most common methods are the *centroid* (centre of mass of the aggregated membership function) and *max-membership* (the crisp value at the peak). These can disagree substantially, which is itself an exam-relevant point.

8.3 Probabilistic Logic and Uncertainty

8.3.1 Probability Spaces

Where fuzzy logic models partial truth, probabilistic logic models ignorance. The formal foundation is the *probability space*, a triple (Ω, Σ, P):

- Ω is the finite, non-empty set of **elementary events** (also called *possible worlds* or *states*).
- $\Sigma = 2^{\Omega}$ is the power set of Ω, the set of all **events**.
- $P : \Sigma \to \mathbb{R}$ is a **probability measure** satisfying Kolmogorov's axioms.

Each event $E \in \Sigma$ is a set of possible worlds; intuitively, E corresponds to the set of worlds in which some proposition holds.

Kolmogorov's axioms (finite version):

1. **Non-negativity:** $P(E) \geq 0$ for all $E \in \Sigma$.
2. **Normalisation:** $P(\Omega) = 1$.
3. **Finite additivity:** If $E_1, \ldots, E_n$ are pairwise disjoint, then

$$P(E_1 \cup \cdots \cup E_n) = P(E_1) + \cdots + P(E_n).$$

A rich collection of derived properties follows immediately. For example:

$$P(\Omega \setminus E) = 1 - P(E), \qquad A \subseteq B \Rightarrow P(A) \leq P(B),$$

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

The last identity handles the case where A and B overlap; additivity applies only to disjoint events, so the intersection must be subtracted to avoid double-counting.

8.3.2 Propositional Probabilistic Logic

Probabilistic logic lifts probability theory into the logical setting by identifying propositional interpretations with possible worlds. Given atoms $A_1, \ldots, A_n$, let Ω

be the set of all 2^n classical truth-value assignments $I : \{A_1, \ldots, A_n\} \to \{0, 1\}$. Each propositional formula F is identified with the event

$$E_F = \{\, I \in \Omega : I \models F \,\},$$

and the probability of F is $P(F) = P(E_F)$.

This connection immediately yields a crucial difference from fuzzy logic:

- If T is a *tautology*, then every interpretation satisfies it, so $E_T = \Omega$ and $P(T) = 1$.
- If C is a *contradiction*, then no interpretation satisfies it, so $E_C = \emptyset$ and $P(C) = P(\emptyset) = 0$.

Fuzzy logic does not share this property: we saw that $\mathcal{I}(F \vee \neg F)$ can be strictly less than 1. This is appropriate for vague predicates where the law of excluded middle should not hold, but it means probabilistic logic is a strictly stronger extension of classical logic in this respect.

A **probabilistic knowledge base** consists of probabilistic formulas of the form $F : p$, meaning the probability measure satisfies $P(F) = p$. From such constraints one can derive bounds on other probabilities. For instance, given $A \wedge B : 0.3$ and $A : 0.8$, one can derive $P(B) \leq 0.5$ by a chain of equalities and inequalities exploiting the axioms and classical logical equivalences.

8.4 Conditional Probability and Reasoning

Many real inference problems are inherently conditional: we want the probability of some conclusion *given* that certain evidence is in place. The **conditional probability** of B given A is:

$$P(B \mid A) = \frac{P(B \cap A)}{P(A)}, \qquad P(A) > 0.$$

In propositional probabilistic logic this reads:

$$P(G \mid F) = \frac{P(G \wedge F)}{P(F)}.$$

A probabilistic knowledge base can also contain *conditional* probabilistic formulas $(G \mid F)[p]$, meaning that $P(G \mid F) = p$ must hold for all satisfying measures. The conditional reasoning example from the lecture illustrates how non-trivial bounds can be derived:

Worked Example

Consider the knowledge base:

$$\begin{aligned}
&\mathit{Bird} : 0.8\\
&(\mathit{Bird} \mid \mathit{Penguin})[1]\\
&(\mathit{Penguin} \mid \mathit{Bird})[0.2]\\
&(\mathit{Flies} \mid \mathit{Penguin})[0]\\
&(\mathit{Flies} \mid \mathit{Bird})[0.7]
\end{aligned}$$

Claim: $P(\mathit{Penguin}) \geq 0.16$.
Derivation:

$$\begin{aligned}
P(\mathit{Penguin}) &= P((\mathit{Bird} \vee \neg \mathit{Bird}) \wedge \mathit{Penguin}) && \text{(ta}\\
&= P((\mathit{Bird} \wedge \mathit{Penguin}) \vee (\neg \mathit{Bird} \wedge \mathit{Penguin})) && \text{(distrib}\\
&= P(\mathit{Bird} \wedge \mathit{Penguin}) + P(\neg \mathit{Bird} \wedge \mathit{Penguin}) && \text{(ad}\\
&= P(\mathit{Penguin} \mid \mathit{Bird}) \cdot P(\mathit{Bird}) + P(\mathit{Penguin} \mid \neg \mathit{Bird}) \cdot P(\neg \mathit{Bird}) && \text{(cond}\\
&\geq P(\mathit{Penguin} \mid \mathit{Bird}) \cdot P(\mathit{Bird}) && \text{(non-neg}\\
&= 0.2 \times 0.8 = 0.16.
\end{aligned}$$

Note that the conditional $(\mathit{Bird} \mid \mathit{Penguin})[1]$ says every penguin is a bird, but does not by itself pin down $P(\mathit{Penguin})$; the lower bound comes from the unconditional bird frequency and the conditional rate of penguins among birds.

8.4.1 A Note on Bayes' Theorem

The definition of conditional probability immediately yields **Bayes' theorem**, which is the cornerstone of probabilistic AI reasoning:

$$P(H \mid E) = \frac{P(E \mid H)\, P(H)}{P(E)}.$$

Here H is a hypothesis and E is observed evidence. $P(H)$ is the *prior* probability of the hypothesis, $P(E \mid H)$ is the *likelihood* of the evidence under that hypothesis, and $P(H \mid E)$ is the updated *posterior*. Bayes' theorem is not covered at length in Session 8, but it underpins most of the probabilistic reasoning techniques encountered later in the course (Chapter 13 of Russell & Norvig).

8.5 Comparing the Three Frameworks

The chapter has now introduced three distinct frameworks for reasoning beyond classical binary truth. The following table contrasts their key properties:

	Classical Logic	**Fuzzy Logic**	**Probabilistic Logic**
Truth values	$\{0, 1\}$	$[0, 1]$	$[0, 1]$
Propositions are	True or false	True to a degree	True or false
Values represent	Truth/falsity	Degree of membership	Probability of truth
Addresses	(perfect knowledge)	Vagueness	Uncertainty
$A \vee \neg A$	Always 1	Can be < 1	Always 1
$A \wedge \neg A$	Always 0	Can be > 0	Always 0
Conjunction	Boolean AND	min (or another t-norm)	$P(A \cap B) = P(A) \cdot P(B)$ if independent
Disjunction	Boolean OR	max (or another s-norm)	$P(A \cup B) = P(A) + P(B) - P(A \cap B)$
Complement	$\neg A$	$1 - \mu_A(x)$	$1 - P(A)$
Tautologies	$P = 1$ forced	Not guaranteed	$P = 1$ forced

The central insight is that fuzzy logic and probabilistic logic both use the interval $[0, 1]$, but they are answering completely different questions. A fuzzy membership value of 0.7 for *warm* says that the temperature is "70 % warm" in a graded semantic sense. A probability of 0.7 says that, in 70 % of possible worlds consistent with our knowledge, the temperature is warm. The vagueness/uncertainty distinction maps directly onto this difference in interpretation.

8.6 Dempster–Shafer Theory (Supplementary)

Key Idea

Dempster–Shafer theory is not covered in the Session 8 slides and should be treated as supplementary material. It is included here for completeness and for students who encounter it in the wider literature.

Dempster–Shafer (DS) theory, also called *evidence theory*, provides a framework that sits between probability theory and classical set theory. Its central motivation is to model situations where evidence supports a set of hypotheses without being specific enough to single out which one is true. Probability theory requires a precise distribution over all atomic outcomes; DS theory allows mass to be distributed over *sets* of outcomes, encoding a fundamentally weaker form of knowledge.

8.6.1 The Mass Function

Let $\Theta = \{\theta_1, \ldots, \theta_n\}$ be a finite frame of discernment (the set of all mutually exclusive and exhaustive hypotheses). The key object is the **basic probability assignment** (or *mass function*):

$$m : 2^{\Theta} \to [0, 1], \quad m(\emptyset) = 0, \quad \sum_{A \subseteq \Theta} m(A) = 1.$$

Each subset $A \subseteq \Theta$ is called a *focal element* if $m(A) > 0$. The mass $m(A)$ represents the portion of belief that is committed *specifically* to the set A and cannot be further subdivided among the individual elements of A based on current evidence.

8.6.2 Belief and Plausibility

From the mass function, two probability-like measures are derived, and they *bracket* the true probability of any event:

$$\mathrm{Bel}(A) = \sum_{B \subseteq A} m(B), \qquad \mathrm{Pl}(A) = \sum_{B \cap A \neq \emptyset} m(B).$$

Belief $\mathrm{Bel}(A)$ sums the mass of all subsets fully contained in A: it captures the evidence that *necessarily* supports A, with no support to anything outside A.

Plausibility $\mathrm{Pl}(A)$ sums the mass of all subsets that *intersect* A: it captures the evidence that is at least *consistent* with A.

For any event A it always holds that:

$$\mathrm{Bel}(A) \leq P(A) \leq \mathrm{Pl}(A).$$

The gap $[\mathrm{Bel}(A), \mathrm{Pl}(A)]$ is the *interval of ignorance*: the wider this gap, the more uncommitted ("ignorant") the current evidence is about whether A holds.

(!) Exam Trap

The direction of the inequality matters: $\mathrm{Bel}(A) \leq \mathrm{Pl}(A)$, always. Bel is a lower bound on probability, Pl is an upper bound. Swapping them is a common error.

8.6.3 Worked Example: Bel and Pl from a Mass Function

Worked Example

Suppose $\Theta = \{H, D\}$ (hypotheses: "Healthy" and "Diseased"), and a diagnostic test yields the mass function:

$$m(\{H\}) = 0.3, \qquad m(\{D\}) = 0.5, \qquad m(\{H, D\}) = 0.2.$$

The mass 0.2 on $\Theta = \{H, D\}$ captures evidence that is positive but non-specific — it supports the frame as a whole but cannot distinguish between the two hypotheses.

Compute belief and plausibility for $A = \{D\}$:

$$\text{Bel}(\{D\}) = m(\{D\}) = 0.5.$$
$$\text{Pl}(\{D\}) = m(\{D\}) + m(\{H, D\}) = 0.5 + 0.2 = 0.7.$$

The true probability of D lies somewhere in $[0.5, 0.7]$. The gap of 0.2 corresponds exactly to the uncommitted mass on Θ.

For comparison, $A = \{H\}$:

$$\text{Bel}(\{H\}) = m(\{H\}) = 0.3.$$
$$\text{Pl}(\{H\}) = m(\{H\}) + m(\{H, D\}) = 0.3 + 0.2 = 0.5.$$

Note that $\text{Bel}(\{H\}) + \text{Bel}(\{D\}) = 0.3 + 0.5 = 0.8 \neq 1$; DS theory does *not* require belief in a set and its complement to sum to 1. This is the key departure from classical probability.

8.6.4 DS Theory vs. Probability vs. Fuzzy Logic

The extended comparison table below situates DS theory alongside the other two frameworks:

	Fuzzy Logic	Probability	Dempster–Shafer
Models	Vagueness (partial truth)	Uncertainty (degree of belief)	Evidence/ignorance
Key object	Membership function $\mu_A : U \to [0,1]$	Probability measure P	Mass function $m : 2^\Theta \to [0,1]$
Mass/measure on	Individual elements	Singletons (atoms)	Arbitrary subsets
Additivity	Not required	Sums to 1 over atoms	Sums to 1 over all subsets
Complement rule	$1 - \mu_A$	$1 - P(A)$	$\mathrm{Bel}(A) + \mathrm{Bel}(A^C) \leq 1$
Ignorance	Not modelled separately	Zero probability	Gap $[\mathrm{Bel}, \mathrm{Pl}]$
Tautologies	Not always 1	Always $P = 1$	$\mathrm{Bel}(\Theta) = 1$

The key advantage of DS theory is its ability to represent *ignorance explicitly*: the mass on Θ as a whole is not yet committed to any sub-hypothesis. Ordinary probability theory forces this mass to be divided among singletons, which can introduce spurious precision.

8.7 Common Exam Traps Revisited

Before closing, it is worth gathering the principal conceptual hazards of this topic in one place.

(!) Exam Trap

Vagueness vs. uncertainty. These are the two distinct phenomena this chapter addresses. Vagueness is about the semantics of a predicate — its extension has no sharp boundary. Uncertainty is about an agent's knowledge — the fact is determinate, the agent just does not know it. Answering an exam question about "why fuzzy logic is needed" with "because we don't know the exact value" confuses the two.

(!) Exam Trap

Fuzzy AND = min, not multiplication (by default). The lecture's primary t-norm is min. The algebraic t-norm $x \cdot y$ is a valid alternative but gives a strictly smaller result when both arguments are in $(0, 1)$. Read the question carefully to see which t-norm is in use.

(!) Exam Trap

$A \vee \neg A$ **is not a fuzzy tautology.** $\max(\mu_A, 1 - \mu_A) = 1$ only at the boundary values 0 and 1. For $\mu_A = 0.6$: $\max(0.6, 0.4) = 0.6 \neq 1$.

(!) Exam Trap

Probabilistic logic does preserve tautologies. Every probability measure assigns probability 1 to tautological formulas and 0 to contradictions. This holds because the events corresponding to tautologies and contradictions are Ω and $\emptyset$ respectively.

(!) Exam Trap

Bel $\leq$ Pl, not the reverse. In Dempster–Shafer theory, belief is always a lower bound and plausibility is always an upper bound. The interval $[\mathrm{Bel}(A), \mathrm{Pl}(A)]$ widens with ignorance and collapses to a point when all mass is on singletons (recovering ordinary probability).

In a Nutshell

Chapter Summary

- **Vagueness** concerns predicates without sharp boundaries (*tall*, *warm*, *heap*); it is a semantic property of language, not a gap in the knower's information.
- **Uncertainty** concerns incomplete knowledge about a determinate fact; it is an epistemic property of the agent, not of the proposition.
- **Fuzzy set theory** models vagueness via membership functions $\mu_A : U \to [0, 1]$. Standard (Zadeh) operations: intersection = min, union

= max, complement = $1-\mu$. The laws of non-contradiction and excluded middle fail intentionally for vague predicates.

- **t-norms and s-norms** are the axiomatic generalisations of AND and OR in fuzzy logic. The Zadeh min/max pair is the largest t-norm; the algebraic pair ($x \cdot y$, $x + y - xy$) and the quotient pair are also valid. There are infinitely many valid families.
- **Probabilistic logic** models uncertainty via probability measures over possible worlds, satisfying Kolmogorov's three axioms. Tautologies always receive probability 1; contradictions always receive probability 0.
- **Conditional probability** $P(B \mid A) = P(B \cap A)/P(A)$ allows probabilistic knowledge bases to encode and propagate conditional degrees of belief.
- **Dempster–Shafer theory** (supplementary) generalises probability by assigning mass to *sets* of hypotheses. Belief $\mathrm{Bel}(A)$ and plausibility $\mathrm{Pl}(A)$ bracket the true probability, with the gap representing uncommitted evidence.
- The key exam trap: fuzzy logic and probabilistic logic both use $[0, 1]$ but they model fundamentally different things. Confusing them is the central conceptual error to avoid.

Chapter 9

Machine Learning I: Foundations and Methodology

9.1 What Is Machine Learning?

Classical AI programs encode knowledge explicitly: a human expert distils their understanding into rules, symbols, and logical relations that the system then applies mechanically. This approach works well for domains that are fully understood and cleanly formalised, but it breaks down wherever the relevant knowledge is hard to articulate—pattern recognition, speech and vision, low-level motor control, and countless others. Machine learning (ML) offers a different path: instead of programming the solution, we supply *data* and let the algorithm discover regularities on its own.

Tom Mitchell's widely cited definition captures the idea precisely. A computer program is said to learn from experience E with respect to some task T and performance measure P if its performance at T, as measured by P, improves with experience E. The three-way triple (T, P, E) is not merely a rhetorical device; it forces us to be precise about what we want the system to do, how we will know whether it is succeeding, and what data we will use to drive improvement. Consider learning to play checkers: the task T is playing checkers, the performance measure P is the percentage of games won in a tournament, and the experience E is playing against itself. The same template applies to a medical diagnosis system (T: assign diagnoses, P: fraction correctly classified, E: annotated patient records) or a speech recogniser (T: transcribe audio, P: word-error rate, E: labelled recordings).

Why has ML become so pervasive now? Two material conditions have shifted: the flood of digitally collected data has grown explosively, and the computational power available—both in raw CPU/GPU throughput and in distributed cloud infrastructure—has expanded to match. These conditions make it practical to train models on billions of examples, something that was purely theoretical twenty years ago.

Key Idea

Machine learning replaces hand-crafted rules with data-driven generalisation. A learner improves its performance measure P on task T through experience E. Without *inductive bias*—prior assumptions about the form of the solution—no generalisation from finite data is possible.

9.1.1 Inductive Bias

Any learning system that goes beyond memorising its training examples must make assumptions about which regularities are plausible. These assumptions are called the **inductive bias** of the learner. Formally, let D be a set of training data drawn from some universe U. Let L be a learner and let $L(x, D)$ denote its prediction for a new element $x \in U$ after training on D. The inductive bias of L is any minimal set of assertions B such that

$$\forall x \in U : B \wedge D \wedge x_i \models L(x, D),$$

where $a \models b$ means that b is deductively entailed by a. In other words, B is the prior knowledge the learner brings to the task that is *not* derived from the examples.

A learner with no inductive bias at all can do nothing but memorise: it has no basis on which to predict labels for unseen inputs. Every practical algorithm therefore encodes some bias, whether explicitly (a decision tree prefers short trees; a linear model assumes a linear decision surface) or implicitly (a neural network's architecture constrains the function class it can represent). The **Inductive Learning Hypothesis** states: any hypothesis that approximates the target function well over a sufficiently large training set will also approximate it well on previously unseen examples. This is the bedrock assumption that makes supervised learning coherent.

(!) Exam Trap

Inductive bias ≠ statistical bias. These two uses of the same word refer to entirely different concepts. Inductive bias is the set of prior assumptions a learner needs in order to generalise at all. Statistical bias (discussed in Section 9.4) is the systematic error that results from a model that is too simple to capture the true relationship in the data. Confusing them is a common exam error.

9.2 Types of Learning

The ML literature distinguishes three classical learning paradigms, differing in the kind of feedback available to the learner.

9.2.1 Supervised Learning

In supervised learning the training data consists of *labelled* examples: each input x_i is paired with a desired output y_i. The learner's goal is to find a function $h : X \to Y$ that maps inputs to outputs in a way that generalises beyond the training set. The metaphor of an "experienced teacher" is apt—the algorithm is guided at every step by correct answers.

Two sub-problems arise depending on the type of output:

- **Classification.** The output y is a discrete category. Examples: spam detection (spam / not spam), image recognition (dog / cat / car), medical diagnosis (disease present / absent). The learned function is called a *classifier* and its decision boundaries partition the input space into regions corresponding to each class.
- **Regression.** The output y is a continuous numerical value. Examples: predicting house prices from square footage and location, forecasting tomorrow's temperature, estimating a patient's blood glucose level. The learned function traces a curve or surface through the data.

9.2.2 Unsupervised Learning

Unsupervised learning operates on *unlabelled* data: the algorithm receives inputs $x_1, \ldots, x_n$ but no corresponding targets. The task is to discover intrinsic structure in the data.

Representative problems include:

- **Clustering.** Group similar examples together without predefined categories. k-means and hierarchical clustering are canonical algorithms; the output is a partition (or dendrogram) of the data.
- **Density estimation.** Learn the probability distribution that generated the data.
- **Dimensionality reduction.** Find a compact low-dimensional representation that preserves the essential structure—for instance, principal component analysis.

Because there are no labels to use as a training signal, success is harder to define and evaluation is often qualitative or task-specific.

9.2.3 Reinforcement Learning

Reinforcement learning (RL) lies between the two extremes. An *agent* interacts with an environment, observing state and taking actions; after each action the environment returns a scalar *reward*. The agent's goal is to learn a *policy*—a mapping from states to actions—that maximises cumulative reward over time. The environment acts as a "weak teacher": it tells the agent how well it did, but not what the correct action was. Classic applications include game playing (chess, Go, video games) and robot control, where hand-labelling every decision would be impractical.

Table 9.1: Comparison of the three classical learning paradigms.

Paradigm	**Training signal**	**Goal**	**Examples**
Supervised	Labelled pairs (x, y)	Learn $h : X \to Y$	Classification, regression
Unsupervised	Unlabelled inputs x	Find structure in data	Clustering, dim. reduction
Reinforcement	Scalar reward signal	Maximise cumulative reward	Game playing, robotics

Beyond these three, the lecture notes mention **semi-supervised learning** (only a fraction of examples carry labels) and **self-supervised learning** (the learner constructs its own supervision signal from the raw data, as modern language models do). Transfer learning, where knowledge gained on one task is reused on another, is increasingly important in practice.

9.3 Features and the ML Pipeline

Before any learning algorithm can operate, raw data must be converted into a numerical form it can process. This is the role of **features**.

9.3.1 Feature Spaces

The *vector space model* embeds each data point x in $\mathbb{R}^n$ via a representation function

$$r : D \to \mathbb{R}^n, \qquad r(x) = (f_1, f_2, \ldots, f_n).$$

The n dimensions are called *features*; they can be anything measurable—pixel intensities, word counts, sensor readings, engineered ratios. Good features are informative, compact, and robust to irrelevant variation. Poor feature choices doom an algorithm regardless of its sophistication, which is why feature engineering and, more recently, feature *learning* (letting the model discover its own representations) are central concerns.

For text data a standard choice is TF-IDF: each document becomes a vector whose i-th component is $\mathrm{tf}(t_i, d) \cdot \log(|D|/\mathrm{df}(t_i))$, where tf is the within-document term frequency and df is the number of documents containing the term. TF-IDF down-weights ubiquitous words such as "the" and highlights discriminative vocabulary. Similarity between documents is then measured with the cosine:

$$\mathrm{sim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|}.$$

Cosine similarity is length-invariant, which is appropriate for text (a long and a short document on the same topic should be similar).

9.3.2 Train, Validation, and Test Splits

A fundamental principle of empirical ML is that the data used to train a model must be *different* from the data used to evaluate it. If we measure performance on the training set, we obtain an optimistic and misleading estimate: the model may have memorised noise in the training data that is not present elsewhere.

The standard practice is therefore to partition the available data into three disjoint subsets:

1. **Training set.** The data the algorithm actually learns from—it adjusts its parameters to minimise error here.

2. **Validation set.** A held-out portion used to monitor performance during development, to choose among candidate models, and to tune hyperparameters (settings of the algorithm that are not learned directly from data, such as tree depth or regularisation strength).
3. **Test set.** A completely held-out portion that is touched *only once*, at the very end, to report final generalisation performance.

(!) Exam Trap

Never tune on the test set. If you use test-set performance to guide any modelling decisions— selecting features, setting hyperparameters, picking among architectures— the test set is no longer an unbiased estimate of how the model will perform on new data. The number you report will be optimistic, and the model may fail in deployment. This is one of the most common and consequential mistakes in applied ML.

9.3.3 Generalisation

The central goal of supervised learning is **generalisation**: performing well on inputs the model has never seen. The gap between training-set performance and test-set performance is the most direct indicator of how well a model generalises. A small gap (both quantities being satisfactory) indicates good generalisation; a large gap indicates that the model has specialised too closely to the training data.

9.4 Overfitting, Underfitting, and the Bias-Variance Tradeoff

The tension between model simplicity and model expressiveness is one of the central themes of ML theory and practice.

9.4.1 Overfitting and Underfitting

Overfitting occurs when a model is complex enough to memorise idiosyncratic details—including noise—in the training set. Such a model achieves very low training error but fails on fresh data, because the noise it memorised does not recur. The lecture illustrates this with a classification boundary: a highly convoluted boundary may separate all training points perfectly, but a new sample drawn from

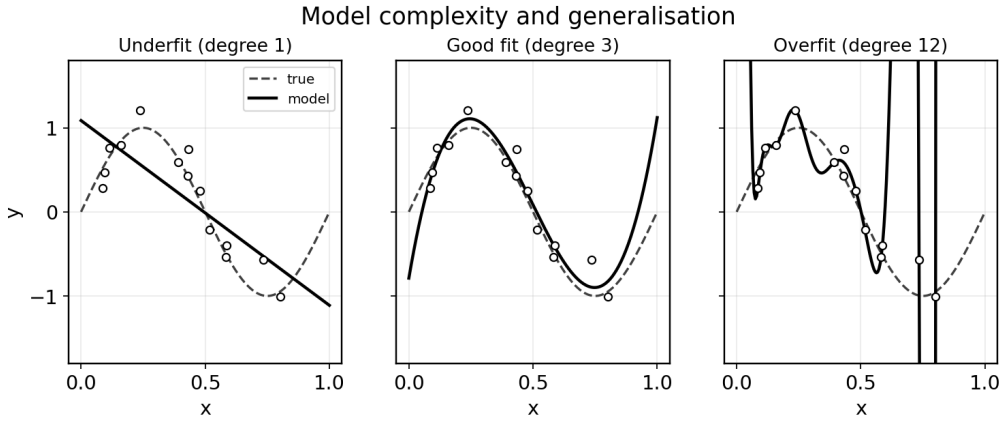

Figure 9.1: Underfitting, a good fit, and overfitting for polynomials of increasing degree.

the same distribution will see the boundary in different places and be misclassified. Occam's Razor—*prefer the simpler of two equally good explanations*—is the guiding intuition: unnecessary complexity should be penalised.

Underfitting is the opposite failure: a model that is too simple to capture the true structure in the data. A linear decision boundary applied to a genuinely non-linear problem, or a shallow polynomial fit to a complex curve, will perform poorly on both training and test data. Underfitting is sometimes overlooked but equally harmful.

9.4.2 The Bias-Variance Decomposition

The bias-variance decomposition provides a precise mathematical framework for understanding these failures. Consider a regression problem where the true relationship is $y = f(x) + \epsilon$, with ϵ being zero-mean noise of variance σ^2. We train a model $\hat{f}$ on a training set D drawn from this distribution. The expected squared error at a test point x decomposes as

$$\mathbb{E}\Big[(y - \hat{f}(x))^2\Big] = \text{Bias}^2(\hat{f}(x)) + \text{Var}(\hat{f}(x)) + \sigma^2,$$

where the three terms are defined and interpreted as follows.

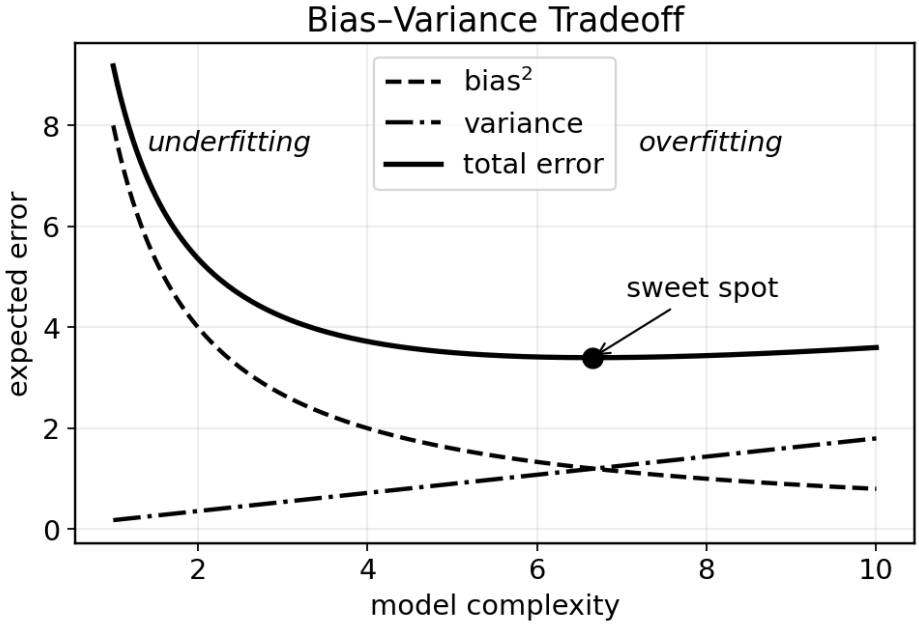

Figure 9.2: The bias–variance tradeoff: total error is minimised at intermediate model complexity.

Bias. The *bias* of an estimator measures systematic error—the average discrepancy between the model's predictions and the true function:

$$\text{Bias}(\hat{f}(x)) = \mathbb{E}[\hat{f}(x)] - f(x).$$

A high-bias model makes the same kind of mistake regardless of which training set it sees. This corresponds to **underfitting**: the model's hypothesis class is too restricted to represent the true relationship.

Variance. The *variance* of an estimator measures sensitivity to fluctuations in the training data:

$$\text{Var}(\hat{f}(x)) = \mathbb{E}\left[\left(\hat{f}(x) - \mathbb{E}[\hat{f}(x)]\right)^2\right].$$

A high-variance model changes dramatically when trained on different samples from the same distribution. This corresponds to **overfitting**: the model tracks noise rather than signal.

Irreducible noise. The term σ^2 is the variance of the noise in the data-generating process itself. No model, however expressive, can eliminate this error—it is irreducible.

The tradeoff. As model complexity increases (say, increasing the degree of a polynomial), bias tends to *decrease* (the model can represent more complex functions) while variance tends to *increase* (the model becomes more sensitive to the particular training sample). Total error is therefore a U-shaped function of complexity, with an optimal point somewhere in the middle. This is the bias-variance tradeoff.

Key Idea

$$\text{Total expected error} = \underbrace{\text{Bias}^2(\hat{f})}_{\text{underfitting}} + \underbrace{\text{Var}(\hat{f})}_{\text{overfitting}} + \underbrace{\sigma^2}_{\text{irreducible noise}}$$

High bias ⇒ model too simple (underfitting). High variance ⇒ model too complex (overfitting). The irreducible noise σ^2 sets a floor on achievable error.

(!) Exam Trap

It is a persistent exam trap to conflate *high bias* with *high variance*. Remember: high bias means the model is systematically wrong in the same way—it *underfits*. High variance means the model is right on average but wildly inconsistent across training sets—it *overfits*.

9.4.3 Avoiding Overfitting

The lecture identifies several complementary strategies. First, *detecting* overfitting requires holding out a validation set and comparing training-set and validation-set performance; a large gap is the warning sign. Once detected, overfitting can be addressed by:

- using a less complex model (reducing the hypothesis class);
- **regularisation**—artificially constraining model capacity, for instance by penalising large parameter values (ℓ_2 or ℓ_1 regularisation), by removing or deactivating parts of the model (dropout in neural networks), or by injecting

noise during training;

- collecting more training data, which dilutes the influence of individual noisy examples.

9.5 Model Selection and Cross-Validation

A single train/validation/test split is sensitive to which particular examples happen to end up in each subset. With limited data, this randomness can substantially distort our estimate of a model's true generalisation ability. Cross-validation addresses this by reusing the data more efficiently.

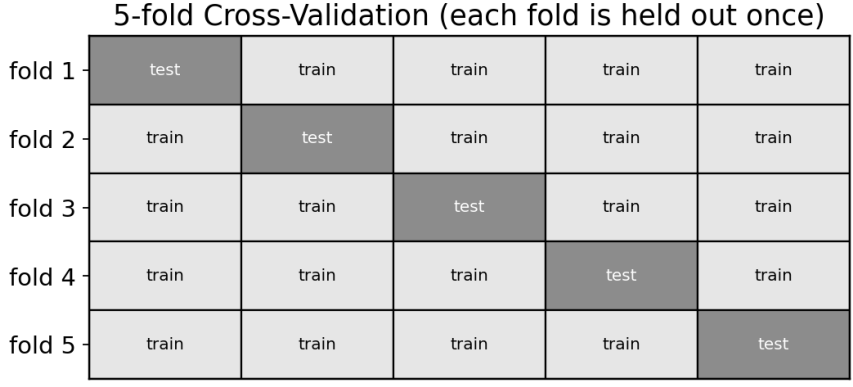

Figure 9.3: k-fold cross-validation: each fold serves once as the validation set.

9.5.1 k-Fold Cross-Validation

In **k-fold cross-validation** the training data is partitioned into k equally sized non-overlapping subsets called *folds*. The procedure runs k rounds. In round i, fold i is held out as the validation set and the model is trained on the remaining $k-1$ folds. This yields k validation-error estimates. The cross-validation error is their average:

$$\mathrm{CV}(k) = \frac{1}{k} \sum_{i=1}^{k} \mathrm{Error}_i.$$

Typical choices for k are 5 and 10. The final reported model is usually retrained on the entire training dataset using the hyperparameters selected by cross-validation.

9.5.2 Leave-One-Out Cross-Validation

Leave-one-out cross-validation (LOO-CV) is the special case $k = n$, where n is the number of training examples. Each round holds out exactly one example and trains on the rest. LOO-CV is essentially unbiased as an estimate of generalisation error because the training set is as large as possible, but it is computationally expensive (requiring n complete training runs) and can have high variance. For large datasets, k-fold CV is preferred.

Key Idea

Cross-validation serves two purposes: **model selection** (choosing among different algorithms or architectures) and **hyperparameter tuning** (finding the best settings for a given algorithm). In both cases, the test set must remain untouched until the very end. Cross-validation operates entirely within the training portion of the data.

(!) Exam Trap

Do not use CV to evaluate on the test set. Cross-validation is a method for estimating generalisation error within the training data, used to select models and tune hyperparameters. It does not replace the held-out test set. After CV selects a model, retrain on all training data and then evaluate once on the test set.

9.6 Evaluation Metrics

Choosing the right performance measure is as important as choosing the right algorithm.

9.6.1 The Confusion Matrix

For binary classification, all outcomes can be summarised in a 2×2 **confusion matrix**:

	Predicted positive	**Predicted negative**
Actually positive	True Positive (TP)	False Negative (FN)
Actually negative	False Positive (FP)	True Negative (TN)

Every scalar metric is a function of these four counts.

9.6.2 Accuracy, Precision, and Recall

Accuracy is the fraction of all examples classified correctly:

$$\mathrm{acc} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}.$$

Accuracy is intuitive but can be misleading when classes are imbalanced. If 95% of emails are not spam, a classifier that always predicts "not spam" achieves 95% accuracy while being completely useless.

Precision measures the fraction of positive predictions that are correct—it quantifies how much we can trust a positive prediction:

$$\mathrm{pre}(h) = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}.$$

Recall (also called *sensitivity*) measures the fraction of actual positives that are detected—it quantifies completeness:

$$\mathrm{rec}(h) = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

There is an inherent tradeoff between precision and recall. Increasing the decision threshold of a classifier raises precision (fewer false positives) but lowers recall (more positives are missed), and vice versa.

F_1 **score** is the harmonic mean of precision and recall, giving a single number that balances both:

$$F_1(h) = \frac{2\,\mathrm{TP}}{2\,\mathrm{TP} + \mathrm{FP} + \mathrm{FN}}.$$

The harmonic mean is appropriate here because it is dominated by the *smaller* of the two quantities: an F_1 score is high only when both precision and recall are high. The parameterised F_β score weights recall β times as heavily as precision, useful when false negatives are more costly than false positives (e.g., medical screening).

9.7 Worked Example: Hypothesis Evaluation

Worked Example

Setup. We have ten data points $d_1, \ldots, d_{10}$, of which four are positive (green) and six are negative (red). Evaluate the following four hypotheses.

h_1 predicts *all* ten points positive: [TP = 4, TN = 0, FP = 6, FN = 0].

$$\text{pre}(h_1) = \tfrac{4}{10} = 0.4, \quad \text{rec}(h_1) = \tfrac{4}{4} = 1.0, \quad \text{acc}(h_1) = \tfrac{4}{10} = 0.4.$$

h_2 predicts *nothing* positive (the empty hypothesis): [TP = 0, TN = 6, FP = 0, FN = 4].

$$\text{pre}(h_2) = \text{undefined}, \quad \text{rec}(h_2) = 0, \quad \text{acc}(h_2) = \tfrac{6}{10} = 0.6.$$

Note that h_2 achieves higher accuracy than h_1 while having zero recall—a reminder that accuracy alone is insufficient.
h_4 correctly identifies all four positives but also includes two false positives: [TP = 4, TN = 4, FP = 2, FN = 0].

$$\text{pre}(h_4) = \tfrac{4}{6} \approx 0.67, \quad \text{rec}(h_4) = 1.0, \quad \text{acc}(h_4) = \tfrac{8}{10} = 0.8.$$

h_4 dominates h_1 on all metrics except precision (same recall, higher accuracy). Among these hypotheses h_4 is the best: it is *complete* (covers all positives) but only moderately *correct* (admits two false positives). The general strategy for hypothesis improvement is: *specialise an incorrect hypothesis* (to eliminate false positives) and *generalise an incomplete hypothesis* (to recover false negatives).

9.8 Putting It All Together: The Standard ML Workflow

A typical supervised learning project proceeds through the following stages.

1. **Problem formulation.** Define the task T, choose a performance measure P, and identify what experience E is available.
2. **Data collection and preprocessing.** Gather raw data, handle missing values, remove or flag outliers, and transform inputs into a suitable form.

3. **Feature engineering.** Construct the feature vectors $r(x) \in \mathbb{R}^n$ that the algorithm will consume. Normalise features if necessary (many algorithms are sensitive to scale).
4. **Data splitting.** Divide data into training, validation, and test sets. Keep the test set sealed.
5. **Model selection and training.** Choose a hypothesis class (decision tree, linear model, neural network, ...), train on the training set, and use cross-validation on the training portion to tune hyperparameters.
6. **Evaluation.** Evaluate the selected model once on the test set and report the appropriate metric (accuracy, F_1, mean squared error, etc.).
7. **Iteration.** If performance is insufficient, return to an earlier stage—collect more data, engineer better features, choose a different model family, or adjust regularisation.

The bias-variance tradeoff provides the diagnostic framework at step 6: if training error is high, the model underfits (increase capacity or improve features); if training error is low but test error is high, the model overfits (regularise, simplify, or collect more data).

In a Nutshell

Chapter Summary: Machine Learning I

- **ML definition (Mitchell):** a program learns if its performance P on task T improves with experience E.
- **Three paradigms:** supervised (labelled data, learn $h : X \to Y$), unsupervised (unlabelled data, find structure), reinforcement (reward signal, learn a policy).
- **Classification vs. regression:** discrete vs. continuous output in the supervised setting.
- **Inductive bias:** every learner requires prior assumptions to generalise; bias-free learners can only memorise.
- **Data splits:** training $\to$ learn parameters; validation $\to$ tune hyperparameters; test $\to$ report final performance (touch once only).
- **Bias-variance tradeoff:** Error $=$ Bias^2 $+$ Variance $+ \sigma^2$. High bias $\Leftrightarrow$ underfitting; high variance $\Leftrightarrow$ overfitting.
- **Cross-validation:** k-fold CV reuses training data to estimate generali-

sation error and select models without touching the test set.

- **Metrics:** accuracy, precision, recall, F_1 score all derive from the confusion matrix (TP, TN, FP, FN).
- **Key exam traps:** inductive bias $\neq$ statistical bias; high bias = underfit; high variance = overfit; never tune on test set; CV operates within training data only.

9.9 Going Further: Oral-Exam Deep Cuts

Q: Can a model have high bias *and* high variance at the same time?
A: Yes. They are independent error components. A poorly chosen model can both systematically miss the target (bias) *and* swing wildly with the training sample (variance) — e.g. a high-degree polynomial fit to too few, mis-specified features. The "tradeoff" describes the typical tension as you vary capacity, not a law that one must be low.

Q: Why does k-fold cross-validation give a better generalisation estimate than a single train/validation split?
A: A single split wastes data and gives a high-variance estimate that depends on which points happened to land in validation. k-fold averages the error over k disjoint validation folds, so every point is used for both training and validation (in different folds), reducing the variance of the estimate and using the data more efficiently.

Q: Why is tuning hyperparameters on the test set "cheating," and what is the correct setup?
A: Because once you select a model by its test performance, the test set has influenced the model and no longer estimates *unseen* performance — you get an optimistically biased number (information leakage). The correct setup is a three-way split (or nested CV): train on training data, select on validation, and report *once* on a test set you never touched during development.

Q: "A bias-free learner that makes no assumptions would be ideal." React.
A: False — this is the point of *inductive bias* and the No-Free-Lunch idea. With no assumptions, every hypothesis consistent with the training data is equally plausible, so the learner cannot generalise to unseen inputs at all. Useful learning

requires a bias (e.g. a preference for simpler hypotheses); the art is choosing one matched to the problem.

Q: When would you optimise for precision rather than recall, and vice versa?
A: Precision matters when false positives are costly (e.g. spam filter deleting real mail); recall matters when false negatives are costly (e.g. screening for a serious disease). The F_1 score balances them as a harmonic mean, which punishes a model that sacrifices one for the other.

Chapter 10

Machine Learning II: Decision Trees and Ensembles

Decision trees and ensemble methods represent two of the most practically successful families of supervised learning algorithms. Decision trees are attractive because they produce transparent, human-readable models — a learned tree can be inspected, printed, and explained to a domain expert. Ensemble methods, and random forests in particular, trade away some of that transparency in exchange for substantially better generalisation performance. Understanding both, and especially understanding *why* ensembles outperform single trees, is a central theme of this chapter.

10.1 Decision Trees

10.1.1 Axis-Aligned Splits and Recursive Partitioning

A decision tree partitions the feature space by a sequence of binary tests, each of which compares a single feature against a threshold. Because every test involves only one feature at a time, the resulting boundaries are axis-aligned — they run parallel to the coordinate axes. The geometric picture is helpful: each internal node carves the current region of feature space into two sub-regions by a hyperplane perpendicular to one coordinate axis. After enough splits the leaf regions are (ideally) pure with respect to the target class.

The learning algorithm proceeds recursively. Given a set of labelled examples and a set of candidate features, the algorithm chooses the feature that best

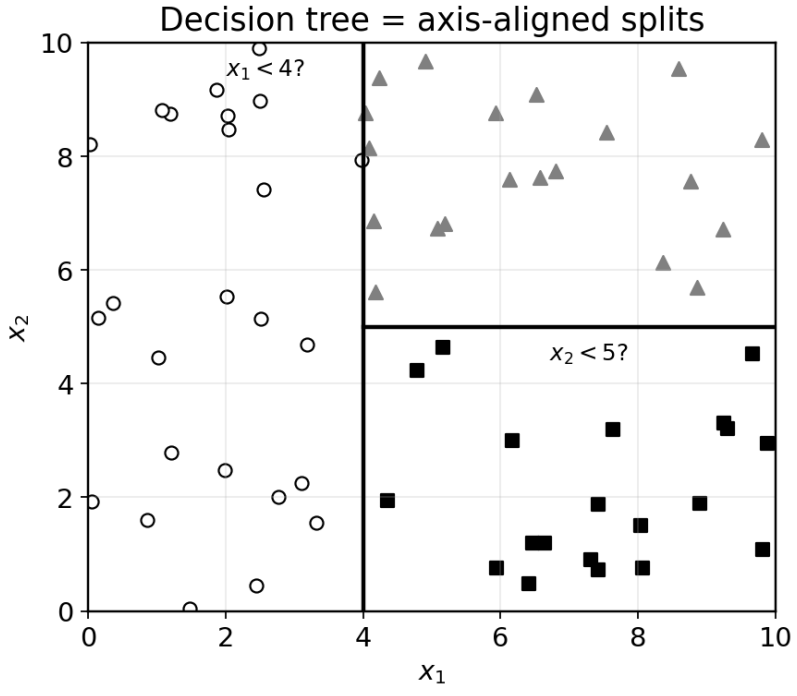

Figure 10.1: A decision tree carves the feature space into axis-aligned rectangles; each internal node tests one feature against a threshold.

separates the classes, creates one branch for each possible value (or, for continuous features, two branches for $\leq$ and $>$ the chosen threshold), and then recurses on each subset. The recursion terminates when a node becomes *pure* (all examples have the same label), when no features remain, or when the node is too small to split further. The formal pseudocode from the lecture reads:

1. If the example set is empty, return the majority class of the parent node.
2. If all examples share the same class label, return a leaf with that label.
3. If the attribute set is empty, return a leaf with the majority class.
4. Otherwise, choose the best attribute A, add a test node, and recurse on each branch with the remaining attributes.

This greedy top-down construction is the basis of the **ID3** algorithm (Quinlan, 1986). ID3 embodies a particular *inductive bias*: it prefers shorter trees and trees that place high-information-gain attributes near the root. The search through

hypothesis space is breadth-first and greedy — once a split is made, it is never revisited.

10.1.2 Splitting Criteria

The key design choice in any tree learner is the *splitting criterion*: how do we measure the quality of a potential split? Two measures dominate the literature and the exam: **entropy / information gain** (used by ID3) and **Gini impurity** (used by CART, Breiman 1983).

Entropy and Information Gain

Entropy comes from information theory. For a node containing examples drawn from c classes, let p_i be the proportion of examples belonging to class i. The entropy of the node is

$$H(S) \;=\; -\sum_{i=1}^{c} p_i \log_2 p_i$$

where, by convention, $0 \log_2 0 = 0$. Entropy measures the *impurity* of a node: it equals zero when the node is perfectly pure (one class dominates completely) and reaches its maximum of $\log_2 c$ when all classes are equally represented. For binary classification it peaks at 1 bit when $p_1 = p_2 = 0.5$.

Key Idea

Entropy $H(S) = 0$ means the node is pure — no uncertainty about the class label. Entropy $H(S) = \log_2 c$ means maximum uncertainty (uniform distribution over c classes). The tree learner always tries to reduce entropy.

To evaluate a candidate split on attribute A, we compute the **information gain**:

$$\text{Gain}(S,\, A) \;=\; H(S) \;-\; \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|}\, H(S_v)$$

Here S_v is the subset of examples for which attribute A takes value v, and $|S_v|/|S|$ weights each child's entropy by its relative size. Information gain is therefore the *expected reduction in entropy* obtained by learning the value of A. ID3 always splits on the attribute with the highest information gain.

Gini Impurity

An alternative to entropy is **Gini impurity**:

$$\text{Gini}(S) \;=\; 1 - \sum_{i=1}^{c} p_i^2$$

Gini impurity can be interpreted as the probability that a randomly chosen example would be misclassified if it were labelled randomly according to the class distribution in S. Like entropy, it is zero for a pure node and maximal for a uniform distribution. Computationally, Gini is slightly cheaper (no logarithms), and CART uses it by default. In practice the two criteria produce very similar trees on most datasets — the choice rarely matters much for final accuracy.

(!) Exam Trap

Exam trap: the lecture introduces both criteria. If asked "which criterion does ID3 use?", answer *information gain (entropy).* If asked about CART, answer *Gini impurity.* Do not confuse Gini impurity with the Gini coefficient from economics — they are different quantities.

10.2 A Worked Information-Gain Computation

The lecture provides a concrete example that is worth working through carefully. The dataset contains $n = 12$ examples and a binary target ("go to party": yes or no), with 8 positive and 4 negative examples. We evaluate two candidate attributes: **tired** (yes/no) and **raining** (yes/no).

Step 1 — Root entropy.

$$H(S) \;=\; -\frac{4}{12}\log_2\frac{4}{12} - \frac{8}{12}\log_2\frac{8}{12} \;\approx\; 0.918 \text{ bits}$$

Step 2 — Entropy of children under *tired*. The 4 examples with `tired=yes` are all negative ($p_{\text{no}} = 1$), so $H(S_{\text{yes}}) = 0$. The 8 examples with `tired=no` split evenly (4/4), so $H(S_{\text{no}}) = 1$.

Step 3 — Information gain for *tired*.

$$\text{Gain}(S, \text{tired}) \;=\; 0.918 - \frac{4}{12}\cdot 0 - \frac{8}{12}\cdot 1 \;=\; 0.918 - 0.667 \;\approx\; 0.251 \text{ bits}$$

Step 4 — Similarly for *raining*. The lecture computes Gain(S, raining) $\approx$ 0.168 bits.

Conclusion. ID3 selects `tired` as the root split because $0.251 > 0.168$. This illustrates the general principle: an attribute that produces very pure children (even if one child is large and impure) can still outscore an attribute that produces moderate purity everywhere.

Worked Example

Worked Example — Gini check on the same root. Using Gini at the root: $\text{Gini}(S) = 1 - (4/12)^2 - (8/12)^2 = 1 - 0.111 - 0.444 = 0.444$. After splitting on `tired`: $\Delta\text{Gini} = 0.444 - (4/12) \cdot 0 - (8/12) \cdot 0.5 = 0.444 - 0.333 = 0.111$. After splitting on `raining` one similarly obtains a smaller reduction. Both criteria agree: `tired` is the better split.

10.3 Overfitting and Pruning

A fully grown decision tree — one that keeps splitting until every leaf is pure — will memorise the training data perfectly. On unseen data, however, it tends to perform poorly because it has learned idiosyncratic noise in the training set rather than the underlying pattern. This is *overfitting*: the model has high variance and low bias. Deep trees are especially prone to this because they can represent very complex, jagged decision boundaries that happen to fit training noise.

The lecture states this directly: "Deep decision trees tend to learn irregular patterns (overfitting)." Two standard remedies exist.

Depth limiting (pre-pruning). One simply stops growing the tree once it reaches a maximum depth $d_{\max}$, or once the number of examples in a node falls below a threshold. This prevents the tree from ever becoming complex enough to overfit, at the cost of possibly underfitting if $d_{\max}$ is set too aggressively.

Post-pruning. The tree is first grown to full depth, then branches that do not improve generalisation performance (estimated on a held-out validation set, or via cross-validation) are collapsed back to leaves. The leaf then predicts the majority class of all training examples that fell into the pruned subtree. Post-pruning tends to produce better results than pre-pruning because it can observe the actual structure of the data before deciding what to remove.

In practice, depth limiting is computationally simpler and is the default in most libraries. The deeper remedy — building many trees and averaging them —

is the subject of the next section.

10.4 Ensemble Methods

An ensemble combines the predictions of multiple models to produce a single output. The central intuition is that independent errors made by different models cancel when averaged: if each model errs on different examples, the majority vote (for classification) or average (for regression) is more reliable than any individual model. Two ensemble strategies are directly relevant to this course: **bagging** and **random forests**.

10.4.1 Bagging: Bootstrap Aggregating

Bagging was introduced by Breiman (1996). The idea is simple:

1. Draw m bootstrap samples $D_1, \ldots, D_m$ from the original training set D of size n. Each D_i is formed by sampling n examples *with replacement* from D.
2. Train a separate full decision tree on each D_i.
3. For a new input $\mathbf{x}$, let each tree vote. Return the majority class (classification) or the mean prediction (regression).

Because sampling is done *with replacement*, each bootstrap sample includes on average about 63.2% of the original training examples (at least once), with the remaining 36.8% never selected. This 36.8% figure follows from $(1 - 1/n)^n \to e^{-1} \approx 0.368$ as $n \to \infty$.

Why Bagging Reduces Variance

This is one of the most important theoretical points in the lecture and a classic exam topic. To understand it, recall the bias-variance decomposition. A deep decision tree has *low bias* (it can fit complex boundaries) but *high variance* (small changes in the training set lead to very different trees). Bagging does not change the bias of the individual models — each tree is still grown to full depth and is still capable of overfitting. What bagging does is *average* the predictions of many such trees.

If m models each have variance σ^2 and their errors are uncorrelated (independent), then the variance of the average prediction is σ^2/m. The more trees, the

lower the variance of the ensemble. In practice the trees are not fully independent — they are all trained on samples from the same D — but even partial decorrelation yields a substantial reduction in variance.

(!) Exam Trap

Exam trap: **bagging reduces variance, not bias.** A single deep tree already has low bias. Averaging many such trees preserves that low bias while suppressing the high variance. If you have an underfitting (high-bias) tree, bagging will not fix it — you need a more powerful base learner, not more copies of a weak one.

10.4.2 Random Forests

Random forests (Ho, 1995; Breiman and Cutler, 2001) add one further ingredient on top of bagging: **random feature subset selection at each split**.

When growing each tree in a bagged ensemble, the usual procedure considers all p features at every split. This means that if one or two features are highly informative, they will appear near the root of almost every tree. The resulting trees are highly correlated with one another, which limits how much variance reduction the ensemble can achieve.

Random forests break this correlation by a simple device: at each candidate split, instead of evaluating all p features, the algorithm randomly samples k features and considers only those. A common rule of thumb is $k = \lfloor \sqrt{p} \rfloor$ for classification and $k = \lfloor p/3 \rfloor$ for regression. Because different splits — and therefore different trees — see different random subsets of features, the resulting trees disagree more with one another. They are more *diverse*. Averaging diverse-but-still-accurate models yields much lower ensemble variance than averaging correlated models.

The lecture puts it clearly: "Individual classifiers in the ensemble should be both accurate and diverse. To build a good ensemble, one needs a way to make sure that individual classifiers are substantially different from one another while still having at least some accuracy on their own."

Key Idea

Random Forest = Bagging + random feature subsampling per split. Bagging decorrelates trees by training on different bootstrap samples. Feature subsampling further decorrelates trees by preventing the strongest feature from dominating every split in every tree.

10.4.3 Out-of-Bag Error

A convenient side effect of bootstrap sampling is that each tree is trained on roughly 63.2% of the data; the remaining $\approx 36.8\%$ — the **out-of-bag (OOB)** examples — were never seen by that tree during training. For each training example x_i, we can therefore collect the predictions of all trees for which x_i was OOB and aggregate them. This gives a generalisation estimate that is essentially free: no separate validation set is needed, and no data is wasted on validation.

(!) Exam Trap

Exam trap: OOB error is a *free* generalisation estimate because it uses examples the individual tree never trained on. It is not the same as a train error, and it is not computed by holding out a separate set manually. Approximately 37% of training examples are OOB for any given tree.

10.4.4 Feature Importance

Because random forests aggregate many trees, they also provide a useful diagnostic: **feature importance**. The most common measure counts, for each feature, how much it reduces impurity (entropy or Gini) across all splits in all trees, weighted by the fraction of examples that pass through each split. Features that are used near the root of many trees and that produce large impurity reductions are ranked as most important. This gives practitioners a rough ranking of which input variables are predictive, which is valuable for feature selection and for understanding the problem.

10.5 Comparison: Single Tree vs. Bagging vs. Random Forest

Property	**Single Tree (ID3)**	**Bagging**	**Random Forest**
Training procedure	One tree on full data	m trees, each on a bootstrap sample	m trees, bootstrap + random feature subset per split
Variance	High (deep trees)	Lower — averaging reduces variance	Lowest — extra decorrelation from feature subsampling
Bias	Low (deep trees)	Same as single tree	Slightly higher than single tree (fewer features at each split)
Interpretability	Full (white box)	Reduced — no single tree to inspect	Low (black box ensemble)
Overfitting risk	High	Moderate	Low
OOB error available	No	Yes (bootstrap by design)	Yes
Feature importance	Structural only	Marginal (all features used)	Built-in impurity-based ranking
Hyperparameters	Depth, min-samples	Number of trees m	m, features per split k, depth, min-samples, ...
Typical performance	Baseline	Better than single tree	State of the art for tabular data

The table makes the progression clear. Moving from a single tree to bagging cuts variance without changing bias. Moving from bagging to a random forest cuts variance further by decorrelating the individual trees. The price paid is a loss

of interpretability: there is no longer a single human-readable tree to examine.

10.6 Exam Traps and Common Pitfalls

This section collects the most frequently tested points and the mistakes students commonly make.

(!) Exam Trap

Bagging reduces variance, not bias. This cannot be overstated. Students often confuse bagging with boosting (which reduces bias). Bagging averages high-variance, low-bias models to get a low-variance, still-low-bias ensemble. A high-bias model will still have high bias after bagging.

(!) Exam Trap

Random Forest $\neq$ Bagging. The distinguishing feature of random forests is the random feature subset at each split. Bagging alone uses all features at every split. If an exam question asks what makes random forests different from simple bagged trees, the answer is: *random feature subsampling per split.*

(!) Exam Trap

OOB definition. Out-of-bag examples are those not included in a particular bootstrap sample. Since sampling is with replacement, on average $\approx 36.8\%$ of training examples are OOB for each tree. The OOB error is computed by averaging the predictions of all trees for which each example was out-of-bag. It is a free generalisation estimate — no held-out set is required.

(!) Exam Trap

Entropy uses $\log_2$. The formula $H(S) = -\sum_i p_i \log_2 p_i$ measures entropy in *bits*. Using ln gives entropy in nats; the choice of base is a convention and does not change which attribute has the highest information gain, but numerical answers depend on the base. Unless otherwise specified, assume

base 2 in this course.

(!) Exam Trap

High-cardinality attributes game information gain. An attribute like a unique customer ID has maximum information gain on the training set because it perfectly splits each example into its own leaf — but it generalises terribly. This is a known weakness of vanilla ID3. C4.5 addresses it with the *gain ratio*, which penalises attributes with many values, but this extension is not covered in the lecture slides.

(!) Exam Trap

ID3 is greedy with no backtracking. Once ID3 commits to a split, it never reconsiders it. The result may not be the globally optimal tree. This is another reason ensembles are preferred in practice: they implicitly explore many different split orderings by training on different data samples.

In a Nutshell

Chapter Summary — Machine Learning II

Decision trees recursively partition feature space via axis-aligned splits. The ID3 algorithm selects the attribute with the highest *information gain* at each node. Information gain $= H(\text{parent}) - \sum_k \frac{n_k}{n} H(\text{child}_k)$, where $H(S) = -\sum_i p_i \log_2 p_i$. Gini impurity $= 1 - \sum_i p_i^2$ is an alternative that avoids logarithms. Deep trees overfit; remedies include depth limiting and post-pruning.

Bagging (bootstrap aggregating) trains m trees on independent bootstrap samples and averages their predictions. It reduces *variance* (not bias) by averaging decorrelated models.

Random forests add random feature subsampling at each split on top of bagging. This further decorrelates the trees, yielding lower ensemble variance than plain bagging. The OOB error provides a free generalisation estimate: $\approx 37\%$ of training examples are out-of-bag for each tree.

Key exam facts: (1) bagging reduces variance, not bias; (2) the extra ingredient in RF is per-split random feature subsampling; (3) OOB $\approx 37\%$;

(4) entropy uses $\log_2$.

10.7 Going Further: Oral-Exam Deep Cuts

Q: Show *why* bagging reduces variance but not bias.
A: Averaging B models, each of variance σ^2 with pairwise correlation ρ, gives ensemble variance $\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2$. The second term vanishes as $B \to \infty$, so variance falls. But all base models share the same distribution, hence the same expected prediction: $\mathbb{E}[\text{ensemble}] = \mathbb{E}[\text{single model}]$, so the bias is unchanged. That is why you bag *low-bias, high-variance* learners (deep unpruned trees).

Q: What is the *one* thing a random forest adds over bagging, and why does it help?
A: Per-split *random feature subsampling* (consider only $\approx \sqrt{p}$ features at each split). In the variance formula, the irreducible floor is $\rho\sigma^2$. Plain bagging leaves trees strongly *correlated* (a dominant feature splits near the root of almost every tree), so ρ is large. Feature subsampling *decorrelates* the trees, lowering ρ and pushing the floor down. So: bagging averages; RF averages *and* decorrelates.

Q: Derive the $\approx 37\%$ out-of-bag figure.
A: A bootstrap sample draws n items with replacement. The probability a given item is *never* chosen is $(1 - 1/n)^n \to e^{-1} \approx 0.368$. So $\approx 37\%$ of examples are out-of-bag for each tree; evaluating each example only on the trees that did *not* see it yields a free, CV-like generalisation estimate.

Q: Information gain is biased — toward what, and how is it fixed?
A: Toward high-cardinality (many-valued) attributes: splitting on a near-unique ID maximises information gain but generalises terribly. The fix is the *gain ratio*, which normalises information gain by the intrinsic entropy of the split itself, penalising splits with many tiny branches.

Q: Can a random forest overfit? Does adding more trees overfit?
A: Adding more trees does *not* overfit — the ensemble error converges as B grows (more trees only reduce estimation variance). A forest *can* still overfit if the individual trees are grown on too little data or the features are very noisy, but the classic "too many trees" worry is unfounded.

Chapter 11

Machine Learning III: Support Vector Machines and Kernels

This chapter covers the two classical linear classifiers treated in Session 11: the perceptron and support vector machines (SVMs). The perceptron serves as a conceptual warm-up — a single-neuron model whose limitations motivate the design of the SVM. SVMs then push linear classification to its theoretical optimum by choosing the hyperplane that *maximises the margin*, and they extend far beyond linear problems through the kernel trick. The second half of the chapter is devoted to a thorough treatment of every important kernel introduced in the lecture: linear, polynomial, and radial basis function (RBF), together with a brief look at tree kernels for structured data.

11.1 The Perceptron

11.1.1 Biological Motivation and the Basic Model

The perceptron (Rosenblatt, 1958) is one of the earliest and most influential models in supervised learning. Its design is directly inspired by the neuron: a biological neuron receives weighted inputs along many dendrites, sums them up, and fires an output along a single axon when the total input exceeds a threshold. The perceptron abstracts this as follows. Given a d-dimensional input vector $\vec{x} \in \mathbb{R}^d$, each component x_i is multiplied by a synaptic weight $w_i \in \mathbb{R}$. A bias weight w_0 acts as a trainable threshold. For notational compactness one augments

the input with a constant component:

$$\vec{x} = (1,\, x_1,\, x_2,\, \ldots,\, x_d)^\top \in \mathbb{R}^{d+1}, \qquad \vec{w} = (w_0,\, w_1,\, w_2,\, \ldots,\, w_d)^\top \in \mathbb{R}^{d+1}.$$

The **preactivation** is then simply the dot product

$$s = \vec{x} \cdot \vec{w} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d,$$

and the **output** applies the Heaviside step function:

$$y = \Theta(s) = \begin{cases} 1 & \text{if } s \geq 0, \\ 0 & \text{otherwise.} \end{cases}$$

The output $y \in \{0, 1\}$ assigns a binary class label to the input. The decision boundary is the $(d-1)$-dimensional hyperplane orthogonal to $\vec{w}$: points with $\vec{x} \cdot \vec{w} > 0$ are mapped to class 1, all others to class 0.

11.1.2 Perceptron Learning Rule

Training uses a labelled dataset $\mathcal{D} = \{(\vec{x}^n, t^n)\}_{n=1}^{|\mathcal{D}|}$, where $t^n \in \{0, 1\}$ is the target label provided by the teacher and $\vec{x}^n$ is the n-th training example. The learning rule is derived by minimising the mean squared error

$$E[\vec{w};\, \mathcal{D}] = \tfrac{1}{2} \sum_{n=1}^{|\mathcal{D}|} \big(t^n - y(\vec{x}^n)\big)^2.$$

Because Θ is not differentiable, the gradient is taken with respect to the preactivation rather than the output. The result is the classical **perceptron update rule**:

$$\boxed{\Delta \vec{w} = \epsilon \sum_{n=1}^{|\mathcal{D}|} \big(t^n - y(\vec{x}^n)\big)\, \vec{x}^n}$$

where $\epsilon > 0$ is the **learning rate** (a small positive constant), t^n is the target for example n, $y(\vec{x}^n) = \Theta(\vec{x}^n \cdot \vec{w})$ is the current prediction, and $\vec{x}^n$ is the input vector. The update is applied in **batch mode** (sum over all examples at once) or **stochastic mode** (one randomly chosen example at a time). For sufficiently small ϵ, stochastic gradient descent approximates batch gradient descent arbitrarily closely.

11.1.3 Linear Separability, Convergence, and XOR

Key Idea

The perceptron converges to a correct solution *if and only if* the training data are linearly separable and the learning rate ϵ is sufficiently small. For non-linearly separable data the weights cycle without converging.

The perceptron implements a hyperplane classifier and can therefore represent AND, OR, NAND, and NOR on Boolean inputs. It cannot, however, implement XOR, because the two XOR classes are not linearly separable — there is no straight line that puts both "1" outputs on one side. One fix is to add a hand-crafted product feature $x_3 = x_1 \cdot x_2$, turning XOR into a linearly separable problem in a three-dimensional feature space. This insight — that the right choice of features unlocks linear separability — foreshadows the kernel trick developed in the SVM section below. The XOR failure triggered a temporary decline of interest in artificial neural networks in the 1970s; it was only the introduction of multi-layer perceptrons with backpropagation that resolved the issue at scale.

(!) Exam Trap

The perceptron output is $\Theta(s) \in \{0, 1\}$ (Heaviside), *not* $\text{sign}(s) \in \{-1, +1\}$. SVMs use sign; confusing the two is a common exam error. Also: the bias w_0 is absorbed into the weight vector via the constant input $x_0 = 1$, so no separate threshold variable is needed.

11.2 Support Vector Machines: The Core Idea

11.2.1 From Linear Separation to Maximum Margin

SVMs were introduced by Vapnik and Lerner in 1963 as the "Generalized Portrait" algorithm and later extended to non-linear classifiers. Like the perceptron, an SVM finds a separating hyperplane; unlike the perceptron, it selects the *unique* hyperplane that maximises the distance to the nearest training points of either class — the **maximum-margin hyperplane**.

Formally, training data are pairs (x_i, y_i) with $x_i \in \mathbb{R}^n$ and $y_i \in \{+1, -1\}$. A separating hyperplane is defined by a weight vector $\vec{w} \in \mathbb{R}^n$ and a bias $b \in \mathbb{R}$:

$$H = \{\vec{x} \in \mathbb{R}^n \mid \langle \vec{w}, \vec{x} \rangle + b = 0\}.$$

Points with $\langle \vec{w}, \vec{x} \rangle + b > 0$ belong to the positive class; points with $\langle \vec{w}, \vec{x} \rangle + b < 0$ to the negative class. Two parallel hyperplanes, one tangent to the closest positive example and one tangent to the closest negative example, satisfy:

$$\langle \vec{w}, x_1 \rangle + b = +1 \quad \text{and} \quad \langle \vec{w}, x_2 \rangle + b = -1.$$

Subtracting gives $\langle \vec{w}, (x_1 - x_2) \rangle = 2$. Projecting onto the unit normal $\vec{w}/\|\vec{w}\|$ yields the margin width:

$$\boxed{\text{margin} = \frac{2}{\|\vec{w}\|}.}$$

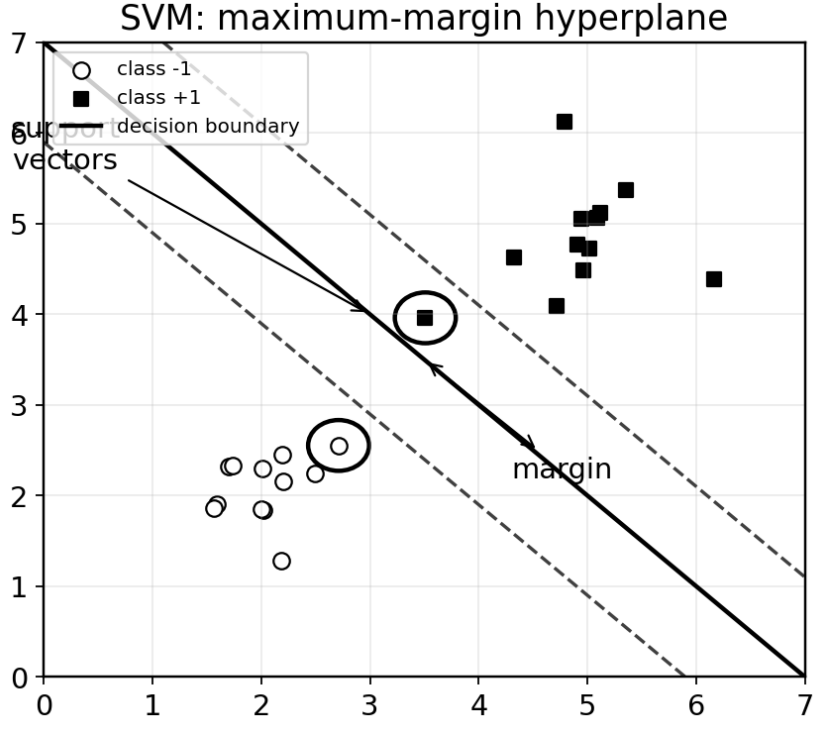

Figure 11.1: The SVM chooses the hyperplane that maximises the margin. The circled points on the margin are the support vectors; removing any other point would not change the boundary.

11.2.2 The Primal Optimisation Problem

Maximising the margin $2/\|\vec{w}\|$ is equivalent to **minimising** $\frac{1}{2}\|\vec{w}\|^2$, subject to the constraint that all training examples are classified correctly with a functional margin of at least 1:

$$\min_{\vec{w},\,b} \; \frac{1}{2}\|\vec{w}\|^2 \quad \text{subject to} \quad y_i\big(\langle \vec{w}, \vec{x}_i \rangle + b\big) \geq 1 \quad \forall\, i.$$

This is a convex quadratic programme with linear constraints, which has a unique global minimum.

(!) Exam Trap

The margin equals $2/\|\vec{w}\|$, *not* $1/\|\vec{w}\|$. The factor of 2 arises because both the $+1$ and -1 margin hyperplanes contribute equally to the total gap.

11.2.3 Lagrangian Formulation and Support Vectors

Introducing Lagrange multipliers $\alpha_i \geq 0$ for each training constraint, the Lagrangian is

$$L(\vec{w},\, b,\, \alpha) = \frac{1}{2}\|\vec{w}\|^2 - \sum_{i=1}^{N} \alpha_i \big[y_i\big(\langle \vec{x}_i, \vec{w} \rangle + b\big) - 1 \big].$$

Setting the gradient with respect to $\vec{w}$ to zero gives

$$\vec{w} = \sum_{i=1}^{N} \alpha_i \, y_i \, \vec{x}_i,$$

so the weight vector is a linear combination of the training examples. The crucial consequence follows from the Karush–Kuhn–Tucker (KKT) complementary slackness condition:

$$\alpha_i \cdot \big[y_i\big(\langle \vec{x}_i, \vec{w} \rangle + b\big) - 1 \big] = 0 \quad \forall\, i.$$

Either $\alpha_i = 0$ (the point is inside the margin and contributes nothing) or $y_i(\langle \vec{x}_i, \vec{w} \rangle + b) = 1$ (the point lies exactly on the margin boundary). The training examples for which $\alpha_i > 0$ are the **support vectors** — the only points that determine the hyperplane. All other points can be discarded without changing the solution.

Key Idea

Only support vectors matter. They are the points closest to the decision boundary (i.e. on the margin), characterised by $\alpha_i > 0$. Removing any training point that is *not* a support vector leaves the hyperplane unchanged.

11.2.4 Soft-Margin SVMs

When data are not perfectly linearly separable — a common situation in practice — the hard-margin constraints $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$ can never all be satisfied simultaneously. The **soft-margin** extension introduces a slack variable $\xi_i \geq 0$ for each training point, allowing it to violate the margin by at most ξ_i:

$$y_i\big(\langle \vec{w}, \vec{x}_i \rangle + b\big) \geq 1 - \xi_i.$$

The objective becomes

$$\min_{\vec{w},\, b,\, \xi} \; \frac{1}{2}\|\vec{w}\|^2 + C \sum_{i=1}^{N} \xi_i,$$

where $C > 0$ is a regularisation hyperparameter. A **small** C penalises margin violations only weakly, so the optimiser prefers a wider margin and tolerates more misclassifications — high regularisation, low variance, risk of underfitting. A **large** C penalises violations heavily, driving the solver towards fewer training errors and a narrower margin — low regularisation, risk of overfitting.

(!) Exam Trap

The direction of the C trade-off is a frequent exam slip. Large $C \Rightarrow$ narrower margin $\Rightarrow$ fewer training errors $\Rightarrow$ higher risk of overfitting. Small $C \Rightarrow$ wider margin $\Rightarrow$ more training errors tolerated $\Rightarrow$ stronger regularisation.

11.3 The Dual Formulation and the Kernel Trick

11.3.1 Dot Products Are All You Need

After eliminating $\vec{w}$ in the Lagrangian, the dual problem depends on the training data only through pairwise **dot products** $\langle \vec{x}_i, \vec{x}_j \rangle$. The decision function takes

the form

$$f(\vec{x}) = \operatorname{sign}\Big(\sum_{i:\,\alpha_i > 0} \alpha_i \, y_i \, \langle \vec{x}_i, \vec{x} \rangle + b \Big),$$

where the sum runs over the support vectors only. This observation — that the optimisation and prediction depend on the data exclusively through dot products — is the key structural fact that enables the kernel trick.

11.3.2 Lifting Data into Feature Space

Consider data that is not linearly separable in the input space $\mathcal{X}$. If there exists a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ into a higher-dimensional (or even infinite-dimensional) Hilbert space $\mathcal{H}$ such that the mapped data *are* linearly separable, then we can run the SVM in $\mathcal{H}$ by simply replacing every dot product with $\langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle$. Computing Φ explicitly, however, may be prohibitively expensive or impossible when $\mathcal{H}$ is infinite-dimensional.

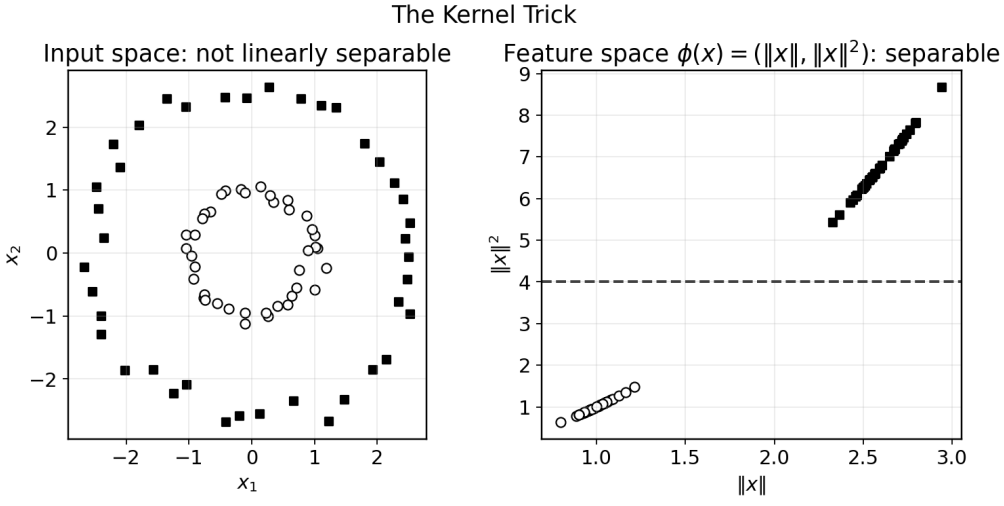

Figure 11.2: The kernel trick: data not linearly separable in input space (left) becomes separable in a higher-dimensional feature space (right).

11.3.3 The Kernel Trick

A function $K : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a **kernel** if there exists an inner-product space $(\mathcal{H}, \langle \cdot, \cdot \rangle)$ and a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ such that

$$K(\vec{x}, \vec{x}') = \langle \Phi(\vec{x}),\, \Phi(\vec{x}') \rangle.$$

The kernel computes the inner product in $\mathcal{H}$ *without explicitly evaluating* Φ. This is the **kernel trick**: wherever the dual SVM uses a dot product $\langle \vec{x}_i, \vec{x}_j \rangle$, substitute a kernel evaluation $K(\vec{x}_i, \vec{x}_j)$. The new decision function is

$$\boxed{f(\vec{x}) = \operatorname{sign}\Big(\sum_{i:\, \alpha_i > 0} \alpha_i \, y_i \, K(\vec{x}_i, \vec{x}) + b \Big).}$$

The kernel determines what kind of decision boundary the SVM can find. Different kernels implicitly lift the data into different feature spaces, producing different boundary shapes in the original input space.

Key Idea

The kernel trick avoids computing Φ explicitly. It is valid whenever the SVM objective depends on the data only through dot products — which is always the case in the dual formulation. Even when Φ maps into an infinite-dimensional space (as the RBF kernel does), the kernel evaluation remains a single scalar computation.

11.4 Important Kernels

The lecture introduces three standard kernels and illustrates the use of structured (tree) kernels for non-vectorial data. Each kernel implicitly defines a different feature space and therefore a different class of decision boundaries.

11.4.1 Linear Kernel

$$K(\vec{x}, \vec{x}') = \langle \vec{x},\, \vec{x}' \rangle$$

The linear kernel applies no mapping at all: Φ is the identity, and $\mathcal{H} = \mathcal{X}$. Using this kernel in the SVM produces exactly the same linear maximum-margin classifier as the standard (primal) hard-margin SVM derived above. Its main

advantages are simplicity and speed: there are no kernel hyperparameters to tune, and the resulting quadratic programme is as small as possible.

The linear kernel is the right choice when the data are already (nearly) linearly separable in the original input space, or when the feature dimension is very high (e.g. in text classification with bag-of-words representations, where the input space may have tens of thousands of dimensions). Using a more expressive kernel in such cases adds complexity without benefit and increases the risk of overfitting.

11.4.2 Polynomial Kernel

$$K(\vec{x}, \vec{x}') = \left(\langle \vec{x},\, \vec{x}' \rangle + c\right)^d$$

Here $d \geq 1$ is the polynomial degree and $c \geq 0$ is a constant that trades off the influence of higher-degree versus lower-degree terms. Setting $c = 0$ gives a homogeneous polynomial kernel (only degree-d terms); setting $c = 1$ is the most common choice and includes all degrees up to d.

To understand why this works as a kernel, consider a simple example with $\vec{x} \in \mathbb{R}^2$ and $d = 2$, $c = 1$:

$$K(\vec{x}, \vec{x}') = \left(x_1 x_1' + x_2 x_2' + 1\right)^2 = x_1^2 {x_1'}^2 + x_2^2 {x_2'}^2 + 2x_1 x_2 x_1' x_2' + 2x_1 x_1' + 2x_2 x_2' + 1.$$

This equals the dot product $\langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle$ where

$$\Phi(\vec{x}) = \left(x_1^2,\; x_2^2,\; \sqrt{2}\, x_1 x_2,\; \sqrt{2}\, x_1,\; \sqrt{2}\, x_2,\; 1\right)^\top.$$

Expanding the degree-d binomial in general, the implicit feature map includes all monomials of degree 1 through d — every cross-product between input components up to that order. The polynomial kernel therefore "computes the relationships between each pair of observations" in a degree-d feature space: data points that overlap on the raw input axes may become linearly separable when their higher-order interactions are taken into account.

Figure 11.3 (the figure above) illustrates the effect concretely. The horizontal axis is a single raw feature — the dosage of a drug in milligrams. On this 1-D axis the green class (drug effective) and the red class (drug not effective) interleave: they overlap and cannot be separated by a threshold. The polynomial kernel lifts each data point upward into a 2-D plane by computing pairwise relationships between all observations. In this higher-dimensional representation the two classes separate cleanly, and the SVM can draw a straight boundary — which, projected back onto the original axis, corresponds to a curved (polynomial)

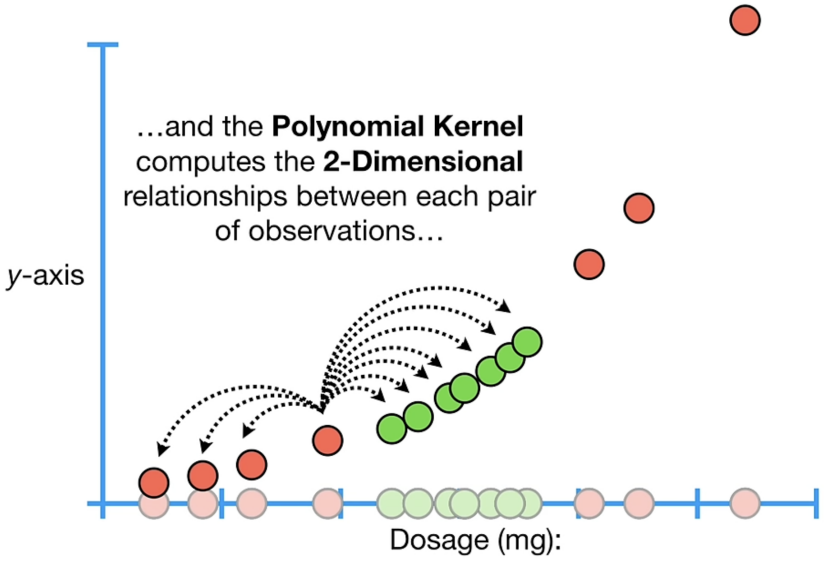

Figure 11.3: Polynomial kernel intuition (after StatQuest): a 1-D dosage axis is lifted into a higher-dimensional space where the polynomial kernel computes the relationship between each pair of observations; the two classes (here shaded) that overlap on the raw axis become separable by a straight boundary in feature space.

decision boundary. This is the kernel trick in miniature: what looks like a nonlinear boundary in input space is, from the perspective of feature space, a perfectly ordinary hyperplane.

The degree d is the principal hyperparameter. Small d (e.g. $d = 2$ or $d = 3$) gives smooth, interpretable boundaries and is appropriate when domain knowledge suggests a moderate degree of interaction. Large d can fit very complex patterns but also leads to oscillatory boundaries and overfitting on small datasets.

11.4.3 Gaussian (Radial Basis Function) Kernel

$$K(\vec{x}, \vec{x}') = \exp\!\left(-\gamma \, \|\vec{x} - \vec{x}'\|^2\right)$$

The parameter $\gamma > 0$ (sometimes written as $1/(2\sigma^2)$, where σ is a bandwidth) controls the locality of the kernel. The Gaussian or **radial basis function (RBF) kernel** is a measure of **similarity**: when $\vec{x}$ and $\vec{x}'$ are close, $\|\vec{x} - \vec{x}'\|^2$

is small and $K \approx 1$; as the points move apart the kernel decays exponentially towards 0.

The feature space corresponding to the RBF kernel is *infinite-dimensional*: the Taylor expansion of $\exp(-\gamma\|\vec{x}\|^2)$ yields an infinite sequence of polynomial terms of every degree. It is of course impossible to compute Φ explicitly, but the kernel value itself is just a single exponential — this is precisely the power of the kernel trick.

The RBF kernel is part of the core course material (see lecture slide 19). It is arguably the most widely used kernel in practice: it requires choosing only a single parameter γ and can represent arbitrarily complex, localised decision boundaries.

Key Idea

Effect of γ on model complexity. Large γ means the kernel decays very steeply: only training points that are extremely close to a test point have any influence. The resulting boundary becomes highly local and wiggly, tracking the training data closely — high variance, risk of overfitting. Small γ means the kernel decays gently: distant points still contribute, producing smoother, more global boundaries — higher bias, better generalisation for small datasets.

(!) Exam Trap

A large γ in the RBF kernel does *not* mean a simpler model. It means a more complex, localised model that can overfit. Think of γ as the inverse of the neighbourhood radius.

11.4.4 Tree Kernels and Structured Data

The three kernels above all assume that data live in a vector space. Many real-world domains produce structured objects — parse trees, XML DOM trees, molecular graphs — for which no natural vector embedding exists. The lecture presents the **Parse Tree Kernel** of Collins and Duffy and several variants (SimTK, LeftTK, SetTK, SoftTK) for DOM trees.

The key idea is to describe a tree T by the multiset of all its subtrees $\{t_1, t_2, \ldots, t_k\}$, forming an implicit (very high-dimensional) feature vector $(\phi_{t_1}(T), \ldots, \phi_{t_k}(T$ where $\phi_t(T)$ counts how often subtree t appears in T. The kernel between two

trees is then

$$K(T, T') = \sum_{v \in V(T),\, v' \in V(T')} \Delta(v, v'),$$

where $\Delta(v, v')$ counts the number of common subtrees rooted at nodes v and v' and is computed recursively in polynomial time. This shows that the kernel framework is not restricted to numerical data: as long as a valid inner product can be defined over a structured space, the full SVM machinery applies. The choice of kernel determines which structural similarities the SVM is sensitive to — an exact analogue of how the choice among linear, polynomial, and RBF kernels determines the geometry of the boundary for vector data.

11.5 Kernel Comparison

Table 11.1 summarises the three main kernels from the lecture.

Table 11.1: Comparison of the three standard kernels introduced in the lecture.

Kernel	**Formula**	**What it does**	**When to use**
Linear	$K(\vec{x}, \vec{x}') = \langle \vec{x}, \vec{x}' \rangle$	No mapping; SVM boundary is linear in input space.	Data (nearly) linearly separable; high-dimensional input; fast baseline.
Polynomial (degree d)	$K(\vec{x}, \vec{x}') = (\langle \vec{x}, \vec{x}' \rangle + c)^d$	Implicitly includes all monomials up to degree d; curved boundary.	Moderate nonlinearity; degree known from domain; image classification.
Gaussian / RBF	$K(\vec{x}, \vec{x}') = \exp(-\gamma \lVert \vec{x} - \vec{x}' \rVert^2)$	Infinite-dimensional feature space; similarity decays with distance.	General nonlinear classification; default choice; tune γ and C.

11.6 Worked Examples

11.6.1 Computing the Margin from $\vec{w}$

Suppose a trained hard-margin SVM has weight vector $\vec{w} = (3, 4)^\top$ and bias $b = -1$. The margin is

$$\text{margin} = \frac{2}{\|\vec{w}\|} = \frac{2}{\sqrt{3^2 + 4^2}} = \frac{2}{\sqrt{9 + 16}} = \frac{2}{\sqrt{25}} = \frac{2}{5} = 0.4.$$

To verify that a point $x = (1, 1)^\top$ with label $y = +1$ is a support vector, check the functional margin:

$$y \cdot \big(\langle \vec{w}, x \rangle + b\big) = 1 \cdot (3 \cdot 1 + 4 \cdot 1 - 1) = 1 \cdot 6 = 6 \neq 1,$$

so this point is not a support vector; it lies in the interior of the positive half-space. A point $x' = (0, 0.3)^\top$ with $y = -1$ would give $(-1)(0{+}1.2{-}1) = (-1)(0.2) = -0.2$, also not a support vector. A support vector x^* with label $+1$ satisfies exactly $\langle \vec{w}, x^* \rangle + b = 1$.

11.6.2 Evaluating the RBF Kernel

Let $\gamma = 0.5$ and two points $\vec{x}_1 = (1, 0)^\top$, $\vec{x}_2 = (3, 0)^\top$.

$$\|\vec{x}_1 - \vec{x}_2\|^2 = (1 - 3)^2 + (0 - 0)^2 = 4.$$

$$K(\vec{x}_1, \vec{x}_2) = \exp(-0.5 \cdot 4) = \exp(-2) \approx 0.135.$$

Now try $\vec{x}_3 = (1.1,\ 0)^\top$, close to $\vec{x}_1$:

$$K(\vec{x}_1, \vec{x}_3) = \exp(-0.5 \cdot (0.1)^2) = \exp(-0.005) \approx 0.995.$$

The kernel correctly captures similarity: nearby points score near 1, distant points score near 0. Increasing γ to 5 would give $K(\vec{x}_1, \vec{x}_2) = e^{-20} \approx 0$ and $K(\vec{x}_1, \vec{x}_3) = e^{-0.05} \approx 0.95$ — a much more local similarity measure.

Worked Example

Kernel decision function. Suppose a trained SVM has two support vectors: $\vec{x}_1 = (0, 0)^\top$ with $y_1 = +1$, $\alpha_1 = 1$, and $\vec{x}_2 = (2, 2)^\top$ with $y_2 = -1$, $\alpha_2 = 1$. Bias $b = 0$. Using the RBF kernel with $\gamma = 1$, classify the test

point $\vec{x} = (1,1)^\top$:

$$K(\vec{x}_1, \vec{x}) = e^{-\|(0,0)-(1,1)\|^2} = e^{-2} \approx 0.135,$$

$$K(\vec{x}_2, \vec{x}) = e^{-\|(2,2)-(1,1)\|^2} = e^{-2} \approx 0.135.$$

$$f(\vec{x}) = \operatorname{sign}\big(1 \cdot (+1) \cdot 0.135 + 1 \cdot (-1) \cdot 0.135 + 0\big) = \operatorname{sign}(0).$$

The test point lies exactly on the decision boundary. A small perturbation towards $\vec{x}_1$ would tip it to class $+1$; towards $\vec{x}_2$, to class -1.

11.7 Multi-Class SVMs

SVMs are inherently binary classifiers (two classes only). Two standard strategies extend them to $k > 2$ classes:

- **One-against-all.** Train k binary SVMs. The i-th SVM treats class i as positive and all other classes as negative. For a test point, feed it into all k models and assign the class whose SVM reports the largest decision value.
- **One-against-one.** Train $k(k-1)/2$ binary SVMs, one for each pair of classes. Each SVM votes for one of its two classes. The test point is assigned to the class receiving the most votes.

One-against-one typically gives better calibration at the cost of training more models; one-against-all is faster to train but can suffer from class-imbalance issues.

11.8 Exam Traps: Summary

(!) Exam Trap

Only support vectors define the hyperplane. Points with $\alpha_i = 0$ (including all correctly classified interior points) contribute nothing to $\vec{w}$ and can be removed from the training set without changing the solution. A common mistake is to assume that all training points influence the boundary.

(!) Exam Trap

Margin is $2/\|\vec{w}\|$, not $1/\|\vec{w}\|$. The factor of 2 arises because the two parallel margin hyperplanes are at signed distances $+1$ and -1 from the central hyperplane.

(!) Exam Trap

Kernel trick avoids computing Φ. You never need to evaluate the feature map Φ explicitly. The kernel $K(\vec{x}_i, \vec{x}_j) = \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle$ computes the inner product in feature space directly as a scalar function of the original inputs. This is essential for the RBF kernel, where Φ maps into an infinite-dimensional space.

(!) Exam Trap

C direction in soft-margin SVMs. Large $C \Rightarrow$ hard constraints, narrow margin, overfitting risk. Small $C \Rightarrow$ loose constraints, wide margin, underfitting risk. Many students invert this.

(!) Exam Trap

Large γ in RBF $\Rightarrow$ overfitting, not simplicity. A large γ makes the kernel very local: each training point influences only its immediate neighbours. The boundary becomes jagged and fits the training data too closely.

(!) Exam Trap

Perceptron output is $\Theta(s) \in \{0, 1\}$, not $\text{sign}(s)$. The SVM uses $\text{sign} \in \{-1, +1\}$. Convergence of the perceptron requires both linear separability of the data *and* a sufficiently small learning rate ϵ. Neither condition alone is sufficient.

In a Nutshell

Chapter Summary: Machine Learning III

- The **perceptron** is a single-neuron linear classifier with output $\Theta(\vec{w} \cdot \vec{x}) \in \{0, 1\}$. It converges iff the data are linearly separable and ϵ is small. Cannot solve XOR.
- An **SVM** finds the *maximum-margin hyperplane* $\langle \vec{w}, \vec{x} \rangle + b = 0$. Margin $= 2/\|\vec{w}\|$. Primal problem: minimise $\frac{1}{2}\|\vec{w}\|^2$ s.t. $y_i(\langle \vec{w}, \vec{x}_i \rangle + b) \geq 1$.
- **Support vectors** are training points on the margin ($\alpha_i > 0$); they are the only points that determine the hyperplane.
- **Soft-margin SVMs** add slack ξ_i and parameter C: large $C \Rightarrow$ narrow margin / overfitting risk; small $C \Rightarrow$ wide margin / regularisation.
- The **kernel trick** replaces every dot product $\langle \vec{x}_i, \vec{x}_j \rangle$ with $K(\vec{x}_i, \vec{x}_j) = \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle$, implicitly mapping data into a higher-dimensional feature space without computing Φ explicitly.
- Three standard kernels: **Linear** ($K = \langle \vec{x}, \vec{x}' \rangle$, no mapping), **Polynomial** ($K = (\langle \vec{x}, \vec{x}' \rangle + c)^d$, all monomials up to degree d), **RBF/Gaussian** ($K = \exp(-\gamma \|\vec{x} - \vec{x}'\|^2)$, infinite-dimensional feature space, locality controlled by γ).
- **Tree kernels** extend SVMs to structured objects (parse trees, DOM trees) by defining inner products over subtree counts.
- Multi-class extension via **one-against-all** (k classifiers) or **one-against-one** ($k(k-1)/2$ classifiers).

11.9 Going Further: Oral-Exam Deep Cuts

Q: What exactly are the support vectors, and what happens to the boundary if you delete one — versus deleting a non-support vector?
A: Support vectors are the training points lying on (or, for soft margins, on/inside) the margin — the points with dual coefficients $\alpha_i > 0$. Since $\vec{w} = \sum_i \alpha_i y_i \vec{x}_i$ and $\alpha_i = 0$ for all others, only the support vectors define the hyperplane. Delete a *non*-support vector and the boundary is *unchanged*; delete a support vector and the boundary can move. This is also why SVMs are sparse and memory-efficient.

Q: A large C in a soft-margin SVM — does it widen or narrow the

margin, and what does it do to bias/variance?
A: The objective is $\frac{1}{2}\|\vec{w}\|^2 + C\sum_i \xi_i$. *Large* C penalises margin violations heavily, so the model tolerates few errors $\Rightarrow$ a *narrow*, hard margin that hugs the data $\Rightarrow$ low bias, high variance (overfitting risk). *Small* C allows more slack $\Rightarrow$ a *wider* margin $\Rightarrow$ higher bias, lower variance. (Note the direction is opposite to a regularisation strength λ — a classic confusion.)

Q: Explain the kernel trick and why you never compute the feature map Φ explicitly.
A: In the dual, data appear *only* as inner products $\langle \vec{x}_i, \vec{x}_j \rangle$. A kernel $K(\vec{x}, \vec{x}') = \langle \Phi(\vec{x}), \Phi(\vec{x}') \rangle$ computes that inner product in the lifted space *directly*, in input-space cost, without ever forming Φ. You avoid Φ because it can be huge or even infinite-dimensional — for the RBF kernel $K = \exp(-\gamma\|\vec{x} - \vec{x}'\|^2)$ the feature space is infinite, so explicit computation is impossible, yet K is trivial to evaluate.

Q: Why *maximise* the margin at all — what is the justification?
A: A wider margin means a less "wiggly" decision boundary and a lower-capacity classifier (structural risk minimisation): it bounds the generalisation error independent of the input dimension. Intuitively, the farther the boundary sits from the nearest points, the more robust it is to noise in those points.

Q: Why move from the primal to the dual formulation?
A: Two reasons. (1) The dual depends on the data only through inner products, which is what enables the kernel trick. (2) The number of dual variables equals the number of *training points*, not the (possibly huge or infinite) feature dimension — so kernelised SVMs stay tractable even in infinite-dimensional feature spaces.

Q: Are SVMs sensitive to feature scaling?
A: Yes, strongly — especially with the RBF kernel and any distance-/norm-based kernel, because $\|\vec{x} - \vec{x}'\|$ is dominated by large-scale features. Always standardise features before training an SVM; this is a common practical gotcha.

Chapter 12

Neural Networks

Neural networks are the backbone of modern machine learning. This chapter traces the conceptual arc from a single artificial neuron all the way to deep multilayer architectures, covering how networks are structured, how they are trained, and what can go wrong. The mathematical machinery — gradient descent, the chain rule, and backpropagation — is developed step by step, and common exam traps are highlighted throughout.

12.1 From the Perceptron to the Multilayer Perceptron

The story of neural networks begins with the **perceptron**, a model of a single biological neuron proposed by Rosenblatt in 1958. The fundamental idea is simple: a neuron receives several input signals, weights them, sums them up, and produces an output by passing that sum through a nonlinear *activation function.* Formally, for a neuron with n inputs $x_1, \ldots, x_n$, weights $w_1, \ldots, w_n$, and bias b, the output is

$$y = g\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{12.1}$$

where g is the activation function. Each weight w_i determines how strongly input x_i influences the neuron; a large positive w_i excites the neuron, a large negative w_i inhibits it. The bias b acts like an adjustable threshold: it shifts the activation function left or right and allows the neuron to fire even when all inputs

are zero (or to remain silent when inputs are moderate).

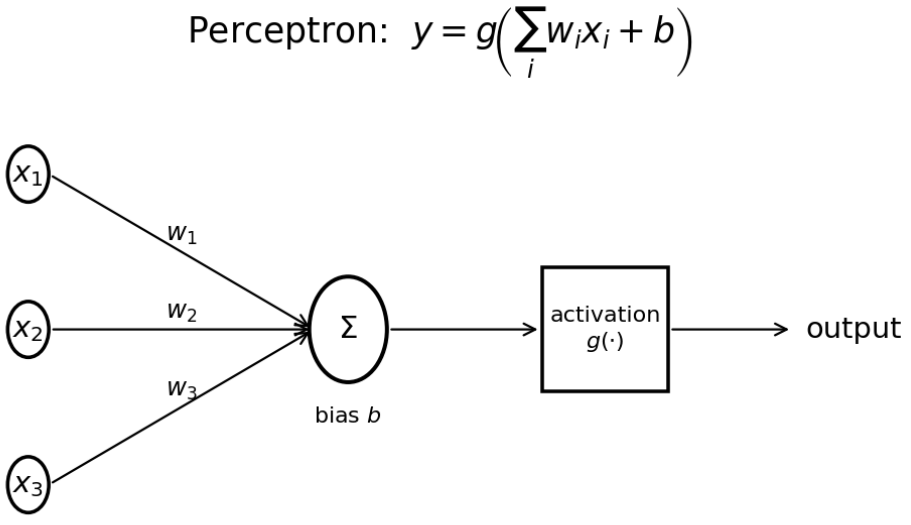

Figure 12.1: A single artificial neuron: weighted sum of inputs plus bias, passed through an activation function.

A single neuron can only represent a linear decision boundary. The historical *XOR problem* — two classes that no line can separate — demonstrated this limitation conclusively. The solution is to stack neurons into **layers** and connect layers sequentially, forming a **Multilayer Perceptron** (MLP), also called a feedforward neural network.

In a layered network, a single layer ℓ receives a vector $\vec{x}$ and applies the same computation to every neuron in parallel. Writing $W^{(\ell)}$ for the weight matrix and $\vec{b}^{(\ell)}$ for the bias vector of layer ℓ, the layer's output is

$$\vec{a}^{(\ell)} = g\left(W^{(\ell)}\vec{x} + \vec{b}^{(\ell)}\right), \tag{12.2}$$

where g is applied element-wise. The *width* of a layer is the number of neurons it contains; the network's *depth* is the number of layers. Layers between input and output are called **hidden layers** because their activations are not directly observed.

12.2 Activation Functions

The activation function g is the element that makes neural networks nonlinear. Without it, stacking multiple linear transformations would collapse into a single linear map — no matter how many layers are added, the network could only compute linear functions of the input. The choice of activation function profoundly affects both the expressiveness and the trainability of a network.

12.2.1 Sigmoid

The **sigmoid** function maps any real number to the open interval $(0, 1)$:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{12.3}$$

Its smooth, S-shaped curve made it the dominant activation function in the early decades of neural network research. The output is naturally interpretable as a probability, which is why sigmoid is still used in the output layer of binary classifiers. However, sigmoid has a critical flaw in deep networks: for very large or very small values of z, the function saturates — its derivative approaches zero. Because backpropagation multiplies gradients layer by layer, a chain of near-zero derivatives causes the gradient to shrink exponentially as it propagates backward through the network. This is the **vanishing gradient problem**, and it effectively prevents the early layers of a deep network from learning.

12.2.2 Hyperbolic Tangent

The **tanh** function is closely related to the sigmoid:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}. \tag{12.4}$$

Its output range is $(-1, 1)$, which means the activations are zero-centred — an advantage over sigmoid, since non-zero-centred activations can introduce a systematic bias in the gradient updates. Nonetheless, tanh also saturates at large magnitudes, so it suffers from the vanishing gradient problem in deep networks just as sigmoid does.

12.2.3 ReLU

The **Rectified Linear Unit** (ReLU) is now the default activation function for hidden layers in most modern networks:

$$\mathrm{ReLU}(z) = \max(0,\, z). \tag{12.5}$$

ReLU is piecewise linear: it outputs zero for negative inputs and the identity for positive inputs. Its derivative is 1 for all positive inputs and 0 for negative inputs. This has a decisive advantage: for positive pre-activations, the gradient is simply 1, and it does *not* shrink as it is multiplied across layers. ReLU therefore largely solves the vanishing gradient problem that hampers sigmoid and tanh. Additionally, ReLU is computationally cheap — it requires only a comparison and a pass-through, no exponentials. The downside is the **dying ReLU** problem: neurons with consistently negative pre-activations produce a zero gradient and receive no weight update, effectively dying during training.

12.2.4 Leaky ReLU

Leaky ReLU addresses the dying neuron problem by allowing a small, non-zero gradient for negative inputs:

$$\mathrm{LeakyReLU}(z) = \begin{cases} z & \text{if } z \geq 0, \\ \alpha z & \text{if } z < 0, \end{cases} \tag{12.6}$$

where α is a small constant (typically 0.01). This keeps the gradient alive for negative pre-activations while preserving the main benefit of ReLU for positive values.

12.2.5 Comparison Table

(!) Exam Trap

Why non-linearity is non-negotiable. Stacking multiple purely linear layers is algebraically equivalent to a single linear transformation: $W^{(2)}(W^{(1)}\vec{x}) = (W^{(2)}W^{(1)})\vec{x}$. No matter how many layers are stacked without nonlinear activations, the network can only compute linear functions. Nonlinear activation functions are what give MLPs their expressive power. In exams, if asked why we need activation functions, the answer is

Table 12.1: Comparison of common activation functions.

Function	Formula	Range	Gradient	Main Problem
Sigmoid	$1/(1+e^{-z})$	$(0,\ 1)$	$\sigma(1-\sigma),\ \leq 0.25$	Vanishing gradient, not zero-centred
Tanh	$\dfrac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$(-1,\ 1)$	$1-\tanh^2,\ \leq 1$	Vanishing gradient (saturates)
ReLU	$\max(0, z)$	$[0,\ \infty)$	1 if $z>0$, else 0	Dying neurons
Leaky ReLU	$\max(\alpha z,\ z)$	$\mathbb{R}$	1 or α	Slightly more complex

always: *to introduce nonlinearity that allows the network to approximate complex functions.*

12.3 Feedforward Computation: The Forward Pass

Given inputs, the MLP computes its output through the **forward pass**: it applies the layers sequentially, feeding each layer's output as the next layer's input. Denoting the number of layers as ℓ, the computation chain is

$$\mathrm{MLP}(\vec{x}) = \left(g^{(\ell)} \circ f^{(\ell)}_{W,\vec{b}} \circ \cdots \circ g^{(1)} \circ f^{(1)}_{W,\vec{b}}\right)(\vec{x}), \tag{12.7}$$

where $f^{(i)}_{W,\vec{b}}(\vec{z}) = W^{(i)}\vec{z} + \vec{b}^{(i)}$ is the affine transformation of layer i and $g^{(i)}$ is its activation function applied pointwise. The network is fully determined once all weight matrices $W^{(i)}$ and bias vectors $\vec{b}^{(i)}$ are specified. The *parameters* of the network are the collection $\theta = \{W^{(i)}, \vec{b}^{(i)}\}_{i=1}^{\ell}$.

12.3.1 Worked Example: One Forward Pass

Consider a single neuron with two inputs $x_1 = 0.5$, $x_2 = -1.0$, weights $w_1 = 0.8$, $w_2 = 0.4$, bias $b = 0.1$, and sigmoid activation.

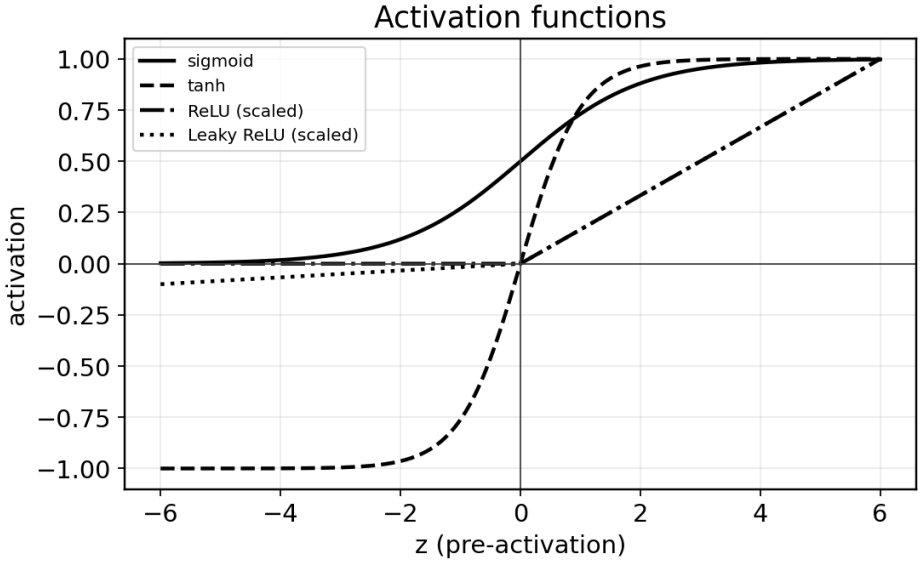

Figure 12.2: Common activation functions (ReLU and Leaky ReLU are scaled to fit the same axes).

Worked Example

Step 1 — Weighted sum:

$$z = w_1x_1 + w_2x_2 + b = (0.8)(0.5) + (0.4)(-1.0) + 0.1 = 0.4 - 0.4 + 0.1 = 0.1.$$

Step 2 — Activation:

$$y = \sigma(z) = \frac{1}{1+e^{-0.1}} \approx \frac{1}{1+0.905} \approx 0.525.$$

The neuron outputs approximately 0.525. If this were a binary classifier, this output is just above the 0.5 decision threshold, so the neuron would predict the positive class.

12.4 The Loss Function

Learning requires a way to measure how badly the network is currently performing. A **loss function** $\mathcal{L}$ assigns a scalar penalty to the network's predictions given

the training data. Large loss means the network is performing poorly; small loss means it is close to the target. The training objective is to minimise $\mathcal{L}$ over the parameter set θ.

An important distinction is between the *loss function* and the *evaluation metric*. The evaluation metric (e.g., accuracy, F1 score) measures the real-world performance that we care about. The loss function is what we actually optimise during training; it must be **differentiable** so that gradient-based methods can be applied. These two quantities should be aligned — the loss should serve as a good proxy for the evaluation metric — but they need not be identical.

12.4.1 Mean Squared Error

For regression tasks, the most common choice is **Mean Squared Error** (MSE):

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \left\| \vec{y}^{(n)} - \text{MLP}(\vec{x}^{(n)};\, \theta) \right\|^2, \tag{12.8}$$

where $\vec{y}^{(n)}$ is the target output for the n-th training example. MSE penalises large errors quadratically, which makes it sensitive to outliers.

12.4.2 Categorical Cross-Entropy

For classification, the MLP output is first passed through a **softmax** layer that converts raw scores (logits) into a probability distribution over classes:

$$\text{softmax}(\vec{z})_k = \frac{e^{z_k}}{\sum_j e^{z_j}}. \tag{12.9}$$

The predicted probability vector $\vec{q}$ can then be compared to the true one-hot distribution $\vec{p}$ using **categorical cross-entropy**:

$$\mathcal{L}_{\text{CE}}(\vec{p},\, \vec{q}) = -\sum_k p_k \log q_k. \tag{12.10}$$

Because $\vec{p}$ is one-hot (exactly one $p_k = 1$, all others 0), this simplifies to the **negative log-likelihood** of the correct class: $\mathcal{L} = -\log q_c$ where c is the true class index. Cross-entropy is closely related to the Kullback–Leibler divergence $D_{\text{KL}}(p\|q)$: since $H(p, q) = H(p) + D_{\text{KL}}(p\|q)$ and the entropy of a one-hot distribution $H(p) = 0$, minimising cross-entropy is equivalent to minimising the KL-divergence to the true distribution.

12.5 Training by Gradient Descent

With a differentiable loss function in hand, training becomes an optimisation problem: find the parameter values θ^* that minimise $\mathcal{L}(\theta)$. The workhorse method is **gradient descent**. Intuitively, the gradient $\nabla_\theta \mathcal{L}$ points in the direction of steepest *increase* of the loss. Moving in the opposite direction — downhill along the negative gradient — reduces the loss. The parameter update rule is

$$\theta \leftarrow \theta - \eta \, \nabla_\theta \mathcal{L}(\theta), \tag{12.11}$$

where $\eta > 0$ is the **learning rate**, a critical hyperparameter that controls the size of each step.

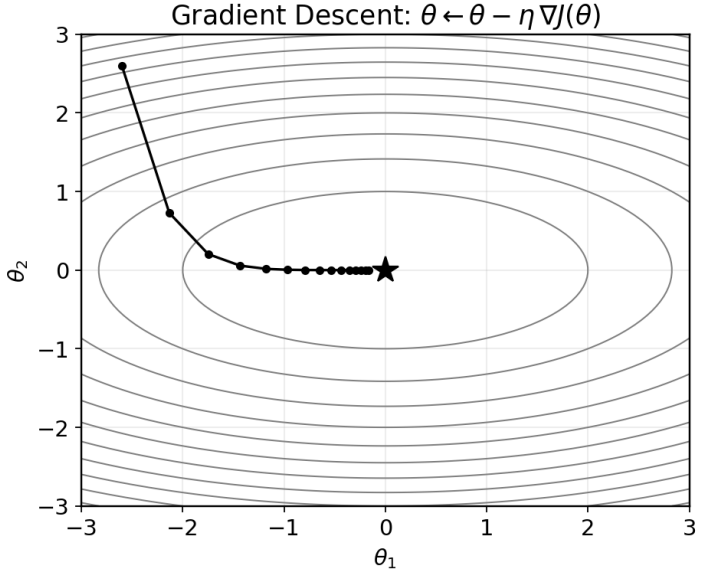

Figure 12.3: Gradient descent steps downhill along the negative gradient toward a minimum of the loss.

12.5.1 The Learning Rate

Choosing an appropriate learning rate is one of the most important and difficult aspects of training. If η is too small, the network will converge, but training will be extremely slow — many thousands of steps may be needed to make noticeable

progress, and the optimiser may get stuck in shallow local minima. If η is too large, the update step will overshoot the minimum: the loss may oscillate or even diverge. In practice, a **learning rate schedule** is commonly used: the learning rate starts at a larger value to make rapid progress and is gradually reduced (decayed) as training progresses, allowing the optimiser to settle into a precise minimum.

(!) Exam Trap

Learning rate trade-off. In exams, remember: too small η = slow convergence, risk of getting stuck; too large η = overshooting, instability, possible divergence. Neither extreme is desirable.

12.5.2 Stochastic, Mini-Batch, and Batch Gradient Descent

The gradient in equation (12.11) is computed over the entire training set, which is called **full-batch gradient descent**. For large datasets this is computationally prohibitive. Three practical variants exist:

- **Stochastic Gradient Descent (SGD):** The gradient is estimated using a single randomly selected training example. The estimate is cheap but very noisy, leading to high variance updates. Despite its instability, the noise can sometimes help escape sharp local minima.
- **Mini-batch gradient descent:** The training set is partitioned into small subsets (*mini-batches*) of size B (typically 32–512 examples). The gradient is computed over each mini-batch. This is the most common approach in practice: it is more stable than single-sample SGD and far cheaper than full-batch.
- **Full-batch gradient descent:** Gradients computed from the entire training set. Stable but expensive and often impractical.

In all cases, one **epoch** is completed when every training example has been used exactly once. Training typically requires many epochs.

12.6 Backpropagation

Gradient descent requires the gradient $\nabla_\theta \mathcal{L}$, which means computing $\partial \mathcal{L}/\partial w$ for every parameter w in the network. Doing this naively for each weight would be

hopelessly expensive. **Backpropagation** is the efficient algorithm that computes all of these gradients simultaneously by applying the **chain rule** of calculus.

12.6.1 The Chain Rule

Recall that for a composition of functions $h = f \circ g$, the chain rule states

$$\frac{\partial h}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}. \tag{12.12}$$

A neural network is precisely a long composition of functions: loss applied to the output of the last activation, which is the output of the last affine transformation, which is the activation of the previous layer, and so on. Backpropagation simply applies the chain rule systematically from the output layer backward to the input layer, reusing intermediate computations.

12.6.2 Forward Pass and Backward Pass

Backpropagation is structured in two phases.

Forward pass. The network processes the input $\vec{x}$ and computes all intermediate activations $\vec{a}^{(1)}, \ldots, \vec{a}^{(\ell)}$ and the final loss $\mathcal{L}$. All pre-activations and activations are stored in memory because they are needed during the backward pass.

Backward pass. Starting from the output, the algorithm propagates the *error signal* — the partial derivative of the loss with respect to each layer's output — back through the network. The loss gradient with respect to the final layer's pre-activation is computed first:

$$\frac{\partial \mathcal{L}}{\partial f^{(\ell)}} = \mathcal{L}' \cdot \left(a^{(\ell)}\right)', \tag{12.13}$$

where $\mathcal{L}'$ is the derivative of the loss with respect to the network output, and $(a^{(\ell)})'$ is the derivative of the activation function. The gradient is then propagated backward layer by layer:

$$\frac{\partial \mathcal{L}}{\partial f^{(i)}} = \frac{\partial \mathcal{L}}{\partial f^{(i+1)}} \cdot W^{(i+1)} \cdot \left(a^{(i)}\right)'. \tag{12.14}$$

At each layer, once the gradient of the loss with respect to the pre-activation $f^{(i)}$ is known, the gradient with respect to the weight matrix $W^{(i)}$ is:

$$\frac{\partial \mathcal{L}}{\partial W^{(i)}} = \frac{\partial \mathcal{L}}{\partial f^{(i)}} \cdot \vec{a}^{(i-1)\top}. \tag{12.15}$$

This is simply the outer product of the error signal and the layer's input. All weight gradients are computed in a single backward sweep; the update rule (12.11) is then applied.

Key Idea

Backpropagation = chain rule, applied efficiently. The algorithm does nothing magical: it is systematic application of the chain rule to the composition of functions that makes up the network. The forward pass computes and stores the activations; the backward pass reuses them to compute gradients from output to input. Each gradient computation is just a multiplication by the local Jacobian (weight matrix times activation derivative).

12.6.3 Vanishing Gradients Revisited

Equation (12.14) reveals exactly why saturating activations are dangerous. Each backward step multiplies by the derivative of the activation function $(a^{(i)})'$. For sigmoid, this derivative is at most 0.25 (achieved at $z = 0$) and shrinks toward zero as $|z|$ grows. Across ℓ layers, the gradient shrinks by a factor of at most 0.25^{ℓ}. For a 10-layer sigmoid network, this is $(0.25)^{10} \approx 10^{-6}$: the gradient that reaches the early layers is so small that the weights there barely move. The network effectively stops learning in its early layers. ReLU avoids this because its derivative for positive inputs is exactly 1, so the gradient does not shrink during the backward pass through ReLU neurons.

12.7 Universal Approximation

A remarkable theoretical result — the **universal approximation theorem** — states that an MLP with nonlinear activations can approximate any continuous function to arbitrary precision on a compact domain. The lecture presents a constructive intuition using **two hidden layers**:

1. The first hidden layer defines hyperplanes in input space, dividing it into half-spaces.

2. The second hidden layer computes logical ANDs over these half-spaces, carving out arbitrary convex regions.
3. The output layer computes logical ORs over those convex regions, assembling any shape.

This shows that two hidden layers are *sufficient* in principle. In practice, however, very deep networks (many layers) often generalise better than wide shallow ones, because depth allows hierarchical feature reuse: early layers learn low-level features (edges, textures), later layers compose them into high-level concepts (objects, categories).

(!) Exam Trap

Slides use 2 hidden layers, not 1. The classic Cybenko (1989) theorem states that a single hidden layer suffices for universal approximation. The lecture's constructive proof, however, uses two hidden layers (hyperplanes $\rightarrow$ convex sets $\rightarrow$ arbitrary regions). Do not conflate these two claims in exams.

12.8 Deep Networks: Benefits and Challenges

A network is considered **deep** if it has more than two layers. Depth adds expressive power — not because it is theoretically necessary (two layers suffice for universal approximation), but because empirical practice consistently shows that deeper networks generalise better on complex tasks. Each additional layer allows the network to build more abstract, compositional representations of the data.

However, depth introduces several new challenges.

12.8.1 Vanishing and Exploding Gradients

As established above, saturating activations cause gradients to vanish during backpropagation. The complementary problem is **exploding gradients**: if the weights or the activation derivatives are large, gradients grow exponentially as they propagate backward, causing numerical instability. The primary remedy is ReLU (or its variants), which keeps the gradient from shrinking. **Residual connections** (skip connections) provide a complementary solution: by adding the input of a block directly to its output ($\vec{y} = F(\vec{x}) + \vec{x}$), the gradient can flow backward

through the identity path without passing through any activation function at all, enabling training of very deep networks (e.g., ResNets with hundreds of layers).

12.8.2 Weight Initialisation

Before training begins, the weights must be initialised. Two principles guide this. First, **symmetry breaking**: if all weights start identical, all neurons in a layer will compute the same function and produce the same gradient — no differentiation can occur. Random initialisation breaks this symmetry. Second, the initialisation should be calibrated to avoid exploding or vanishing signals at the start of training. Schemes such as **Glorot (Xavier)** and **Kaiming (He)** initialisation choose the variance of the random weights based on the number of incoming and outgoing connections and the choice of activation function.

12.8.3 Overfitting and Regularisation

Deep networks are heavily **overparameterised**: they often have far more parameters than training examples. This means they can, in principle, memorise the training set perfectly without learning any generalisable structure. When the training loss is low but the test loss is high, the model has *overfit*. Several regularisation strategies combat this:

- **Weight decay (L2 regularisation):** A penalty term $\lambda \sum_w \|w\|^2$ is added to the loss, discouraging large weights and producing smoother functions. In the update rule, this manifests as subtracting a small fraction of each weight on every step.
- **Dropout:** During training, each neuron's activation is randomly set to zero with probability p (the dropout rate). This prevents co-adaptation of neurons and effectively trains an ensemble of many different sub-networks. At test time, dropout is disabled; activations are rescaled to account for the fraction of neurons that were active during training.
- **Early stopping:** Training is monitored on a held-out *validation set*. When the validation loss starts increasing while the training loss continues to decrease, training is stopped. This prevents the model from over-specialising to the training data.

(!) Exam Trap

Dropout is training-only. During inference (test time), dropout must be disabled. Neurons are not randomly deactivated; instead, activations are rescaled (or, equivalently, inverted dropout is used during training so that the expected activation magnitude is preserved without rescaling at test time). Forgetting this distinction is a common exam error.

12.8.4 The Lottery Ticket Hypothesis

An intriguing empirical observation is that even after training, large networks can be pruned by removing 80–90% of their connections without significant performance loss. The **lottery ticket hypothesis** (Frankle & Carlin, 2019) offers an explanation: overparameterised networks contain many subnetworks (*tickets*), and training essentially finds a *winning ticket* — a subnetwork that, combined with its specific initial weights, trains efficiently. The overparameterisation provides the combinatorial pool from which a successful subnetwork can be found.

12.9 MLP for Classification

A particularly important application of MLPs is classification: mapping an input $\vec{x} \in \mathbb{R}^{d_{\text{in}}}$ to a discrete class label from $\{C_1, \ldots, C_K\}$. Since MLPs map real vectors to real vectors, the trick is to encode the class as a real vector. The standard approach is **one-hot encoding**: class C_k is represented as the K-dimensional vector that is 1 in position k and 0 everywhere else.

The output layer has K neurons (one per class). Their outputs (logits) are passed through the softmax function to produce a probability distribution over classes. The softmax ensures the outputs are non-negative and sum to one. The predicted class is the one with the highest probability.

Training uses categorical cross-entropy loss (12.10), which directly measures how well the predicted probability distribution matches the true one-hot distribution. MLPs have been successfully applied to spam detection, handwritten digit recognition, medical diagnosis, customer churn prediction, and sentiment analysis, among many other tasks.

12.10 Common Exam Traps

(!) Exam Trap

Stacking linear layers is pointless without activations. If every layer is purely linear (no activation function), then the entire network collapses to a single linear map, regardless of depth. The activation function must be nonlinear to give the network expressive power beyond linear models.

(!) Exam Trap

Vanishing gradient. In deep networks with sigmoid or tanh activations, gradients shrink exponentially as they propagate backward. The derivative of sigmoid is at most 0.25; multiplied across many layers, the early-layer gradients approach zero and those layers stop learning. ReLU solves this by having a constant gradient of 1 for positive inputs.

(!) Exam Trap

Backpropagation is the chain rule. Backpropagation is not a separate algorithm with special structure; it is the systematic, computationally efficient application of the chain rule to compute gradients of a composed function. The forward pass stores intermediate activations; the backward pass multiplies local Jacobians in reverse order.

(!) Exam Trap

Universal approximation: slides use 2 hidden layers. The lecture's constructive argument uses two hidden layers (half-planes $\rightarrow$ convex sets $\rightarrow$ arbitrary regions). Cybenko's theorem requires only one, but that is not the argument made in the slides. Cite correctly when asked.

(!) Exam Trap

Learning rate too high or too low. A learning rate that is too low leads to slow convergence and risk of getting stuck in local minima. A learning rate that is too high causes overshooting: the optimiser jumps

over the minimum and the loss may oscillate or diverge. Learning rate schedules (decay over time) are the standard practical solution.

In a Nutshell

Chapter Summary: Neural Networks

- A neuron computes $y = g(\sum_i w_i x_i + b)$: a weighted sum of inputs plus bias, passed through a nonlinear activation function g.
- **Nonlinearity is essential**: without it, stacking layers collapses to a single linear map. Common activations: sigmoid (smooth, range $(0, 1)$, saturates), tanh (zero-centred, range $(-1, 1)$, saturates), ReLU (fast, non-saturating for $z > 0$, default choice), Leaky ReLU (fixes dying neurons).
- The **vanishing gradient problem** occurs when saturating activations (sigmoid, tanh) reduce gradients to near zero across layers. ReLU largely solves this.
- An MLP performs a **forward pass**: sequentially applying affine transformations and activations. All activations are stored for the backward pass.
- A **loss function** ($\mathcal{L}$) measures prediction error. It must be differentiable. MSE for regression; cross-entropy for classification.
- **Gradient descent**: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$. The learning rate η must be tuned. Mini-batch gradient descent is the standard approach.
- **Backpropagation** computes all parameter gradients via the chain rule in a single backward sweep. It is not magic — it is systematic application of calculus to composed functions.
- **Universal approximation**: two hidden layers suffice in theory; in practice, depth enables hierarchical feature learning that generalises better.
- Key regularisation methods: weight decay (penalise large weights), dropout (randomly zero activations during training only), early stopping (halt when validation loss rises).

Chapter 13

Deep Learning: CNNs and Transformers

13.1 What Makes Learning "Deep"?

A neural network is called *deep* when it contains more than two layers of learned parameters — that is, when the computation from raw input to final output passes through multiple successive nonlinear transformations. The distinction between shallow and deep architectures is not merely a matter of counting layers; it fundamentally changes what the network learns and how it represents knowledge.

A classical result in function approximation theory states that a network with two hidden layers can, in principle, approximate any continuous function to arbitrary precision — the so-called *universal approximation* property. One might therefore ask: why go deeper at all? Practice gives a clear answer. Networks with many layers consistently outperform shallow counterparts on complex tasks, often by large margins and with fewer total parameters per unit of representational power. The reason lies in the nature of the features that are learned.

Key Idea

In a deep network, each layer learns progressively more *abstract* features of the input. Early layers detect low-level structure (edges, local textures, short phonemes); middle layers combine these into intermediate patterns (shapes, syllables); late layers encode high-level semantic concepts (faces, words). This *hierarchical feature learning* replaces the need for hand-crafted

feature engineering.

Before the deep learning era, practitioners invested enormous effort in designing domain-specific feature extractors by hand: HOG descriptors for pedestrian detection, MFCC coefficients for speech, or bag-of-words representations for text. Deep learning allows the network to *discover* these hierarchies automatically from data, provided enough examples and compute are available.

13.1.1 Challenges of Depth

Depth introduces a set of well-known training difficulties. First, the *vanishing gradient* problem: when activation functions such as sigmoid saturate at large absolute inputs, their derivatives approach zero. During backpropagation, these near-zero gradients are multiplied across every layer, so the signal reaching early layers becomes exponentially small and those layers barely update. The symmetric problem — *exploding gradients* — occurs when weight matrices amplify signals layer by layer, causing numerical instability.

Two architectural innovations largely solved the vanishing gradient problem. The *Rectified Linear Unit* (ReLU), defined as $\mathrm{ReLU}(x) = \max(0, x)$, has a constant derivative of 1 for all positive inputs, so gradients do not shrink through activation. *Residual connections* (He et al., 2016) add an identity shortcut around a block of layers: if the block computes $F(x)$, the output is $F(x) + x$. Gradients can then flow directly through the identity path without passing through any nonlinearity, enabling networks of hundreds of layers to train reliably.

Depth also raises the risk of *overfitting*. Deep networks are typically highly overparameterized: after training, it is often possible to prune 80 to 90 percent of connections with negligible loss of accuracy. The *Lottery Ticket Hypothesis* (Frankle and Carlin, 2019) interprets this observation by suggesting that overparameterization seeds a pool of candidate subnetworks; training finds and amplifies the "winning ticket" — the subnetwork that happened to be well-initialized for the task. Regularization techniques counteract overfitting: *weight decay* augments the loss with a term $\lambda \sum_{w} \|w\|^2$ that penalizes large weights, *dropout* randomly deactivates a fraction of units each forward pass (effectively training an ensemble of subnetworks), and *early stopping* halts training when validation loss begins to rise.

13.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are the dominant architecture for tasks defined on grid-structured data — images, audio spectrograms, and video frames. Their design exploits two structural regularities of such data: nearby elements are more informative about each other than distant ones (*locality*), and the same pattern can appear at any position (*translation equivariance*).

13.2.1 Convolution, Filters, and Weight Sharing

In a standard fully connected layer, each output neuron maintains a separate weight to every input neuron. For a 224×224 colour image this means each neuron in the first hidden layer would need $224 \times 224 \times 3 = 150{,}528$ independent parameters — millions of parameters even before the first nonlinearity. CNNs replace dense connections with a *convolutional filter* (also called a *kernel*): a small matrix of learned weights, typically 3×3 or 5×5, that is slid across the entire input. At each position, the filter computes a dot product between its weights and the local patch of input pixels beneath it. This operation is the discrete *convolution*.

The crucial insight is *weight sharing*: the *same* filter weights are reused at every spatial position. A filter that detects a horizontal edge will detect horizontal edges everywhere in the image, not just at the location it was "assigned" to. This reduces the number of parameters dramatically and encodes the prior that a useful feature detector should be applicable everywhere.

Worked Example

Tiny convolution example. Suppose the input is a 4×4 grayscale image and we apply a 3×3 filter K:

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

This is a vertical-edge detector (Sobel-like). At the top-left position, the output value is the element-wise product of K with the top-left 3×3 patch of the input, summed together. Slide the filter one step to the right and repeat. The result is a 2×2 *feature map* (for stride 1, no padding). The

nine weights in K are shared across all four output positions — the filter "looks for" the same vertical edge pattern everywhere.

13.2.2 Feature Maps and Depth

A single filter produces a single *feature map*: a two-dimensional activation pattern showing where in the image that filter's pattern was detected. In practice, a convolutional layer applies C filters in parallel, producing C feature maps stacked into a three-dimensional tensor. The first convolutional layer of a deep CNN typically learns low-level detectors — oriented edges, colour blobs. Subsequent layers apply their filters to the previous layer's feature maps, effectively learning detectors for combinations of lower-level patterns. This stacking is precisely the hierarchical feature learning described in the previous section, now implemented through spatially structured convolutions.

13.2.3 Pooling and Translation Equivariance

After a convolutional layer, *pooling* reduces the spatial dimensions of the feature maps. *Max pooling* partitions the feature map into non-overlapping $k \times k$ regions and retains the maximum activation in each region. This serves two purposes: it introduces a degree of *translation invariance* (a small shift of the input changes which cells the maximum falls in, but often not which region "wins"), and it reduces the number of activations propagated to subsequent layers, controlling computational cost and overfitting.

Formally, a convolution operation is *translation equivariant*: if the input is shifted by δ, the output feature map shifts by the same δ. Max pooling on top of equivariance gives approximate *invariance* over the pooling window size. This inductive bias is very well matched to image recognition, where the category of an object does not change because it appears 10 pixels to the left.

Key Idea

CNNs achieve parameter efficiency through weight sharing (each filter is reused at every position) and appropriate depth. A typical deep CNN uses orders of magnitude fewer parameters than a fully connected network of comparable capacity on image data, while also generalising better because

its inductive biases match the structure of natural images.

13.3 From CNNs to Transformers: The Need for Global Context

CNNs are powerful but limited by the size of their receptive fields. A neuron in a convolutional layer only "sees" a local patch of the input; to integrate information across the whole image or whole sentence, the network must stack many layers until the receptive field grows large enough. For sequential data such as language, this is particularly problematic: understanding "The bank where I deposit money" requires relating "bank" to "money" across an arbitrarily long span, something local convolutions handle poorly.

Recurrent networks (RNNs, LSTMs) address sequential dependencies by maintaining a hidden state that is updated token-by-token. But they process sequences *sequentially*, which prevents parallelisation over time steps during training and causes gradients to decay or explode over long sequences.

Transformers (Vaswani et al., 2017) resolve both limitations at once by replacing recurrence with *self-attention*: a mechanism in which every token directly attends to every other token in the sequence, regardless of distance, and the whole computation can be parallelised across positions.

13.4 Self-Attention and the Transformer

13.4.1 Scaled Dot-Product Attention

The core operation of a Transformer is *scaled dot-product attention*. Given a sequence of n input vectors, we derive three matrices by learned linear projections:

- $Q \in \mathbb{R}^{n \times d_k}$ — the *query* matrix,
- $K \in \mathbb{R}^{n \times d_k}$ — the *key* matrix,
- $V \in \mathbb{R}^{n \times d_v}$ — the *value* matrix.

The attention output is then:

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

Intuitively, think of this as a soft retrieval system: each query vector q_i (the

i-th row of Q) asks "which tokens are most relevant to me?" It answers by computing a dot product with every key vector k_j (rows of K); a high dot product means token j is relevant to query i. These scores fill an $n \times n$ attention matrix. Applying softmax row-wise converts the raw scores into a probability distribution over positions. Finally, multiplying by V produces a weighted sum of value vectors — the output for position i is a blend of all values, weighted by how relevant each position was judged to be.

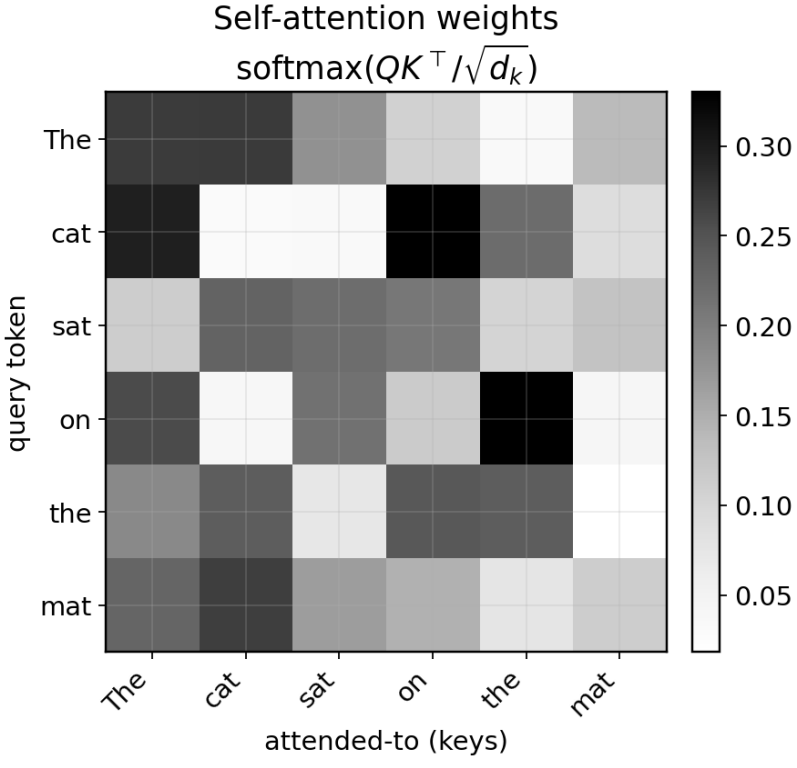

Figure 13.1: Self-attention weights: each query token (row) distributes attention over all tokens (columns) via softmax($QK^\top/\sqrt{d_k}$). Darker cells = higher weight.

13.4.2 Why Divide by $\sqrt{d_k}$?

(!) Exam Trap

This is one of the most frequently tested exam details. Do not skip it. When d_k is large, the dot products $q_i^\top k_j$ tend to grow in magnitude proportionally to $\sqrt{d_k}$ (assuming unit-variance entries). Very large dot products push the softmax into regions where its output is nearly one-hot — almost all weight on a single token — and its gradient becomes nearly zero. Dividing by $\sqrt{d_k}$ rescales the scores to a range where softmax remains well-conditioned and gradients flow effectively during training.

13.4.3 Queries, Keys, and Values — An Analogy

The lecture draws an analogy to information retrieval. Imagine searching for a song: your *query* is what you type into a search engine; the *keys* are metadata tags attached to each song; the *values* are the actual song files returned. The attention mechanism asks: given this query, how well does each key match? The answer determines which values to blend into the output. In language, the query might represent a word whose meaning is ambiguous; the keys represent context words that help resolve the ambiguity; and the values carry the contextual information to be mixed in.

A critical property is that attention is computed between all pairs of positions simultaneously. The self-attention over n tokens requires $O(n^2)$ operations (one dot product for every query-key pair), but all of these can be computed in parallel on a GPU, unlike the sequential steps of an RNN.

13.4.4 Multi-Head Attention

A single attention head learns one way of measuring relevance between tokens. *Multi-head attention* runs h attention heads in parallel, each with its own learned projection matrices W_i^Q, W_i^K, W_i^V. The outputs of all heads are concatenated and linearly projected back to the model dimension:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O$$

where $\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$. Different heads can specialise: one head may capture syntactic subject-verb agreement, another may

track coreference chains, a third may link a pronoun to its antecedent many tokens away. Empirical analysis of trained Transformers confirms that early layers develop heads sensitive to local dependencies, while later layers develop heads that attend globally across the sequence.

13.4.5 Positional Encoding

(!) Exam Trap

Self-attention is *permutation-equivariant*: if you shuffle the order of the input tokens, the set of output vectors is shuffled in exactly the same way, but no information about the original order is retained. The model has no innate sense of whether token j came before or after token i. For language — where word order is grammatically and semantically critical — this is a serious problem. Positional encoding is required.

To inject position information, the Transformer adds a *positional encoding* vector p_i to each token embedding w_i before the first attention layer, giving input $e_i = w_i + p_i$. The original Vaswani et al. (2017) paper used *absolute sinusoidal* encoding: the $2k$-th dimension of p_i is $\sin(i/10000^{2k/d})$ and the $(2k+1)$-th is $\cos(i/10000^{2k/d})$. This choice produces a unique, smooth signal for every position and generalises to sequence lengths not seen during training. Many later models replace this with learned positional embeddings or *relative* position encodings that encode the *difference* between positions rather than their absolute values.

Key Idea

Position encoding is not optional. Without it, a Transformer treats "The dog bit the man" and "The man bit the dog" as identical inputs (same tokens, different order) and would produce the same output for both — a catastrophic failure for any task depending on word order.

13.5 Encoder, Decoder, and the Full Transformer

The original Transformer (Vaswani et al., 2017) was proposed for sequence-to-sequence tasks such as machine translation. Its architecture has two halves.

The Encoder. The encoder reads the full source sequence in one pass. It consists of N identical layers, each containing two sublayers: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. Both sublayers are wrapped with a residual connection and layer normalisation: $\text{output} = \text{LayerNorm}(x + \text{Sublayer}(x))$. Because the encoder attends bidirectionally — each token can attend to every other token to its left *and* right — it builds rich contextual representations of the entire input.

The Decoder. The decoder generates the output sequence token by token. Its N layers contain three sublayers: a *masked* self-attention layer (which prevents each output position from attending to future positions, enforcing left-to-right generation), a *cross-attention* layer (which attends to the encoder's output, allowing the decoder to "look at" the full source), and a feed-forward network. At inference time, the decoder is run autoregressively: the previously generated tokens are fed back as input to predict the next token.

13.6 BERT and GPT: Two Paradigms

The Transformer architecture spawned two dominant families of pre-trained language models, which differ fundamentally in which half of the architecture they use and how they are trained.

13.6.1 BERT — Bidirectional Encoder Representations from Transformers

BERT (Devlin et al., 2019) uses the encoder half of the Transformer and discards the decoder entirely. Its defining training objective is *Masked Language Modelling* (MLM): approximately 15% of the input tokens are randomly replaced with a special `[MASK]` token, and the model is trained to predict the original identity of those tokens from the surrounding context. Because the encoder is bidirectional, BERT can use context from *both sides* of a masked token simultaneously — "The `[MASK]` ate the fish" — making its contextual representations semantically rich.

A second pre-training task, *Next Sentence Prediction* (NSP), trains the model to classify whether two sentences appeared consecutively in the original corpus, encouraging discourse-level understanding. After pre-training on billions of words, BERT is fine-tuned on downstream tasks (question answering, named entity

recognition, textual entailment) by adding a small task-specific head on top of the frozen or lightly adapted encoder.

13.6.2 GPT — Generative Pre-trained Transformer

GPT (Radford et al., 2018, and successors) uses the decoder half of the Transformer. Its training objective is *causal language modelling*: given the previous tokens $w_1, \ldots, w_{t-1}$, predict the next token w_t. The masked self-attention in the decoder enforces left-to-right *unidirectionality* — each token may only attend to tokens that precede it. This makes GPT inherently suited to text generation: at inference time it samples the next token, appends it, and repeats.

GPT models have scaled dramatically: GPT-3 was trained on roughly 570 GB of text with 175 billion parameters; GPT-4's scale is undisclosed but estimated in the trillions. At scale, autoregressive language models exhibit emergent in-context learning: they can solve new tasks from a handful of examples provided in the prompt, without any parameter update.

13.6.3 Comparison Table

Table 13.1: BERT vs. GPT at a glance.

Property	**BERT**	**GPT**
Architecture	Encoder-only	Decoder-only
Directionality	Bidirectional	Unidirectional (left-
Training objective	Masked LM + Next Sentence Prediction	Causal (autoregress
Attention mask	Full (all tokens see all tokens)	Causal (no future t
Primary use	Understanding tasks (classification, QA)	Generation tasks (t
Pre-training data (base)	∼3.4 billion words	∼40 GB text (GPT
Parameters (base)	110 million (BERT-BASE)	117 million (GPT-1
Fine-tuning style	Add task head; fine-tune whole model	In-context or fine-t

(!) Exam Trap

Do not confuse the directionality of BERT and GPT. BERT is bidirectional because its encoder self-attention is unmasked — during masked LM training, the prediction for a masked token at position i draws on all other

positions, including those to the *right* of i. GPT is unidirectional because its decoder self-attention is causally masked — position i can only attend to positions 1 through i, never to future tokens. Mixing this up in an exam answer is a common and costly error.

13.7 A Worked Attention Example

Worked Example

Tiny self-attention (3 tokens, $d_k = 2$).
Suppose we have three tokens with the following query, key, and value matrices (already projected):

$$Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad K = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0.5 & 0.5 \end{pmatrix}$$

Step 1 — compute raw scores $S = QK^\top$:

$$S = \begin{pmatrix} 1\cdot 1 + 0\cdot 0 & 1\cdot 0 + 0\cdot 1 & 1\cdot 1 + 0\cdot 1 \\ 0\cdot 1 + 1\cdot 0 & 0\cdot 0 + 1\cdot 1 & 0\cdot 1 + 1\cdot 1 \\ 1\cdot 1 + 1\cdot 0 & 1\cdot 0 + 1\cdot 1 & 1\cdot 1 + 1\cdot 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

Step 2 — scale by $1/\sqrt{d_k} = 1/\sqrt{2} \approx 0.707$:

$$\hat{S} = \begin{pmatrix} 0.71 & 0 & 0.71 \\ 0 & 0.71 & 0.71 \\ 0.71 & 0.71 & 1.41 \end{pmatrix}$$

Step 3 — apply softmax row-wise (row 1, for illustration):

$$\text{softmax}(0.71,\ 0,\ 0.71) = \frac{(e^{0.71},\ e^{0},\ e^{0.71})}{e^{0.71} + e^{0} + e^{0.71}} \approx (0.39,\ 0.22,\ 0.39)$$

Token 1 attends roughly equally to itself and token 3, and less to token 2.
Step 4 — weighted sum over V for token 1:

$$\text{output}_1 = 0.39 \cdot (1, 0) + 0.22 \cdot (0, 1) + 0.39 \cdot (0.5, 0.5) \approx (0.585,\ 0.415)$$

The output for token 1 mixes information from all three value vectors, weighted by how strongly each key matched its query.

13.8 Transfer Learning and Foundation Models

A recurring theme in deep learning is *transfer learning*: pre-train a large model on a large, general dataset, then fine-tune or adapt it to a specific downstream task. The Transformer architecture is especially amenable to this paradigm. Pre-trained models such as BERT and the GPT family serve as *foundation models* — general-purpose representations that transfer across tasks, domains, and even

modalities.

Research by Ballout et al. (presented in the lecture) demonstrated that pre-trained language models transfer to non-language tasks including computer vision and structured prediction, outperforming non-pre-trained Transformers. Attention weight analysis on arithmetic tasks revealed that early layers develop attention to local structure first, then later layers fan out to global dependencies — a pattern that mirrors human hierarchical problem-solving strategies.

A further result concerns the role of instructions during training. For small models (around 60 million parameters), providing explicit output explanations alongside examples significantly improves performance and generalisation, effectively compensating for limited capacity. For large models (around 3 billion parameters), detailed explanations provide marginal additional benefit — the model learns to generalise from examples regardless.

13.9 Exam Traps: Common Mistakes and How to Avoid Them

(!) Exam Trap

Why divide by $\sqrt{d_k}$? Because large d_k causes dot products to grow in magnitude, pushing softmax into saturation (near-zero gradients). Dividing by $\sqrt{d_k}$ normalises the scale and keeps gradients healthy. This answer must be precise: say "saturates the softmax" and "vanishing gradients in the attention scores", not merely "makes the numbers smaller".

(!) Exam Trap

Positional encoding is not optional. Attention is permutation-equivariant — it produces the same set of outputs regardless of input order. Without positional encoding, the model is entirely order-blind. Sinusoidal encoding (Vaswani et al. original paper) uses *absolute* positions, not relative ones — the lecture slides contain an error claiming relative encoding was used in the original paper; in an exam context, report what the question expects.

(!) Exam Trap

CNN weight sharing. In a convolutional layer, the same filter weights are shared across all spatial positions. This is not merely a computational convenience — it encodes the inductive bias that the same feature detector should be applied everywhere in the input. A fully connected layer does *not* share weights; a convolutional layer always does.

(!) Exam Trap

BERT is bidirectional; GPT is unidirectional. BERT's encoder sees all tokens when predicting masked ones. GPT's decoder is causally masked: token i sees only tokens 1 through i. Do not confuse "bidirectional" with "encoder" in general — the distinction is about the attention mask, not the module name.

(!) Exam Trap

Universal approximation and depth. The slides prove that two hidden layers suffice for universal approximation. This does not mean deeper networks are unnecessary: in practice, deeper networks are more parameter-efficient and generalise better. The universal approximation theorem says *what is possible* in principle, not what is *practical*.

(!) Exam Trap

Dropout is training-only. At test time, dropout is turned off (all units active) and activations are rescaled — or equivalently, inverted dropout is applied during training. Never apply dropout at inference.

In a Nutshell

Chapter Summary: Deep Learning, CNNs, and Transformers

- **Deep learning** replaces hand-crafted features with *learned hierarchical representations*: early layers detect simple patterns, later layers detect complex compositions.
- Key training challenges — vanishing gradients, overfitting, compu-

tational cost — are mitigated by **ReLU**, **residual connections**, **dropout**, **weight decay**, and **early stopping**.

- **CNNs** exploit *locality* and *translation equivariance* via shared convolutional filters. Weight sharing dramatically reduces parameters. Max pooling adds approximate invariance.
- **Self-attention** computes $\text{Attention}(Q, K, V) = \text{softmax}(QK^\top/\sqrt{d_k})V$. The $\sqrt{d_k}$ divisor prevents softmax saturation. Multi-head attention runs multiple heads in parallel to capture diverse relationship types.
- **Positional encoding** is mandatory because self-attention is permutation-equivariant — without it, the model is order-blind.
- The **Transformer** encoder is bidirectional (full self-attention); the decoder is causal (masked self-attention + cross-attention).
- **BERT** (encoder-only, bidirectional, masked LM) excels at understanding tasks. **GPT** (decoder-only, unidirectional, autoregressive LM) excels at text generation.
- Pre-trained foundation models *transfer* to new tasks and even new modalities, with smaller models benefiting more from explicit training instructions than larger models.

Chapter 14

The Exam Paper: *Transformers Struggle to Learn to Search*

This chapter is dedicated to the paper assigned for the oral module exam:

> Abulhair Saparov, Srushti Pawar, Shreyas Pimpalgaonkar, Nitish Joshi, Richard Yuanzhe Pang, Vishakh Padmakumar, Seyed Mehran Kazemi, Najoung Kim, He He (2024). ***Transformers Struggle to Learn to Search.*** arXiv:2412.04703.

The exam format requires you to (1) *briefly present* the paper, (2) *critically evaluate* it, and (3) *place it in a larger context* (about 10–15 minutes), followed by a discussion of the paper and of AI in general. This chapter gives you the substance for all three parts, plus the connection to the classical search algorithms from Chapter 2 and a bank of tricky oral-exam questions. It is no accident that this paper was chosen: it sits exactly at the intersection of two of the agreed focus topics — **(Local) Search** and **Machine Learning** (specifically Transformers).

14.1 What the Paper Does

Research question. Search underlies reasoning, planning and navigation, and large language models (LLMs) are known to perform it poorly. The paper asks

whether this failure is due to (a) too little data, (b) too few parameters, or (c) a *fundamental limitation of the transformer architecture*.

Testbed. The authors reduce "search" to its cleanest possible form: the **graph-connectivity** problem on a *directed acyclic graph* (DAG). Given a DAG, a start vertex and a goal vertex (guaranteed reachable), the model must output the *next vertex on a path* from start to goal. This task is deliberately minimal yet foundational: it is equivalent to *proof search* in a small fragment of propositional logic (each vertex is a fact, each edge an implication). The argument is: if a model cannot do *this*, it has no hope on richer reasoning.

Lookahead. Difficulty is measured by the **lookahead** L: roughly, how many steps one must search forward from the start before committing to the correct first move. Larger lookahead means harder search.

Three training distributions (the central methodological idea):

- **Naïve** (Erdős–Rényi DAGs): almost always produces tiny lookaheads, so a model can succeed using *shortcuts/heuristics* rather than genuine search — and therefore fails when real search is needed.
- **Star**: star-shaped graphs; also exploitable by shortcuts.
- **Balanced**: lookahead made *uniform*, with shortcuts deliberately engineered out. *Only here* does the transformer learn to search almost perfectly, generalising across the trained range of lookaheads.

Key Idea

Whether a transformer learns to search depends dramatically on the **training distribution**, not only on the model. If the data permit shortcuts, the model learns shortcuts; only a carefully balanced distribution forces it to acquire a genuine, generalisable search procedure.

The models are tiny GPT-2-style transformers (e.g. hidden dimension 16, 6–8 layers, one attention head per layer), trained with effectively *unlimited*, procedurally generated data (streaming training). That "limitless data" design is what lets the authors rule out "too little data" as the explanation.

14.2 The Algorithm the Transformer Learns

A second contribution — arguably the more durable one — is *methodological*: a new **mechanistic-interpretability** technique that *reconstructs the computation*

graph of the trained model from its activations and attention patterns, using **activation patching**. Crucially, it does *not* assume the algorithm in advance; it discovers it.

What it finds is an **exponential path-merging** algorithm:

- Each vertex token stores, in its embedding, the **set of vertices reachable from it** (within some number of steps).
- At each layer, the attention mechanism *unions* a vertex's reachable set with that of a vertex on its current frontier.
- Therefore the reachable set can **double in size at every layer**, and the model searches over a number of vertices *exponential in the number of layers*.

This matches the theoretical result (Sanford et al.) that graph connectivity needs only a *logarithmic* number of layers in the graph size.

In a Nutshell

Path-merging in one sentence. Every vertex simultaneously builds up the set of vertices it can reach; each layer merges sets along edges, so reachability spreads exponentially fast in depth — a parallel, all-vertices-at-once style of search.

14.3 Key Results

1. **Scaling does not rescue large graphs.** As the input graph grows, the model has increasing difficulty learning the task, and *adding parameters does not fix it.* Bigger is not the answer.
2. **Chain-of-thought helps only partially.** Permitting intermediate steps (depth-first-search traces, selection-inference) makes the task learnable with a *constant* number of layers, but the model *still fails* on larger graphs.
3. **Conclusion.** Standard training plus scaling will likely *not* give transformers robust search; alternative training (e.g. curriculum learning) or architectures (e.g. looped transformers) may be required.

14.4 Connection to Classical Search Algorithms

This is the bridge to Chapter 2 (and to uninformed/informed search more broadly), and it is exactly where an examiner will probe.

First, a clean distinction. The paper's task is *classical, path-based search* — the path matters — *not* local search. Recall from Chapter 2 that local search (hill climbing, simulated annealing) discards the path and keeps only a current state. Graph connectivity is the opposite: the output *is* a step on a path. So the relevant comparison is to BFS, DFS and A*, not to hill climbing.

Path-merging vs. BFS/DFS. Classical BFS expands a frontier *sequentially*, one layer of the graph per iteration, and would need a number of steps proportional to the path length. The transformer instead computes reachability for *all vertices in parallel* and *doubles* each reachable set per layer — this is essentially repeated squaring of the reachability relation, i.e. computing the **transitive closure** in a logarithmic number of layers. So where BFS is linear and serial, path-merging is logarithmic and parallel.

	Classical search (BFS/DFS/A*)	**Learned transformer (path-merging)**
How it explores	Sequential frontier expansion	All vertices in parallel
Steps needed	$\propto$ path length / depth	$\propto$ log(lookahead) layers
Correctness	Provably complete & correct	Approximate; degrades with graph size
Memory of path	Explicit	Implicit in embeddings
Guarantees	Yes (admissible h for A*)	None

DFS as chain-of-thought. The paper explicitly tests a DFS-trace variant: the model emits the sequence of visited vertices, predicting the next one — exactly the DFS you know from classical search, but produced token by token. This connects the "intermediate steps" of chain-of-thought to the explicit stack-based exploration of DFS.

Heuristics vs. shortcuts. In informed search, a *heuristic* (e.g. an admissible h in A*) is a principled estimate that preserves correctness. The "shortcuts" the transformer learns on the naïve distribution are the opposite: *statistical*

regularities of the training data that happen to give the right answer on easy graphs but break on hard ones. The paper's balanced distribution is precisely an attempt to make such shortcuts useless, forcing a correct procedure.

Why this matters (neuro-symbolic AI). Classical search algorithms are *provably correct and scale gracefully*; a learned transformer only approximates them and degrades with size. This is a strong argument for **neuro-symbolic / hybrid systems** that offload search to a symbolic solver or external tool, rather than hoping a monolithic LLM will internalise it. (This is also a natural place to mention CSP search from Chapter 3: backtracking with arc consistency gives *exact* guarantees that a learned approximator cannot.)

14.5 Critical Evaluation

Being "critical" here means separating what the paper *claims* from what it *demonstrates* — fairly, and with a rebuttal ready.

In a Nutshell

The meta-formula for a critique. "The paper claims **X**. The evidence supports the narrower **X′**. The gap is **Y**. This matters because **Z**. To close it you would need **W**."

Tier 1 — the strong, defensible critiques.

1. **Learnability ≠ expressivity (the title overreaches).** They show a *training/optimisation* difficulty (high seed variance, sensitivity to the data distribution), not that transformers *cannot represent* search. They themselves cite Merrill & Sabharwal (2024): chain-of-thought transformers are Turing-complete. The honest claim is narrower: "hard to *train*, with these tiny models and this setup." *Be ready for:* "but adding parameters does not help" — true only within the tested tiny regime; it refutes trainability-at-scale, not representability.
2. **Extrapolation gap.** Hidden dimension 16, ≤60M parameters. Generalising to 100B+ frontier models is a leap, and the authors hedge ("it is possible that scaling to much larger model sizes may lead to emergent searching ability").
3. **Internal tension: data vs. architecture.** The *positive* result needs a hand-crafted balanced distribution that does not exist in the real world.

But that implies the failure is about *data/shortcuts*, which undercuts the "architectural limitation" framing. You cannot simultaneously claim "it is the distribution" and "it is the architecture."

Tier 2 — good supporting critiques.

- **Interpretability-friendly architecture ≠ a real transformer:** 1-hot embeddings, *concatenated* (not added) positional encodings, no causal mask, a single attention head. The path-merging finding may be partly an artefact of this setup; the rotary/decoder-only robustness check is only in an appendix.
- **"Search" is operationalised very narrowly** (DAG connectivity). The leap to "reasoning" is asserted via the proof-search equivalence, not demonstrated on real tasks.

Being a fair critic. State what the critique does *not* touch: the mechanistic-interpretability method is a genuine contribution and the path-merging finding is well supported; the data-distribution-sensitivity result is robust and important regardless of the overclaim.

Constructive critique ("how would you improve it?"). Test medium-sized models to bridge the extrapolation gap; check whether path-merging appears in *real* pretrained LLMs; try curriculum learning / looped transformers; characterise *theoretically* why the optimisation is hard (the loss landscape).

14.6 Placing It in a Larger Context

- **"Reasoning vs. pattern-matching" in LLMs:** alongside Kambhampati (LLMs & planning), Saparov & He (proof search), Bachmann & Nagarajan (pitfalls of next-token prediction). Recurring theme: LLMs often imitate reasoning via shortcuts.
- **Scaling laws and emergence:** Wei et al. ("emergent abilities") vs. Schaeffer et al. ("emergence is a mirage of metric choice"). This paper is a counterexample to pure scaling optimism.
- **Neuro-symbolic AI:** the case for hybrid systems (LLM + symbolic planner/solver), since classical search is correct and scalable.
- **Mechanistic interpretability** as a growing field: understanding *what* networks compute, not just that they work.
- **Link to the course:** the lectures give the gold-standard search algorithms;

this paper shows learned models do not reliably reach them — a justification for why classical AI methods still matter.

14.7 Tricky Oral-Exam Questions

Q: What is the single most important contribution — the negative or the positive result?
A: Both matter, but frame it carefully. The *positive* result (transformers *can* learn search) is conditional on an artificial balanced distribution; the *negative* result (scaling and chain-of-thought do not fix large graphs) is the practically relevant one. The most durable contribution is arguably *methodological*: the mechanistic reconstruction of the learned algorithm.

Q: Why three distributions? Could they not just use random graphs?
A: Random (naïve) graphs almost never require large lookahead, so a model can score well using shortcuts without ever learning to search. The balanced distribution makes lookahead uniform and engineers shortcuts out, isolating genuine search ability. The three distributions together *prove* the failure is about shortcuts in the data, not inability in principle.

Q: Explain the path-merging algorithm and why it needs only log **layers.**
A: Each vertex stores its reachable set; each layer unions sets along edges, so a set can double per layer. Doubling L times reaches lookahead 2^L, hence $\log_2$(lookahead) layers suffice. It is effectively repeated squaring of the reachability relation (transitive closure), done for all vertices in parallel.

Q: Is this local search or classical search? Justify.
A: Classical, path-based search — the output is a step *on a path*, and the path matters. Local search (Chapter 2) discards the path and optimises a single state. So compare path-merging to BFS/DFS/A*, not to hill climbing.

Q: The title says transformers "struggle to learn" search. Is that fully justified by the evidence?
A: Partially. They show *trainability* is hard for these small models and that scaling does not help *in that regime*. They do *not* show representational impossibility — indeed CoT-transformers are Turing-complete (Merrill & Sabharwal). So the strong architectural reading overreaches; the defensible claim is about learnability under standard training.

Q: How does this connect to the rest of the course?
A: It contrasts *learned* approximate search with the *exact, provably correct* search algorithms from the syllabus (BFS/DFS/A*, CSP backtracking with arc consistency). It motivates neuro-symbolic systems and connects to ML themes of inductive bias, shortcut learning and generalisation.

Q: What would you do next if this were your project?
A: Bridge the scale gap with medium models; probe real pretrained LLMs for path-merging circuits; try curriculum learning and looped/recurrent architectures; and seek a theoretical account of why SGD struggles here (loss-landscape / optimisation analysis).

In a Nutshell

The exam paper — In a Nutshell.

- **Claim:** small transformers can learn graph-connectivity search, but only with a balanced, shortcut-free training distribution; scaling and chain-of-thought do not make large graphs work.
- **Method:** a new mechanistic-interpretability technique reconstructs the learned algorithm — *exponential path-merging* (reachable-set doubling per layer; log-depth, parallel transitive closure).
- **Search link:** classical (path-based), not local. Parallel/logarithmic vs. BFS's serial/linear; correct-but-rigid classical algorithms vs. approximate-but-flexible learned ones $\Rightarrow$ neuro-symbolic argument.
- **Critique:** learnability $\neq$ expressivity; tiny models; data-vs-architecture tension. But the interpretability method and distribution-sensitivity findings are solid.
- **Context:** reasoning-vs-pattern-matching debate, scaling/emergence, neuro-symbolic AI, mechanistic interpretability.

Part I

Exam-Style Quiz

Chapter 15

Practice Questions

This chapter contains **130 exam-style questions** covering every major topic of the course. Work through each section before reading the answers. The questions are numbered within each section so that Question n in a section corresponds exactly to Answer n in the **Answer Key** chapter that follows. Questions are labelled by approximate difficulty: **(R)** recall, **(E)** explain / compare, **(C)** calculation or derivation.

15.1 Local Search

1. (R) Name the three classical failure modes of pure Hill Climbing and explain each in one sentence.
2. (R) What is the key difference between Local Beam Search with k states and running k independent Hill Climbing searches in parallel?
3. (E) Write down the Simulated Annealing acceptance probability for a *downhill* move. Define Δ and T, and explain how the formula behaves when T is large vs. small.
4. (R) What guarantee does Random-Restart Hill Climbing provide, and what guarantee does it *not* provide?
5. (E) Why does Simulated Annealing guarantee the global optimum in theory but not in practice?
6. (E) Explain *Stochastic Beam Search* and state the problem it solves compared with standard Local Beam Search.
7. (E) Naive one-point crossover fails for the Travelling Salesman Problem.

Why, and what is used instead?

8. (R) What is a *memetic algorithm*? Why is it often better than a plain Genetic Algorithm?
9. (E) Describe the three roles of **selection**, **crossover**, and **mutation** in a Genetic Algorithm.
10. (E) What properties must the fitness function of a GA satisfy for selection pressure to be meaningful?
11. (E) Analyse the neighbourhood-size trade-off in local search: what happens with a *small* neighbourhood vs. a *large* one?
12. (E) On the 8-Queens problem, why is *stochastic* Hill Climbing often more successful than *steepest-ascent* Hill Climbing?
13. (R) What distinguishes Local Search from classical search algorithms such as A* with respect to memory usage and what constitutes a "solution"?

15.2 Constraint Satisfaction

1. (R) State the three components of a Constraint Satisfaction Problem (CSP) and define what a *solution* is.
2. (R) Describe what AC-3 does at each step, and what it concludes when a domain becomes empty.
3. (C) What is the time complexity of AC-3? Name the variables and give the intuition behind the bound.
4. (R) Explain MRV and LCV. Which is for variable selection, which is for value selection, and what is the goal of each?
5. (R) What is the *degree heuristic* and when is it typically used in the search?
6. (E) A triangle of three variables (X, Y, Z), each with domain $\{R, G\}$, carries pairwise "$\neq$" constraints. Is this CSP arc-consistent? Does it have a solution? What does this show about AC-3?
7. (E) Define k-consistency and *strong* k-consistency. Explain the relationship to arc consistency and path consistency.
8. (E) Why does strong n-consistency (where n is the number of variables) guarantee a backtrack-free solution? Why do we not always enforce it?
9. (R) How is Local Search adapted for CSPs? Describe the state space, the neighbourhood, and the objective function.

10. (E) Local Search on a CSP finds a state with fewer satisfied constraints than the total number of constraints. What can you conclude, and why?
11. (R) Name the eight RCC-8 spatial relations and state which predicate they are all based on.
12. (R) What is *node consistency* (1-consistency)? Why is it the cheapest consistency concept to enforce?
13. (E) Describe Forward Checking. How does it differ from full arc-consistency propagation, and what is the concept of a *domain wipeout*?

15.3 Planning — Classical and Probabilistic

1. (R) Explain the frame, qualification, and ramification problems in one sentence each. Give a concrete example for each.
2. (R) List the three lists of a STRIPS operator and explain how STRIPS implicitly "solves" the frame problem.
3. (E) How does STRIPS's solution to the frame problem differ from the treatment in Situation Calculus? How many frame axioms does Situation Calculus need in the worst case with n actions and m facts?
4. (R) What does STRIPS assume about the world (which world assumption), and which formalisms do *not* make this assumption?
5. (C) Write a STRIPS action for `put(X,Y)` in the Blocksworld, giving PRE, ADD, and DEL lists.
6. (E) Describe the *ignore-delete* heuristic for planning. Is it admissible? Why or why not?
7. (R) How does PDDL extend STRIPS syntactically, and what important structural separation does it introduce?
8. (E) State-space search vs. plan-space (partial-order) planning — what is the fundamental difference in the search space?
9. (E) Why is classical planning PSPACE-complete rather than merely NP-complete?
10. (R) State the four components of an MDP tuple. Note any component that is sometimes mistakenly included.
11. (R) State the Markov property formally.
12. (E) What distinguishes a *plan* (classical planning) from a *policy* (MDP)?

13. (C) Write the Bellman equation for a fixed policy π. Name every term.
14. (C) Write the Bellman optimality equation for $V^*(s)$. How does it differ from the policy-evaluation equation?
15. (E) Describe the two phases of Policy Iteration and state when the algorithm terminates.
16. (E) What is the main difference between Value Iteration and Policy Iteration in terms of per-iteration cost and number of iterations?
17. (E) Why does $\gamma = 1$ cause convergence problems in non-terminating MDPs?
18. (E) In the grid-world example, why is a step reward of -0.04 useful, and what changes if the step reward is $+0.04$?

15.4 Knowledge Representation

1. (R) What goes into the TBox and what goes into the ABox of a Description Logic knowledge base? Give one example for each.
2. (R) What does the General Concept Inclusion $C \sqsubseteq D$ mean semantically? Which direction does the subset go?
3. (R) List all constructors of the Description Logic ALC and give a one-line meaning for each.
4. (E) Explain the "vacuous truth" trap for the value restriction $\forall R.C$. Give a concrete example.
5. (E) How do you test the subsumption $C \sqsubseteq D$ with the tableaux algorithm? What concept do you form, and what does the result of the algorithm tell you?
6. (R) Name the five standard reasoning tasks for a DL knowledge base (split into TBox and ABox tasks).
7. (R) Open World Assumption vs. Closed World Assumption: which formalism uses which, and what is the practical difference?
8. (E) ALC lacks role hierarchies. What would you need to express $R_1 \sqsubseteq R_2$, and which DL extensions support this?
9. (R) Name all 13 Allen interval relations and explain how the Allen composition table is used.
10. (R) What is the key trade-off in knowledge representation between expressivity and tractability? Give a concrete example.

11. (E) Describe the three OWL dialects (OWL Lite, OWL DL, OWL Full) and how they differ in expressivity and decidability.
12. (R) What distinguishes RCC-8 from metric geometry? Which predicate is the sole primitive of RCC-8?

15.5 Vagueness and Uncertainty

1. (R) What is the fundamental difference between *vagueness* and *uncertainty*? Which does fuzzy logic address, and which does probabilistic logic address?
2. (C) Given $\mu_A(x) = 0.7$ and $\mu_B(x) = 0.4$, compute (a) $\mu_{A \cap B}(x)$, (b) $\mu_{A \cup B}(x)$, (c) $\mu_{A^C}(x)$ using Zadeh's standard operators.
3. (E) Show with a concrete numerical example that the law of excluded middle, $A \vee \neg A$, does not hold in fuzzy logic. Why is this feature intentional rather than a flaw?
4. (R) State the four axioms that a t-norm must satisfy. Give two examples of t-norms.
5. (E) Besides the Zadeh minimum t-norm, what is the *algebraic* t-norm and the corresponding algebraic s-norm? Illustrate with $t(0.5, 0.5)$.
6. (E) Why does probabilistic logic maintain $P(A \vee \neg A) = 1$ while fuzzy logic does not?
7. (R) Write down Kolmogorov's three axioms of probability. Derive $P(A^C) = 1 - P(A)$ from them.
8. (R) Define conditional probability. State the condition under which it is defined.
9. (E) The Sorites (Heap) paradox: describe the paradox, explain why classical logic cannot resolve it, and explain the fuzzy logic resolution.
10. (E) What is the Dempster-Shafer framework? Distinguish between belief $\mathrm{Bel}(A)$ and plausibility $\mathrm{Pl}(A)$, and explain what the interval $[\mathrm{Bel}(A), \mathrm{Pl}(A)]$ represents.
11. (C) If $P(\text{Bird}) = 0.8$ and $P(\text{Penguin} \mid \text{Bird}) = 0.2$, what is the minimum possible value of $P(\text{Penguin})$? State the law of total probability you use.

15.6 Machine Learning Foundations

1. (R) State Mitchell's formal definition of machine learning (the experience–task–performance formulation).
2. (R) What is *inductive bias* and why does every learning algorithm require one? Connect your answer to the No-Free-Lunch theorem.
3. (R) Name the three main categories of machine learning and give one concrete algorithm for each.
4. (C) Write the entropy formula $H(S)$ for a set S. State the extreme values and when they are attained.
5. (C) Write the Information Gain formula $\text{Gain}(S, A)$ and explain what it measures.
6. (E) Why does pure Information Gain favour attributes with many distinct values? What is the standard remedy used in C4.5?
7. (C) State the bias-variance decomposition of expected squared error. Define each term.
8. (E) What is *bagging*? Explain why it reduces variance but not bias.
9. (E) How does Random Forest differ from plain bagging? What is the additional mechanism, and why does it help?
10. (R) What is the Out-of-Bag (OOB) error and why is it a "free" generalization estimate?
11. (R) Describe k-fold cross-validation: procedure and main purpose.
12. (E) Describe the k-means algorithm. What does it optimize, and why does it only find local optima?
13. (E) Decision tree: why is a single ID3 tree considered a *high-variance* model? Name two strategies to fix this.
14. (E) Your decision tree achieves 0% training error and 25% test error. Name the phenomenon, the corresponding bias-variance term, and two distinct remedies.

15.7 Decision Trees and Ensembles

1. (R) Describe the ID3 algorithm step by step. What greedy criterion does it use at each node?
2. (C) A leaf contains 6 examples: all labelled `Yes`. Compute its entropy and

explain what this means for ID3.

3. (E) Why does Information Gain measure the *expected* reduction in uncertainty, and how does the weighted average of child entropies appear in the formula?
4. (E) You add a `UserID` column to your training data. ID3 selects it as the root split, giving 100% training accuracy but 50% test accuracy. Explain what happened and give the standard fix.
5. (E) Define pre-pruning and post-pruning of decision trees. State one concrete criterion for each.
6. (E) A Random Forest fixes a specific weakness of single decision trees. Name the weakness and describe the two complementary mechanisms RF uses to fix it.
7. (E) You are building a medical decision-support system where doctors must audit every recommendation. Compare a single pruned decision tree vs. Random Forest vs. XGBoost, and justify a choice.
8. (R) What is Gini impurity? Write its formula for a binary classification problem and compare it to entropy as a splitting criterion.

15.8 Support Vector Machines

1. (R) State the central idea of a Support Vector Machine. What are *support vectors*?
2. (C) Express the SVM margin as a function of w. Write the primal optimization problem (objective and constraints).
3. (C) Write the SVM Lagrangian $L(w, b, \alpha)$. Pay attention to the sign of the constraint term.
4. (E) State the KKT complementary-slackness condition for SVMs. What does it tell you about which training points are support vectors?
5. (R) Explain the kernel trick in one sentence. Why does it make high-dimensional feature spaces tractable?
6. (C) Write the RBF (Gaussian) kernel in both the γ-notation and the σ-notation. State the relationship between γ and σ.
7. (R) Name three common kernel functions and give their formulas.
8. (E) Soft-margin SVM: what do the slack variables ξ_i allow, and how does the hyperparameter C control the margin-vs.-accuracy trade-off?

9. (E) What is Mercer's condition? When is a function a valid kernel?
10. (R) State the perceptron training rule Δw. Explain each symbol.
11. (R) What does the perceptron convergence theorem guarantee, and what precondition must hold?
12. (E) Why can a single-layer perceptron not learn XOR? What was the historical consequence of this result?
13. (E) Compare the fundamental optimization methods used by SVMs and perceptrons. How do they differ in the type of problem they solve?

15.9 Neural Networks

1. (R) Describe how a Hopfield network stores patterns (the weight-setting rule) and how it retrieves them (the update rule). Write down the energy function $E(y)$.
2. (E) What are *spurious memories* in a Hopfield network? How do they arise, and what is the practical capacity rule of thumb?
3. (R) What is ReLU? Write its formula and explain why it is preferred over sigmoid for training deep networks.
4. (R) State the Universal Approximation Theorem for MLPs. What is its practical limitation?
5. (E) Describe the vanishing gradient problem. How do ReLU and residual connections each address it?
6. (R) Write the residual block computation $y = x + F(x)$. Show how the gradient flows through the identity shortcut.
7. (R) Name three standard regularization techniques for deep networks and give a one-sentence description of each.
8. (R) What is an autoencoder? Describe encoder, bottleneck, and decoder, and name two applications.
9. (R) Write the scaled dot-product attention formula. Define Q, K, and V.
10. (E) Why is the scaling factor $1/\sqrt{d_k}$ necessary in the attention formula?
11. (E) Why do Transformers need positional encoding? Describe the sinusoidal encoding scheme from Vaswani et al. (2017).
12. (E) Compare BERT and GPT with respect to training objective, directionality of context, and typical application.

13. (E) The Lottery Ticket Hypothesis: state the hypothesis and explain its implication for model pruning — why does pruning *before* training hurt?
14. (E) Ramsauer et al. (2020) showed a connection between modern Hopfield networks and the Transformer attention mechanism. Describe the core idea and why the storage capacity comparison matters.

15.10 Deep Learning and Transformers

1. (R) What is *multi-head attention*? Why is it used instead of a single attention head?
2. (E) Describe the full Transformer encoder block: list every sub-layer in order and state the purpose of each.
3. (R) What is *layer normalization* and why is it applied after each sub-layer in the Transformer?
4. (E) What is the computational complexity of self-attention with respect to sequence length n, and why is this a bottleneck for very long sequences?
5. (R) Distinguish pre-training and fine-tuning in the context of large language models.
6. (E) What is the masked self-attention mechanism used in GPT-style decoders? How does the mask enforce the autoregressive property?
7. (R) What is a Variational Autoencoder (VAE)? How does it differ from a standard autoencoder, and what loss function does it minimize?
8. (E) Describe the *batch normalization* technique. How does it stabilize training, and where is it typically applied relative to the activation function?
9. (E) Gradient clipping: what problem does it solve, and what does it do to the gradient vector?
10. (E) Explain the intuition behind weight decay (L2 regularization) as a Bayesian prior on the weights.
11. (E) Compare dropout during training vs. at inference time. What correction must be applied at inference?

Chapter 16

Answer Key

The answers below correspond one-to-one with the questions in the **Practice Questions** chapter. Each section heading and item number matches. Answers are *complete but concise*: aim to capture the key idea in 1–4 sentences as you would write in an exam.

16.1 Local Search

1. **Local maxima**: the current state is better than all neighbours but is not the global optimum; the algorithm stops. **Plateaus**: all neighbours have equal value, so there is no gradient to follow. **Ridges**: high-value regions that are narrow and oriented diagonally relative to the move operators; no single neighbour step stays on the ridge.
2. In Local Beam Search all k beams pool their successors and the globally best k are selected across the entire joint set. In k parallel Hill Climbers each search is isolated and selects only from its own successors. Beam Search lets beams share information and concentrate on promising regions; parallel HC cannot do this.
3. The acceptance probability is $P = e^{\Delta/T}$ where $\Delta = f(s') - f(s) < 0$ is the change in objective value (negative for downhill moves) and T is the current temperature. When T is large, $\Delta/T \approx 0$ so $P \approx 1$ (almost any move is accepted — exploration). When T is small, $\Delta/T \ll 0$ so $P \approx 0$ (downhill moves are almost never accepted — exploitation).
4. With n independent restarts the probability of finding the global optimum

grows monotonically and approaches 1 as $n \to \infty$. However, with any *finite* number of restarts there is **no hard guarantee**; it only increases the probability of success.

5. The theoretical guarantee requires a cooling schedule that decreases *infinitely slowly* (e.g. logarithmic). Practical schedules (linear or exponential) are far faster; at low temperature $e^{\Delta/T} \to 0$, so the algorithm can become trapped in a local optimum from which it cannot escape.
6. Stochastic Beam Search draws the next k states probabilistically in proportion to their fitness rather than deterministically choosing the best k. This solves the *clustering problem* of standard Local Beam Search: without stochasticity all beams tend to converge to the same region, losing diversity.
7. Permutation chromosomes (as in TSP) require every city to appear exactly once; simple one-point crossover produces offspring with duplicate and missing cities, making them invalid tours. **Uniform-Order Crossover** is used instead: a random binary template selects positions from Parent A; the remaining positions are filled in the order they appear in Parent B, guaranteeing a valid permutation.
8. A memetic algorithm is a Genetic Algorithm augmented with a local search step: after each reproduction phase, Hill Climbing (or another local searcher) is applied to every offspring. It combines the global exploration of a GA with the local refinement of HC, typically outperforming either in isolation.
9. **Selection**: chromosomes are chosen for reproduction with probability proportional to fitness, so better solutions reproduce more often. **Crossover**: two parents are recombined at one or more points to produce offspring that inherit partial solutions from each parent. **Mutation**: individual genes are randomly altered with a small probability, maintaining population diversity and enabling exploration of regions unreachable by crossover alone.
10. The fitness function must (a) impose a total ordering on solutions (higher fitness = better) and (b) produce selection probabilities that correlate meaningfully with fitness values (e.g. roulette-wheel proportional to f). If all fitness values are nearly equal, selection pressure collapses and the population drifts randomly rather than converging.
11. A **small** neighbourhood means fast per-step evaluation but many local optima (the algorithm gets trapped easily). A **large** neighbourhood has fewer local optima and can escape more readily, but each step is computationally

expensive and the search slows down. The right size is problem-dependent.

12. Steepest-ascent HC on 8-Queens frequently encounters plateaus where many neighbours have the same conflict count; choosing arbitrarily among tied "best" neighbours does not help the search progress. Stochastic HC picks a *random* improving (or equally good) neighbour, which allows it to explore the plateau and escape more easily.
13. Local Search keeps only the **current state** in memory (no frontier, no path). The goal is to find a good state; the path to it is irrelevant. Classical search (BFS, A*) must reconstruct the path and therefore maintains the full frontier and parent pointers.

16.2 Constraint Satisfaction

1. A CSP is a triple $(\{X_1, \ldots, X_n\}, \{D_1, \ldots, D_n\}, R)$: a set of variables, a set of domains (one per variable), and a set of constraints (relations over subsets of variables). A **solution** is a complete assignment of all variables that satisfies every constraint simultaneously.
2. For each arc (X_i, X_j) in the queue AC-3 removes from D_i every value that has no supporting value in D_j. Whenever D_i shrinks, all arcs pointing *into* X_i are re-added to the queue. If any domain becomes empty the CSP is **inconsistent** and the algorithm terminates.
3. The complexity is $O(c \cdot d^3)$ where c is the number of binary constraints (arcs) and d is the maximum domain size. Intuition: each value can be removed at most d times per arc; each removal may trigger $O(d)$ re-checks, each costing $O(d)$.
4. **MRV** (Minimum Remaining Values) selects the *variable* with the fewest remaining legal values; the goal is to detect failure early (fail-fast). **LCV** (Least Constraining Value) selects the *value* that eliminates the fewest choices from neighbouring domains; the goal is to keep options open (fail-slow). They serve complementary purposes.
5. The degree heuristic selects the variable that participates in the most constraints with *still-unassigned* variables. It is typically used as a tiebreaker when MRV produces a tie.
6. Yes, the CSP is arc-consistent: for every variable and every value there exists a supporting value in each neighbour's domain (e.g. for $X = R$, Y can

take G). However, no solution exists because a triangle is an odd cycle and cannot be properly 2-coloured. This shows that **arc consistency does not imply global consistency**; detecting this failure requires path consistency (3-consistency) or stronger.

7. **k-consistency**: any consistent assignment of $k-1$ variables can be extended to a consistent assignment of a k-th variable. **Strong k-consistency**: j-consistent for all $j \leq k$. Arc consistency equals 2-consistency; path consistency (together with node consistency) equals 3-consistency. Plain 3-consistency does not imply 2-consistency; strong 3-consistency does.
8. If the CSP is strong n-consistent, any consistent partial assignment of size k can always be extended to $k+1$. Therefore a greedy left-to-right assignment never backtracks. We do not always enforce this because the cost of computing k-consistency grows exponentially with k; enforcing arc consistency is cheap and already prunes much of the search space.
9. For CSPs, Local Search uses **complete** (possibly violating) assignments as states. The neighbourhood is defined by changing one variable's value. The objective function is the **number of satisfied constraints**, which we maximise; reaching the global optimum (all constraints satisfied) solves the CSP.
10. You cannot conclude anything definitive. The CSP may be genuinely inconsistent, *or* the search may simply be stuck in a local optimum. Local Search is incomplete and cannot prove inconsistency.
11. The eight RCC-8 relations are: **DC** (Disconnected), **EC** (Externally Connected), **PO** (Partial Overlap), **TPP** (Tangential Proper Part), **TPPi** (its inverse), **NTPP** (Non-Tangential Proper Part), **NTPPi** (its inverse), and **EQ** (Equal). All eight are defined solely in terms of the connection predicate $C(x, y)$ — no metric information is needed.
12. Node consistency (1-consistency) holds when every value in D_i satisfies all *unary* constraints on X_i. It is cheapest because it involves only one variable at a time; enforcement is a single pass through each domain with no propagation queue, costing $O(\sum_i |D_i|)$.
13. Forward Checking (FC) propagates the effect of an assignment only to the *direct unassigned neighbours* of the newly assigned variable, removing values with no support. It is weaker than full arc consistency (which also propagates indirect effects) but much cheaper per step. A **domain wipeout** occurs

when a neighbour's domain becomes empty; FC detects this immediately and triggers backtracking without assigning any further variables.

16.3 Planning — Classical and Probabilistic

1. **Frame**: how do we efficiently represent what does *not* change after an action? (Example: moving Block A leaves Blocks B and C unchanged, but this must be represented somehow.) **Qualification**: actions have potentially endless preconditions that can never be fully enumerated. (Example: "move robot" may fail if the battery is dead, a motor is broken, or the path is blocked.) **Ramification**: actions have unintended side effects. (Example: moving a storage unit also moves all items stored inside it.)
2. A STRIPS operator consists of **PRE** (atoms that must be true before the action), **ADD** (atoms that become true after the action), and **DEL** (atoms that become false after the action). The frame problem is solved implicitly by the convention that every atom *not* mentioned in ADD or DEL remains unchanged — no explicit frame axioms are needed.
3. In Situation Calculus every fact that could change after an action requires its own explicit *frame axiom*, giving $O(n \cdot m)$ axioms for n actions and m facts. STRIPS avoids this by the default-persistence convention, making it far more scalable but less expressive (it uses the Closed World Assumption, while Situation Calculus uses the Open World Assumption).
4. STRIPS uses the **Closed World Assumption (CWA)**: any atom not present in the state description is assumed false. Situation Calculus, Description Logics (OWL), and classical databases using null values do *not* make this assumption.
5. For `put(X,Y)` (place block X on top of block Y): **PRE** = {`ontable`(X), `clear`(X), `cl`
   ADD = {`on`(X, Y)}; **DEL** = {`ontable`(X), `clear`(Y)}.
6. The ignore-delete heuristic removes all DEL lists from every action, producing a relaxed problem in which actions never undo effects. The length of a solution to this relaxed problem lower-bounds the true plan length. However, computing the exact relaxed solution is still NP-hard; in practice a greedy approximation is used, which is **not admissible** (it can overestimate), so optimality of A* is not guaranteed.
7. PDDL uses a LISP-like syntax and adds typing, numeric fluents, condi-

tional effects, and action costs. Its key structural contribution is the **domain/problem separation**: the domain file specifies action schemas and types; the problem file specifies the initial state and goal. This decoupling lets the same domain be reused with different problem instances.

8. **State-space search**: nodes are world states and edges are actions; the algorithm searches from the initial state to the goal state. **Plan-space (partial-order) search**: nodes are partial plans and edges are refinement steps (adding an action, adding an ordering constraint, or adding a causal link). Plan-space planning supports non-linear plans and is the basis of partial-order planners.
9. Plan lengths can be exponential in the number of state atoms, and the state space itself is exponential in the number of propositional atoms. However, the algorithm only needs to store the *current state* at any point, requiring polynomial space rather than polynomial time, which places the problem in PSPACE (and it is PSPACE-complete).
10. An MDP is the 4-tuple (S, A, p, r): states, actions, transition probability $p(s' \mid s, a)$, and reward function $r(s, a)$. The discount factor γ is **not** part of the MDP tuple; it belongs to the objective function specified by the solver.
11. $P(S_{t+1} \mid S_t, A_t, S_{t-1}, A_{t-1}, \ldots) = P(S_{t+1} \mid S_t, A_t)$. The future depends only on the current state and action, not on the full history.
12. A **plan** is a sequence of actions computed for one specific initial state. A **policy** π is a mapping from *every* state to an action; it prescribes what to do regardless of how the current state was reached. A policy covers the entire state space; a plan covers only one trajectory.
13. $V^{\pi}(s) = r(s, \pi(s)) + \gamma \sum_{s'} p(s' \mid s, \pi(s))\, V^{\pi}(s')$. The value of state s under policy π equals the immediate reward plus the γ-discounted expected value of the successor state.
14. $V^{*}(s) = \max_a \big[r(s, a) + \gamma \sum_{s'} p(s' \mid s, a)\, V^{*}(s')\big]$. Compared to the policy-evaluation equation, the action is not fixed by a given policy but is chosen by *maximising* over all available actions, directly encoding optimality.
15. (1) **Policy Evaluation**: given the current policy π, compute V^{π} iteratively using the Bellman equation until convergence. (2) **Policy Improvement**: update the policy greedily: $\pi'(s) = \arg\max_a [r(s, a) + \gamma \sum_{s'} p\, V^{\pi}(s')]$. The algorithm terminates when $\pi' = \pi$ (no improvement), at which point the policy is optimal.

16. **Value Iteration** applies the Bellman optimality equation directly and iteratively without an explicit policy, converging slowly (many iterations, cheap per iteration). **Policy Iteration** alternates a full policy-evaluation pass with a policy-improvement step; it typically needs far fewer iterations (often fewer than 10) but each iteration is more expensive (a full evaluation to convergence).
17. With $\gamma = 1$ the return $G_t = \sum_{k=0}^{\infty} r_{t+k}$ is an infinite sum. If rewards are non-zero and there are no terminal states, this sum diverges, so the Bellman update never converges. Using $\gamma < 1$ turns the sum into a convergent geometric series.
18. A negative step reward (e.g. -0.04 per step) creates an implicit time pressure: the agent must reach the goal quickly because every extra step costs something. A positive step reward (e.g. $+0.04$) makes the optimal policy *never* reach the goal — wandering indefinitely is more rewarding than the finite goal bonus.

16.4 Knowledge Representation

1. The **TBox** (terminological box) contains general statements about concepts and their relationships, e.g. $\texttt{Student} \sqsubseteq \texttt{Person}$. It is analogous to a database schema. The **ABox** (assertion box) contains facts about named individuals, e.g. $\texttt{Mary} : \texttt{Teacher}$ or $(\texttt{Mary}, \texttt{CS600}) : \texttt{teaches}$. It is analogous to the database content.
2. $C \sqsubseteq D$ means "every C is also a D", i.e. $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ in every interpretation $\mathcal{I}$. The inclusion goes from the more specific concept on the left to the more general concept on the right: $\texttt{Flu} \sqsubseteq \texttt{Illness}$ means every Flu instance is an Illness instance, not the other way around.
3. ALC constructors: **atomic concept** A; $\top$ (everything); $\bot$ (nothing); $\neg C$ (complement); $C \sqcap D$ (intersection); $C \sqcup D$ (union); $\forall R.C$ (value restriction: all R-successors are in C); $\exists R.C$ (existential restriction: at least one R-successor is in C). "ALC" stands for Attributive Language with Complement.
4. $\forall R.C$ is *vacuously true* for any individual that has **no** R-successors at all. For example, "all children of x are students" ($\forall \texttt{hasChild}.\texttt{Student}$) is trivially satisfied by a childless individual — even though it feels semantically odd. By contrast, $\exists R.C$ always requires at least one filler.

5. To test $C \sqsubseteq D$ via tableaux, form the concept $C \sqcap \neg D$ and check whether it is **satisfiable**. The tableaux algorithm tries to construct a finite model for $C \sqcap \neg D$ by applying expansion rules. If every branch leads to a contradiction (clash) the concept is unsatisfiable, which proves $C \sqsubseteq D$. If a model is found, $C \sqsubseteq D$ does *not* hold.
6. **TBox tasks**: subsumption ($C \sqsubseteq D$?), concept satisfiability ($C \not\equiv \bot$?), concept equivalence ($C \equiv D$?). **ABox tasks**: instance checking (is individual a in concept C?), knowledge-base consistency (does the KB have any model?), retrieval (which individuals are instances of C?).
7. **OWA** (Open World Assumption): what cannot be derived from the KB is *unknown* — used in Description Logics and OWL. **CWA** (Closed World Assumption): what cannot be derived is assumed *false* — used in STRIPS, classical databases, and Prolog by default. OWA is more realistic for incomplete knowledge domains; CWA is more efficient when the knowledge is complete.
8. ALC does not allow role-inclusion axioms such as $R_1 \sqsubseteq R_2$. To express role hierarchies you need DL extensions such as **SH** (adds transitive roles and a role hierarchy), **SROIQ** (the basis of OWL 2), or similar expressive fragments.
9. The 13 Allen relations (7 basic + 6 inverses): **BEFORE/AFTER**, **MEETS/IS-MET-BY**, **OVERLAPS/IS-OVERLAPPED-BY**, **STARTS/IS-STARTED-BY**, **FINISHES/IS-FINISHED-BY**, **DURING/CONTAINS**, **EQUAL** (symmetric, counts once). The Allen composition table states: given $R(I_1, I_2)$ and $R(I_2, I_3)$, which relations are possible for $R(I_1, I_3)$? It is used for constraint propagation in temporal CSPs — analogous to AC-3.
10. More expressive languages (e.g. full FOL) can represent more complex knowledge, but reasoning becomes harder or undecidable. ALC is less expressive than FOL but subsumption is decidable and efficiently implementable via tableaux. Adding role composition to ALC pushes reasoning outside 2-variable logic, causing exponential or worse complexity. The design goal is to be expressive *enough* while remaining tractable.
11. **OWL Lite**: simple hierarchies and constraints; low DL expressivity; efficient reasoning. **OWL DL**: full Description Logic (SHOIN(D)); all standard reasoning tasks are decidable. **OWL Full**: fully RDF-compatible; no restrictions on usage; reasoning is **undecidable**.

12. RCC-8 is purely **topological**: it is based solely on the connection predicate $C(x, y)$ ("regions x and y share a point") and uses no distances, coordinates, or sizes. This makes it suitable for qualitative spatial reasoning without metric data.

16.5 Vagueness and Uncertainty

1. **Vagueness**: the predicate itself has gradual, imprecise boundaries ("tall", "heap", "reddish"); the property holds to a *degree*. Handled by fuzzy logic. **Uncertainty**: the fact is sharply true or false, but we do not know which; we have incomplete information. Handled by probabilistic logic.
2. Using Zadeh standard operators: (a) $\mu_{A\cap B}(x) = \min(0.7, 0.4) = 0.4$. (b) $\mu_{A\cup B}(x) = \max(0.7, 0.4) = 0.7$. (c) $\mu_{A^C}(x) = 1 - 0.7 = 0.3$.
3. Let $\mu_A(x) = 0.6$. Then $\mu_{A^C}(x) = 1 - 0.6 = 0.4$. The union gives $\max(0.6, 0.4) = 0.6 \neq 1$. So $A \vee \neg A$ is only 0.6-true — not a tautology. This is *intentional*: for a vague predicate the transition zone is genuine, and forcing a binary $\{0, 1\}$ answer would be the paradox fuzzy logic is designed to avoid.
4. A t-norm $t : [0, 1]^2 \to [0, 1]$ must satisfy: (1) **neutral element 1**: $t(x, 1) = x$; (2) **commutativity**: $t(x, y) = t(y, x)$; (3) **associativity**: $t(t(x, y), z) = t(x, t(y, z))$; (4) **monotone increasing** in both arguments. Examples: $t_{\min}(x, y) = \min(x, y)$ (Zadeh) and $t.(x, y) = x \cdot y$ (algebraic product).
5. Algebraic t-norm: $t(x, y) = x \cdot y$. At $x = y = 0.5$: $t(0.5, 0.5) = 0.25$ (vs. 0.5 for the Zadeh minimum). Algebraic s-norm: $s(x, y) = x + y - x \cdot y$. Both satisfy the t-/s-norm axioms and generalise AND/OR, but yield stricter (t) and more lenient (s) results than min/max for intermediate values.
6. In probabilistic logic, formulas are associated with events in a classical probability space. A tautology corresponds to Ω (probability 1); a contradiction corresponds to $\emptyset$ (probability 0). In fuzzy logic, truth values are computed directly: $I(F \vee \neg F) = \max(I(F), 1 - I(F))$, which equals $I(F)$ when $I(F) \geq 0.5$ and $1 - I(F)$ otherwise — neither is necessarily 1.
7. Kolmogorov's axioms: (1) $P(A) \geq 0$ for all events A. (2) $P(\Omega) = 1$. (3) $P(A \cup B) = P(A) + P(B)$ for disjoint A, B. **Derivation**: A and A^C are disjoint and $A \cup A^C = \Omega$, so $P(A) + P(A^C) = P(\Omega) = 1$, giving $P(A^C) = 1 - P(A)$.

8. $P(B \mid A) = P(B \cap A)/P(A)$, defined only when $P(A) > 0$. It expresses the probability of event B given that A is known to have occurred.
9. **Paradox**: "A heap of sand stays a heap if one grain is removed." By induction: even a single grain constitutes a heap — contradiction. Classical logic forces a binary threshold that does not exist in reality. Fuzzy logic resolves this by assigning a gradually decreasing membership value $\mu_{\text{Heap}}(n)$ as grains decrease; no sharp cut-off, no contradiction.
10. Dempster-Shafer theory assigns *mass* m to subsets of a frame of discernment. $\text{Bel}(A) = \sum_{B \subseteq A} m(B)$ is the total mass committed to A and its subsets — what we can believe at minimum. $\text{Pl}(A) = 1 - \text{Bel}(\neg A)$ is the mass that does not rule out A. The interval $[\text{Bel}(A), \text{Pl}(A)]$ quantifies *ignorance*: a wide interval means little evidence, a narrow interval means a clear picture.
11. By the law of total probability: $P(\text{Penguin}) = P(\text{Penguin} \mid \text{Bird}) \cdot P(\text{Bird}) + P(\text{Penguin} \mid \neg\text{Bird}) \cdot P(\neg\text{Bird}) \geq 0.2 \times 0.8 = 0.16$ (since the second term is non-negative).

16.6 Machine Learning Foundations

1. A program is said to *learn* from **experience** E with respect to **task** T and **performance measure** P if its performance on T, measured by P, improves with E (Mitchell, 1997).
2. Inductive bias is the set of prior assumptions a learner uses to generalise from training examples to unseen data (e.g. "the decision boundary is linear"). Without any bias there is no reason to prefer one hypothesis over another that fits the same training data, so no generalisation is possible. The No-Free-Lunch theorem formalises this: averaged over all possible tasks, no learner outperforms random guessing without some bias.
3. (1) **Supervised** (labelled data, classification or regression) — e.g. decision tree, SVM. (2) **Unsupervised** (unlabelled data, find structure) — e.g. k-means. (3) **Reinforcement** (reward signal, learn a policy) — e.g. Q-learning, value iteration.
4. $H(S) = -\sum_i p_i \log_2 p_i$. $H = 0$ when all examples have the same label (pure node). $H = \log_2 c$ (maximum) when c classes are uniformly distributed.
5. $\text{Gain}(S, A) = H(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} H(S_v)$. It measures the expected reduction in entropy (impurity) achieved by splitting on attribute A. ID3

greedily chooses the attribute with the highest gain at each node.

6. An attribute with many distinct values (e.g. a unique ID) can split every example into its own branch, achieving zero entropy in every child and maximum gain — but with zero generalisation power. The standard remedy is **Gain Ratio** (used in C4.5): $\text{GainRatio}(S, A) = \text{Gain}(S, A)/\text{SplitInfo}(S, A)$, which penalises high-cardinality attributes.
7. $\mathbb{E}[(y - \hat{y})^2] = \text{Bias}[\hat{y}]^2 + \text{Var}[\hat{y}] + \sigma^2$. **Bias**: systematic error from overly simple models (underfitting). **Variance**: sensitivity to fluctuations in the training data. σ^2: irreducible noise inherent in the data.
8. Bagging (Bootstrap Aggregation) trains many models, each on an independently drawn bootstrap sample (sampling with replacement), and averages their predictions. Averaging reduces *variance* because uncorrelated errors cancel. It does *not* reduce bias because all models share the same model class; the systematic error remains.
9. Random Forest adds a second mechanism on top of bagging: at each split only a random subset of features (typically $\sqrt{p}$ for classification) is considered. This *decorrelates* the trees — without it all trees would pick the same dominant feature at the root — amplifying the variance reduction from averaging.
10. In bootstrap sampling, approximately 37% of training examples are left out of any given tree (the out-of-bag samples). These can serve as a held-out test set for that specific tree, giving a generalisation estimate at no extra data cost — "free" because no separate validation set is needed.
11. Data are split into k equal folds. In iteration i, the model is trained on all folds except i and tested on fold i. The k test errors are averaged. Purpose: a low-variance estimate of generalisation performance, usable for model selection and hyperparameter tuning.
12. k-means iterates: (1) assign each point to its nearest centroid; (2) recompute each centroid as the mean of its assigned points. It minimises the within-cluster sum of squared distances. Because the algorithm is a greedy alternating optimisation, it is only guaranteed to converge to a *local* minimum; the result depends on the random initialisation.
13. Because the recursive greedy split uses the specific training data at each node; a small perturbation early in the tree can flip an early split decision and cascade into a structurally very different subtree. Remedies include **pruning**

(pre- or post-) and **ensemble methods** (Random Forest, boosting).

14. The phenomenon is **overfitting**, corresponding to **high variance** (low bias). Two remedies: (1) Prune the tree (pre- or post-pruning). (2) Use a Random Forest or another ensemble to average out variance.

16.7 Decision Trees and Ensembles

1. ID3 proceeds recursively: (1) If all examples have the same class, return a leaf. (2) Compute Gain(S, A) for every remaining attribute A. (3) Select the attribute A^* with the highest gain as the split attribute. (4) For each value v of A^*, recursively build a subtree on S_v. If no attributes remain, return the majority class as a leaf.
2. $H(S) = -1 \cdot \log_2(1) - 0 \cdot \log_2(0) = 0$ bits (using the convention $0 \log_2 0 = 0$). Zero entropy means the node is **pure**; ID3 stops splitting here and labels it `Yes`. No information would be gained by any further split.
3. Information Gain uses a *weighted* average of child entropies (weights $= |S_v|/|S|$) rather than the minimum child entropy. This ensures the measure captures how much total uncertainty is eliminated per split; without the weighting, a single tiny pure child alongside a large impure one would look deceptively good.
4. **What happened**: `UserID` has a unique value for every example, so splitting on it creates one pure leaf per example — zero entropy in every child, maximum Information Gain, but zero generalisation (pure memorisation). **Fix**: use **Gain Ratio** (C4.5), which normalises Gain by SplitInfo and heavily penalises high-cardinality attributes; alternatively, remove columns that have no predictive signal.
5. **Pre-pruning** (early stopping): impose a maximum depth, a minimum leaf size, or a minimum required Gain before allowing a split. **Post-pruning** (reduced-error pruning): grow the full tree, then collapse subtrees whose removal does not increase validation error. Post-pruning typically outperforms pre-pruning because it uses the complete tree as a starting point.
6. Single trees suffer from **high variance**: small data perturbations cause large structural changes. RF fixes this with two mechanisms: (1) **Bagging**: train each tree on a different bootstrap sample; averaging reduces variance. (2) **Random feature subsampling**: at each split, consider only $\sqrt{p}$ features;

this *decorrelates* trees so averaging is effective.

7. A strong defensible choice is a **single pruned decision tree**. Auditability is a hard regulatory requirement (GDPR Article 22): a doctor can trace any prediction through the if-then-else nodes. Random Forest and XGBoost would gain perhaps 2–5% accuracy but are essentially black boxes. An alternative is Random Forest with SHAP values for post-hoc explanation. XGBoost is the wrong choice when raw interpretability is mandatory.
8. For a binary classification problem with class proportions p and $1-p$: $\text{Gini}(S) = 1 - p^2 - (1-p)^2 = 2p(1-p)$. Like entropy, Gini is 0 when the node is pure and maximal when classes are balanced ($p = 0.5$, giving Gini $= 0.5$). Gini is used in CART; entropy is used in ID3/C4.5. Both are similar in practice, but Gini is slightly faster to compute (no logarithm).

16.8 Support Vector Machines

1. SVMs find the hyperplane that maximises the **margin** — the distance from the hyperplane to the nearest training examples of either class. **Support vectors** are the training examples that lie exactly on the margin boundary; only they determine the solution.
2. Margin $= 2/\|w\|$. SVM solves: $\min_{w,b} \ \frac{1}{2}\|w\|^2$ subject to $y_i(\langle x_i, w\rangle + b) \geq 1$ for all i. This is a convex quadratic program.
3. $L(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_i \alpha_i\big(y_i(\langle x_i, w\rangle + b) - 1\big)$, with $\alpha_i \geq 0$. The sign of the constraint term is **minus** (we subtract the constraint penalty). Optimising the Lagrangian yields $w = \sum_i \alpha_i y_i x_i$.
4. KKT complementary slackness: $\alpha_i \cdot (y_i(\langle x_i, w\rangle + b) - 1) = 0$. Therefore either $\alpha_i = 0$ (the point is not a support vector and contributes nothing to w) or $y_i(\langle x_i, w\rangle + b) = 1$ (the point lies exactly on the margin and is a support vector).
5. Replace the inner product $\langle x, x'\rangle$ with a kernel function $K(x, x') = \langle \phi(x), \phi(x')\rangle$ that computes the inner product in a higher-dimensional feature space without explicitly applying ϕ. This makes even infinite-dimensional feature spaces (e.g. RBF) computationally tractable.
6. γ-notation: $K(x, y) = e^{-\gamma\|x-y\|^2}$. σ-notation: $K(x, y) = e^{-\|x-y\|^2/(2\sigma^2)}$. They are equivalent with $\gamma = 1/(2\sigma^2)$.
7. Three kernels: (1) **Linear**: $K(x, y) = \langle x, y\rangle$. (2) **Polynomial**: $K(x, y) =$

$(\langle x, y\rangle + c)^d$. (3) **RBF/Gaussian**: $K(x, y) = e^{-\gamma\|x-y\|^2}$; maps to an infinite-dimensional space; often the default choice.

8. Slack variables $\xi_i \geq 0$ allow training points to violate the margin constraint: $y_i(\langle x_i, w\rangle + b) \geq 1 - \xi_i$. The objective becomes $\frac{1}{2}\|w\|^2 + C\sum_i \xi_i$. **Large** C: violations are costly, margin is small, risk of overfitting. **Small** C: violations are cheap, margin is large, more regularisation (may underfit). $C = \infty$ recovers the hard-margin SVM.
9. A function K is a valid kernel (Mercer's condition) if and only if the Gram matrix $K_{ij} = K(x_i, x_j)$ is **positive semi-definite** and symmetric for every finite set of points. This guarantees the existence of a feature mapping ϕ such that $K(x, y) = \langle\phi(x), \phi(y)\rangle$.
10. $\Delta w = \varepsilon\sum_n (t_n - y(x_n))\, x_n$ where ε is the learning rate, t_n is the target label for example n, and $y(x_n)$ is the perceptron's prediction. Applied per example (online) or summed over all examples (batch).
11. If the data are **linearly separable** and the learning rate ε is sufficiently small, the perceptron training rule converges in finitely many steps to a separating hyperplane. For non-linearly separable data the algorithm never converges.
12. XOR is not linearly separable: no single hyperplane separates $\{(0,0),(1,1)\}$ from $\{(0,1),(1,0)\}$. Minsky and Papert (1969) proved this, leading to the **first AI winter** for neural networks because the limitation was (incorrectly) assumed to apply to all neural networks. A multi-layer perceptron with one hidden layer solves XOR.
13. **Perceptron**: gradient descent on a (typically MSE) error function; iterative, may not converge for non-separable data. **SVM**: constrained quadratic optimisation solved globally via Lagrange multipliers and KKT conditions. They represent entirely different classes of optimisation: gradient-based vs. convex programming.

16.9 Neural Networks

1. **Storage**: weights are set such that target patterns are local energy minima (e.g. via Hebbian learning: $w_{ij} = \sum_\mu y_i^\mu y_j^\mu$). **Retrieval**: initialise with a (possibly corrupted) query pattern; update each neuron sequentially: $y_i^{(t+1)} = \text{sign}\big(\sum_{j\neq i} w_{ij} y_j + b_i\big)$. **Energy function**: $E(y) = -\sum_{ij} w_{ij} y_i y_j -$

$\sum_i b_i y_i$. Each sequential update can only decrease or maintain E, guaranteeing convergence to a local minimum.

2. Spurious memories are stable states of the Hopfield network that are *not* among the intended stored patterns. They arise as *superpositions* of stored patterns when the Hebbian weight matrix has additional local energy minima. They reduce effective capacity; the practical rule of thumb is that reliable retrieval holds for up to approximately $0.138N$ patterns for a network of N neurons.
3. $\mathrm{ReLU}(x) = \max(0, x)$. For $x > 0$ the derivative is exactly 1 (no saturation, no vanishing gradient), whereas sigmoid's derivative approaches 0 for large $|x|$. This allows gradients to flow through many layers without shrinking, enabling the training of deep networks.
4. A single hidden layer MLP with sufficiently many neurons and a non-linear activation can approximate any continuous function on a compact domain to arbitrary precision. Practical limitation: the required width can be *exponential* in the input dimension; deeper networks can achieve the same approximation with polynomially fewer parameters.
5. **Vanishing gradient**: in deep networks with sigmoid or tanh activations, the gradient shrinks exponentially as it is backpropagated through layers. **ReLU** eliminates saturation for positive activations (gradient = 1). **Residual connections** add an identity shortcut: $\partial L/\partial x = \partial L/\partial y \cdot (1 + \partial F/\partial x)$; the "1" guarantees a direct gradient path, enabling networks with 100+ layers.
6. $y = x + F(x)$. Gradient: $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \left(1 + \frac{\partial F}{\partial x}\right)$. The identity term "1" provides a direct shortcut so even if $\partial F/\partial x$ is very small, the gradient still flows back through the shortcut without vanishing.
7. (1) **Dropout**: randomly deactivate neurons during training (with probability p); effectively trains an ensemble of subnetworks. (2) **Weight decay (L2)**: add $\lambda\|w\|^2$ to the loss, penalising large weights. (3) **Early stopping**: halt training when validation loss starts increasing, even if training loss is still decreasing.
8. An **autoencoder** has an encoder that compresses input x to a low-dimensional latent code z, and a decoder that reconstructs x from z. The bottleneck forces a compact representation. Applications: (1) non-linear dimensionality reduction and (2) anomaly detection (high reconstruction error $\Rightarrow$ anomaly).
9. $\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\big(QK^\top/\sqrt{d_k}\big)\,V$. Q (query) = what we are

looking for; K (key) = what each position offers; V (value) = the actual content to aggregate.

10. For large d_k the dot products $q \cdot k$ grow in variance (proportionally to d_k), pushing the softmax into a very peaked distribution where almost all weight falls on one entry and most gradients vanish. Dividing by $\sqrt{d_k}$ keeps the variance stable and the softmax in a trainable regime.
11. Transformers process all tokens in parallel and have no recurrence; without additional information they are **permutation-invariant** and cannot distinguish word order. Positional encoding adds sequence position information. The sinusoidal scheme (Vaswani et al. 2017): $\text{PE}(\text{pos}, 2i) = \sin(\text{pos}/10000^{2i/d})$, $\text{PE}(\text{pos}, 2i{+}1) = \cos(\text{pos}/10000^{2i/d})$, added to the token embeddings before the attention layers.
12. **BERT**: trained via masked language modelling (predict masked tokens); **bidirectional** (sees the whole context); primarily used for classification and understanding tasks. **GPT**: trained autoregressively (predict the next token); **unidirectional** (left-to-right, causal mask); primarily used for text generation. Both are Transformer-based and use pre-training followed by fine-tuning.
13. Modern Hopfield networks (with continuous states and an exponential energy function) store patterns via a softmax-style update whose update rule is mathematically **equivalent to scaled dot-product attention**: the query "retrieves" the closest stored key pattern. Their capacity is exponential in d (vs. $0.138N$ for classical Hopfield nets), matching the empirical expressive power of Transformer attention.
14. The Lottery Ticket Hypothesis (Frankle & Carlin, 2019): large overparameterised networks contain small subnetworks ("winning tickets") that, trained with their *original* initialisation, reach comparable performance to the full network. Pruning *before* training hurts because we do not yet know which subnetwork will be the winning ticket; the winning ticket is identified only through training.

16.10 Deep Learning and Transformers

1. **Multi-head attention** runs h attention heads in parallel, each with its own learned projection matrices W_i^Q, W_i^K, W_i^V. The outputs are concatenated and projected back. Multiple heads let the model attend to information

from different representation subspaces simultaneously (e.g. one head for syntax, another for coreference).

2. A Transformer encoder block in order: (1) **Multi-head self-attention** — each token attends to all others. (2) **Residual add & Layer Norm** — stabilises training. (3) **Feed-forward sub-layer** (two linear layers with ReLU) — applies a per-token non-linear transformation. (4) **Residual add & Layer Norm** again.
3. Layer normalisation normalises the activations of each token across the feature dimension (zero mean, unit variance) and then applies learned scale and shift parameters. Applied after each sub-layer (or before, in pre-LN variants), it stabilises the distribution of activations and prevents exploding or vanishing activations in deep stacks.
4. Self-attention computes pairwise attention scores for all $n \times n$ token pairs, giving $O(n^2)$ time and memory complexity. For very long sequences (e.g. $n >$ 10,000) this becomes a bottleneck. Efficient attention variants (Longformer, FlashAttention) reduce this via sparse attention patterns or IO-aware tiling.
5. **Pre-training**: train a large model on massive unlabelled data with a self-supervised objective (masked LM or next-token prediction) to learn general language representations. **Fine-tuning**: continue training on a small labelled task-specific dataset, adapting the pre-trained weights to the target task. Fine-tuning is far cheaper than training from scratch.
6. Masked self-attention in the decoder allows each position to attend only to positions *earlier* in the sequence (and itself). The mask is an upper-triangular matrix of $-\infty$ values added to the attention logits before softmax, which zeroes out future positions. This enforces the autoregressive property: the model can only condition on past tokens when predicting the next one.
7. A **VAE** adds a probabilistic latent space: the encoder outputs parameters (μ, σ) of a Gaussian; the decoder receives a sample $z \sim \mathcal{N}(\mu, \sigma^2)$. Loss: reconstruction error + KL divergence $D_{\mathrm{KL}}(q(z|x)\|p(z))$ (regularises the latent space to be smooth and generative). A standard autoencoder has no probabilistic structure and cannot generate new samples by sampling z.
8. Batch normalisation normalises each feature across the mini-batch (zero mean, unit variance) and applies learned γ and β parameters. It stabilises training by reducing internal covariate shift, acts as a mild regulariser, and allows higher learning rates. It is typically applied *before* the activation

function (though the order is debated).

9. Gradient clipping addresses the **exploding gradient** problem: when the gradient norm exceeds a threshold τ, the gradient vector is rescaled so its norm equals τ: $g \leftarrow \tau \cdot g/\|g\|$. The direction of the gradient is unchanged; only its magnitude is clipped.
10. Weight decay adds $\frac{\lambda}{2}\|w\|^2$ to the loss, penalising large weights. From a Bayesian perspective this is equivalent to placing a zero-mean Gaussian prior on the weights: MAP estimation with that prior yields the same update as gradient descent with L2 regularisation. It encourages the model to prefer simpler hypotheses (smaller weights) when the data alone do not strongly dictate a value.
11. During **training**, each neuron is independently set to zero with probability p; the remaining activations are used as-is. During **inference**, dropout is *disabled* (all neurons are active) and activations are scaled by $(1-p)$ (or equivalently, during training activations are scaled by $1/(1-p)$, the *inverted dropout* convention) to keep the expected activation magnitude the same. Without this correction the network output would be systematically larger at inference than during training.