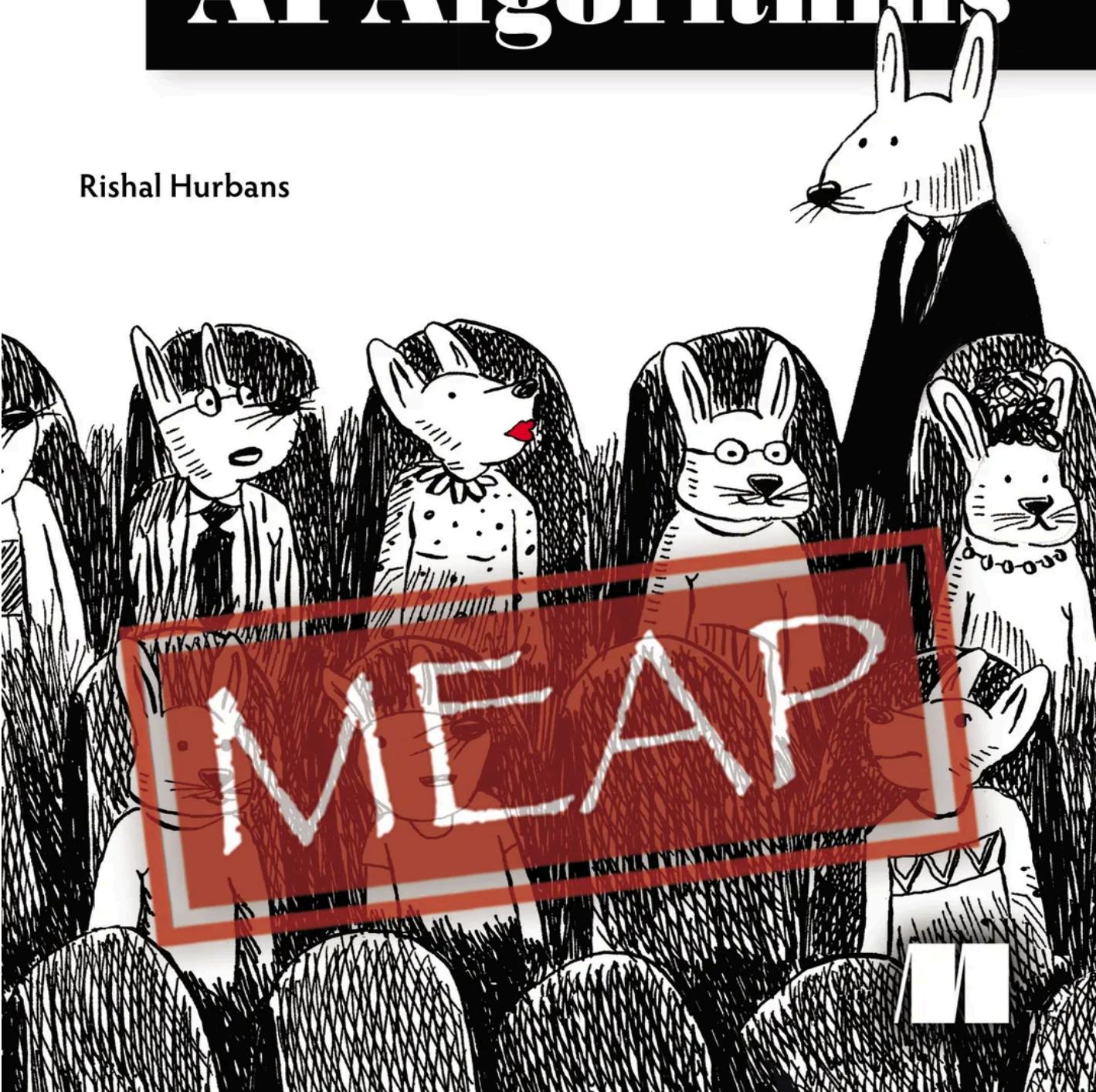


SECOND EDITION

**grokking**

# AI Algorithms

Rishal Hurbans





**MEAP Edition**  
**Manning Early Access Program**

**Grokking AI Algorithms, Second Edition**  
**Version 2**

**Copyright 2025 Manning Publications**

For more information on this and other Manning titles go to [manning.com](https://manning.com).

# welcome

---

Thank you for purchasing the MEAP for *Grokking AI Algorithms, Second Edition*! I'm excited to have you join me on this journey and appreciate your early support.

This book was written for software developers and anyone in the tech industry who wants to move beyond the buzzwords and truly grasp the intuition behind artificial intelligence. To get the most from this book, you should have a handle on basic programming concepts like variables, loops, and functions; experience in any language is perfectly fine. A basic understanding of math concepts, like how functions are represented on a graph, will also be helpful. My goal is to make these powerful algorithms accessible through visual explanations and practical examples, rather than deep dives into academic theory and mathematical proofs.

I've always been driven by a passion for making complex topics intuitive. I wanted to create the book I wished I had when I was starting out: one that builds a deep, practical understanding of how these algorithms work, why they were designed the way they are, and where they can be applied to solve real-world problems.

*Grokking AI Algorithms* will take you on a tour through the landscape of artificial intelligence. We'll start with the fundamentals of search and planning, which form the bedrock of many intelligent systems. From there, we'll explore fascinating nature-inspired approaches like evolutionary algorithms and swarm intelligence. Then, we'll dive into the data-driven world of machine learning, uncovering how models like neural networks learn from examples. Finally, we'll venture into the state-of-the-art with introductions to reinforcement learning, large language models, and generative image models.

The second edition builds upon the previous edition's deep learning concepts to include two new chapters aimed at exploring modern generative models, including the intuition and inner workings of LLMs with Transformers, and image generation with Diffusion.

Your feedback while the book is in early access is invaluable. It will play an important role in shaping the final book and ensuring the explanations are as clear and effective as possible. Please share your questions, comments, and suggestions in the [LiveBook discussion forum](#).

Thanks again for your interest!

—Rishal Hurbans

# *brief contents*

---

## **CHAPTERS**

- 1 Intuition of AI*
- 2 Search fundamentals*
- 3 Intelligent search*
- 4 Evolutionary algorithms*
- 5 Advanced evolutionary approaches*
- 6 Swarm intelligence: Ants*
- 7 Swarm intelligence: Particles*
- 8 Machine learning*
- 9 Artificial neural networks*
- 10 Reinforcement learning*
- 11 Large Language Models (LLMs)*
- 12 Generative Image Models*

# 1 *Intuition of AI*

## This chapter covers

- The definition of AI as we know it
- Intuition concepts underpinning AI and important terminology
- Defining problem types and approaches to solve them
- An overview of the AI algorithms discussed in this book
- Real-world use cases for AI algorithms

Artificial intelligence is no longer a niche area of research; it is a foundational tool in modern software engineering. For professionals in the technology industry, understanding the mechanics behind these systems—rather than just their APIs—is becoming a requisite skill. This book aims to bridge that gap. You will learn exactly how different classes of AI work conceptually, alongside the basic structures of their implementations. We will trace the evolution of algorithmic intelligence, moving from the explicit logic of Search and Evolutionary Algorithms through the statistical principles of Machine Learning, and finally into the complex architectures of Deep Learning and Generative AI. By understanding these foundational principles, you will be better equipped to reason about, implement, and innovate with the systems that are reshaping the world. You can expect to learn with relatable analogies, practical examples, and hand-drawn illustrations. Let's crack open the black box.

## 1.1 What is Artificial Intelligence (AI)?

Intelligence is a bit of a mystery. Philosophers, psychologists, and engineers all view intelligence differently. Yet, we recognize it everywhere: in the collective work of ants, the flocking of birds, and our own thinking and behavior. Great minds have long debated the nature of intelligence. Salvador Dalí saw it as *ambition*; Einstein tied it to *imagination*; Stephen Hawking defined it as the *ability to adapt*. While there is no single agreed upon definition, we generally use human behavior as the benchmark.

At a minimum, intelligence means being *autonomous* and *adaptive*. Autonomous entities act without constant instruction, while adaptive ones adjust to changing environments. Whether biological or mechanical, the fuel for intelligence is always data. The sights we see, the sounds we hear, and the measurements of our world are all inputs. We consume, process, and act on this data; therefore, understanding AI begins with understanding the data that powers it.

### 1.1.1 Defining AI

Generative AI, like natural language models and chatbots, has become the “public face” of AI. However, in research and practice, AI is far more diverse. The complex systems we use today are built upon many foundational algorithms geared to solve different problems—ranging from specific tasks, like effectively searching information, to general challenges, like understanding language. Throughout this book, we will be exploring these algorithms, their use cases, and developing the intuition leading to modern machine learning, artificial neural networks, and generative models.

For the sake of sanity, let’s define AI as systems that perform tasks typically requiring human intelligence. This includes simulating senses like vision and hearing, or mastering language to reason about complex problems.

- Playing and winning complex games
- Detecting cancer tumors from body image scans
- Generating artwork based on a natural language prompt
- Autonomous self-driving car
- Chatbots encoded with the history of information on the internet

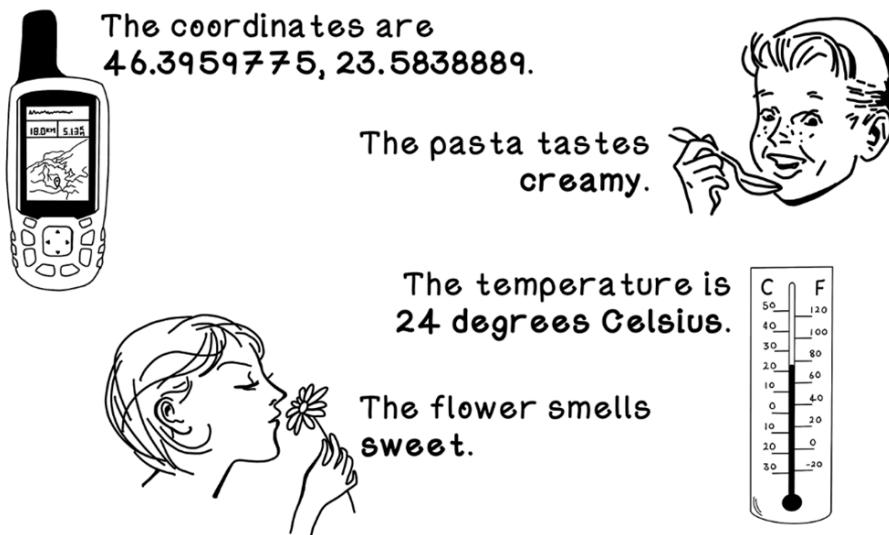
Douglas Hofstadter famously quipped, “AI is whatever hasn’t been done yet”. A calculator was once considered intelligent; now it is taken for granted. Whether an algorithm fits a strict academic definition matters less than its utility. If it solves a complex problem autonomously, it belongs in our toolkit.

The algorithms in this book have been classified as AI algorithms in the past or present; whether they fit a specific definition of AI or not doesn’t really matter. What matters is that they are useful and intuition about them forms a robust understanding from foundational to sophisticated applications.

### 1.1.2 Data is the fuel for AI algorithms

Data is the fuel that makes AI algorithms work. With the incorrect choice of data, badly represented data, or missing data, algorithms perform poorly, so the outcome is only as good as the data provided. The world is filled with data, and that data exists in forms that we can't even sense. Data can represent values that are measured numerically, such as the current temperature in the Arctic, the number of cats in a field, or your current age in days. All these examples involve capturing accurate numeric values based on facts. It's difficult to misinterpret this data. The temperature at a specific location at a specific point in time is absolutely true and is not subject to any bias. This type of data is known as *quantitative data*. However, cherry-picking (or sampling) specific data points intentionally or unintentionally can create bias.

Data can also represent values of observations, such as the smell of a flower or one's subjective review of a movie. This type of data is known as *qualitative data* and is sometimes difficult to interpret because it's not an absolute truth, but a perception of someone's truth. Figure 1.1 illustrates some examples of the quantitative and qualitative data around us.



**Figure 1.1 Examples of data around us**

Data is raw facts about things, so recordings of it should have no bias. In the real world, however, data is collected, recorded, and related by people based on a specific context with a specific understanding of how the data may be used. The act of constructing meaningful insights to answer questions based on data is creating *information*. Furthermore, the act of utilizing information with experiences and consciously applying it creates *knowledge*. This is partly what we try to simulate with AI algorithms.

For example, when dealing with a patient at a hospital:

- *Data*: 38°C.
- *Information*: The patient has a fever.
- *Knowledge*: We should administer medication to lower the fever.

Figure 1.2 shows how quantitative and qualitative data can be interpreted. Standardized instruments such as clocks, calculators, and scales are usually used to measure quantitative data, whereas our senses of smell, sound, taste, touch, and sight, as well as our opinionated thoughts, are usually used to create qualitative data.

	Quantitative	Qualitative
Instruments		
Cappuccino example	<ul style="list-style-type: none"> <li>- 350 ml volume cup</li> <li>- 91°C in temperature</li> <li>- 226 grams in weight</li> <li>- Porcelain cup</li> <li>- Beans from Africa</li> </ul>	<ul style="list-style-type: none"> <li>- Creamy texture</li> <li>- Strong taste with a hint of chocolate</li> <li>- Coffee is golden brown in color</li> <li>- Cup is white in color</li> <li>- Smells rich</li> </ul>

**Figure 1.2 Qualitative data versus quantitative data**

Data, information, and knowledge can be interpreted differently by different people, based on their level of understanding of that domain and their outlook on the world, and this fact has consequences for the quality of solutions we build—making the “scientific method” aspect of creating technology hugely important. By following repeatable scientific processes to capture data, conduct experiments, and accurately report findings, we can strive towards more accurate results and better solutions when processing data with algorithms.

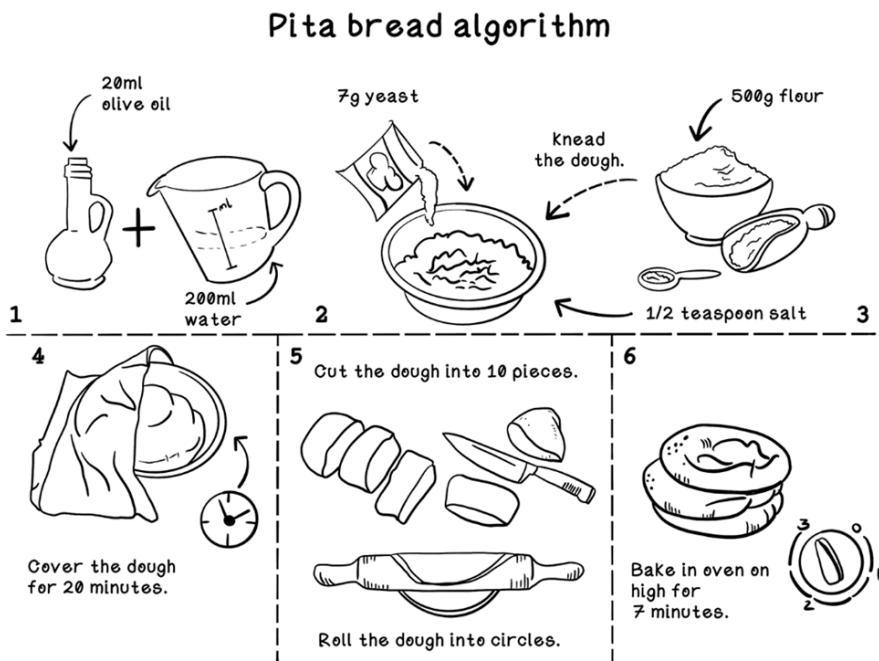
### 1.1.3 Algorithms are like recipes

We now have a loose definition of AI and an understanding of the importance of data. Because we will be exploring several AI algorithms throughout this book, it is useful to understand exactly what an algorithm is. An *algorithm* is a set of instructions and rules provided as a specification to accomplish a goal. Algorithms typically accept inputs, and after several finite steps where the algorithm progresses through varying states, an output is produced.

Even something as simple as reading a book can be represented as an algorithm. Here's an example of the steps involved in reading this book:

1. Find the book Grokking AI Algorithms.
2. Open the book.
3. While unread pages remain,
  - a. Read page.
  - b. Think about what you have learned.
  - c. Turn to the next page.
4. Think about how you can apply your learnings in the real world.

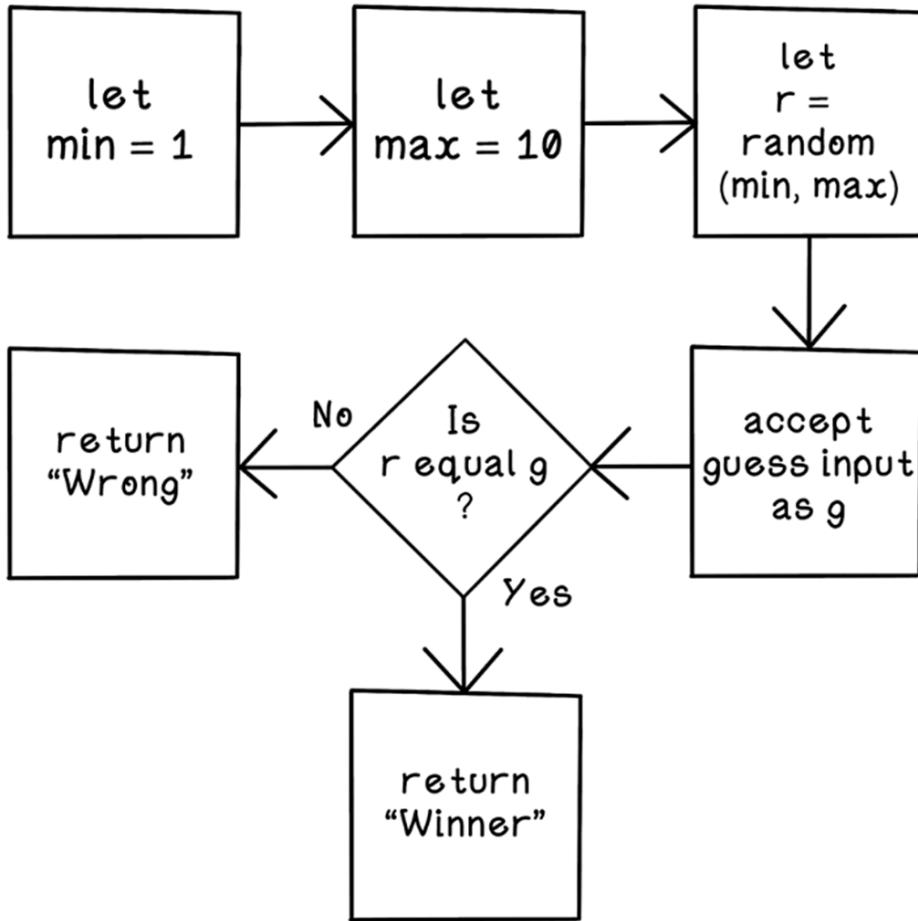
An algorithm can be seen as a recipe (in figure 1.3). Given some ingredients and tools as inputs, and instructions for creating a specific dish, a meal is produced as the output.



**Figure 1.3 An example showing that an algorithm is like a recipe**

Algorithms are used in many different solutions. For example, we can enable live video chat across the world through compression algorithms, and we can navigate cities through map applications that use real-time routing algorithms. Even a simple “Hello World” program has many algorithms at play to translate the human-readable programming language into machine code and execute the instructions on specific hardware. You can find algorithms everywhere if you look closely enough.

To illustrate something more closely related to the algorithms in this book, figure 1.4 shows a number-guessing-game algorithm represented as a flow chart. The computer generates a random number in a given range, and the player attempts to guess that number. Notice that the algorithm has discrete steps that perform an action or determine a decision before moving to the next operation. We will be seeing flow charts like this throughout the book to explain the lifecycles of algorithms.



**Figure 1.4** A number-guessing-game algorithm flow chart

#### 1.1.4 Algorithms vs. models

Algorithms are the logic; models are the result. To understand how AI systems are built, we must distinguish between the process (the recipe) and the representation (the meal). In this book, you will see two distinct patterns depending on the type of AI we are using.

#### THE ALGORITHM AS THE ACTIVE SOLVER

Some algorithms solve the problem actively in real-time. For example, a search algorithm is the star of the show when navigating a map or calculating the best move in a game. These algorithms are deployed to production and run fresh every time to solve problems based on the current situation.

## THE ALGORITHM AS THE BUILDER

In machine learning and deep learning, the relationship changes. Here, the algorithm is a “builder”. Its job is to look at data and construct a representation of the world. The algorithm is used to train a model, and the model is the artifact that is eventually deployed to production. This model contains the intelligence—pre-trained by the algorithm—that is used for problem-solving.

### 1.2 The evolution of AI

A look back at the strides made in AI is useful for understanding that old techniques and new ideas can be harnessed together to solve problems in innovative ways.

AI is not a new idea. History is filled with myths of mechanical men and autonomous “thinking” machines. Looking back, we find that we’re standing on the shoulders of giants. Perhaps we, ourselves, can contribute to the pool of knowledge in a small way.

Looking at past developments highlights the importance of understanding the fundamentals of AI; algorithms from decades ago are critical in many modern AI implementations. This book starts with fundamental algorithms that help build the intuition of problem-solving and gradually moves to more interesting and modern approaches.

Figure 1.5 isn’t an exhaustive list of achievements in AI—it is simply a small set of examples. History is filled with many more breakthroughs.

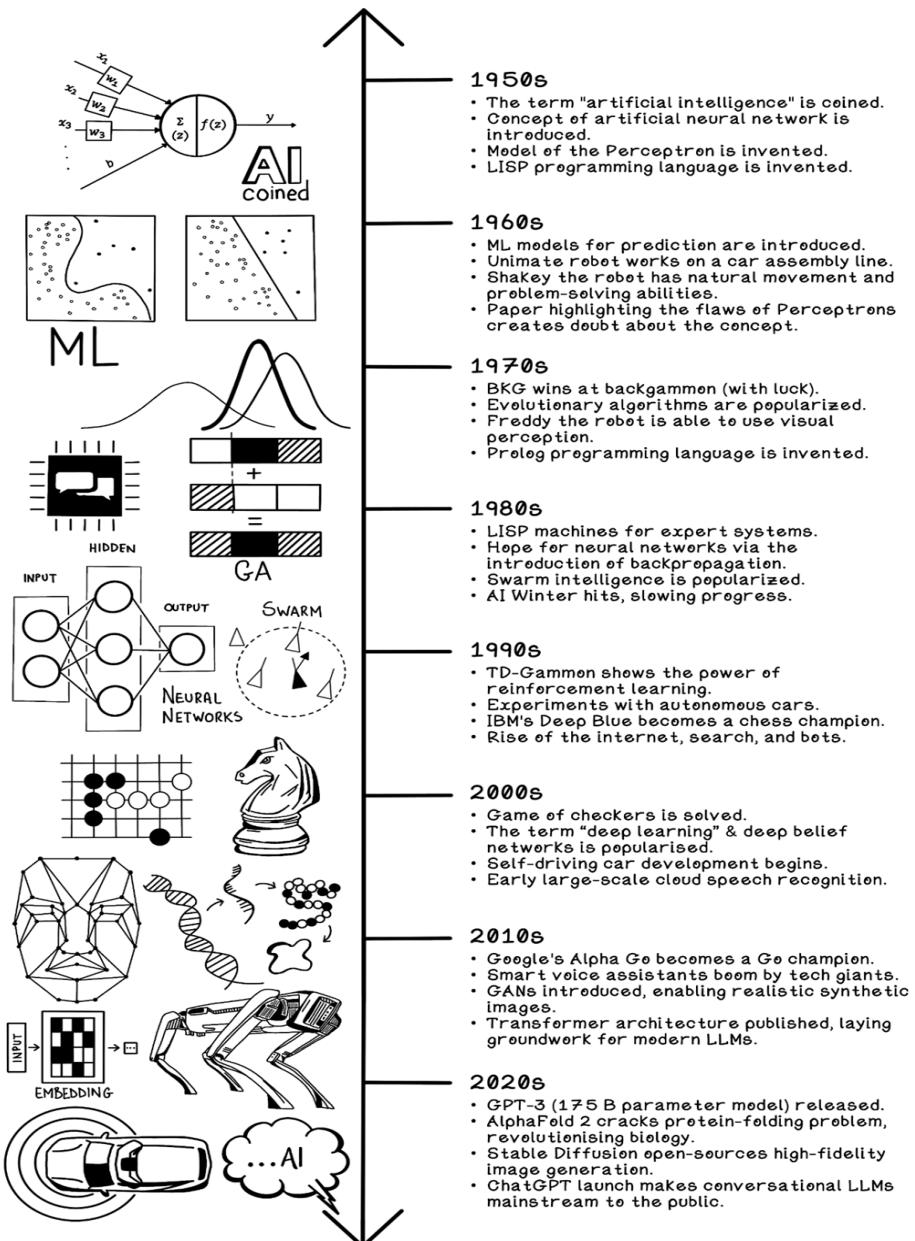


Figure 1.5 The evolution of AI

## 1.3 Different types of problems

AI algorithms are powerful, but they are not silver bullets that can solve any problem...yet. But what are *problems*? This section looks at different types of problems that we usually experience in computer science. This intuition can help us identify these problems in the real world and guide the choice of algorithms used. A grasp of this terminology will be useful as you progress through the book.

### 1.3.1 Search problems: Find a path to a solution

A *search problem* involves a situation that has multiple possible solutions, each of which represents a sequence of steps (path) towards a goal. Some solutions contain overlapping paths to success; some are better than others; and some are cheaper to achieve than others. A “better” solution is determined by the specific context; a “cheaper” solution means computationally cheaper to execute.

For example, determining the shortest path between cities on a map: many routes may be available, with different distances and traffic conditions, but some routes are better than others. Many AI algorithms are based on the concept of searching a space of possible solutions.

### 1.3.2 Optimization problems: Find a good solution

An *optimization problem* involves a situation in which there are a vast number of valid solutions and the absolute-best solution is difficult to find. Optimization problems usually have an enormous number of possibilities, each of which differs in how well it solves the problem.

An example is packing luggage in the trunk of a car. The goal is to maximize the use of space while adhering to strict constraints, such as the fixed dimensions of the trunk or a weight limit. Many combinations are available, but valid solutions must fit within these boundaries. If the trunk is packed effectively, more luggage can fit in it.

#### LOCAL BEST VERSUS GLOBAL BEST

Because optimization problems have many solutions, and because these solutions exist at different points in the search space, the concept of local bests and global bests comes into play. A *local best* solution is the best solution within a specific area in the search space, and a *global best* is the best solution in the entire search space. The challenge for AI is avoiding the trap of a local best (a good solution) while missing the global best (the perfect solution).

For example, imagine you’re searching for the best restaurant. You may find the best restaurant in your local area, but it may not necessarily be the best restaurant in the country or the best restaurant in the world. The best in the world is the global best.

### 1.3.3 Prediction and classification problems: Learn from patterns in data

*Prediction problems* are problems where we have data about something and want to try to find patterns and predict a numerical value. For example, we might have data about different vehicles and their engine sizes, as well as each vehicle's fuel consumption. Can we predict the fuel consumption of a new model of vehicle, given its engine size? If there's a correlation in the historic data between engine sizes and fuel consumption, this prediction is possible.

*Classification problems* are similar to prediction problems, but instead of trying to find an exact value such as fuel consumption, we try to find a category. If we have the physical dimensions of a vehicle, its engine size, and the number of seats, can we predict whether that vehicle is a motorcycle, sedan, or sport-utility vehicle? Classification problems require finding patterns in the data that group examples into categories.

A helpful rule of thumb to remember the difference is: Prediction outputs a quantity (a number), while Classification outputs a label (a category).

### 1.3.4 Clustering problems: Identify patterns in data

*Clustering problems* are scenarios where trends and relationships are uncovered from data. Different aspects of the data are used to group examples in different ways. For example, given cost and location data about restaurants, we may find that younger people tend to frequent locations where the food is cheaper. Clustering aims to find relationships in data even when a precise question is not being asked. This approach is also useful for gaining a better understanding of data to inform what you might be able to do with it.

### 1.3.5 Deterministic models: Same result each time it's calculated

*Deterministic models* are models where given a specific input, it returns a consistent output. If you input 100°C into a unit conversion model, the output will always be 212°F. It doesn't matter what time of day it is, where you are, or how many times you repeat the calculation; the relationship is fixed and the result never varies.

### 1.3.6 Probabilistic models: Potentially different result each time it's calculated

*Probabilistic models* are models that, given a specific input, return an outcome from a set of possible outcomes. Probabilistic models usually have an element of controlled randomness that contributes to the possible set of outcomes. If you type the phrase "The best pet is a..." on your phone, it might autocomplete various words based on words you've used in the past. It might assign "cat" 40%, "dog" 35%, and "goldfish" 25%. If you run this model once, it might pick "cat". If you run it again, the element of randomness might cause it to pick "dog". The input is the same, but the output varies based on the probability distribution.

## 1.4 Intuition of AI concepts

Trying to make sense of different but similar words and concepts in AI can be daunting. In this section, we demystify them and form a road map of the topics covered throughout this book.

Let's dive into the different levels of AI, introduced with figure 1.6.

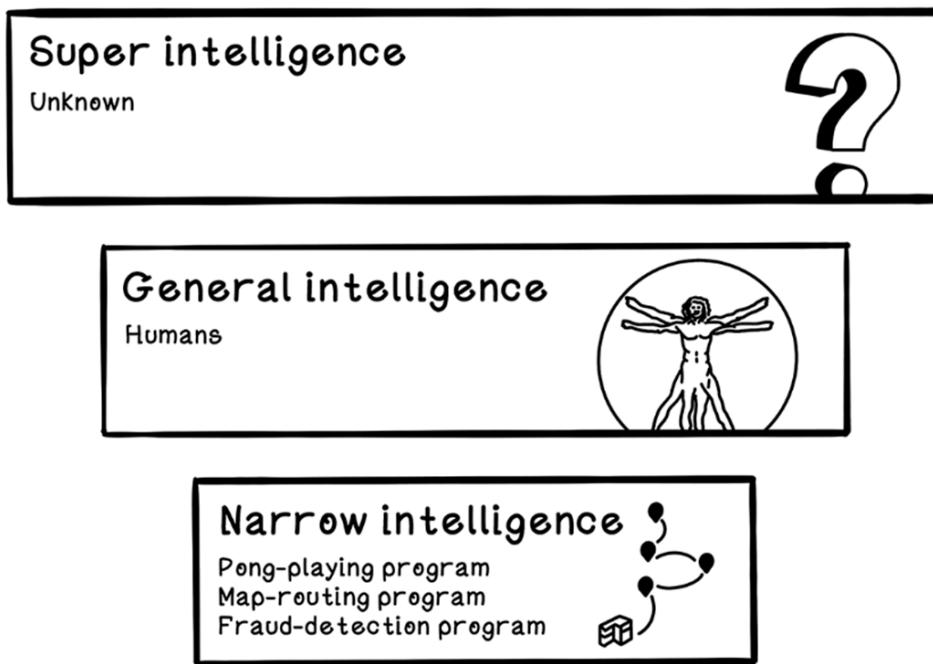


Figure 1.6 Levels of AI

### 1.4.1 Narrow intelligence: Specific-purpose solutions

*Artificial Narrow Intelligence (ANI)* systems specialize in a single domain. They cannot transfer knowledge; a model trained to analyze spending behavior, for example, cannot identify cats in an image. However, distinct narrow systems can be combined to simulate broader intelligence. A voice assistant, for instance, stacks speech recognition, web search, and recommendation algorithms to interact naturally with users. Different narrow intelligence systems can be combined in sensible ways to create something greater. A modern voice assistant isn't one smart brain; it is a combination of distinct narrow models (speech-to-text + web search + audio generation) working together to simulate a conversation.

### 1.4.2 General intelligence: Humanlike solutions

*Artificial General intelligence (AGI)* mirrors human adaptability: the ability to apply knowledge from one problem to another. For example, if you felt pain when touching something extremely hot as a child, you can extrapolate and know that other things that are hot may have a chance of hurting you. Just as a child generalises that “extremely hot” implies “danger” across different objects, AGI would integrate memory, reasoning, and sensory input to solve novel problems. Unlike the voice assistant, which would require a code update to learn a new skill, an AGI could simply figure it out. If you asked a narrow assistant to perform a task it wasn't programmed for (like negotiating a better price for your flight), it would fail. An AGI, however, would understand the intent of “saving money,” apply reasoning from other contexts, and attempt the negotiation. While true AGI remains elusive, modern large language models represent a significant step toward this flexible reasoning.

### 1.4.3 Super intelligence: The great unknown

*Super Intelligence (ASI)* refers to systems that surpass the brightest human minds in every field. While often depicted in sci-fi as apocalyptic, the core definition is simply intelligence beyond our comprehension. Whether humans can create something smarter than ourselves remains a philosophical debate, making ASI a realm of pure speculation for now.

### 1.4.4 Old AI and new AI

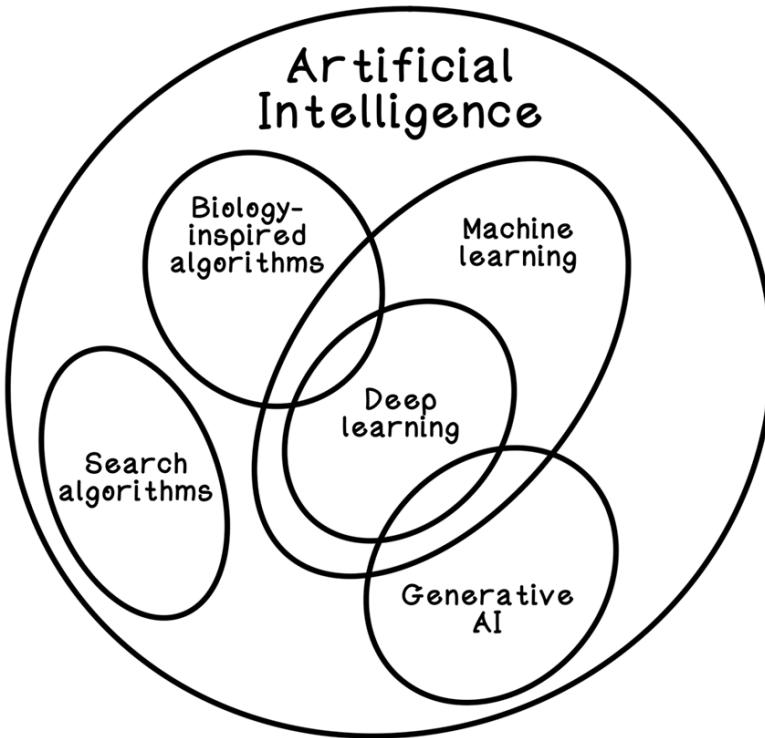
Sometimes, the notions of old AI and new AI are used.

*Old AI* is explicit logic and search that relies on humans to encode the rules of the world. The machine doesn't “learn”; it calculates solutions based on logic and rules provided by programmers. A classic example is the Minimax algorithm in chess: the human defines how pieces move and how to score a board position, and the AI uses computational power and smart branching to search future moves and find the best one.

*New AI* is learning from data and flips the Old AI approach. Instead of being told the rules or how to score a situation, these models analyze vast datasets to figure it out themselves. A modern neural network playing chess, for example, isn't following hard-coded heuristics; it has played millions of games against itself to learn patterns of victory that human programmers might not even understand.

We learn about both because, while search algorithms are often categorized as “Old AI” they are not obsolete. In fact, they are often paired with modern techniques to solve difficult problems. For example, Large Language Models utilize search strategies to determine the best sequence of words to generate. You need to understand the logic of search before you can understand how a neural network “searches” for an optimal solution.

Figure 1.7 illustrates the relationship between some of the different concepts within artificial intelligence.



**Figure 1.7 Categorization of concepts within AI**

### 1.4.5 Search algorithms

Search algorithms are the bedrock of problem-solving. They are essential when a goal requires a sequence of actions, like navigating a maze or calculating the winning move in chess. Instead of brute-forcing every possibility—which could take thousands of hours—smart search algorithms evaluate future states to find the optimal path efficiently. We start our journey here: Chapter 2 covers the fundamental search algorithms, and Chapter 3 dives into intelligent search methods that strategize rather than guess.

### 1.4.6 Biology-inspired algorithms

When we look at the world around us, we notice incredible things happening in different creatures, plants, and other living organisms—like the cooperation of ants in gathering food, the flocking of birds when migrating, estimating how organic brains work, and the evolution of different organisms to produce stronger offspring. By observing and learning from these phenomena, we've gained knowledge of how these organic systems operate and of how simple rules can result in incredible emergent intelligent behavior.

*Evolutionary Algorithms:* Inspired by Darwinian theory, these algorithms use reproduction and mutation to “evolve” code that improves over generations. We explore this survival of the fittest in Chapters 4 (Evolutionary algorithms) and Chapter 5 (Advanced evolutionary approaches).

*Swarm Intelligence:* This mimics the collective power of “dumb” individuals acting smart as a group. Chapter 6 (Ant colony optimization) explores intelligent path finding inspired by ants, while Chapter 7 (Particle Swarm Optimization) covers solving optimization problems inspired by how animals flock.

### 1.4.7 Machine learning algorithms

Chapter 8 (Machine learning) takes a statistical approach to training models to learn from data. The umbrella of machine learning has a variety of algorithms that can be harnessed to improve understanding of relationships in data, to make decisions, and to make predictions based on that data.

There are three main approaches in machine learning:

- *Supervised learning* means training models with algorithms when the training data has known outcomes for a question being asked, such as determining the type of fruit if we have a set of data that includes the weight, color, texture, and fruit label for each example.
- *Unsupervised learning* uncovers hidden relationships and structures within the data that guide us in asking the dataset relevant questions. It may find patterns in properties of similar fruits and group them accordingly, which can inform the exact questions we want to ask the data. These core concepts and algorithms help us create a foundation for exploring advanced algorithms in the future.
- *Reinforcement learning* is inspired by behavioral psychology. In short, it describes rewarding an individual if a useful action was performed and penalizing that individual if an unfavorable action was performed. When a child achieves good results on their report card, they are usually rewarded, but poor performance sometimes results in punishment, reinforcing the behavior of achieving good results. Chapter 10 (Reinforcement learning) will explore how reinforcement learning can be used to train intelligent models.

### 1.4.8 Deep learning algorithms

*Deep learning* is a broader family of approaches and algorithms that are used to achieve narrow intelligence and strive toward general intelligence. Deep learning usually implies that the approach is attempting to solve a problem in a more general way like linguistic intelligence or spatial reasoning, or it is being applied to problems that require more generalization such as speech recognition and computer vision. Deep learning approaches usually employ many layers of artificial neural networks (ANNs). By leveraging different layers of intelligent components, each layer solves specialized problems; together, the layers solve complex problems toward a greater goal. Identifying any object in an image, for example, is a general problem, but it can be broken into understanding color, recognizing shapes of objects, and identifying relationships among objects to achieve a goal. We will dive into how artificial neural networks operate in Chapter 9 (ANNs).

### 1.4.9 Generative models

Generative AI marks the shift from analyzing existing data to creating new content. Instead of just classifying an image or predicting a number, these models generate entirely new text, code, and visuals that have never existed before.

*Large Language Models (LLMs)*: We explore the architecture that revolutionized AI: the Transformer. Chapter 11 (Large Language Models) explains how these models master context and attention to converse, write, and reason with human-level fluency.

*Generative Image Models*: How does a computer dream? Chapter 12 (Generative Image Models) dives into the mechanics of creativity. We will look at how Diffusion models and U-Nets learn to sculpt pure random noise into structured, high-fidelity artwork.

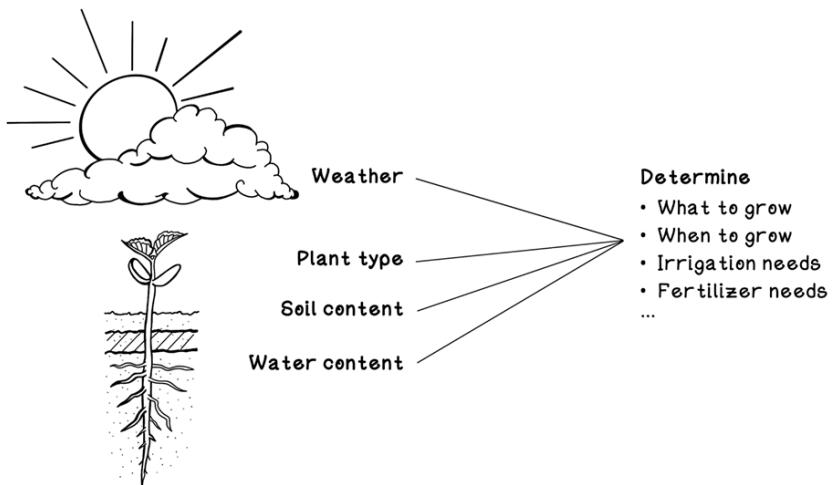
## 1.5 Some uses for AI algorithms

The uses for AI techniques are potentially endless. The applications of AI are limited only by the availability of data. Wherever there is a complex problem and historical data, there is potential for optimization. This section explores how AI transforms specific industries.

### 1.5.1 Agriculture: Optimizing plant growth

*The problem*: One of the most important industries that sustain human life is farming. Farming is a high-stakes balancing act. A single crop's success depends on hundreds of interacting variables—soil pH, moisture levels, microbial health, and unpredictable weather patterns. It is impossible for a human to perfectly calculate how these factors interact in real-time.

*The solution*: Modern farms utilize sensors to capture this environment as data. AI algorithms analyze these vast datasets to find hidden patterns—identifying exactly which combination of water, fertilizer, and timing results in the highest yield. Instead of guessing, farmers get real-time, data-driven recommendations to maximize growth and minimize waste (Figure 1.8).



**Figure 1.8 Using data to optimize crop farming**

### 1.5.2 Banking: Preventing fraudulent transactions

*The problem:* As banking moved from physical branches to digital networks, the volume of transactions exploded. With millions of payments happening every second, it is impossible for humans to manually review them for suspicious activity. Furthermore, static security rules (like “flag all transactions over \$10,000”) are rigid and easily bypassed by clever criminals: just move \$9,999 instead.

*The solution:* This is a classic case of anomaly detection. AI algorithms analyze a user’s historical data to learn their specific “pattern of life”—where they shop, how much they spend, and at what times. When a transaction deviates from this learned baseline (like a sudden high-value purchase in a foreign country), the model flags it in milliseconds, stopping the theft before the money leaves the account.

### 1.5.3 Cybersecurity: Safeguarding email inboxes

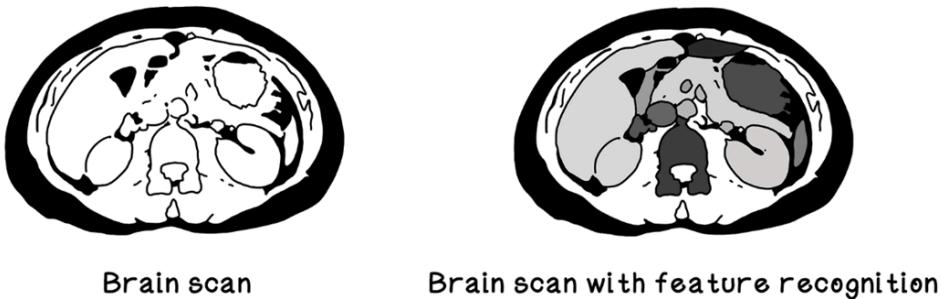
*The problem:* Email scams are a primary entry point for cyberattacks. In the past, spam filters relied on “blacklists” of bad words (like “lottery” or “winner”). However, scammers easily bypassed these by changing spellings or using social engineering tactics that looked like legitimate business requests.

*The solution:* Modern AI uses Natural Language Processing (NLP) to “read” emails. Instead of just looking for keywords like “lottery”, the AI analyzes the context and intent of the message. It can determine that an email claiming to be from your CEO asking for gift cards—even if it has no typos—is suspicious based on the sender’s writing style and the unusual request. This keeps inboxes clean and prevents users from clicking dangerous links.

#### 1.5.4 Health care: Diagnosing patients

*The problem:* Radiologists and doctors must review thousands of X-rays, MRIs, and CT scans to detect diseases. Fatigue can lead to human error, and subtle early signs of conditions like cancer or pneumonia can be imperceptible to the naked human eye.

*The solution:* This is a prime application for *Computer Vision*. AI models are trained on millions of medical images to recognize the visual textures of disease. These systems can scan images in seconds, highlighting potential tumors or fractures with high accuracy (figure 1.9). They act as a “second set of eyes”, ensuring that doctors don’t miss critical diagnoses and allowing treatment to begin earlier.



**Figure 1.9 Using machine learning for feature recognition in brain scans**

#### 1.5.5 Logistics: Finding the best delivery route

*The problem:* Logistics is essentially the Traveling Salesperson Problem on a massive scale. A delivery driver needs to visit dozens of locations in the shortest time possible. While this is mathematically difficult on its own, the real world adds chaos like traffic jams, road closures, fuel costs, and variable vehicle sizes.

*The solution:* AI algorithms transform this from a static math problem into a dynamic one. Algorithms like Ant Colony Optimization or Genetic Algorithms can simulate millions of potential routes in seconds to find the optimal path. Beyond just driving, AI can solve the 3D Bin Packing problem—computing exactly how to stack boxes inside the truck to maximize space and ensure the right packages are accessible at the right stops.

### 1.5.6 Fitness and Health: Optimizing your body

*The problem:* Generic health advice (like “sleep 8 hours”, “walk 10,000 steps”) is a statistical average that fails at the individual level. The human body is a complex biological machine where the relationship between stress, nutrition, and recovery varies wildly from person to person.

*The solution:* Wearable technology now provides the raw data—heart rate variability, blood oxygen, and sleep cycles. AI algorithms can process this constant stream of time-series data to build a personalized model of your physiology. Instead of a static training plan, the AI acts as a dynamic feedback loop. It detects subtle signals of fatigue that a human might miss, automatically adjusting today’s workout intensity to prevent injury or prescribing specific recovery protocols to peak in an athletic event.

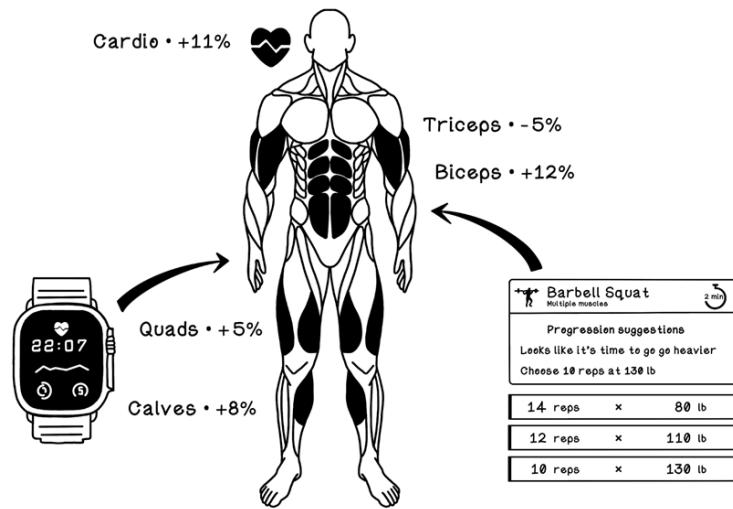


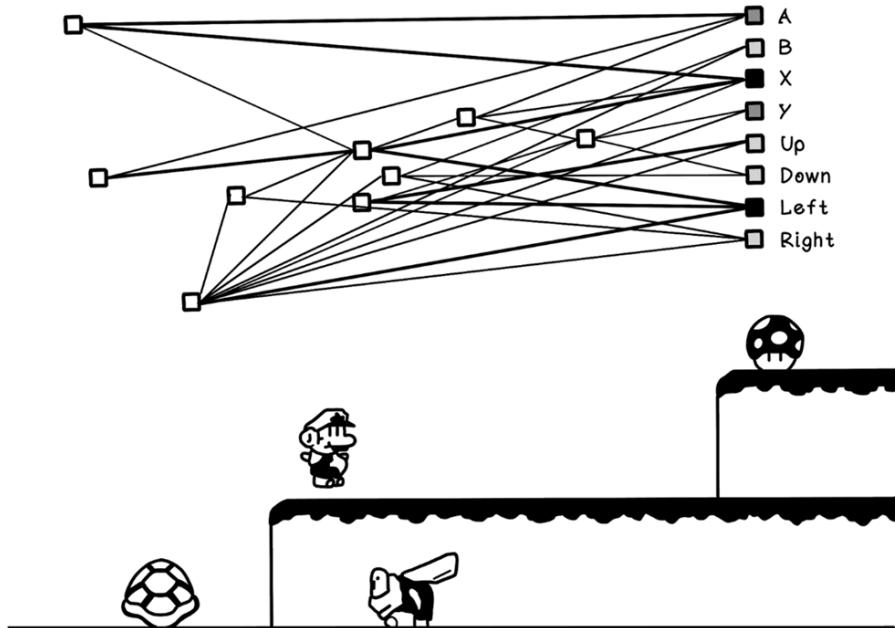
Figure 1.10 Using sensors and AI to guide fitness & health

### 1.5.7 Games: Adapting in complexity

Games are the ultimate benchmark for intelligence because they require strategy, planning, and adaptation.

In simple games like Tic-Tac-Toe or Chess, an AI can use search algorithms to calculate future moves and pick the statistically best option. However, games like Go or modern real-time strategy video games have search spaces larger than the number of atoms in the universe; a computer cannot simply “calculate” a guaranteed win.

To solve this, researchers use Reinforcement Learning. Instead of following hard-coded rules, the AI plays against itself millions of times. It is rewarded for good moves and penalized for bad ones, eventually developing a form of digital “intuition”. This approach allowed AlphaGo to defeat human world champions in 2016. The goal isn’t just to win games, but to develop general-purpose agents that can apply this strategic planning to real-world chaos (Figure 1.11).

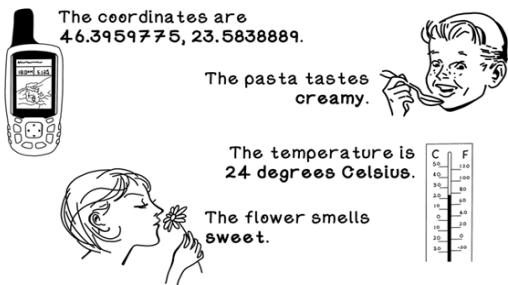


**Figure 1.11 Using neural networks to learn how to play games**

We have now defined the fuel (data) that powers our systems and outlined the menu of problems—from prediction to optimization—that we aim to solve. But knowing the ingredients isn’t enough; we need to learn how to cook. It is time to move from definitions to implementation. In the next chapter, we will dive into Search algorithms—the fundamental logic that allows machines to navigate complex choices and plan their path to victory.

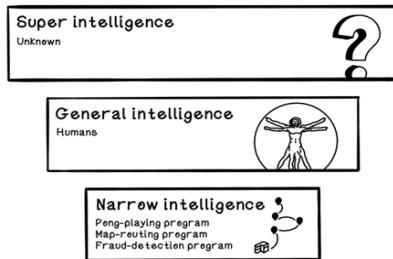
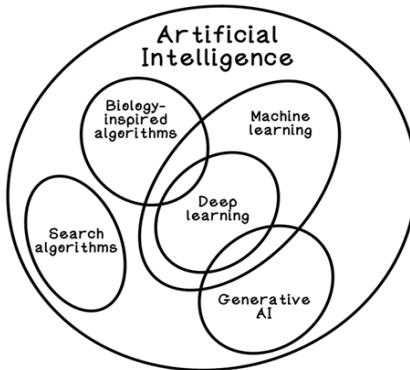
## 1.6 Summary of Intuition of AI

AI: Autonomously exhibits intelligent behavior & adapts



It's all about the data: Garbage in, garbage out

AI algorithms solve general problems with similar repeatable recipes



AI algorithms may lean toward narrow intelligence or general intelligence

Applications of AI algorithms are potentially endless



Brain scan



Brain scan with feature recognition

Build responsibly with ethics and humanity in mind

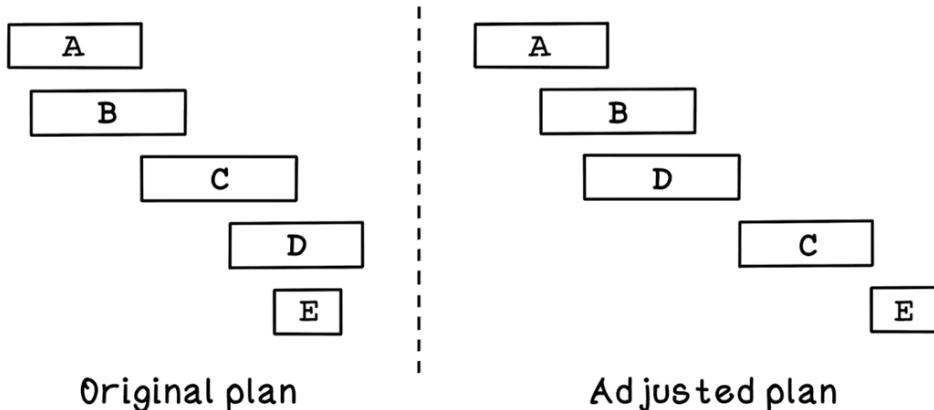
# 2 *Search fundamentals*

## This chapter covers

- The intuition of planning and searching
- Identifying problems suited to be solved using search algorithms
- Representing problem spaces in a way suitable to be processed by search algorithms
- Understanding and designing fundamental search algorithms to solve problems

## 2.1 What are planning and searching?

The ability to plan before acting is a hallmark of intelligence. Before going on a trip to a different country, before starting a new project, before writing functions in code, we plan. *Planning* happens at different levels of detail to strive for the best possible outcome when carrying out the tasks involved to accomplish goals (figure 2.1).



**Figure 2.1 Example of how plans change in projects**

Plans rarely work out perfectly in the way that we envision at the start. We live in a world where things are constantly changing, so it is impossible to account for all the variables and unknowns along the way. Regardless of the plan we started with, we almost always deviate. We need to (again) make a new plan from our current point going forward as unexpected events occur. As a result, the final plan that is carried out is usually quite different from the original one.

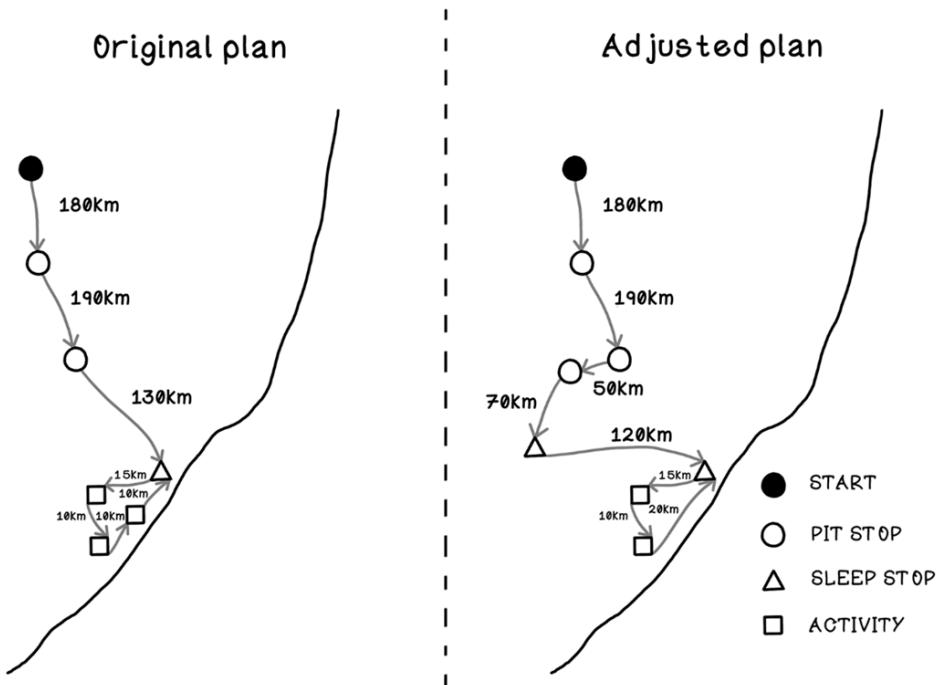
*Searching* is a way to guide planning by creating steps in a plan. When we plan a trip, for example, we search for routes to take, evaluate the stops along the way and what they offer, and search for accommodations and activities that we like and can afford. Depending on the results of these searches, the plan changes.

Suppose that we have settled on a trip to the beach, which is 500 kilometers (~310 miles) away, with two stops: one at a petting zoo and one at a pizza restaurant. We will sleep at a lodge close to the beach on arrival and do three activities. The trip to the destination will take approximately 8 hours. We're also taking a shortcut private road after the restaurant, but it's only open until 2:00.

We start the trip, and everything is going according to plan. We stop at the petting zoo and see some cute animals. We drive on and start getting hungry; it's time for the stop at the restaurant. But to our surprise, the restaurant recently went out of business. We need to adjust our plan and find another place to eat. This involves searching for a close-by restaurant of our liking, and adjusting our plan.

After driving around for a while, we find a restaurant, enjoy a pizza, and get back on the road. Upon approaching the shortcut private road, we realize that it's 2:20. The road is closed; yet again, we need to adjust our plan. We search for a detour and find that it will add 120 kilometers (~75 miles) to our drive, and we will need to find accommodations for the night at a different lodge before we even get to the beach. We search for a place to sleep and plot out our new route. Because we lost some time, we can do only two activities at the destination. The plan has been adjusted heavily through searching for different options that satisfy each new situation, but we end up having a great adventure en route to the beach.

This example shows how search is used for planning and influences planning toward desirable outcomes. As the environment changes, our goals may change slightly, and our path to them inevitably needs to be adjusted (figure 2.2). Adjustments in plans can almost never be anticipated but need to be made.



**Figure 2.2 Original plan versus adjusted plan for a road trip**

Searching involves evaluating future states toward a goal with the aim of finding an optimal path of states until the goal is reached. This chapter centers on different approaches to searching depending on different types of problems. Searching is an older but powerful tool for developing intelligent algorithms.

## 2.2 Cost of computation: The reason for smart algorithms

In programming, functions consist of operations, and because of the way that traditional computers work, different functions use different amounts of processing time. The more computation required, the more expensive the function is. *Big O notation* is used to describe the complexity of a function or algorithm. Big O notation models the number of operations required as the input size increases. Here are some examples and associated complexities:

- $O(1)$  is like shaking hands with one person. It takes the same time regardless of how many people are in the room. In code, this might be a single operation that prints "Hello World".
- $O(n)$  is like shaking hands with everyone in the room. If the room has 100 people, it takes 100 handshakes. In code, this could be a function that iterates over a list and prints each item.
- $O(n^2)$  is like everyone in the room shaking hands with everyone else. It becomes chaotic very quickly. In code, this might be a function that compares every item in a list with every other item in the list.

Figure 2.3 depicts different costs of algorithms. Algorithms that require more operations to explore as the size of the input increases are the worst-performing; algorithms that require a more constant number of operations as the number of inputs increases scale better to large datasets. Note: A "slower" algorithm might actually run faster if it can be parallelized across many processors (like on a GPU). We aim for low Big O complexity, but we also care about memory usage and hardware efficiency.

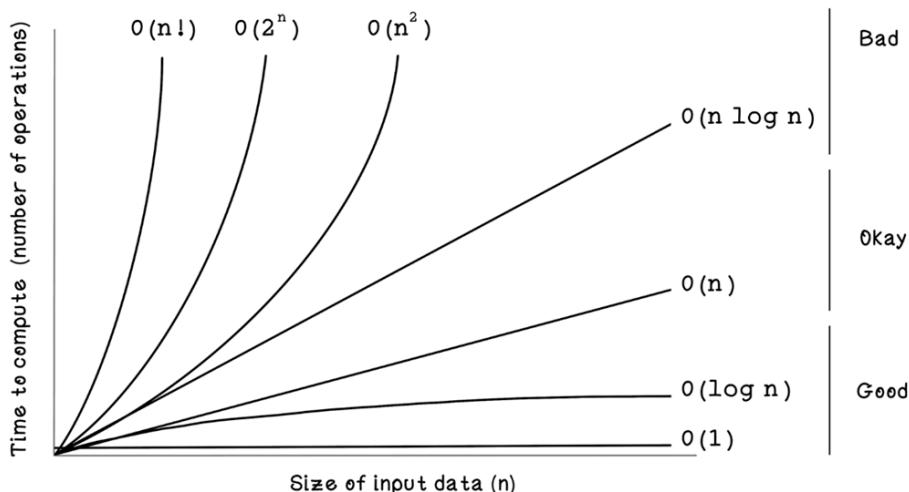


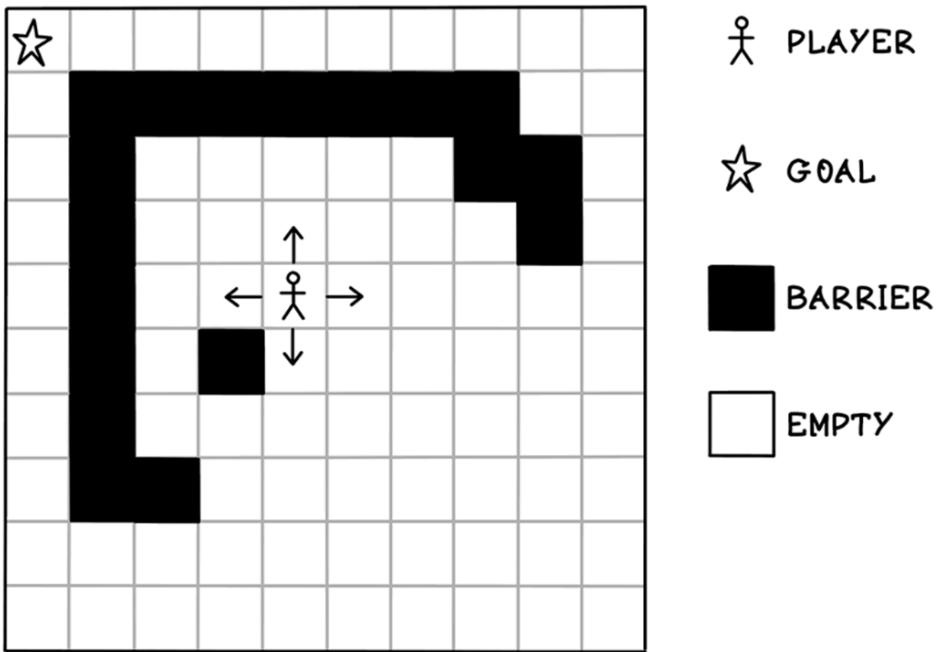
Figure 2.3 Big O complexity

Understanding that different algorithms have different computation costs is important because addressing this is the entire purpose of intelligent algorithms that solve problems well and efficiently. Theoretically, we can solve almost any problem by brute-forcing every possible option until we find the best one, but in reality, the computation could take hours, years, or even millenia, which makes it infeasible for real-world scenarios.

## 2.3 Problems applicable to searching algorithms

Almost any problem that requires a series of decisions to be made can be solved with search algorithms. Depending on the problem and the size of the search space, different algorithms may be employed to help solve it. Depending on the search algorithm selected and the configurations used, the optimal solution or a *best available solution* may be found. In other words, a good solution will be found, but it might not necessarily be the best one. When we speak about a “good solution” or “optimal solution,” we are referring to the performance of the solution in addressing the problem at hand.

One scenario where search algorithms are useful is finding the shortest path to the goal in a maze. Suppose that we’re in a square maze that is 10 blocks by 10 blocks (figure 2.4). There’s a goal that we want to reach and barriers that we cannot step into or over. The objective is to find a path to the goal while avoiding barriers with as few steps as possible by moving north, south, east, or west. In this example, the “player” cannot move diagonally.



**Figure 2.4 An example of the maze problem**

How can we find the shortest path to the goal while avoiding barriers? By thinking about the problem as a human, we can try each path possibility and count the moves. Using trial and error, we can find the paths that are the shortest, since this maze is relatively small.

Using the example maze, figure 2.5 depicts some possible paths to reach the goal, although, note that we don't reach the goal in option 1.

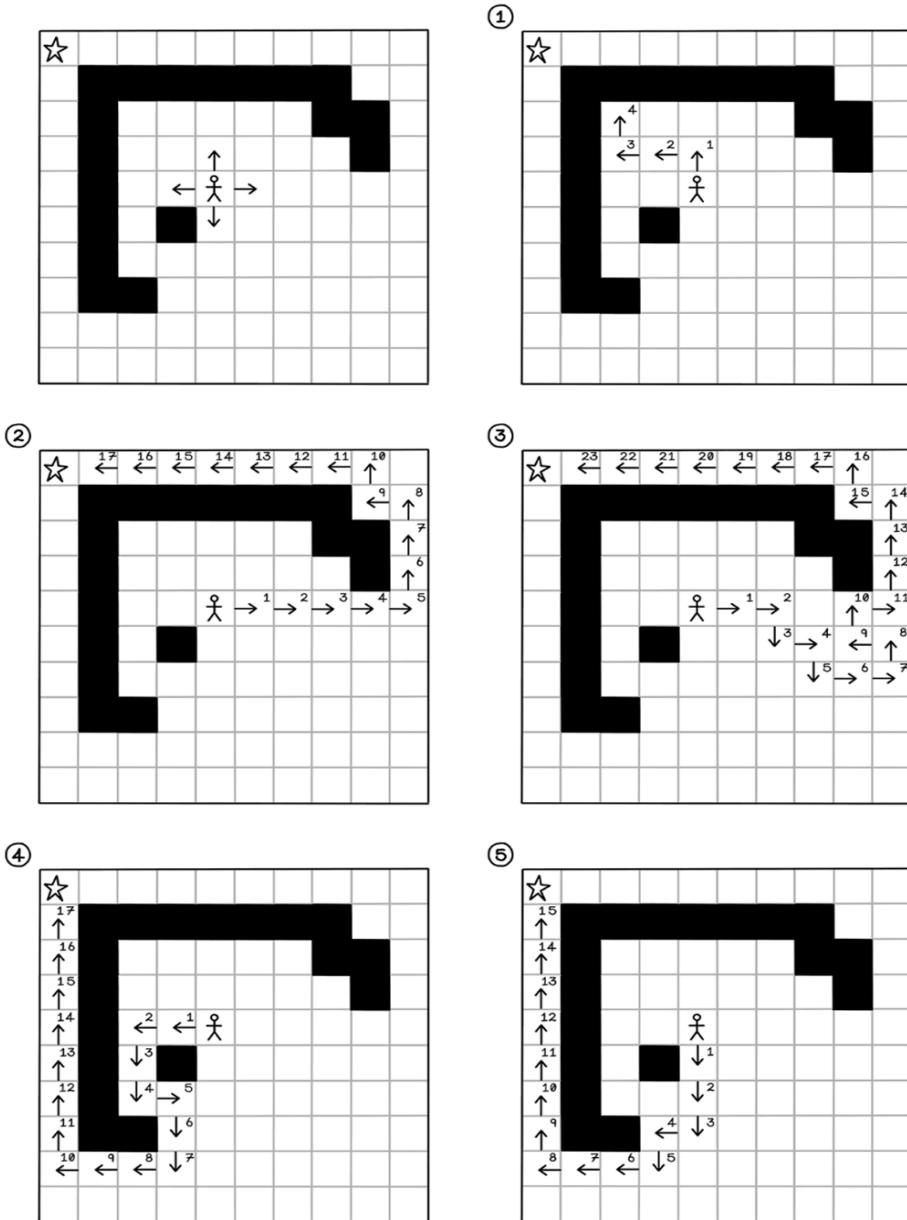
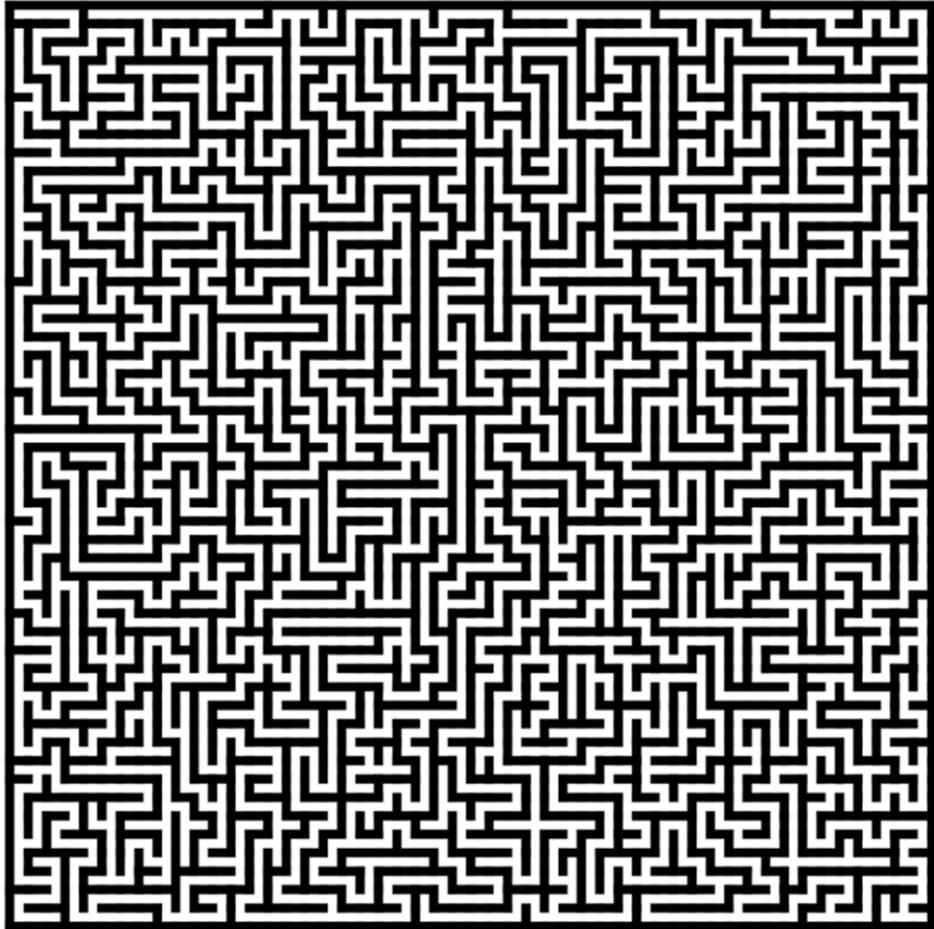


Figure 2.5 Examples of possible paths to the maze problem

By looking at the maze and counting blocks in different directions, we can find several solutions to the problem. Five attempts have been made to find four successful solutions out of an unknown number of solutions. It will take exhaustive effort to compute all possible solutions by hand:

- Attempt 1 is not a valid solution. It took 4 actions, and the goal was not found.
- Attempt 2 is a valid solution, taking 17 actions to find the goal.
- Attempt 3 is a valid solution, taking 23 actions to find the goal.
- Attempt 4 is a valid solution, taking 17 actions to find the goal.
- Attempt 5 is the best valid solution, taking 15 actions to find the goal.  
Although this attempt is the best one, it was found by chance.

If the maze were a lot larger, like the one in figure 2.6, it would take an immense amount of time to compute the best possible path manually. Search algorithms can help.



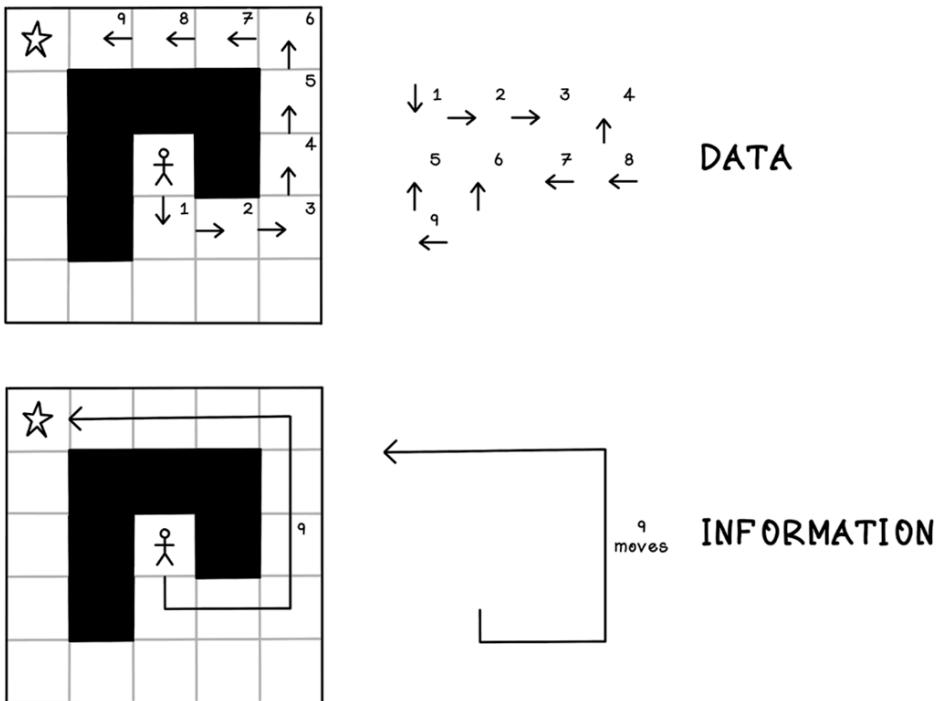
**Figure 2.6 A large example of the maze problem**

Our power as humans is to perceive a problem visually, understand it, and find solutions. We understand and interpret data and information in an abstract way. A computer cannot yet understand generalized information in the natural form that we do completely. The problem space needs to be represented in a way that is suitable for computation.

## 2.4 Representing state: Creating a framework to represent problem spaces and solutions

When representing data in a way that a computer can understand, we need to encode it logically so that it can be understood objectively. Although the data will be encoded “subjectively” by the person who performs the task, there should be a consistent way to repeat it.

Let's clarify the difference between data and information. *Data* is raw facts about something, and *information* is an interpretation of those facts that provides insight about the data in the specific domain. Information requires context and processing of data to provide meaning. As an example, each individual distance traveled in the maze example is data, and the sum of the total distance traveled is information. Depending on the perspective, level of detail, and desired outcome, classifying something as data or information can be dependent on the context (figure 2.7).



**Figure 2.7 Data versus information**

*Data structures* are concepts in computer science used to represent data in a way that is appropriate for efficient processing by algorithms. A data structure is an abstract data type consisting of data and operations organized in a specific way. The data structure we use is influenced by the type of problem and the desired goal.

An example of a data structure is an *array* - simply a collection of data. Different types of arrays have different properties that make them good for different purposes. Depending on the programming language used, an array could allow each value to be of a different type or require each value to be the same type, or the array may disallow duplicate values. These different types of arrays usually have different names. The features and constraints of different data structures also enable more efficient computation (figure 2.8).

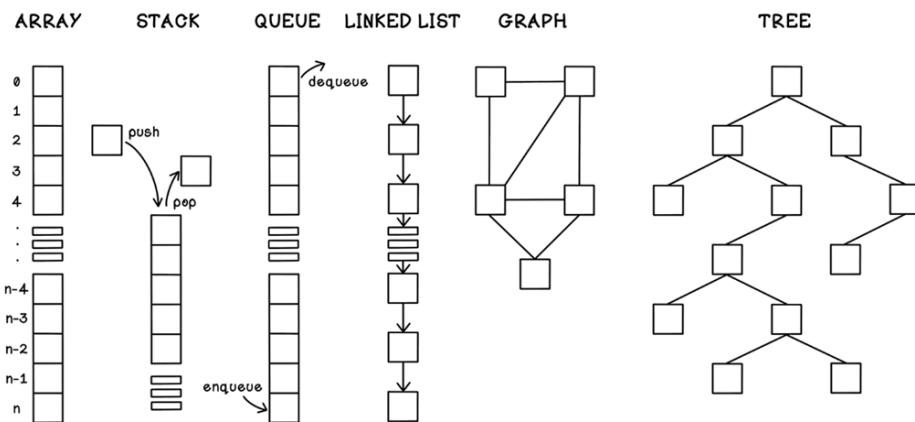
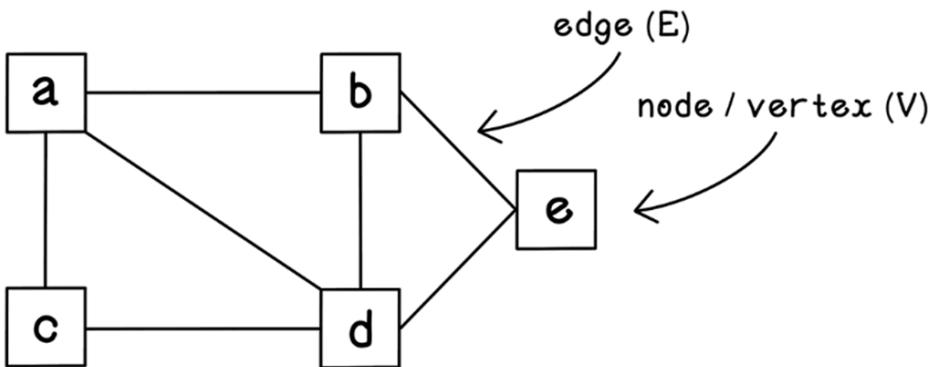


Figure 2.8 Data structures used with algorithms

### 2.4.1 Graphs: Representing search problems and solutions

A *graph* is a data structure containing several states with connections among them. Each state in a graph is called a *node* (or a *vertex*), and a connection between two states is called an *edge*. Graphs are studied in *graph theory* in mathematics and are used to model relationships among objects. Graphs are useful data structures that are easy for humans to understand, due to the ease of representing them visually as well as to their strong logical nature, which is ideal for processing via various algorithms (figure 2.9). We'll be seeing graph structures throughout the book.



$$V = \{a, b, c, d, e\}$$

$$E = \{ab, ac, ad, bd, be, cd, de\}$$

**Figure 2.9** The notation used to represent graphs

Figure 2.10 is a graph of the trip to the beach example that we discussed in the first section of this chapter. Each stop is a node on the graph; each edge between nodes represents points traveled between; and the weights on each edge indicate the distance traveled.

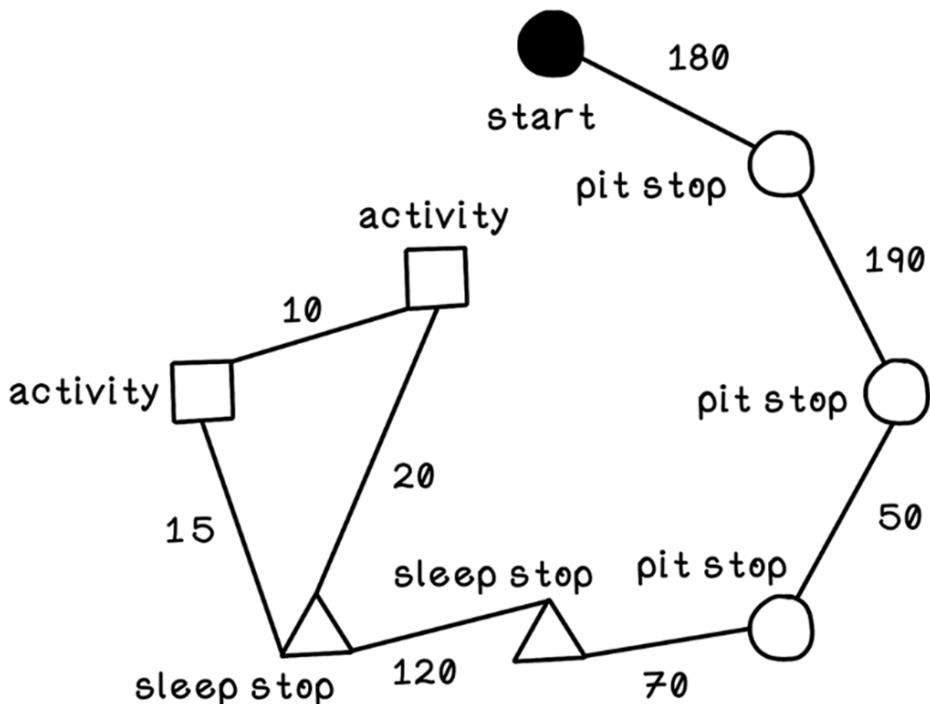
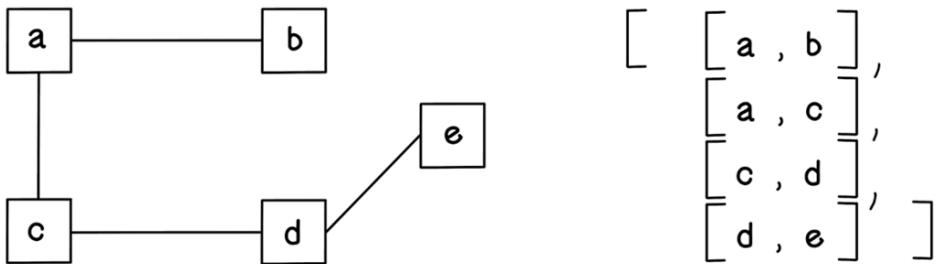


Figure 2.10 The example road trip represented as a graph

#### 2.4.2 Representing a graph as a concrete data structure

A graph can be represented in several ways for efficient processing by algorithms. At its core, a graph can be represented by an array of arrays that indicates relationships among nodes, as shown in figure 2.11. It is sometimes useful to have another array that simply lists all nodes in the graph so that the distinct nodes do not need to be inferred from the relationships each time.

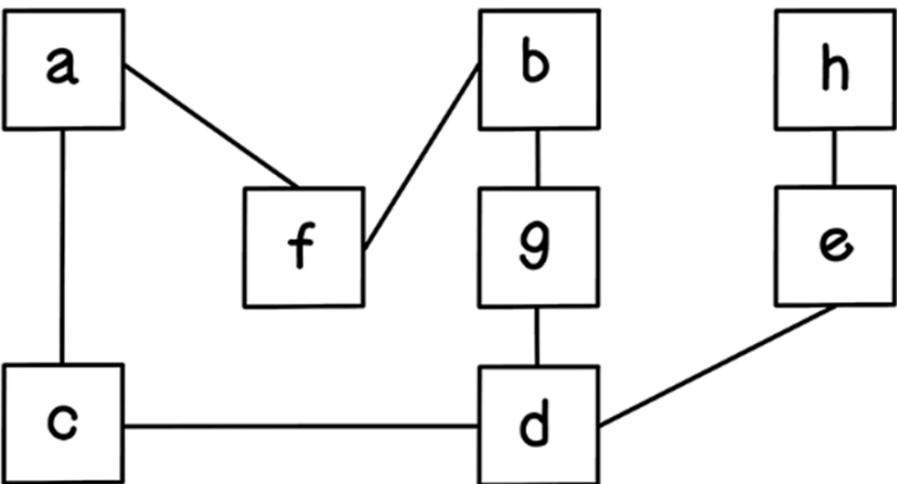


**Figure 2.11 Representing a graph as an array of arrays**

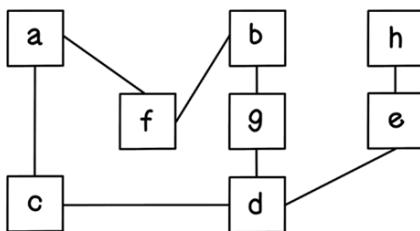
Other representations of graphs include an incidence matrix, an adjacency matrix, and an adjacency list. By looking at the names of these representations, you see that the adjacency of nodes in a graph is important. An *adjacent node* is a node that is connected directly to another node.

#### EXERCISE: REPRESENT A GRAPH AS A MATRIX

How would you represent the following graph using edge arrays?



## SOLUTION: REPRESENT A GRAPH AS A MATRIX



```
[ [ a, c ],
  [ a, f ],
  [ b, g ],
  [ b, f ],
  [ c, d ],
  [ d, g ],
  [ d, e ],
  [ e, h ] ]
```

Array of edges

	a	b	c	d	e	f	g	h
a	0	0	1	0	0	1	0	0
b	0	0	0	0	0	1	1	0
c	1	0	0	1	0	0	0	0
d	0	0	1	0	1	0	1	0
e	0	0	0	1	0	0	0	1
f	1	1	0	0	0	0	0	0
g	0	1	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0

Adjacency matrix

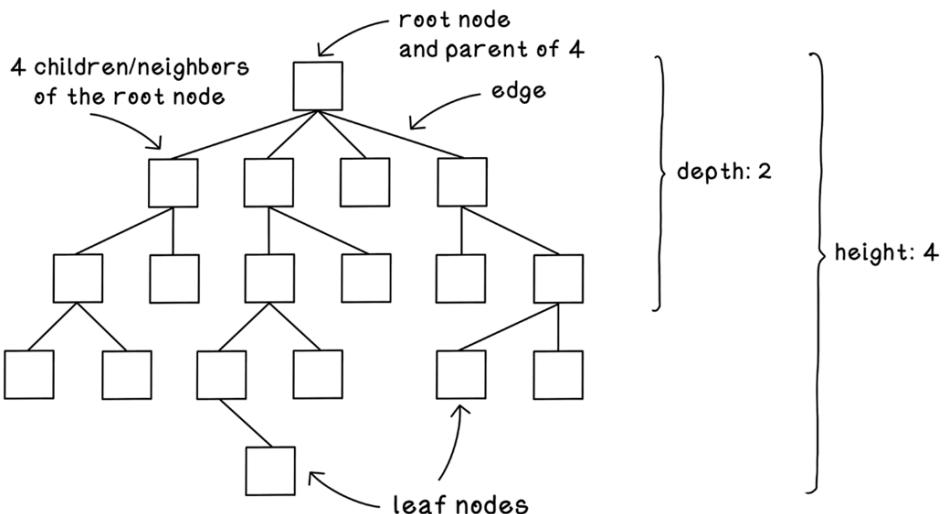
## 2.4.3 Trees: The concrete structures used to represent search solutions

A *tree* is a popular data structure that simulates a hierarchy of values or objects. A *hierarchy* is an arrangement of things in which a single object is related to several other objects below it. A tree is a connected acyclic graph—meaning all nodes are connected, but no loops (cycles) exist.

In a tree, the value or object represented at a specific point is called a *node*. Trees typically have a single root node with zero or more child nodes that could contain subtrees. Let's take a deep breath and jump into some terminology. When a node has connected nodes, the root node is called the *parent*. You can apply this thinking recursively. A child node may have its own children, which may also contain subtrees or their children. Each child node has a single parent node. A node without any children is a *leaf node*.

Trees also have a total height. The level of specific nodes is called a *depth*.

The terminology used to relate family members is heavily used in working with trees. Keep this analogy in mind, it will help you connect the concepts in the tree data structure. Note that in figure 2.12, the height and depth are indexed from 0 from the root node. The root node has 4 child nodes, the tree height is 4 and the depth in this example is 2. Depth is always based on where in the tree the algorithm is during traversal.



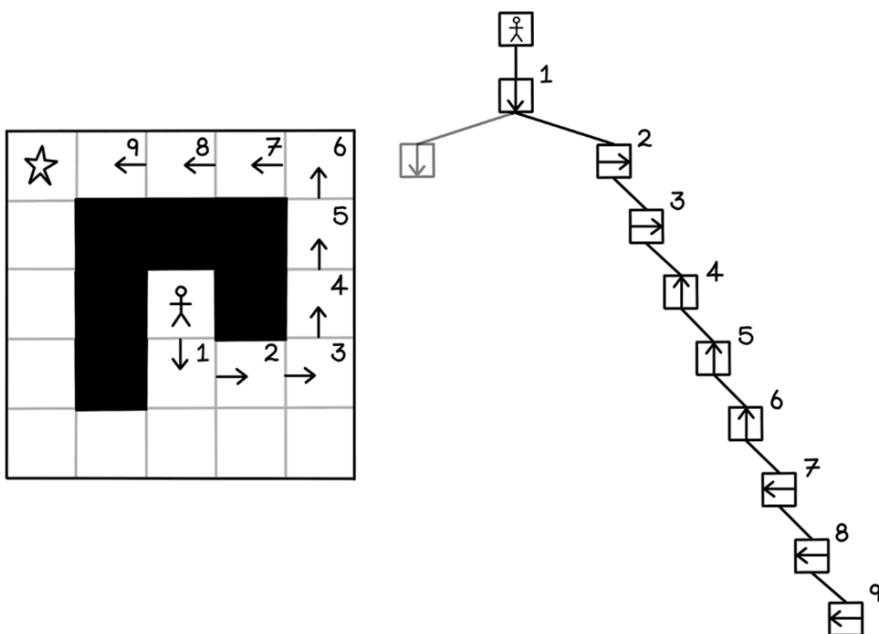
**Figure 2.12 The main attributes of a tree**

The topmost node in a tree is called the *root node*. A node directly connected to one or more other nodes is called a *parent node*. The nodes connected to a parent node are called *child nodes* or *neighbors*. Nodes connected to the same parent node are called *siblings*. A connection between two nodes is called an *edge*.

A *path* is a sequence of nodes and edges connecting nodes that are not directly connected. A node connected to another node by following a path away from the root node is called a *descendent*, and a node connected to another node by following a path toward the root node is called an *ancestor*. A node with no children is called a *leaf node*.

The term *degree* is used to describe the number of children a node has; therefore, a leaf node has degree zero.

Figure 2.13 represents a path from the start point to the goal for the maze problem. This path contains nine nodes that represent different moves being made in the maze.



**Figure 2.13 A solution to the maze problem represented as a tree**

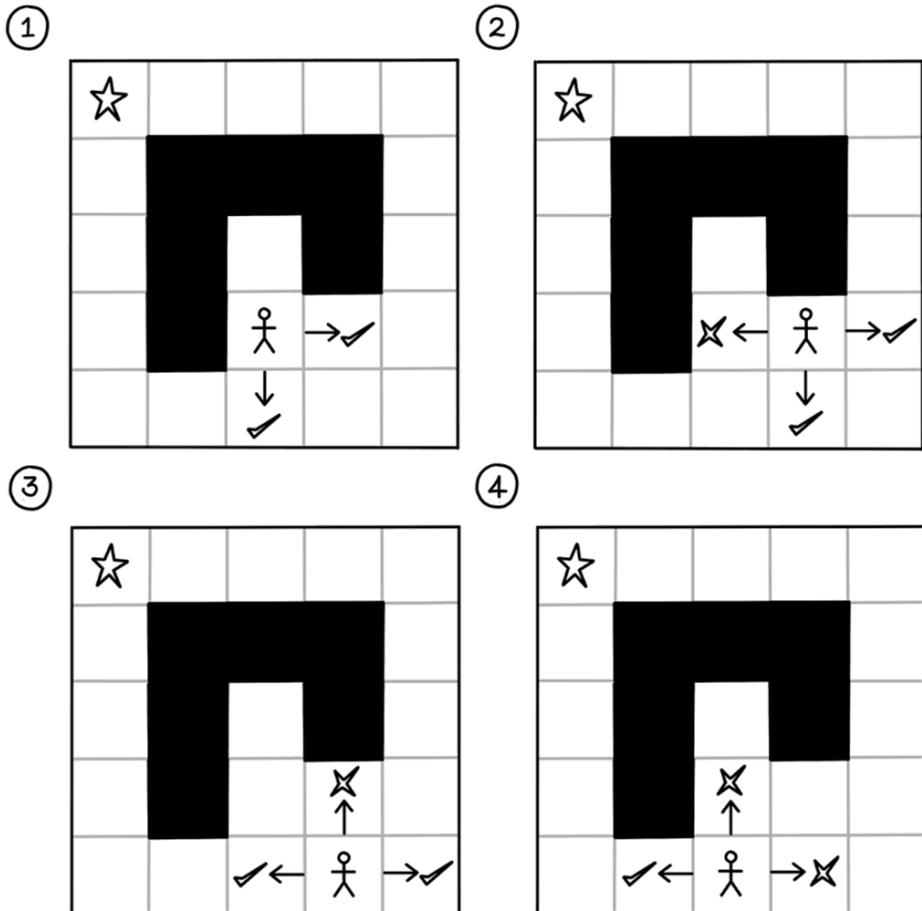
Trees are the fundamental data structure for many search algorithms, which we will be diving into next. Sorting algorithms are also useful in solving certain problems and computing solutions more efficiently. If you’re interested in learning more about sorting algorithms, take a look at *Grokking Algorithms* (Manning Publications).

## 2.5 Uninformed search: Looking blindly for solutions

*Uninformed search* is also known as *unguided search*, *blind search*, or *brute-force search*. Uninformed search algorithms have no additional information about the domain of the problem apart from the representation of the problem, which is usually a tree.

Think about exploring things you want to learn. Some people might look at a wide breadth of different topics and learn the basics of each, whereas other people might choose one narrow topic and explore its subtopics in-depth. This is what breadth-first search (BFS) and depth-first search (DFS) involve, respectively. *Depth-first search* explores a specific path from the start until it finds a goal at the utmost depth. *Breadth-first search* explores all options at a specific depth before moving to options deeper in the tree.

Consider the maze scenario (figure 2.14). In attempting to find an optimal path to the goal, assume the following simple constraint to prevent getting stuck in an endless loop and prevent cycles in our tree: *the player cannot move into a block that they have previously occupied*. Because uninformed algorithms attempt every possible option at every node, creating a cycle will cause the algorithm to fail catastrophically.

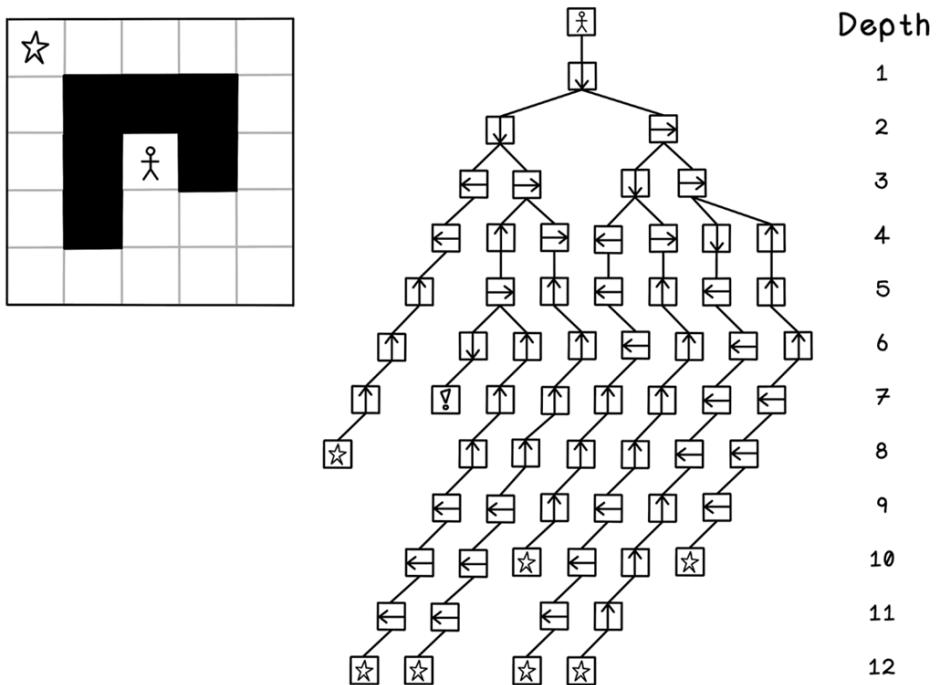


**Figure 2.14 The constraint for the maze problem**

This constraint prevents cycles in the path to the goal in our scenario. But this constraint will introduce problems if, in a different maze with different constraints or rules, moving into a previously occupied block more than once is required for the optimal solution.

In figure 2.15, all possible paths in the tree are represented to highlight the different options available. This tree contains seven paths that lead to the goal and one path that results in an invalid solution, given the constraint of not moving to previously occupied blocks. It's important to understand that in this small maze, representing all the possibilities is feasible. The entire point of search algorithms, however, is to search or generate these trees iteratively, because generating the entire tree of possibilities up front is computationally expensive and not efficient at all.

It is also important to note that the term *visiting* is used to indicate different things here. The player visits blocks in the maze. The algorithm also visits nodes in the tree. The order of choices will influence the order of nodes being visited in the tree. In the maze example, the priority order of movement is north, south, east, and then west.



**Figure 2.15 All possible movement options represented as a tree**

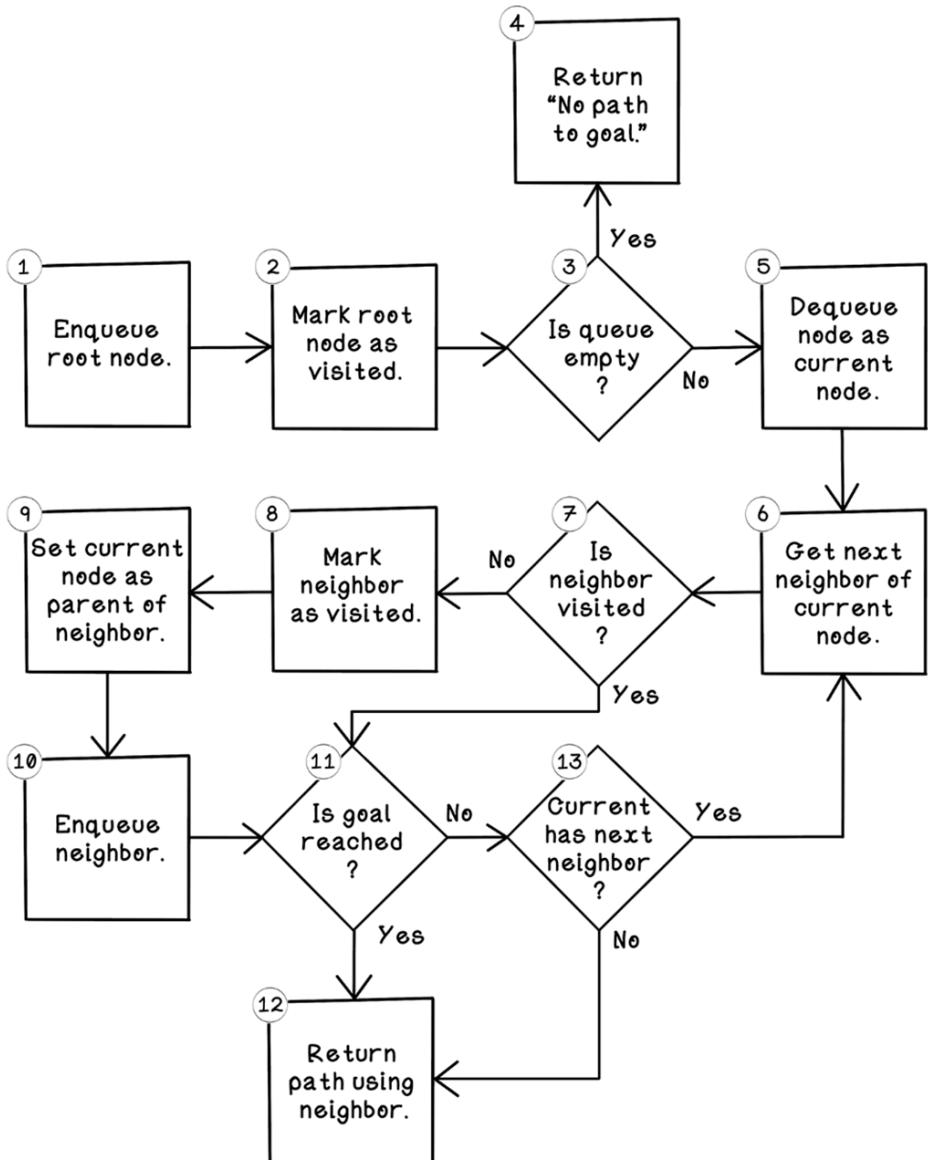
Now that we understand the ideas behind trees and the maze example, let's explore how search algorithms can generate trees that seek out paths to the goal.

## 2.6 Breadth-first search: Looking wide before looking deep

*Breadth-first search* is an algorithm used to traverse or generate a tree. This algorithm starts at a specific node, usually the root, and explores every node at that depth before exploring the next depth of nodes. Think of BFS like dropping a stone into a calm pond. The ripples expand uniformly in all directions, touching everything 1 meter away, then everything 2 meters away, and so on. The algorithm mimics this expanding ring, visiting all neighbors at the current depth before moving outward. This guarantees finding the shortest path in unweighted graphs (graphs where all edges have the same cost).

The breadth-first search algorithm is best implemented by using a first-in, first-out queue in which the current depths of nodes are processed, and their children are queued to be processed later. This order of processing is exactly what we require when implementing this algorithm.

Figure 2.16 is a flow chart describing the sequence of steps involved in the breadth-first search algorithm.

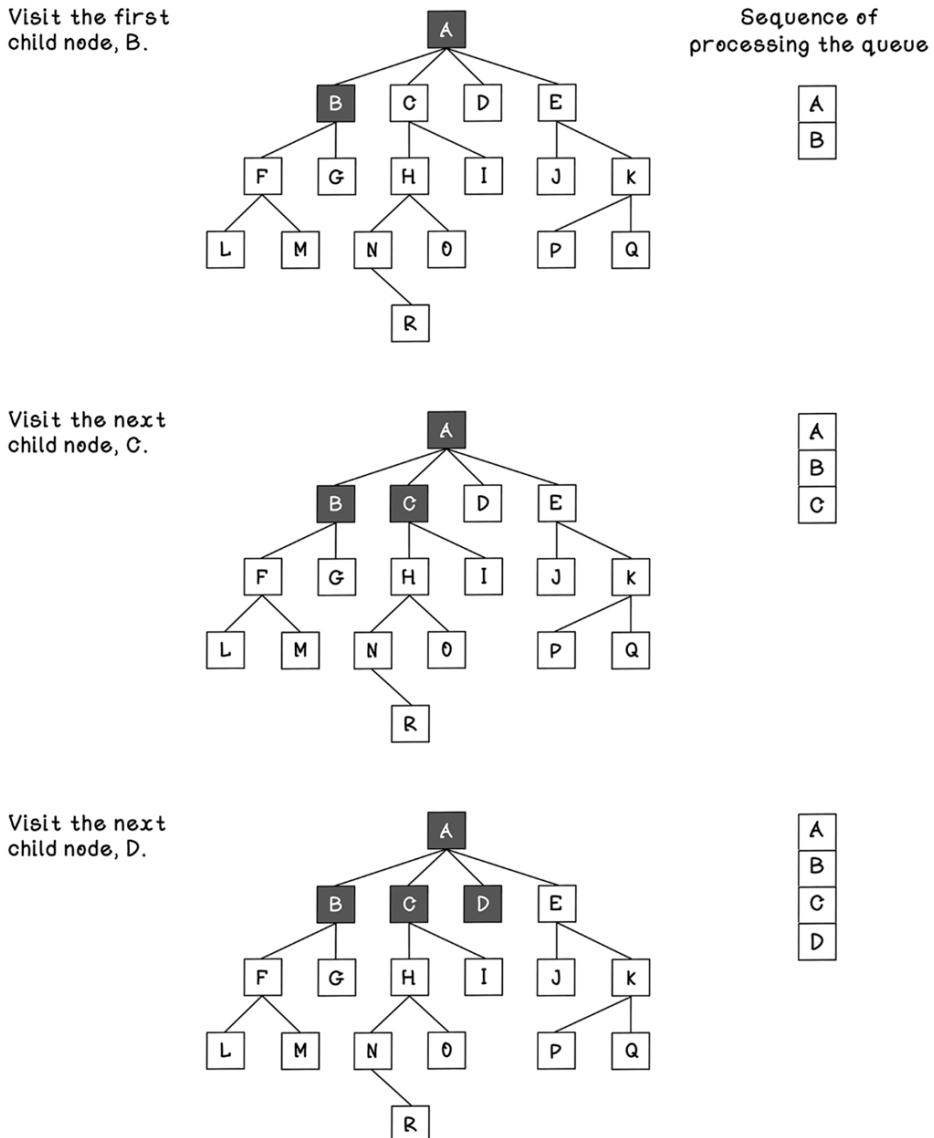


**Figure 2.16 Flow of the breadth-first search algorithm**

Let's walk through each step in the algorithm to learn exactly what operations happen:

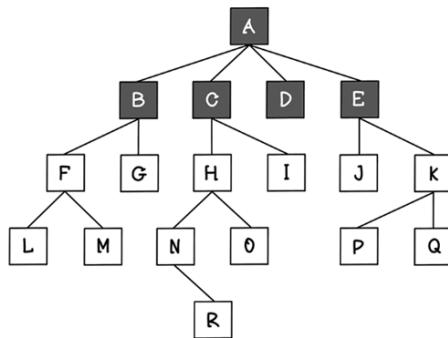
1. *Enqueue root node*—The breadth-first search algorithm is best implemented with a queue. Objects are processed in the sequence in which they are added to the queue. This process is also known as *first in, first out* (FIFO). The first step is adding the root node to the queue. This node will represent the starting position of the player on the map.
2. *Mark root node as visited*—Now that the root node has been added to the queue for processing, it is marked as visited to prevent it from being revisited for no reason.
3. *Is queue empty?*—If the queue is empty (all nodes have been processed after many iterations), and if no path has been returned in step 12 of the algorithm, there is no path to the goal. If there are still nodes in the queue, the algorithm can continue its search to find the goal.
4. *Return No path to goal*—This message is the one possible exit from the algorithm if no path to the goal exists.
5. *Dequeue node as current node*—By pulling the next object from the queue and setting it as the current node of interest, we can explore its possibilities. When the algorithm starts, the current node will be the root node.
6. *Get next neighbor of current node*—This step involves getting the next possible move in the maze from the current position by referencing the maze and determining whether a north, south, east, or west movement is possible.
7. *Is neighbor visited?*—If the current neighbor has not been visited, it hasn't been explored yet and can be processed now.
8. *Mark neighbor as visited*—This step indicates that this neighbor node has been visited.
9. *Set current node as parent of neighbor*—Set the origin node as the parent of the current neighbor. This step is important for tracing the path from the current neighbor to the root node. From a map perspective, the origin is the position that the player moved from, and the current neighbor is the position that the player moved to.
10. *Enqueue neighbor*—The neighbor node is queued for its children to be explored later. This queuing mechanism allows nodes from each depth to be processed in that order.
11. *Is goal reached?*—This step determines whether the current neighbor contains the goal that the algorithm is searching for.
12. *Return path using neighbor*—By referencing the parent of the neighbor node, then the parent of that node, and so on, the path from the goal to the root will be described. The root node will be a node without a parent.
13. *Current has next neighbor?*—If the current node has more possible moves to make in the maze, jump to step 6 for that move.

Let's walk through what that would look like in a simple tree. Notice that as the tree is explored and nodes are added to the FIFO queue, the nodes are processed in the order desired by leveraging the queue (figures 2.17 and 2.18).



**Figure 2.17 The sequence of tree processing using breadth-first search (part 1)**

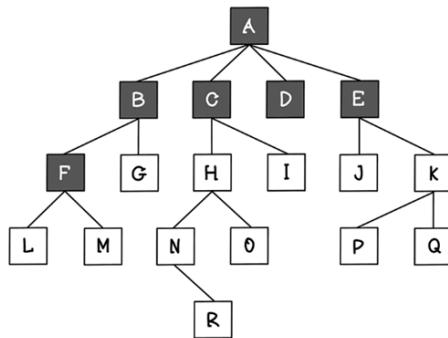
Visit the final node on this depth, E.



Sequence of processing the queue

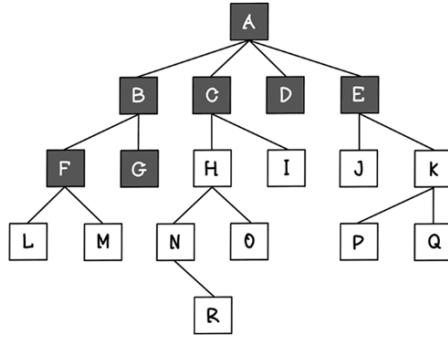
A
B
C
D
E

Visit the first child of the first neighbor of A. This is F, first child of B.



A
B
C
D
E
F

Visit the next child of B. This is G.

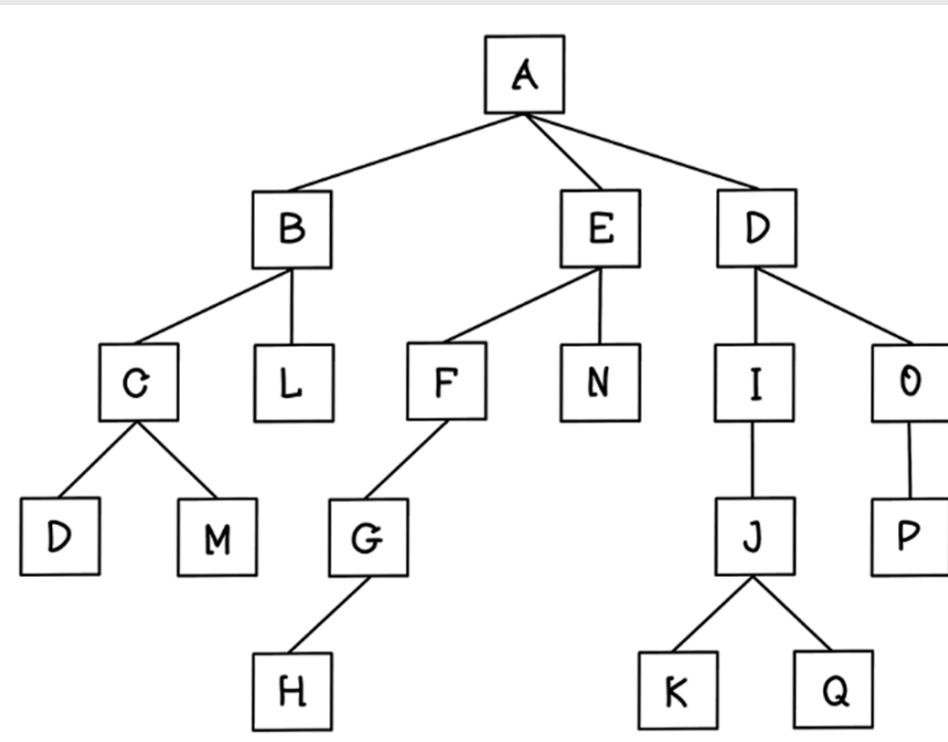


A
B
C
D
E
F
G

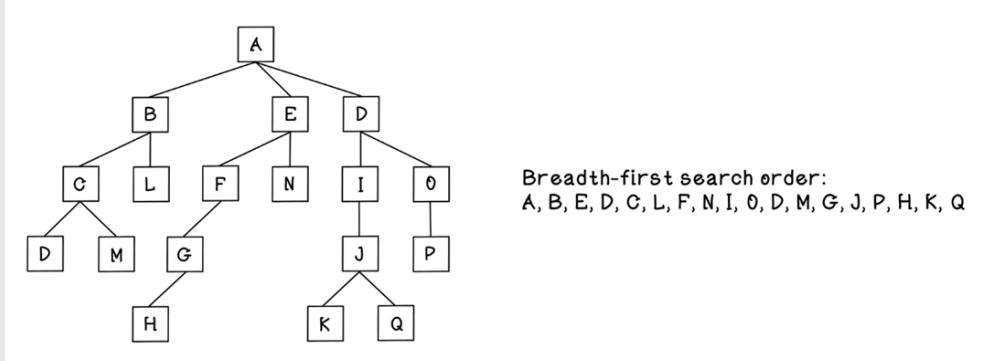
Figure 2.18 The sequence of tree processing using breadth-first search ( part 2)

**EXERCISE: DETERMINE THE PATH TO THE SOLUTION**

What would be the order of visits using breadth-first search for the following tree?



### SOLUTION: DETERMINE THE PATH TO THE SOLUTION



In the maze example, the algorithm needs to understand the current position of the player in the maze, evaluate all possible choices for movement, and repeat that logic for each choice of movement made until the goal is reached. By doing this, the algorithm generates a tree with a single path to the goal.

It is important to understand that the process of visiting nodes in a tree is used to generate nodes in a tree. We are simply finding related nodes through a mechanism.

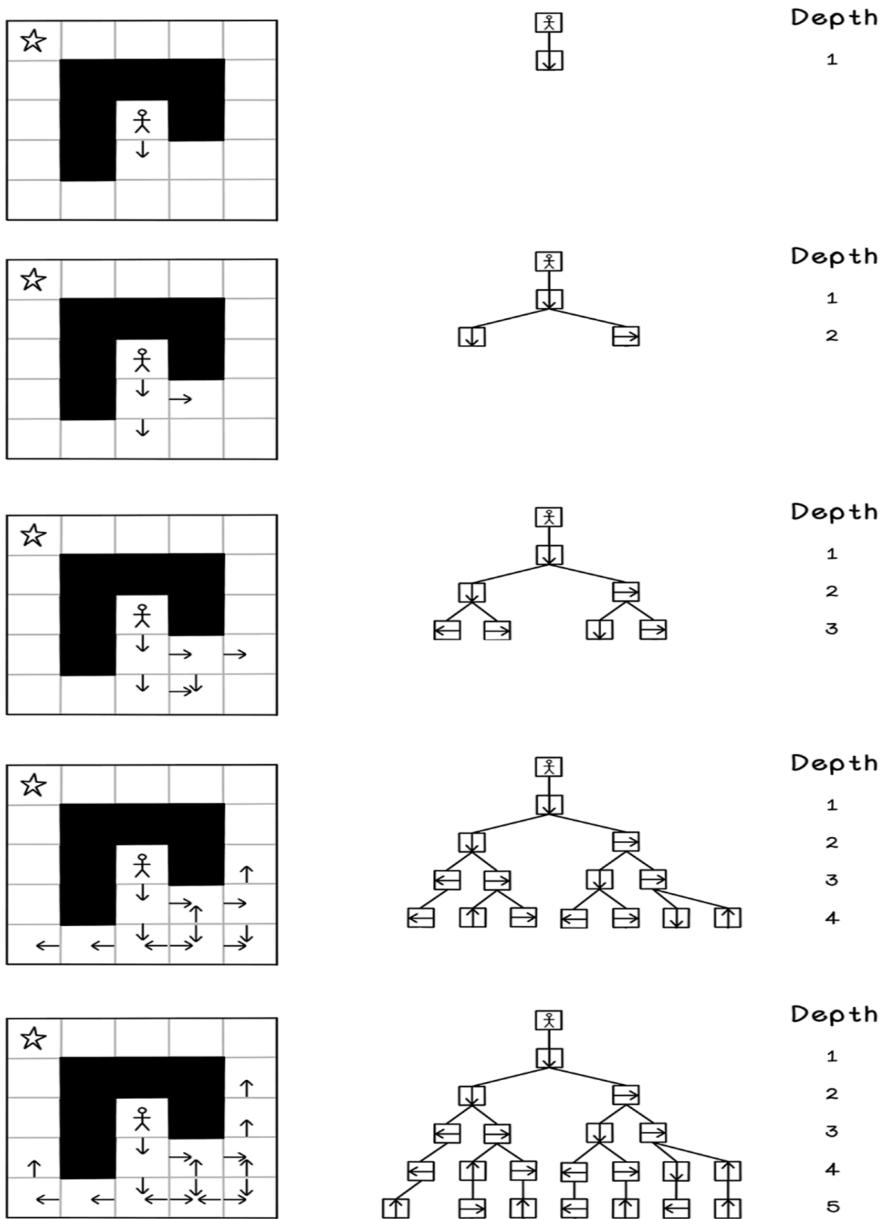
Each path to the goal consists of a series of moves to reach the goal. The number of moves in the path is the distance to reach the goal for that path, which we will call the *cost*. The number of moves also equals the number of nodes visited in the path, from the root node to the leaf node that contains the goal. The algorithm moves down the tree depth by depth until it finds a goal; then it returns the first path that got it to the goal as the solution. There may be a more optimal path to the goal, but because breadth-first search is uninformed, it is not guaranteed to find that path.

---

**NOTE** In the maze example, all search algorithms used terminate when they've found a solution to the goal. It is possible to allow these algorithms to find multiple solutions with a small tweak to each algorithm, but the best use cases for search algorithms find a single goal, since it's often too expensive to explore the entire tree of possibilities.

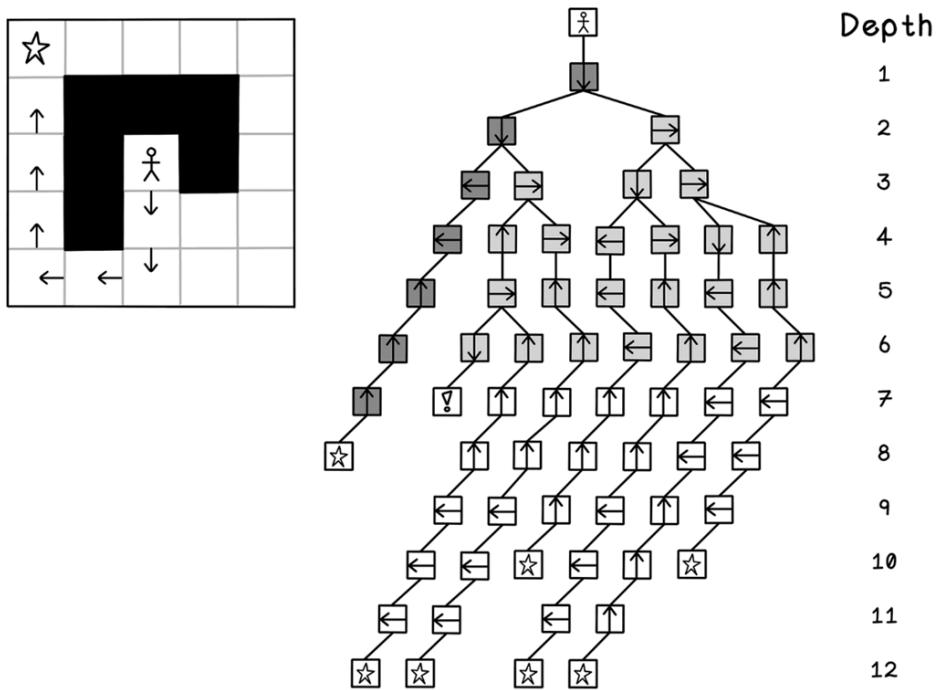
---

Figure 2.19 shows the generation of a tree using movements in the maze. Because the tree is generated using breadth-first search, each depth is generated to completion before looking at the next depth (figure 2.20).



**Figure 2.19** Maze movement tree generation using breadth-first search

In figure 2.19 the BFS algorithm has explored up to depth 5 of the tree, and notice that on the actual map, we see two paths to the goal emerge from its search thus far.



**Figure 2.20 Nodes visited in the entire tree after breadth-first search**

Figure 2.20 illustrates the entire tree of possibilities (remember this is only done for learning purposes). Notice that at depth 7, BFS has found a path to the goal successfully. So our solution is south, south, west, west, north, north, north.

### PYTHON CODE SAMPLE

As mentioned previously, the breadth-first search algorithm uses a queue to generate a tree one depth at a time. Having a structure to store visited nodes is critical to prevent getting stuck in cyclic loops; and setting the parent of each node is important for determining a path from the starting point in the maze to the goal.

```

def run_bfs(maze_puzzle, current_point, visited_points):
    queue = deque()
    # Append the current node to the queue
    queue.append(current_point)
    visited_points.append(current_point)
    # Keep searching while there are nodes in the queue
    while queue:
        current_point = queue.popleft() #A
        neighbors = maze_puzzle.get_neighbors(current_point) #B
        for neighbor in neighbors: #C
            if not is_in_visited_points(neighbor, visited_points): #D
                neighbor.set_parent(current_point) #D
                queue.append(neighbor) #D
                visited_points.append(neighbor) #D
                if maze_puzzle.get_current_point_value(neighbor) == '*': #E
                    return neighbor #E
    return 'No path to the goal found.' #F

#A Set the next node in the queue as the current node
#B Get the neighbors of the current node
#C Iterate through the neighbors of the current node
#D Add the neighbor to the queue if it hasn't been visited
#E Return the path to the current neighbor if it is the goal
#F In the case that no path to the goal was found

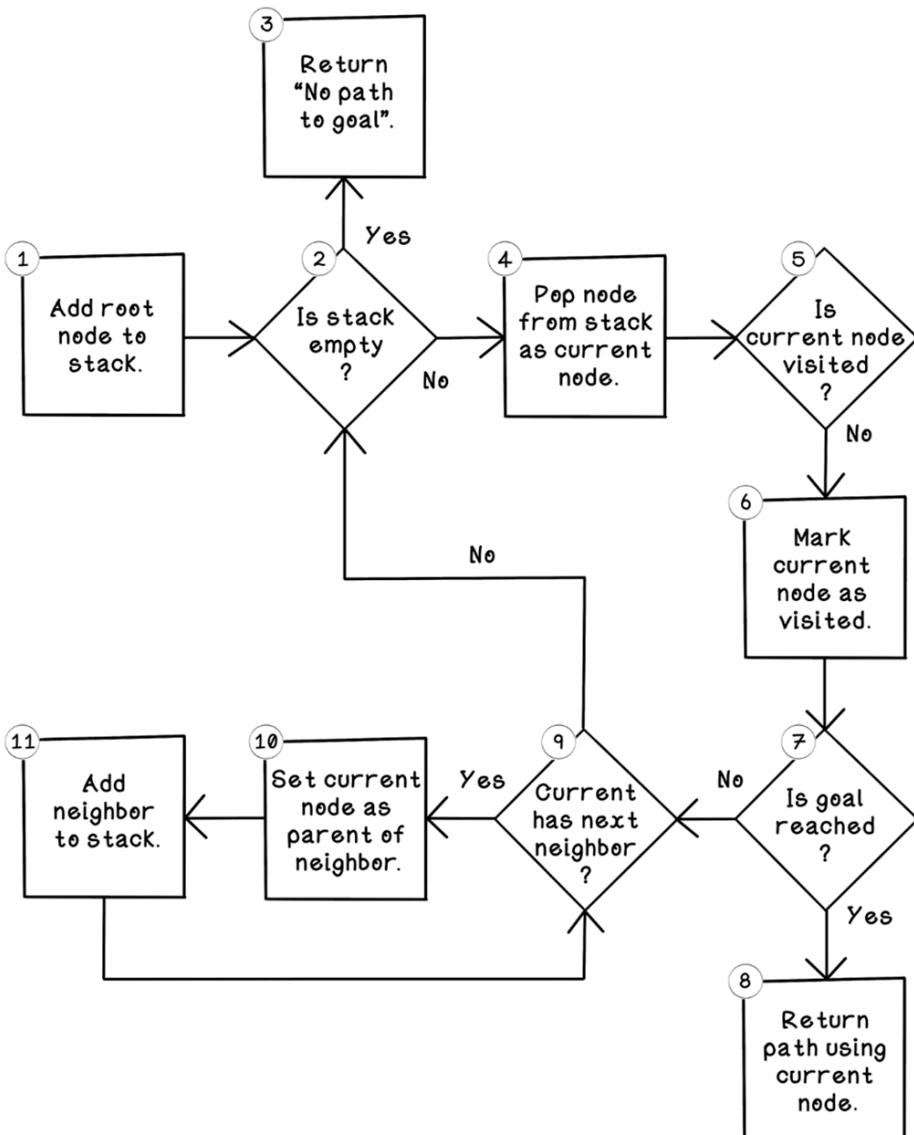
```

## 2.7 Depth-first search: Looking deep before looking wide

*Depth-first search* is another algorithm used to traverse a tree or generate nodes and paths in a tree.

Think of DFS like a maze explorer who commits to a single path, walking as far as possible until hitting a dead end. Only then do they backtrack to the last intersection to try a different turn. Unlike the cautious, expanding nature of BFS, DFS is aggressive and dives deep immediately.

Figure 2.21 illustrates the general flow of the depth-first search algorithm.



**Figure 2.21 Flow of the depth-first search algorithm**

Let's walk through the flow of the depth-first search algorithm:

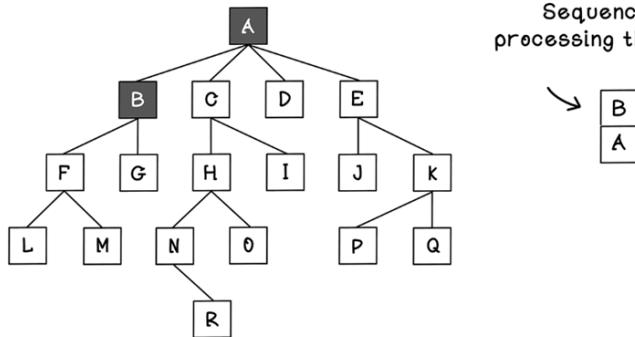
1. *Add root node to stack*—The depth-first search algorithm can be implemented by using a stack in which the last object added is processed first. This process is known as *last in, first out* (LIFO). The first step is adding the root node to the stack.

2. *Is stack empty?*—If the stack is empty and no path has been returned in step 8 of the algorithm, there is no path to the goal. If there are still nodes in the stack, the algorithm can continue its search to find the goal.
3. *Return No path to goal*—This return is the one possible exit from the algorithm if no path to the goal exists.
4. *Pop node from stack as current node*—By pulling the next object from the stack and setting it as the current node of interest, we can explore its possibilities.
5. *Is current node visited?*—If the current node has not been visited, it hasn't been explored yet and can be processed now.
6. *Mark current node as visited*—This step indicates that this node has been visited to prevent unnecessary repeat processing of it.
7. *Is goal reached?*—This step determines whether the current neighbor contains the goal that the algorithm is searching for.
8. *Return path using current node*—By referencing the parent of the current node, then the parent of that node, and so on, the path from the goal to the root is described. The root node will be a node without a parent.
9. *Current has next neighbor?*—If the current node has more possible moves to make in the maze, that move can be added to the stack to be processed. Otherwise, the algorithm can jump to step 2, where the next object in the stack can be processed if the stack is not empty. The nature of the LIFO stack allows the algorithm to process all nodes to a leaf node depth before backtracking to visit other children of the root node.
10. *Set current node as parent of neighbor*—Set the origin node as the parent of the current neighbor. This step is important for tracing the path from the current neighbor to the root node. From a map perspective, the origin is the position that the player moved from, and the current neighbor is the position that the player moved to.
11. *Add neighbor to stack*—The neighbor node is added to the stack for its children to be explored later. Again, this stacking mechanism allows nodes to be processed to the utmost depth before processing neighbors at shallow depths.

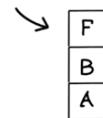
Figures 2.22 and 2.23 explore how the LIFO stack is used to visit nodes in the order desired by depth-first search. Notice that nodes get pushed onto and popped from the stack as the depths of the nodes visited progress. The term *push* describes adding objects to a stack, and the term *pop* describes removing the topmost object from the stack.

Similarly to breadth-first search, visit the first child of node A. This is B.

Sequence of processing the stack



Instead of visiting other child nodes of A, the first child of B is visited – in this case, F.



Again, the first child of F is visited. This is L.

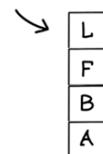
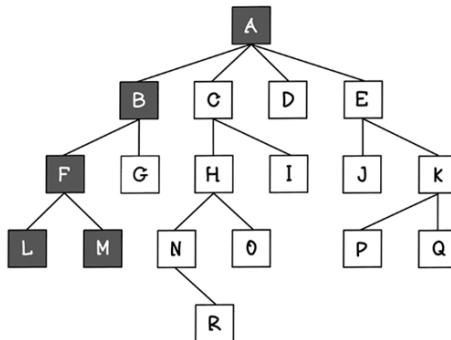
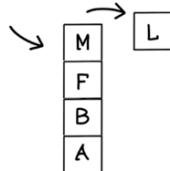


Figure 2.22 The sequence of tree processing using depth-first search (part 1)

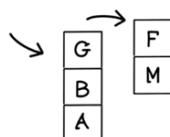
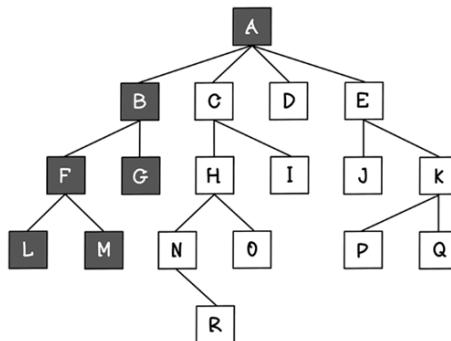
Because L is a leaf node (it has no children), the algorithm backtracks to visit the next child of F, which is M.



Sequence of processing the stack



Because M is a leaf node, the algorithm backtracks to visit the next child of B. Because F has no unvisited children, this child is G.



Finally, because all children of B have been visited, the algorithm backtracks to the next child of A, which is C.

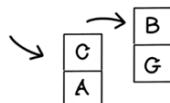
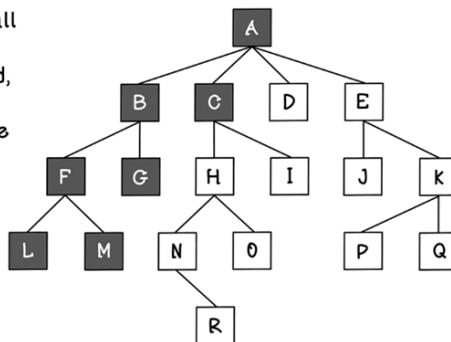
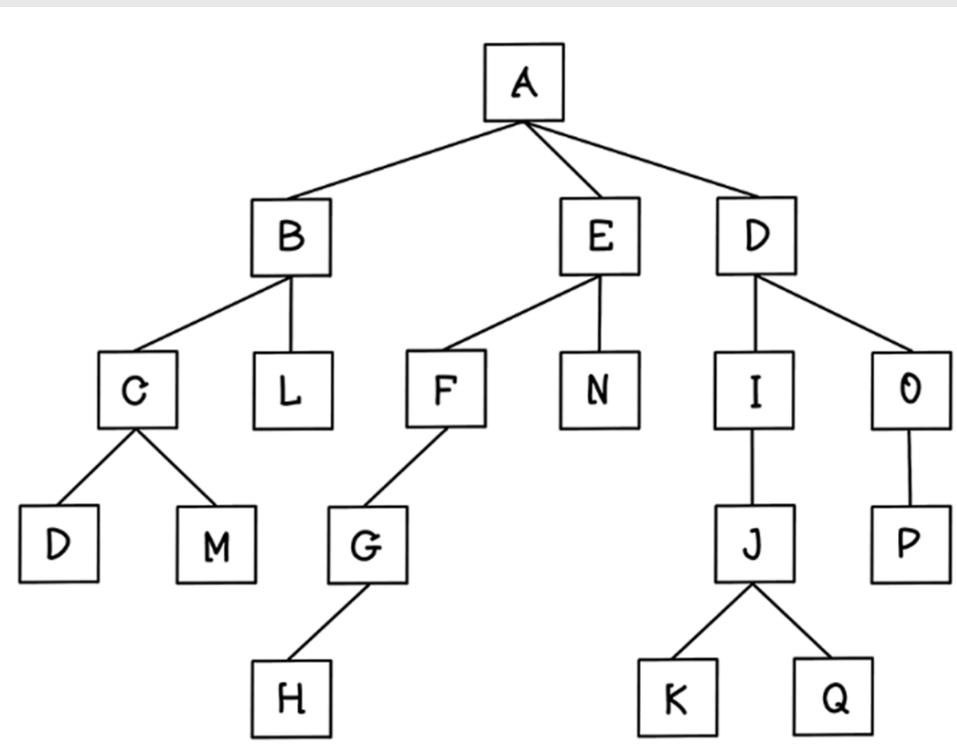


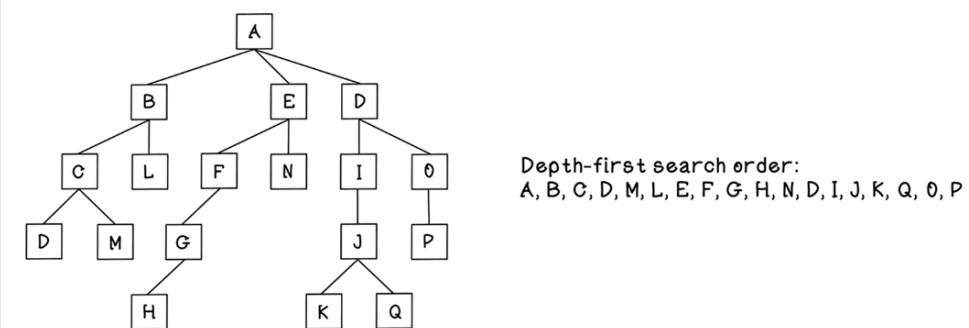
Figure 2.23 The sequence of tree processing using depth-first search (part 2)

**EXERCISE: DETERMINE THE PATH TO THE SOLUTION**

What would the order of visits be in depth-first search for the following tree?



### SOLUTION: DETERMINE THE PATH TO THE SOLUTION



It is important to understand that the order of children matters substantially when using depth-first search, as the algorithm explores the first child until it finds leaf nodes before backtracking.

In the maze example, the order of movement (north, south, east, and west) influences the path to the goal that the algorithm finds. A change in order will result in a different solution. The forks represented in figures 2.24 and 2.25 don't matter; what matters is the order of the movement choices in our maze example.

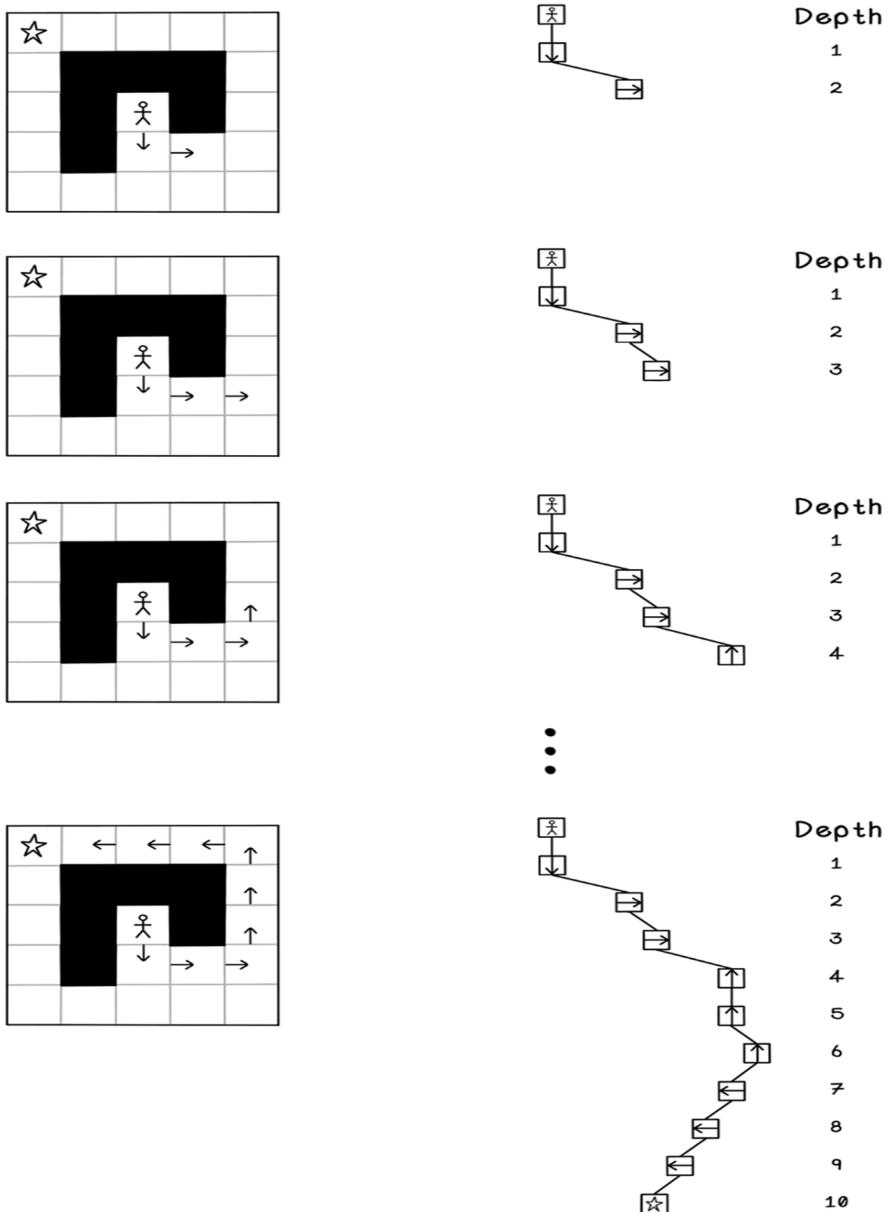
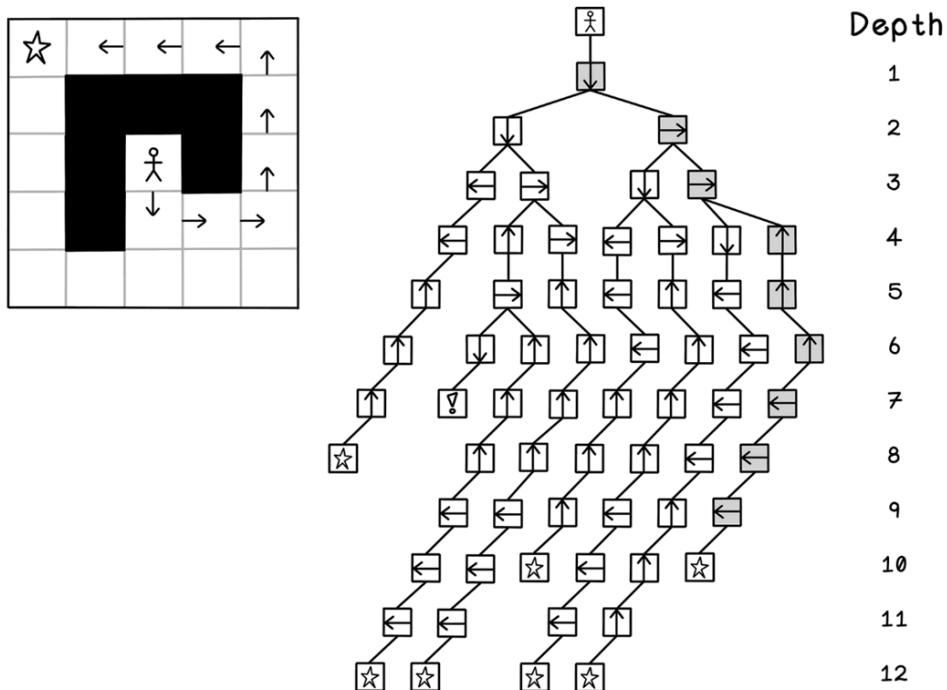


Figure 2.24 Maze movement tree generation using depth-first search

Notice in figure 2.24, unlike BFS, the DFS algorithm has exploited one path, and (by chance) using that path has led to finding the goal with south, east, east, north, north, north, west, west, west as the solution.



**Figure 2.25 Nodes visited in the entire tree after depth-first search**

Again, figure 2.25 shows our full tree, and highlights the path taken by the DFS algorithm.

### PYTHON CODE SAMPLE

Although the depth-first search algorithm can be implemented with a recursive function, we're looking at an implementation that is achieved with a stack to better represent the order in which nodes are visited and processed. It is important to keep track of the visited points so that the same nodes do not get visited unnecessarily, creating cyclic loops:

```

def run_dfs(maze_game, current_point):
    visited_points = [] #A
    stack = [current_point] #A
    while stack: #B
        next_point = stack.pop() #C
        if not is_in_visited_points(next_point, visited_points): #D
            visited_points.append(next_point) #D
            if maze_game.get_current_point_value(next_point) == '*': #E
                return next_point #E
            else:
                neighbors = maze_game.get_neighbors(next_point) #F
                for neighbor in neighbors: #F
                    neighbor.set_parent(next_point) #F
                    stack.append(neighbor) #F
    return 'No path to the goal found.'

```

#A Append the current node to the stack

#B Keep searching while there are nodes in the stack

#C Set the next node in the stack as the current node

#D If the current node hasn't already been exploited, search it

#E Return the path to the current neighbor if it is the goal

#F Add the current node's neighbors to the stack

Now that we have explored both approaches, how do you choose between them? It usually comes down to two factors: the need for the shortest path versus the constraints on memory.

Breadth-First Search (BFS) is your choice when you absolutely need the shortest path (like a GPS finding the fastest route). However, there is a cost: BFS must store every node at the current depth in memory before moving to the next. In a massive graph, this “expanding ring” can consume huge amounts of RAM.

Depth-First Search (DFS) is your choice when memory is limited. Because it dives deep along a single path, it only needs to store the nodes on that specific path. It is very memory efficient. The downside is that it's not guaranteed to find the shortest path; it will simply return the first solution it bumps into.

## 2.8 Use cases for uninformed search algorithms

While the following real-world examples are technically Graphs (which may contain loops/cycles), we can still use search algorithms on them by keeping track of visited nodes to avoid going in circles. Uninformed search algorithms are versatile and useful in several real-world use cases, such as:

- *Finding paths between nodes in a network*—Historically, this was fundamental to the creation of ARPANET (the precursor to the Internet) in 1969, where routing algorithms were needed to pass messages between the first four connected universities. When two computers need to communicate over a network, the connection passes through many connected computers and devices. Search algorithms can be used to establish a path in that network between two devices.
- *Crawling web pages*—Web searches allow us to find information on the internet across a vast number of web pages. To index these web pages, crawlers typically read the information on each page, as well as follow each link on that page recursively. Search algorithms are useful for creating crawlers, metadata structures, and relationships between content. A famous example is Google's initial architecture (1998), where Brin and Page described using URL servers to feed search algorithms that crawled millions of pages to map the web's structure.
- *Finding social network connections*—Social media applications contain many people and their relationships. Bob may be friends with Alice, for example, but not direct friends with John, so Bob and John are indirectly related via Alice. A social media application can suggest that Bob and John should become friends because they may know each other through their mutual friendship with Alice. A classic application is the “Oracle of Bacon” viral meme, which uses Breadth-First Search on IMDb data to find the shortest path of co-stars linking any actor to Kevin Bacon.

## 2.9 Optional: More about graph categories

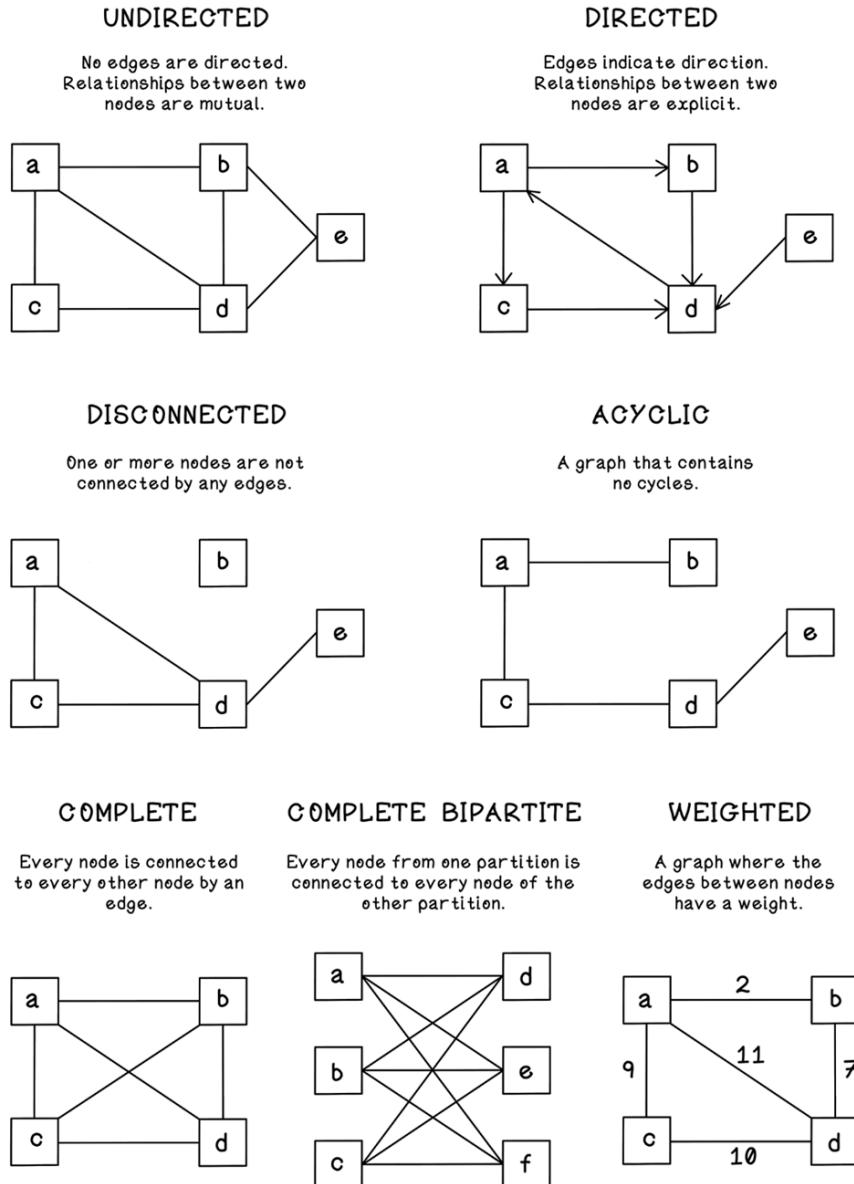
Graphs are useful for many computer science and mathematical problems, and due to the nature of different types of graphs, different principles and algorithms may apply to specific categories of graphs. A graph is categorized based on its overall structure, number of nodes, number of edges, and interconnectivity between nodes.

These categories of graphs are good to know about, as they are common and sometimes referenced in search and other AI algorithms:

- *Undirected graph*—No edges are directed. Relationships between two nodes are mutual. As with roads between cities, there are roads traveling in both directions.
- *Directed graph*—Edges indicate direction. Relationships between two nodes are explicit. As in a graph representing a child of a parent, the child cannot be the parent of its parent.
- *Disconnected graph*—One or more nodes are not connected by any edges. As in a graph representing physical contact between continents, some nodes are not connected. Like continents, some are connected by land, and others are separated by oceans.

- *Acyclic graph*—A graph that contains no cycles. Think of university course prerequisites: you must take Calc 101 to take Calc 201. You cannot have a situation where a course eventually requires itself as a prerequisite.
- *Complete graph*—Every node is connected to every other node by an edge. As in the lines of communication in a small team, everyone talks to everyone else to collaborate.
- *Complete bipartite graph*—A *vertex partition* is a grouping of vertices. Given a vertex partition, every node from one partition is connected to every node of the other partition with edges. Like at a cheese-tasting event, every person (Group A) connects with every cheese (Group B), but people don't taste people, and cheese doesn't taste cheese.
- *Weighted graph*—A graph in which the edges between nodes have a weight. As in the distance between cities, some cities are farther than others. The connections "weigh" more.

It is useful to understand the different types of graphs to best describe the problem and use the most efficient algorithm for processing (figure 2.26). Some of these categories of graphs are discussed in upcoming chapters, such as chapter 6 on ant colony optimization and chapter 8 on neural networks.



**Figure 2.26 Types of graphs**

## 2.10 Optional: More ways to represent graphs

Depending on the context, other encodings of graphs may be more efficient for processing or easier to work with, depending on the programming language and tools you're using.

### 2.10.1 Incidence matrix

An incidence matrix uses a matrix where the height is the number of nodes and the width is the number of edges. Each column represents a specific edge. Why would you choose this over an *Adjacency Matrix*? An Incidence Matrix is particularly useful when the edges are more important than the nodes. For example, in circuit design or physical flow networks, we care deeply about the connections (edges).

- *In Undirected Graphs:* The column for an edge contains a 1 in the rows corresponding to the two connected nodes, and 0 everywhere else.
- *In Directed Graphs:* We use signs to indicate direction. A common convention is to place a 1 for the source node (where the edge starts) and a -1 for the destination node (where the edge ends).

Note: Mathematical conventions vary; some systems might swap the -1 and 1, but the concept of using opposite signs to denote direction remains the same. An incidence matrix can be used to represent both directed and undirected graphs (figure 2.27).

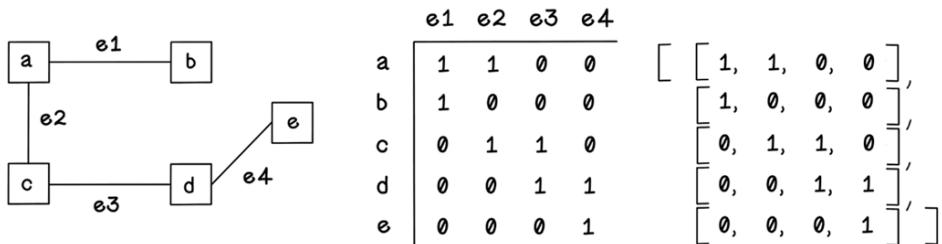
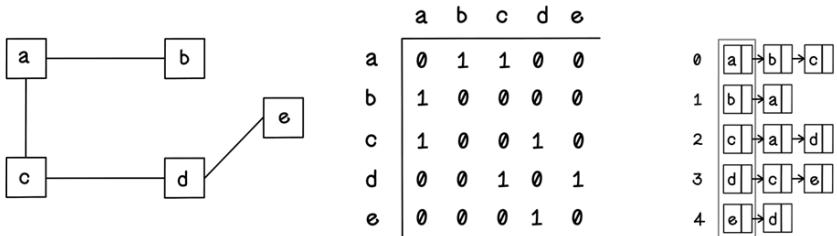


Figure 2.27 Representing a graph as an incidence matrix

### 2.10.2 Adjacency list

An *adjacency list* uses linked lists in which the size of the initial list is the number of nodes in the graph and each value represents the connected nodes for a specific node. An adjacency list can be used to represent both directed and undirected graphs (figure 2.28).



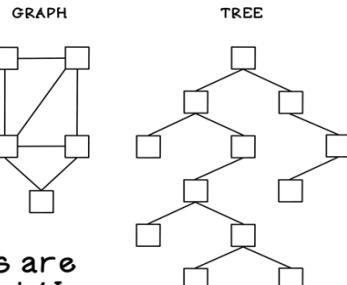
**Figure 2.28 Representing a graph as an adjacency list**

Graphs are also interesting and useful data structures because they can easily be represented as mathematical equations, which are the backing for all algorithms we use. You can find more information about this topic throughout the book.

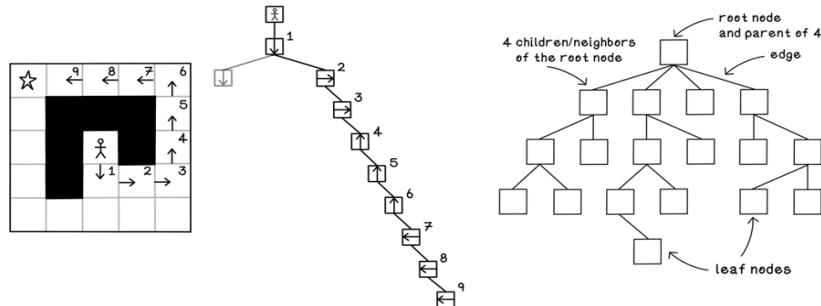
## 2.11 Summary of search fundamentals

Data structures are important in every AI algorithm

Search algorithms are useful in finding solutions where there are many options, and a solution lies somewhere in that tree

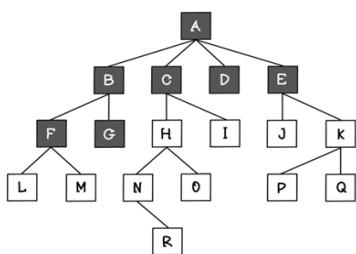


Graphs and tree structures are widely used in algorithms and AI

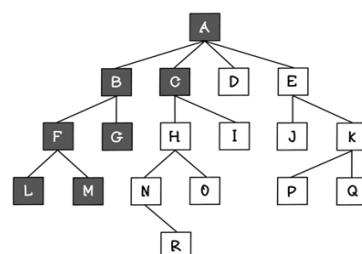


Uninformed search is blind and can be computationally expensive - using the correct data structure helps

Breadth-first search looks wide before looking deep



Depth-first search looks deep before looking wide



# 3 *Intelligent search*

## This chapter covers

- Understanding and designing heuristics for guided search
- Identifying problems suited to being solved with guided search approaches
- Understanding and designing a guided search algorithm
- Designing a search algorithm to play a two-player game

## 3.1 Defining heuristics: Designing educated guesses

Now that we have an idea of how uninformed search algorithms work from chapter 2, we can explore how they can be improved by seeing more information about the problem. For this, we use informed search. *Informed search* means that the algorithm has some context of the specific problem being solved, instead of blindly searching breadth-first or depth-first. Heuristics are a way to represent this information.

Often called a *rule of thumb*, a *heuristic* is a rule or set of rules used to evaluate a state. It can be used to define criteria that a state must satisfy or measure the performance of a specific state. A heuristic is used when a clear method of finding an optimal solution is not possible. A heuristic can be interpreted as an educated guess and should be seen more as a guideline rather than as a scientific truth with respect to the problem that is being solved.

When you're ordering a pizza at a restaurant, for example, your heuristic of how good it is may be defined by the ingredients and type of base used. If you enjoy extra tomato sauce, extra cheese, mushrooms, and pineapple on a thick base with crunchy crust, a pizza that includes more of these will be more appealing to you and achieve a better score for your heuristic. A pizza that contains fewer of those attributes will be less appealing and achieve a poorer score.

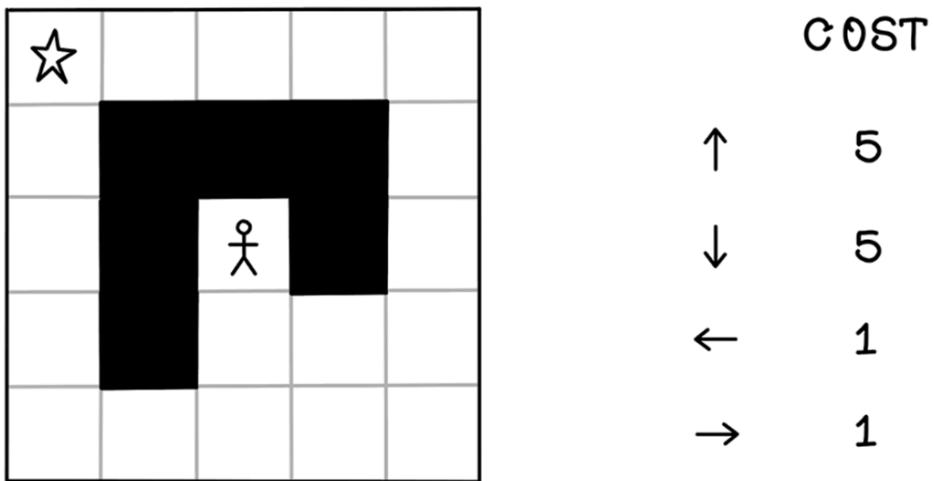
Another example is a GPS routing problem. The heuristic may be “Good paths minimize time in traffic and minimize distance traveled” or “Good paths minimize toll fees and maximize good road conditions.” A poor heuristic would be to minimize straight-line distance between two points. This heuristic might work for birds or planes, but for everyday commuting, we use roads, which rarely offer a straight-line between two destinations.

Let’s look at an example of checking whether an uploaded file is an audio clip in a library of copyrighted content. Because audio clips are frequencies of sound, one way to achieve this is to search every time-slice of the uploaded clip with every clip in the library. This task will require an extreme amount of computation. A primitive start to building a better search could be defining a heuristic that minimizes the difference in distribution of frequencies between the two clips, as shown in figure 3.1. Notice that the frequencies are identical but shifted slightly. Given these distributions we can be pretty certain that it is a copyrighted clip. This solution may not be perfect, but it is a good start toward a less-expensive algorithm.



**Figure 3.1 Comparison of two audio clips using frequency distribution**

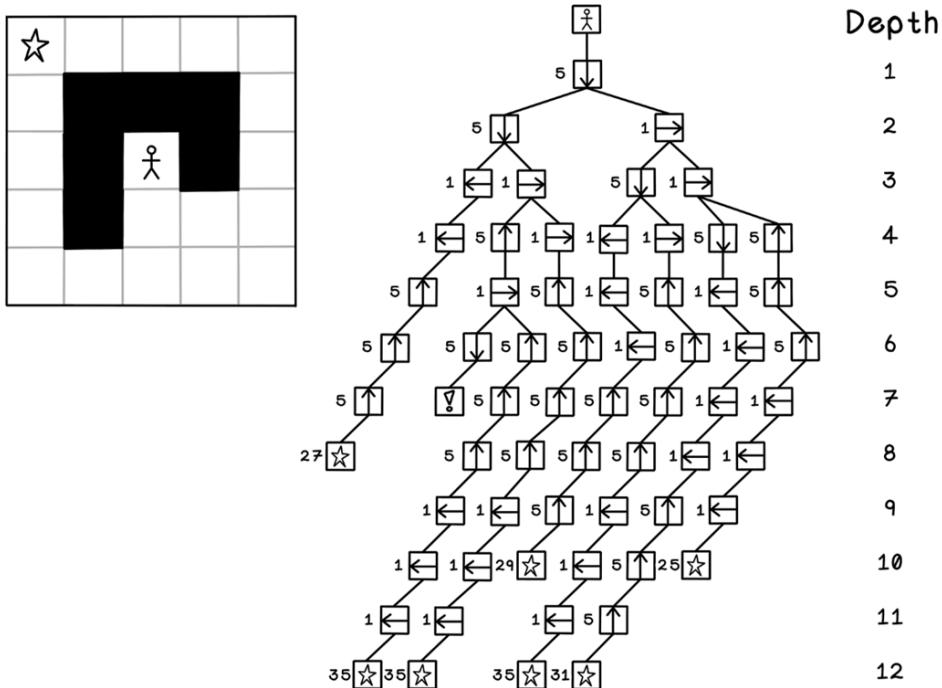
Heuristics are problem-specific, and a good heuristic can help optimize solutions substantially. The maze scenario from chapter 2 will be adjusted to demonstrate the concept of creating heuristics by introducing an interesting dynamic. Instead of treating all movements the same way and measuring better solutions purely by paths with fewer actions (shallow depth in the tree), movements in different directions now cost different amounts. For the sake of this example, imagine there’s been some strange shift in the gravity of our maze, and moving north or south now costs five times as much as moving east or west (figure 3.2).



**Figure 3.2 Adjustments to the maze example: gravity**

In the adjusted maze scenario, the factors influencing the best possible path to the goal are the number of actions taken and the sum of the cost for each action in a respective path.

In figure 3.3, all possible paths in the tree are represented to highlight the options available, indicating the costs of the respective actions. Again, this example demonstrates the search space in the trivial maze scenario and does not often apply to real-life scenarios. The algorithm will be generating the tree as part of the search.



**Figure 3.3 All possible movement options represented as a tree**

A heuristic for the maze problem can be defined as follows: “Good paths minimize cost of movement and minimize total moves to reach the goal.” This simple heuristic helps guide which nodes are visited because we are applying some domain knowledge to solve the problem.

**THOUGHT EXPERIMENT: GIVEN THE FOLLOWING SCENARIO, WHAT HEURISTIC CAN YOU IMAGINE?**

Several miners specialize in different types of mining, including diamond, gold, and platinum. All the miners are productive in any mine, but they mine faster in mines that align with their specialties. Several mines that can contain diamonds, gold, and platinum are spread across an area, and depots appear at different distances between mines. If the problem is to distribute miners to maximize their efficiency and reduce travel time, what could a heuristic be?

#### THOUGHT EXPERIMENT: POSSIBLE SOLUTION

A sensible heuristic would include assigning each miner to a mine of their specialty and tasking them with traveling to the depot closest to that mine. This can also be interpreted as minimizing assigning miners to mines that are not their specialty and minimizing the distance traveled to depots.

## 3.2 Informed search: Looking for solutions with guidance

*Informed search*, also known as *heuristic search*, is an algorithm that uses both breadth-first search and depth-first search approaches combined with some intelligence. The search is guided by heuristics, given some knowledge of the problem at hand.

We can employ several informed search algorithms, depending on the nature of the problem, including Greedy Search (also known as Best-first Search). The most popular and useful informed search algorithm, however, is A\*.

### 3.2.1 A\* search

A\* search is pronounced “A star search.” The A\* algorithm usually improves performance by estimating heuristics to minimize the cost of the next node visited.

Total cost is calculated with two metrics: the total distance from the start node to the current node and the estimated cost of moving to a specific node by using a heuristic. When we are attempting to minimize cost, a lower value indicates a better-performing solution (figure 3.4).

$$f(n) = g(n) + h(n)$$

**g(n): is the Past:** The known cost. The effort you already spent to travel from the start to the current node.

**h(n): is the Future:** The estimated cost. The "guess" of how much effort remains to reach the goal.

**f(n): is the Total:** The combination of history and prediction.

**Figure 3.4 The function for the A\* search algorithm**

A\* is famous because it guarantees finding the absolute shortest path. However, this guarantee only holds true if your heuristic follows a strict rule: It must never overestimate the cost to the goal.

In technical terms, the heuristic must be *Admissible* and *Consistent*.

- *Admissible:* The estimated cost must be less than or equal to the actual cost. Think of it as being "optimistic". If the heuristic thinks the goal is 10 miles away, but it's actually 5 miles, A\* might mistakenly ignore the best path.
- *Consistent:* If you are searching a graph (which allows loops), the heuristic also needs to be consistent (meaning the estimated cost shouldn't drop erratically as you get closer).

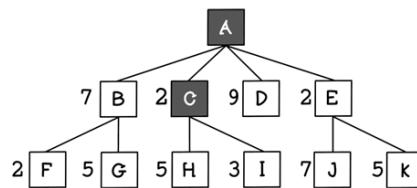
The following example of processing is an abstract example of how a tree is visited using heuristics to guide the search. The focus is on the heuristic calculations for the different nodes in the tree.

Breadth-first search visits all nodes on each depth before moving to the next depth. Depth-first search visits all nodes down to the final depth before traversing back to the root and visiting the next path.

A\* search is different in that it does not have a predefined pattern to follow; nodes are visited in the order based on their heuristic costs. To achieve this, the A\* algorithm uses a *Priority Queue* to manage the nodes it needs to explore.

Unlike a stack (last in, first out—LIFO) or a standard queue (first in first out—FIFO), a *Priority Queue* automatically keeps the “most promising” node at the front. This ensures that the algorithm always processes the node with the lowest estimated total cost next. Note that the algorithm does not know the costs of all nodes up front. Costs are calculated as the tree is explored or generated, and each node visited is added to this queue, which means nodes that cost more than nodes already visited are ignored, saving computation time (figures 3.5, 3.6, and 3.7).

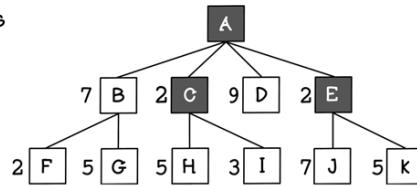
Given a tree representing nodes and their heuristic scores, A\* will visit the first child with the lowest cost. In this case, it is C, with a cost of 2.



Sequence of processing the priority queue  
 ↘  
 C 2  
 A

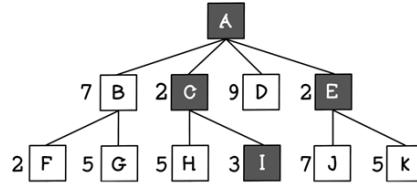
When two nodes cost the same, the node whose score was calculated first is selected.

Because E also has a cost of 2 and is a child of A, it will be the next node visited.



↗  
 E 2  
 C 2  
 A

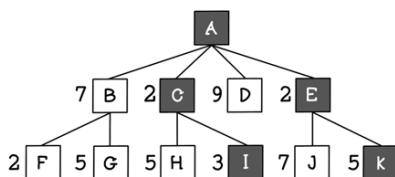
Then A\* will visit the lowest-cost node from children of A or children of nodes it has already visited.



↗  
 I 3  
 E 2  
 C 2  
 A

Figure 3.5 The sequence of tree processing using A\* search (part 1)

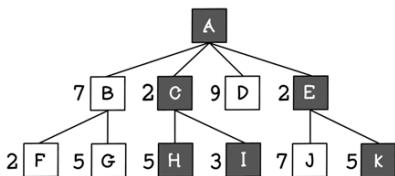
The next lowest-cost node is K, a child of E.



Sequence of processing the priority queue

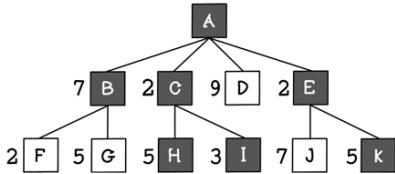
K	5
I	3
E	2
C	2
A	

The next lowest-cost node is H, a child of C.



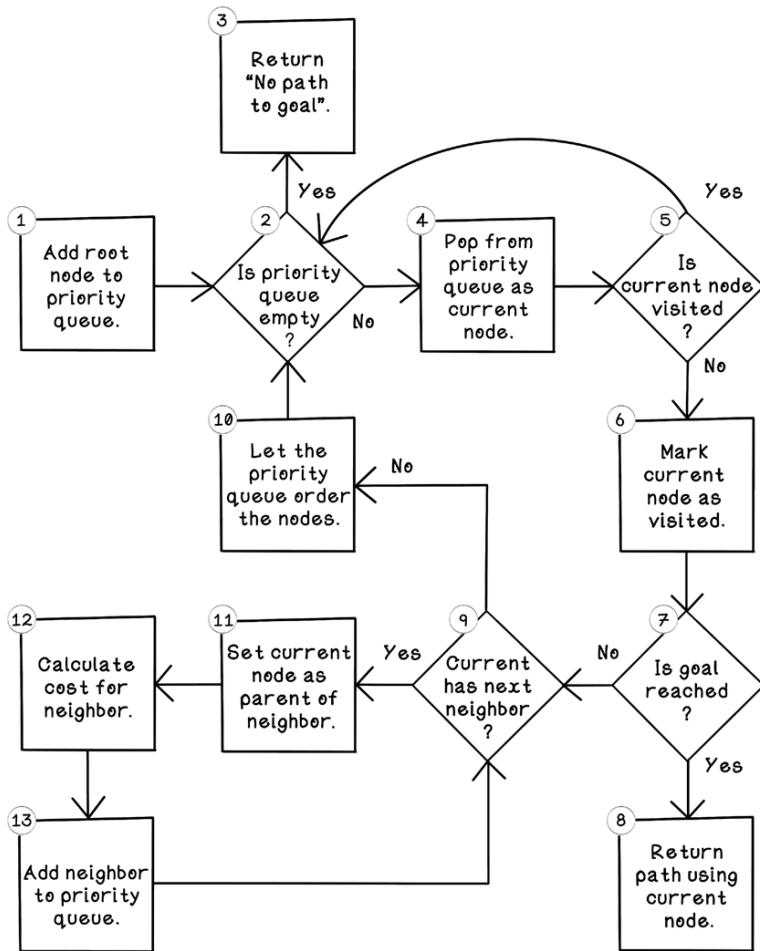
H	5
K	5
I	3
E	2
C	2
A	

A direct child of A is visited, because it has the lowest cost of children of A and children of all other nodes visited.



B	7
H	5
K	5
I	3
E	2
C	2
A	

Figure 3.6 The sequence of tree processing using A\* search (part 2)



**Figure 3.7 Flow for the A\* search algorithm**

Let's walk through the flow of the A\* search algorithm:

1. *Add root node to priority queue*—The A\* algorithm uses a Priority Queue (often implemented as a min-heap) to ensure we always explore the most promising path first. The first step is adding the root node to this queue.
2. *Is priority queue empty?*—If the queue is empty, and no path has been returned in step 8 of the algorithm, there is no path to the goal. If there are still nodes in the queue, the algorithm can continue its search.
3. *Return No path to goal*—This step is the one possible exit from the algorithm if no path to the goal exists.

4. *Pop from priority queue as current node*—By pulling the lowest-cost object from the queue and setting it as the current node of interest, we can explore its possibilities.
5. *Is current node visited?*—If the current node has not been visited, it hasn't been explored yet and can be processed now.
6. *Mark current node as visited*—This step indicates that this node has been visited to prevent unnecessary repeat processing.
7. *Is goal reached?*—This step determines whether the current neighbor contains the goal that the algorithm is searching for.
8. *Return path using current node*—By referencing the parent of the current node, then the parent of that node, and so on, the path from the goal to the root is described. The root node will be a node without a parent.
9. *Current has next neighbor?*—If the current node has more possible moves to make in the maze example, that move can be added to be processed. Otherwise, the algorithm can jump to step 2, in which the next object in the priority queue can be processed if it is not empty. The nature of the priority queue allows the algorithm to switch between branches. If a neighbor is added that is cheaper than the current path, the algorithm effectively “jumps” to explore that more promising node next.
10. *Let the priority queue order the nodes*—because we are using a Priority Queue, the node with the lowest total cost effectively floats to the top. We don't need to manually sort; the structure ensures the next node we pop is always the best candidate.
11. *Set current node as parent of neighbor*—Set the origin node as the parent of the current neighbor. This step is important for tracing the path from the current neighbor to the root node. From a map perspective, the origin is the position that the player moved from, and the current neighbor is the position that the player moved to.
12. *Calculate cost for neighbor*—The cost function is critical for guiding the A\* algorithm. The cost is calculated by summing the distance from the root node with the heuristic score for the next move. More-intelligent heuristics will directly influence the A\* algorithm for better performance.
13. *Add neighbor to priority queue*—The neighbor node is added to the priority queue for its children to be explored later.

Unlike depth-first search, the order of child nodes is determined by their estimated total cost. If two nodes have the same cost, the first node added is usually visited first. Figures 3.8, 3.9, and 3.10 illustrate an example of the sequence of processing the tree.

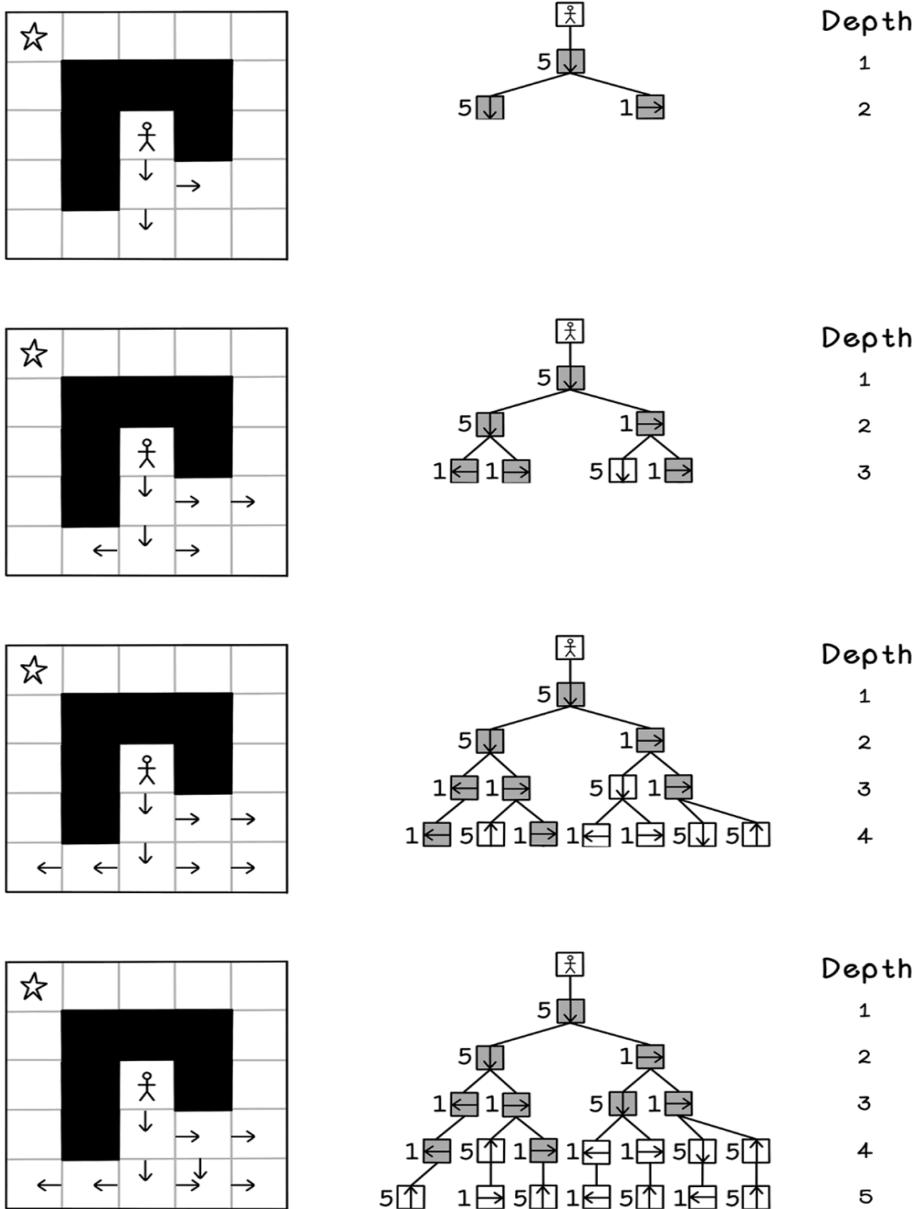
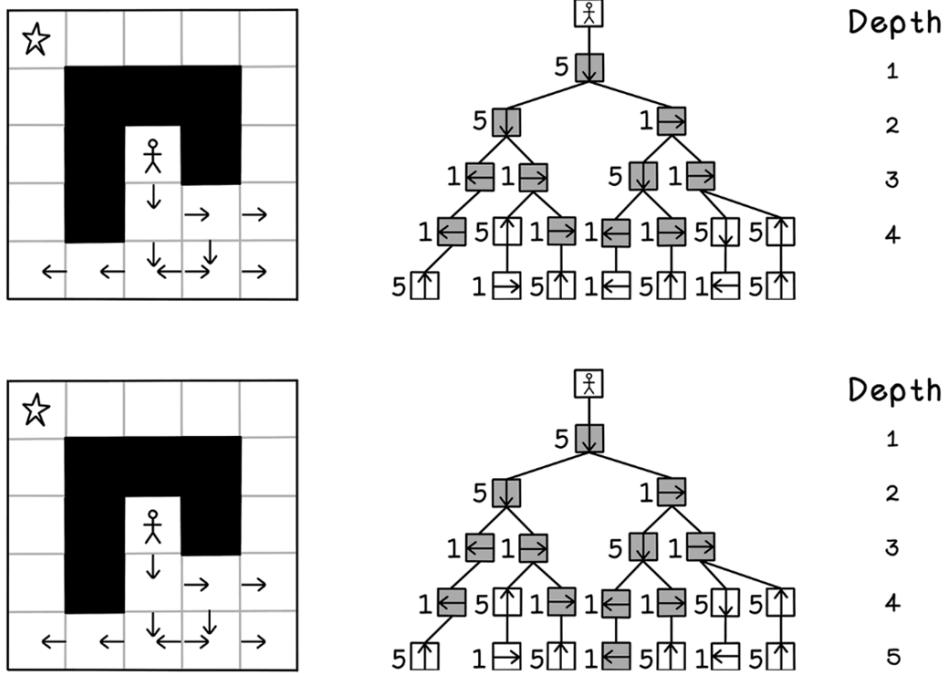


Figure 3.8 The sequence of tree processing using A\* search (part 1)



**Figure 3.9 The sequence of tree processing using A\* search (part 2)**

Notice in figure 3.8 and figure 3.9 how the A\* algorithm visits nodes both “breadth-wise” and “depth-wise” to find a solution.

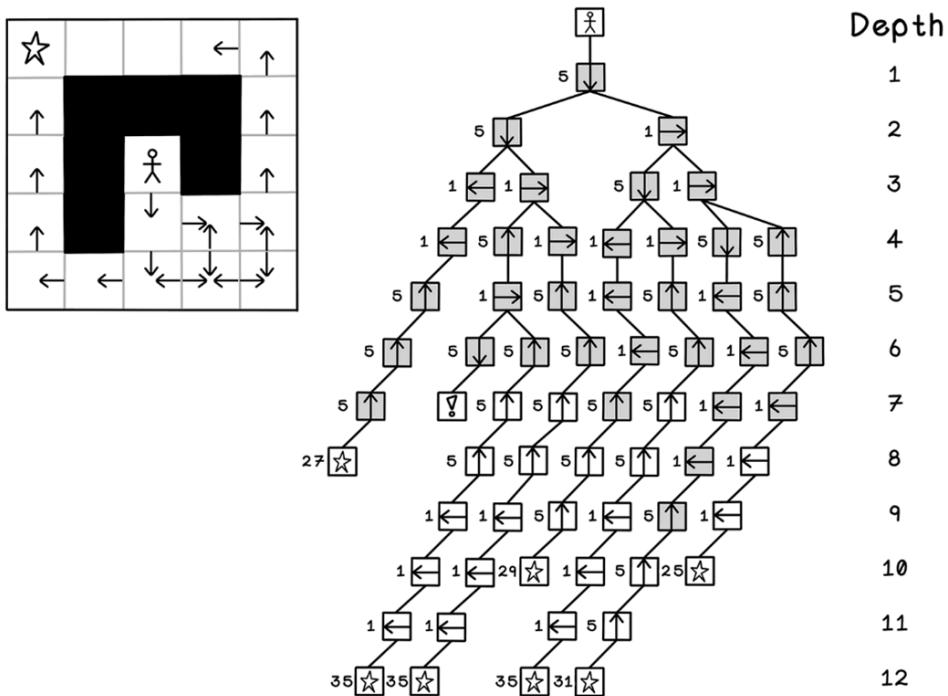


Figure 3.10 Nodes visited in the entire tree after A\* search

In figure 3.10, notice that there are several paths to the goal, but the A\* algorithm finds a path to the goal while minimizing the cost to achieve it, with fewer moves and cheaper move costs based on north and south moves being more expensive (because of the “gravity” rule that we added to this scenario).

### PYTHON CODE SAMPLE

The A\* algorithm uses a Priority Queue (often referred to as the “open set”) to manage the nodes it needs to explore. A priority queue automatically keeps the “most promising” node at the front. This ensures that the algorithm always processes the node with the lowest estimated total cost ( $f$ ) next.

```

def run_astar(maze_game, start_point):
    visited_points = []
    goal_point = find_goal_point(maze_game)
    priority_queue = []
    tie = itertools.count()
    start_point.cost = 0
    heapq.heappush(priority_queue, (start_point.cost, next(tie), start_point))

    while priority_queue:
        _, _, next_point = heapq.heappop(priority_queue)

        if is_in_visited_points(next_point, visited_points):
            continue

        visited_points.append(next_point)

        if maze_game.get_current_point_value(next_point) == mp.MazePuzzle.GOAL:
            return next_point

        for neighbor in maze_game.get_neighbors(next_point):
            neighbor.set_parent(next_point)
            neighbor.cost = determine_cost(next_point, neighbor, goal_point)
            heapq.heappush(priority_queue, (neighbor.cost, next(tie), neighbor))

    return "No path to the goal found"

```

The functions for calculating the cost are critical. For A\* to work, we calculate two values: the actual cost so far ( $g$ ) and the heuristic estimate ( $h$ ).The following functions describe how these costs are derived. In our maze example, we use a “Manhattan Distance” heuristic (measuring total grid steps) to estimate the remaining distance.

```

def determine_cost(origin, target, goal_point):
    # g: distance traveled (edges)
    g_cost = mp.get_path_length(target)
    # movement cost for this step (captures gravity penalty)
    move_cost = mp.get_move_cost(origin, target)
    # h: Manhattan distance to the goal
    h_cost = abs(target.x - goal_point.x) + abs(target.y - goal_point.y)
    return g_cost + move_cost + h_cost

```

Uninformed search algorithms such as Breadth-First Search (BFS) explore layers exhaustively and result in the optimal solution (in unweighted graphs). Depth-First Search (DFS), while useful, does not guarantee the shortest path.

A\* search is a superior approach when a sensible heuristic can be created. It computes more efficiently than uninformed algorithms because it ignores nodes with high estimated costs. However, if the heuristic is flawed or “inadmissible” (over-pessimistic), A\* loses its guarantee of finding the optimal solution.

### 3.2.2 Use cases for informed search algorithms

Informed search algorithms are versatile and useful for many real-world use cases in which heuristics can be defined, such as the following:

- *Path finding for autonomous game characters in video games*—Game developers often use this algorithm to control the movement of enemy units in a game in which the goal is to find the human player within an environment.
- *Text Generation and Decoding in NLP* — When AI models (like chatbots) write sentences, they don't just pick the single most likely next word. They often use heuristic search (like *Beam Search*) to look ahead at multiple possible “branches” of a sentence to find the sequence of words that makes the most sense grammatically and logically.
- *Telecommunications network routing*—Guided search algorithms can be used to find the shortest paths for network traffic in telecommunications networks to improve performance. Servers/network nodes and connections can be represented as searchable graphs of nodes and edges.
- *Single-player games and puzzles*—Informed search algorithms can be used to solve single-player games and puzzles such as the Rubik's Cube, because each move is a decision in a tree of possibilities until the goal state is found.

## 3.3 Adversarial search: Looking for solutions in a changing environment

The search example of the maze game involves a single actor: the “player”. The environment is affected only by the single player; so, that player generates all possibilities. The goal until now was to maximize the benefit for the player: choosing paths to the goal with the shortest distance and cost.

*Adversarial search* is characterized by opposition or conflict. Adversarial problems require us to anticipate, understand, and counteract the actions of the opponent in pursuit of a goal. Examples of adversarial problems include two-player turn-based games such as Tic-Tac-Toe and Connect Four. The players take turns for the opportunity to change the state of the environment of the game to their favor. A set of rules dictates how the environment may be changed and what the winning and end states are.

These are often called Zero-Sum Games: for one player to win (+10), the other must lose (-10). The total sum of the outcome is always zero.

### 3.3.1 A simple adversarial problem

This section uses the game of Connect Four to explore adversarial problems. Connect Four (figure 3.11) is a game consisting of a grid in which players take turns dropping tokens into one of the columns. The tokens in a specific column pile up, and any player who manages to create four adjacent sequences of their tokens—vertically, horizontally, or diagonally—wins. If the grid is full, with no winner, the game results in a draw.

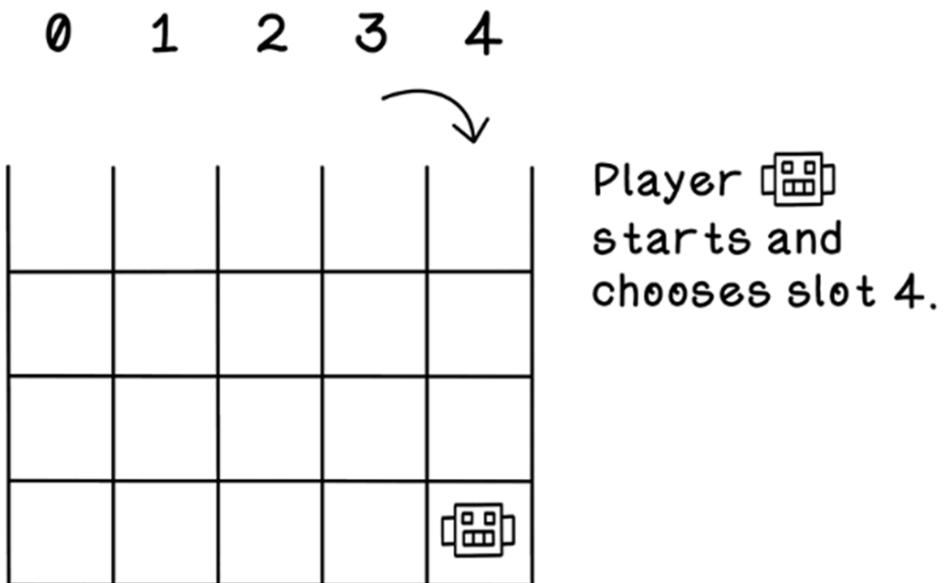


Figure 3.11 The game of Connect Four

### 3.3.2 Min-max search: Simulate actions and choose the best future

*Min-max search* aims to build a tree of possible outcomes based on moves that each player could make and favor paths that are advantageous to the agent while avoiding paths that are favorable to the opponent. To do so, this type of search simulates possible moves assuming that both players play perfectly (the opponent will always choose the move that hurts the agent most) and scores the state based on a heuristic after making the respective move. Min-max search attempts to discover as many states in the future as possible; but due to memory and computation limitations, discovering the entire game tree may not be realistic, so it searches to a specified depth. Min-max search simulates the turns taken by each player, so the depth specified is directly linked to the number of turns between both players. A depth of 4, for example, means that each player has had 2 turns. Player A makes a move, player B makes a move, player A makes another move, and Player B makes another move.

## HEURISTICS

The min-max algorithm uses a heuristic score to make decisions. This score is defined by a crafted heuristic and is not automatically learned by the algorithm. If we have a specific game state, every possible valid outcome of a move from that state will be a child node in the game tree.

Assume that we have a heuristic that provides a score in which positive numbers are better than negative numbers. By simulating every possible valid move, the min-max search algorithm tries to minimize making moves where the opponent will have an advantage or a winning state and maximize making moves that give the agent an advantage or a winning state.

Figure 3.12 illustrates a min-max search tree. In this figure, the leaf nodes are the only nodes where the heuristic score is calculated, since these states indicate a winner or a draw. The other nodes in the tree indicate states that are in progress. Starting at the depth where the heuristic is calculated and moving upward, either the child with the minimum score or the child with the maximum score is chosen, depending on whose turn is next in the future simulated states. Starting at the top, the agent attempts to maximize its score; and after each alternating turn, the intention changes, because the aim is to maximize the score for the agent and minimize the score for the opponent.

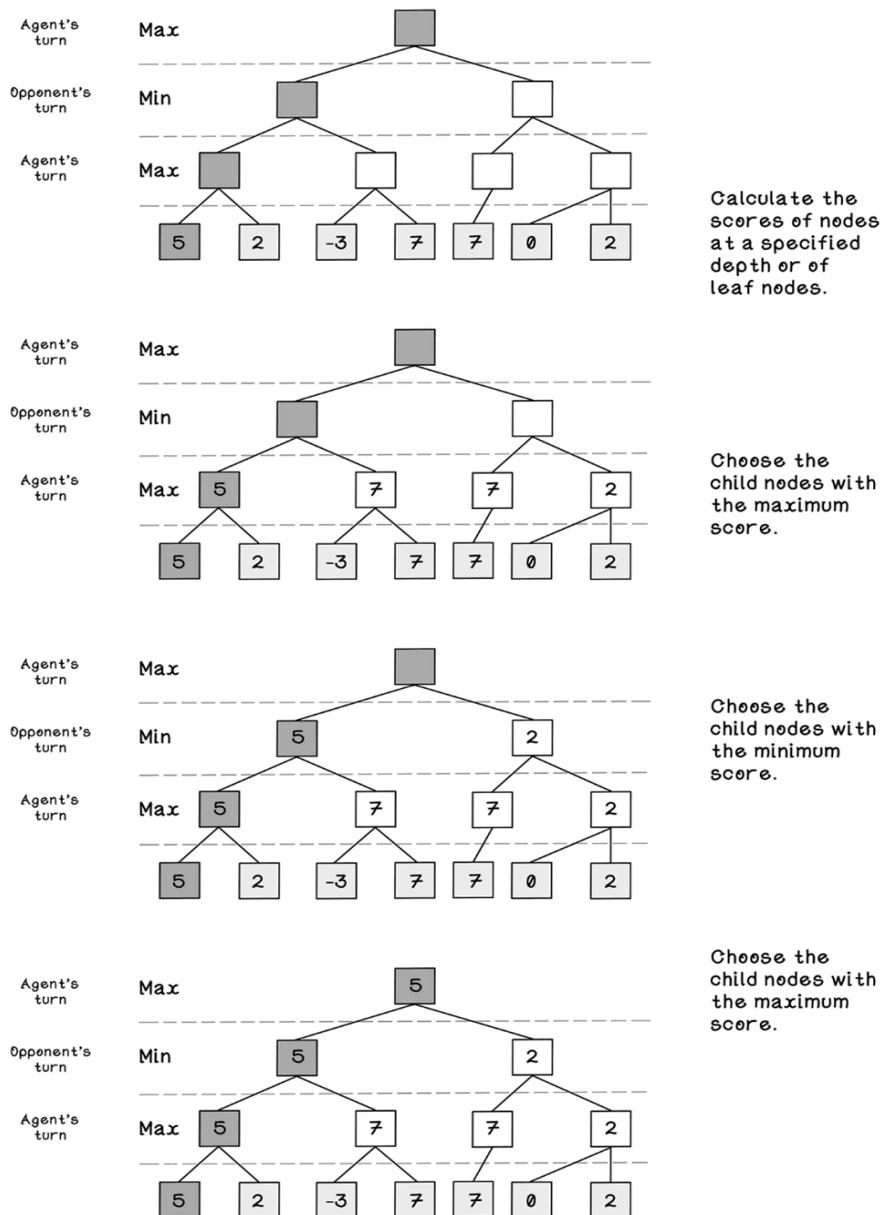
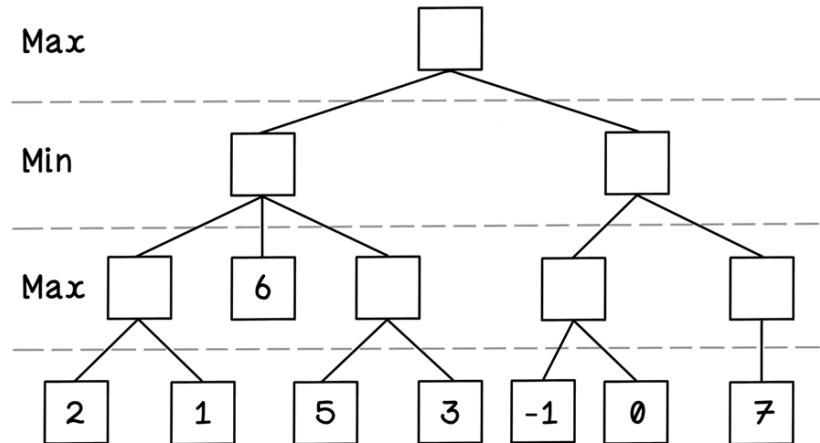
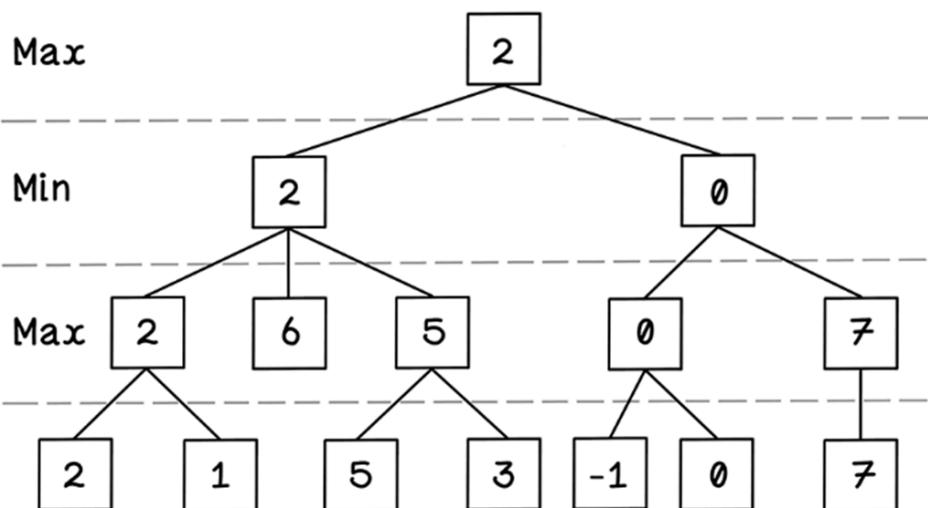


Figure 3.12 The sequence of tree processing using min-max search

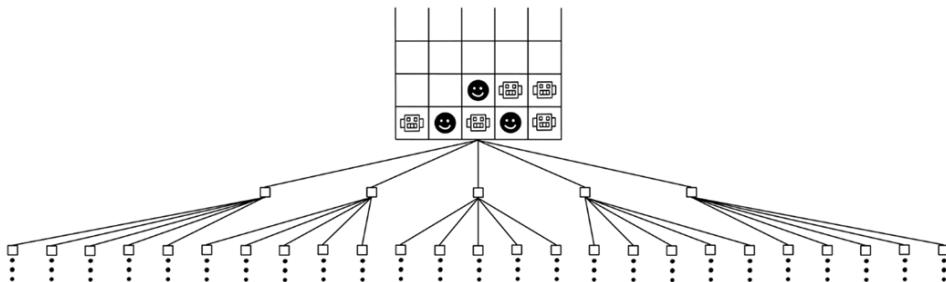
EXERCISE: WHAT VALUES WOULD PROPAGATE IN THE FOLLOWING MIN-MAX TREE?



SOLUTION: WHAT VALUES WOULD PROPAGATE IN THE FOLLOWING MIN-MAX TREE?

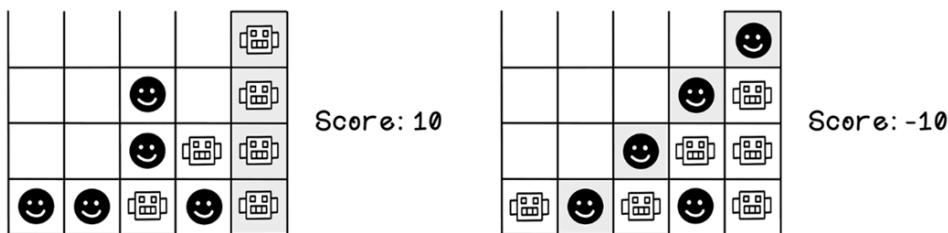


Because the min-max search algorithm simulates possible outcomes, in games that offer a multitude of choices, the game tree explodes, and it quickly becomes too computationally expensive to explore the entire tree. In the simple example of Connect Four played on a 5 x 4 block board, the number of possibilities already makes exploring the entire game tree on every turn inefficient (figure 3.13).

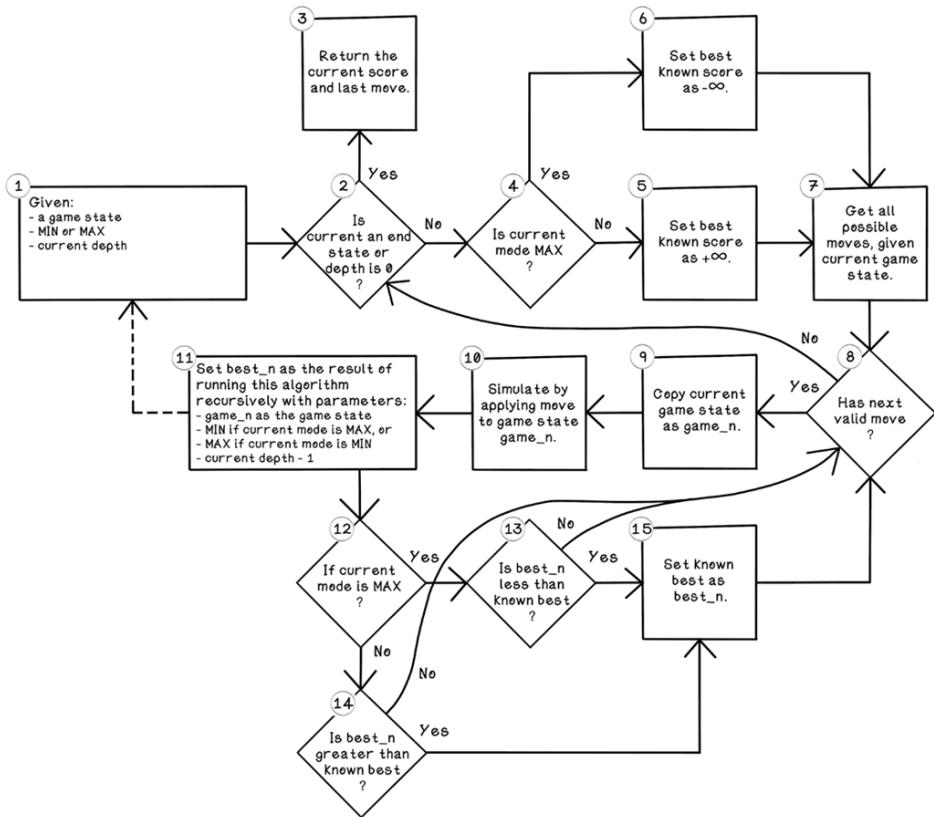


**Figure 3.13 The explosion of possibilities while searching the game tree**

To use min-max search in the Connect Four example, the algorithm essentially makes all possible moves from a current game state; then it determines all possible moves from each of those states until it finds the path that is most favorable. Game states that result in a win for the agent return a score of 10, and states that result in a win for the opponent return a score of -10. Min-max search tries to maximize the positive score for the agent (figures 3.14 and 3.15).



**Figure 3.14 Scoring for the agent versus scoring for the opponent**



**Figure 3.15 Flow for the min-max search algorithm**

Although the flow chart for the min-max search algorithm looks complex due to its size, it really isn't. The number of conditions that check whether the current state is to maximize or minimize causes the chart to bloat.

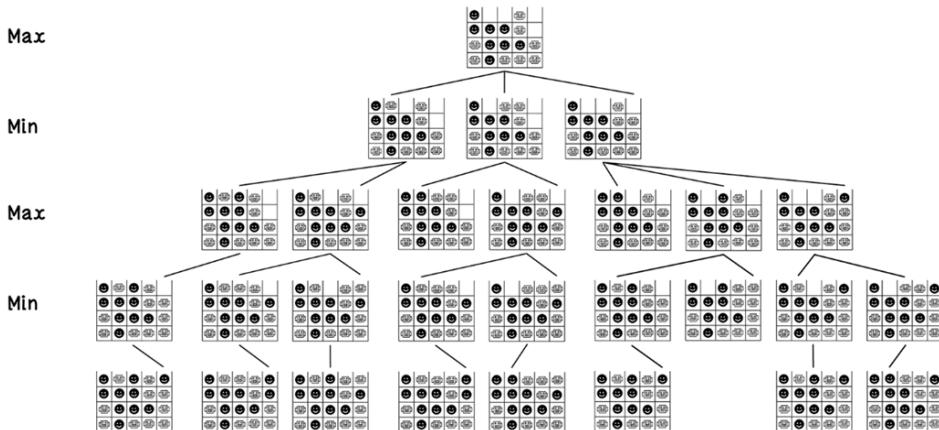
Let's walk through the flow of the min-max search algorithm:

1. *Given a game state, whether the current mode is minimization or maximization, and a current depth, the algorithm can start*—It is important to understand the inputs for the algorithm, as the min-max search algorithm is recursive. A recursive algorithm calls itself in one or more of its steps. It is important for a recursive algorithm to have an exit condition to prevent it from calling itself forever.

2. *Is current an end state or depth is 0?*—This condition determines whether the current state of the game is a terminal state or whether the desired depth has been reached. A terminal state is one in which one of the players has won or the game is a draw. A score of 10 represents a win for the agent, and a score of -10 represents a win for the opponent, and a score of 0 indicates a draw. A depth is specified, because traversing the entire tree of possibilities to all end states is computationally expensive and will likely take too long on the average computer. By specifying a depth, the algorithm can look a few turns into the future to determine whether a terminal state exists.
3. *Return the current score and last move*—The score for the current state is returned if the current state is a terminal game state or if the specified depth has been reached.
4. *Is current mode MAX?*—If the current iteration of the algorithm is in the maximize state, it tries to maximize the score for the agent.
5. *Set best known score as  $+\infty$* —If the current mode is to minimize the score, the best score is set to positive infinity, because we know that the scores returned by the game states will always be less. In actual implementation, a really large number is used rather than infinity.
6. *Set best known score as  $-\infty$* —If the current mode is to maximize the score, the best score is set to negative infinity, because we know that the scores returned by the game states will always be more. In actual implementation, a really large negative number is used rather than infinity.
7. *Get all possible moves, given current game state*—This step specifies a list of possible moves that can be made, given the current game state. As the game progresses, not all moves available at the start may be available anymore. In the Connect Four example, a column may be filled; therefore, a move selecting that column is invalid.
8. *Has next valid move?*—If any possible moves have not been simulated yet and there are no more valid moves to make, the algorithm short-circuits to returning the best move in that instance of the function call.
9. *Copy current game state as game\_n*—A copy of the current game state is required to perform simulations of possible future moves on it.
10. *Simulate by applying move to game state game\_n*—This step applies the current move of interest to the copied game state.
11. *Set best\_n as the result of running this algorithm recursively*—Here's where recursion comes into play. *best\_n* is a variable used to store the next best move, and we're making the algorithm explore future possibilities from this move.
12. *If current mode is MAX?*—When the recursive call returns a best candidate, this condition determines whether the current mode is to maximize the score.

13. *Is best\_n less than known best?*—This step determines whether the algorithm has found a better score than one previously found if the mode is to maximize the score.
14. *Is best\_n greater than known best?*—This step determines whether the algorithm has found a better score than one previously found if the mode is to minimize the score.
15. *Set known best as best\_n*—If the new best score is found, set the known best as that score.

Given the Connect Four example at a specific state, the min-max search algorithm generates the tree shown in figure 3.16. From the start state, every possible move is explored. Then each move from that state is explored until a terminal state is found—either the board is full or a player has won.



**Figure 3.16 A representation of the possible states in a Connect Four game**

The highlighted nodes in figure 3.17 are terminal state nodes in which draws are scored as 0, losses are scored as -10, and wins are scored as 10. Because the algorithm aims to maximize its score, a positive number is required, whereas opponent wins are scored with a negative number.

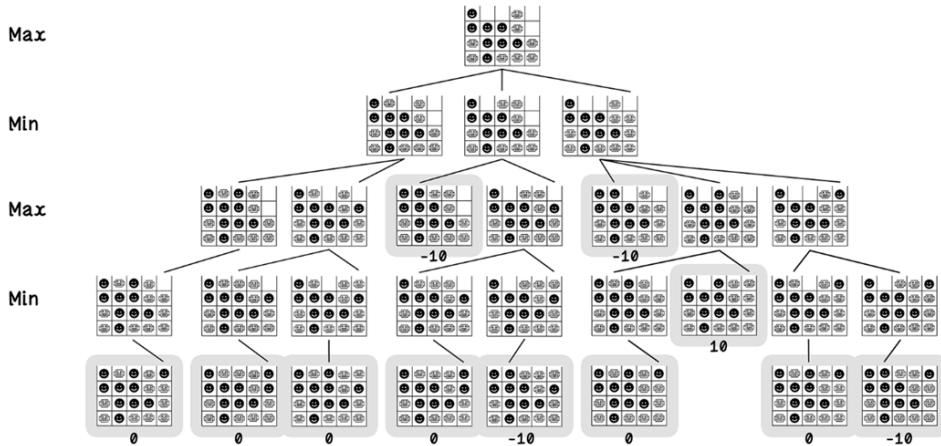


Figure 3.17 The possible end states in a Connect Four game

When these scores are known, the min-max algorithm starts at the lowest depth and chooses the node whose score is the minimum value (figure 3.18).

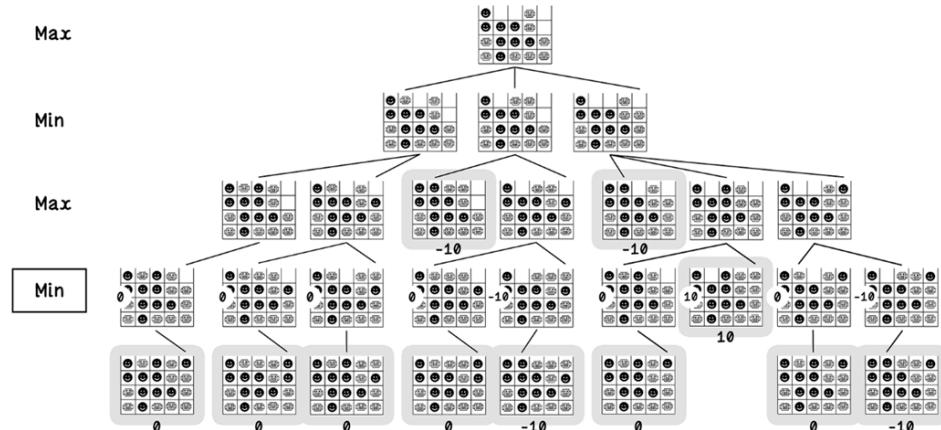
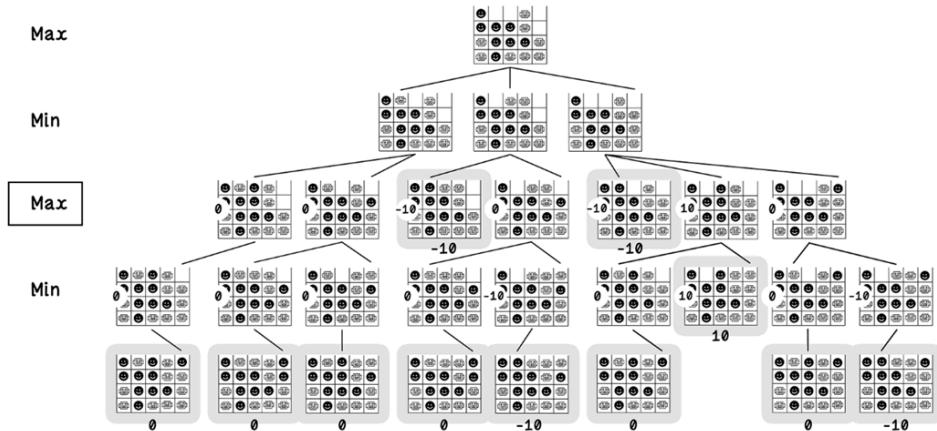


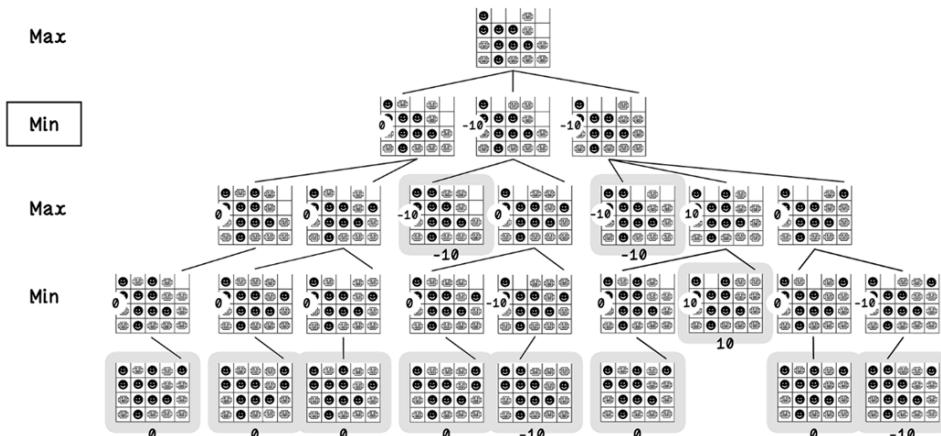
Figure 3.18 The possible scores for end states in a Connect Four game (part 1)

Then, at the next depth, the algorithm chooses the node whose score is the maximum value (figure 3.19).



**Figure 3.19 The possible scores for end states in a Connect Four game (part 2)**

Finally, at the next depth, nodes whose score is the minimum are chosen, and the root node chooses the maximum of the options. By following the nodes and score selected and intuitively applying ourselves to the problem, we see that the algorithm selects a path to a draw to avoid a loss. If the algorithm selects the path to the win, there is a high likelihood of a loss in the next turn. The algorithm assumes that the opponent will always make the smartest move to maximize their chance of winning (figure 3.20).



**Figure 3.20 The possible scores for end states in a Connect Four game (part 3)**

The simplified tree in figure 3.21 represents the outcome of the min-max search algorithm for the given game state example.

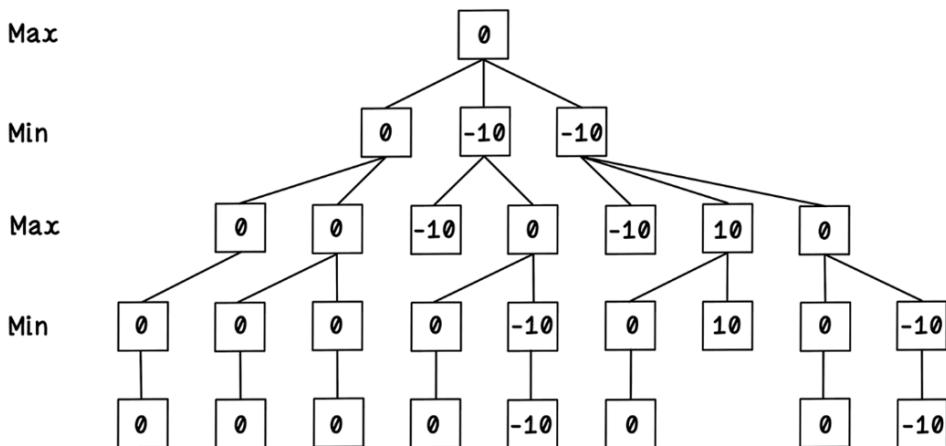


Figure 3.21 Simplified game tree with min-max scoring

### PYTHON CODE SAMPLE

The min-max search algorithm is implemented to be a recursive function. The function is provided with the current state, desired depth to search, minimization or maximization mode, and last move. The algorithm terminates by returning the best move and score for every child at every depth in the tree. Comparing the code with the flow chart in figure 3.15, we notice that the tedious conditions of checking whether the current mode is maximizing or minimizing are not as apparent. In the code, 1 or -1 represents the intention to maximize or minimize, respectively. By using some clever logic, the best score, conditions, and switching states can be done via the principle of negative multiplication, in which a negative number multiplied by another negative number results in a positive. So if -1 indicates the opponent's turn, multiplying it by -1 results in 1, which indicates the agent's turn. Then, for the next turn, 1 multiplied by -1 results in -1 to indicate the opponent's turn again:

```

def minmax(connect, depth, min_or_max, move):
    current_score = connect.get_score_for_ai()
    current_is_board_full = connect.is_board_full()
    if current_score != 0 or current_is_board_full or depth == 0:
        return Move(move, current_score)

    best_score = INFINITY_NEGATIVE * min_or_max
    best_max_move = -1
    moves = random.sample(range(0, connect.board_size_y + 1),
    connect.board_size_x)
    for slot in moves:
        neighbor = copy.deepcopy(connect)
        move_outcome = neighbor.play_move(slot)
        if move_outcome:
            best = minmax(neighbor, depth - 1, min_or_max * -1, slot) #A
            if (min_or_max == MAX and best.value > best_score) or (min_or_max == MIN and best.value < best_score): #B
                best_score = best.value #B
                best_max_move = best.move #B
    return Move(best_max_move, best_score)

#A Recursively call minmax for the next state after playing a move
#B Update the best score and best move

```

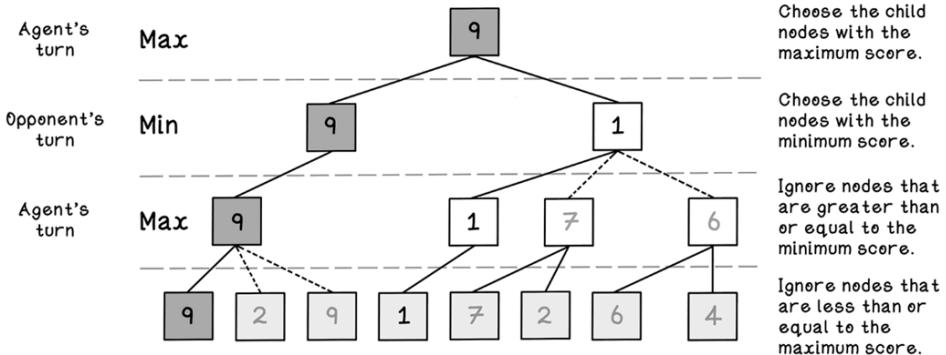
### 3.3.3 Alpha-beta pruning: Optimize by exploring the sensible paths only

*Alpha-beta pruning* is a technique used with the min-max search algorithm to short-circuit exploring areas of the game tree that are known to produce poor solutions. This technique optimizes the min-max search algorithm to save computation, because insignificant paths are ignored.

Imagine you are using a map to find the fastest route home. Route A takes 30 minutes. You then check Route B. The app tells you that the first segment of Route B involves a ferry ride that takes 45 minutes.

You stop calculating Route B immediately. You don't need to know how fast the roads are after the ferry. Since the first step alone is already longer than the entirety of Route A, Route B is guaranteed to be worse. You "prune" that route from your calculations.

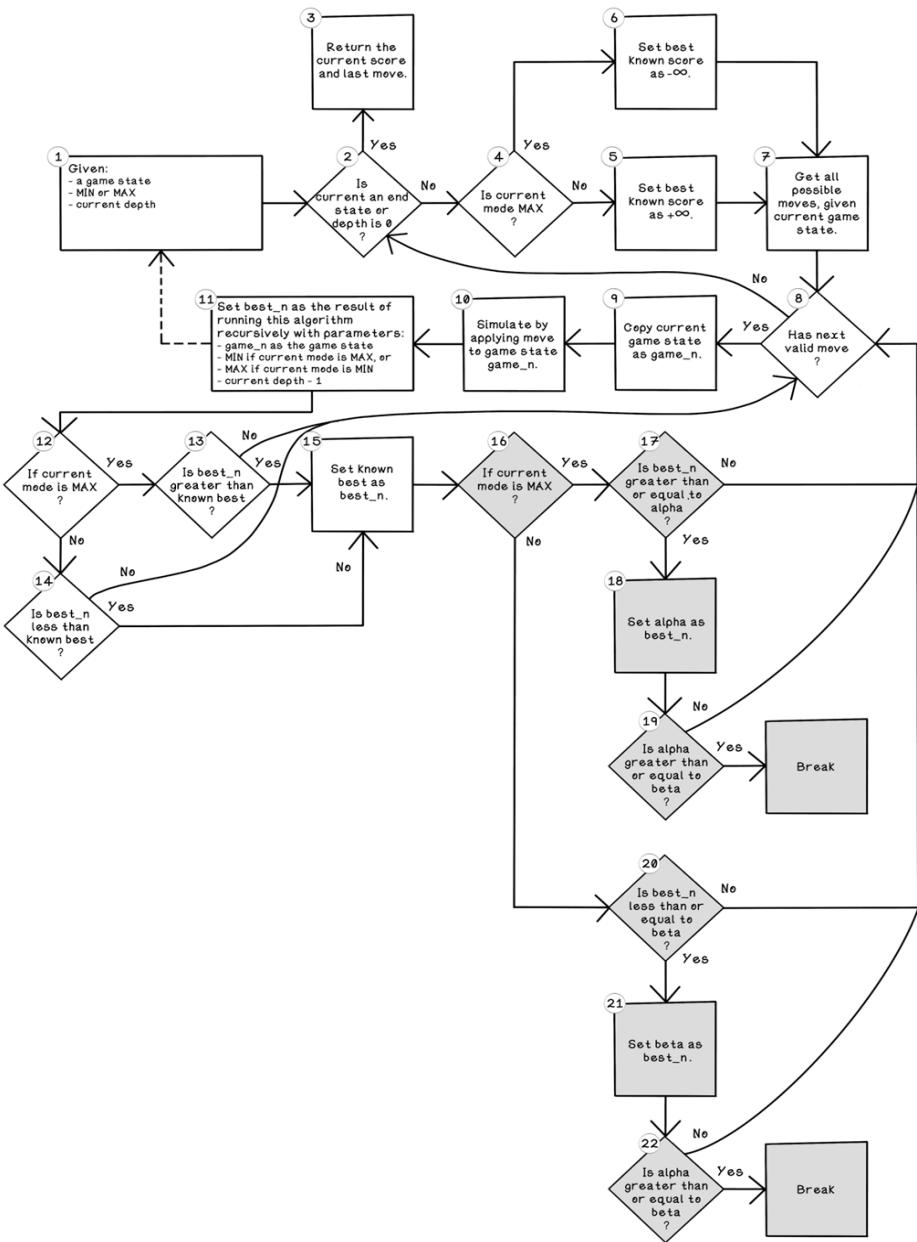
Because we know how the Connect Four example game tree explodes, we clearly see that ignoring more paths will improve performance significantly (figure 3.22).



**Figure 3.22 An example of alpha-beta pruning**

The alpha-beta pruning algorithm works by storing the best score for the maximizing player and the best score for the minimizing player as alpha and beta, respectively. Initially, alpha is set as  $-\infty$ , and beta is set as  $\infty$ —the worst score for each player. If the best score of the minimizing player is less than the best score of the maximizing player, it is logical that other child paths of the nodes already visited would not affect the best score.

Figure 3.23 illustrates the changes made in the min-max search flow to accommodate the optimization of alpha-beta pruning. The highlighted blocks are the additional steps in the min-max search algorithm flow.



**Figure 3.23 Flow for the min-max search algorithm with alpha-beta pruning**

The following steps are additions to the min-max search algorithm. These conditions allow termination of exploration of paths when the best score found will not change the outcome:

1. *Is current mode MAX?*—Again, determine whether the algorithm is currently attempting to maximize or minimize the score.
2. *Is best\_n greater than or equal to alpha?*—If the current mode is to maximize the score and the current best score (best\_n) is greater than alpha, we have found a better path than previously discovered. We update alpha to reflect this new highest value. This tightens the search window, helping us potentially prune future branches, but we do not stop exploring this node yet (unless alpha becomes greater than or equal to beta in step 4).
3. *Set alpha as best\_n.* Set the variable *alpha* as *best\_n*.
4. *Is alpha greater than or equal to beta?*—The score is as good as other scores found, and the rest of the exploration of that node can be ignored by breaking.
5. *Is best\_n less than or equal to beta?*—If the current mode is to minimize the score and the current best score (best\_n) is less than beta, we have found a stronger move for the minimizing player. We update beta to reflect this new lowest value. This tightens the search window from the top, which helps prune future branches if the Maximizer's best option (alpha) meets this new limit.
6. *Set beta as best\_n.* Set the variable *beta* as *best\_n*.
7. *Is alpha greater than or equal to beta?*—The score is as good as other scores found, and the rest of the exploration of that node can be ignored by breaking.

## PYTHON CODE SAMPLE

The code for achieving alpha-beta pruning is largely the same as the code for min-max search, with the addition of keeping track of the alpha and beta values and maintaining those values as the tree is traversed. Note that when minimum(min) is selected the variable *min\_or\_max* is -1, and when maximum(max) is selected, the variable *min\_or\_max* is 1:

```

def minmax(connect, depth, min_or_max, move, alpha, beta):
    current_score = connect.get_score_for_ai()
    current_is_board_full = connect.is_board_full()
    if current_score != 0 or current_is_board_full or depth == 0:
        return Move(move, current_score)

    best_score = INFINITY_NEGATIVE * min_or_max
    best_max_move = -1
    moves = random.sample(range(0, connect.board_size_x), connect.board_size_x)
    for slot in moves:
        neighbor = copy.deepcopy(connect)
        move_outcome = neighbor.play_move(slot)
        if move_outcome:
            # Recursively call minmax for the next state after playing a move
            best = minmax(neighbor, depth - 1, min_or_max * -1, slot, alpha,
beta)
            # Update the best score and best move, ignore irrelevant scores using
alpha beta pruning
            if (min_or_max == MAX and best.value > best_score) or (min_or_max ==
MIN and best.value < best_score):
                best_score = best.value
                best_max_move = best.move
    if min_or_max == MAX: #A
        if best_score > best_score_so_far:
            if best_score > alpha:
                alpha = best_score
    elif min_or_max == MIN: #B
        if best_score < best_score_so_far:
            if best_score < beta:
                beta = best_score
    if alpha >= beta:
        break
    return Move(best_max_move, best_score)

#A If Maximizer
#B If Minimizer

```

### 3.3.4 Use cases for adversarial search algorithms

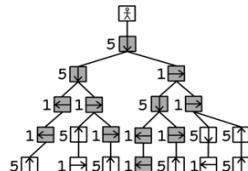
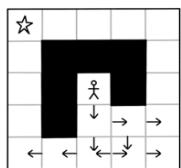
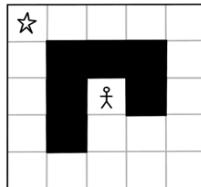
Informed search algorithms are versatile and useful in real-world use cases such as the following:

- *Creating game-playing agents for turn-based games with perfect information*—In some games, two or more players act on the same environment. There have been successful implementations of chess, checkers, and other classic games. Games with perfect information are games that do not have hidden information or random chance involved.
- *Creating game-playing agents for turn-based games with imperfect information*—Unknown future options exist in these games, including games like poker and Scrabble.
- *Adversarial search and ant colony optimization (ACO) for route optimization*—Adversarial search is used in combination with the ACO algorithm (discussed in chapter 6) to optimize package-delivery routes in cities.

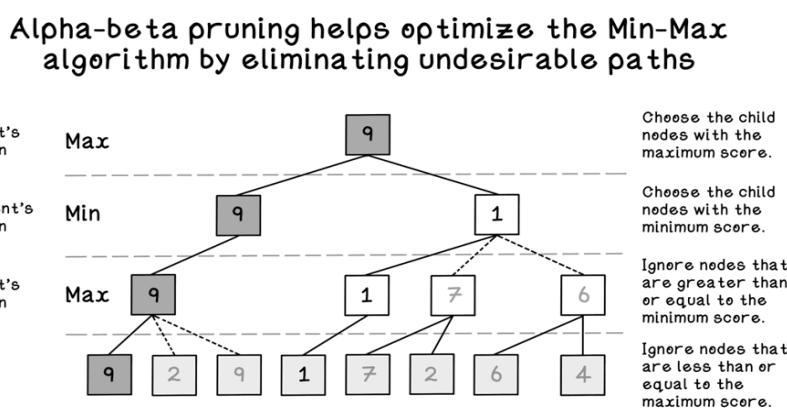
### 3.4 Summary of Intelligent search

Informed search gives search algorithms some intelligence

Heuristics can be tricky  
to think up, but a good  
heuristic is powerful in  
finding solutions more  
efficiently



Score: 10      Score: -10



# 4 Evolutionary algorithms

## This chapter covers

- The inspiration for evolutionary algorithms
- Solving problems with evolutionary algorithms
- Understanding the life cycle of a genetic algorithm
- Designing and developing a genetic algorithm to solve optimization problems

## 4.1 What is evolution?

When you look around at life on Earth, nothing popped into existence fully formed. Everything alive today is the result of a long chain of tiny changes that accumulated over millions of years. This implies that the physical and cognitive characteristics of every living organism are a result of fitting to its environment for survival.

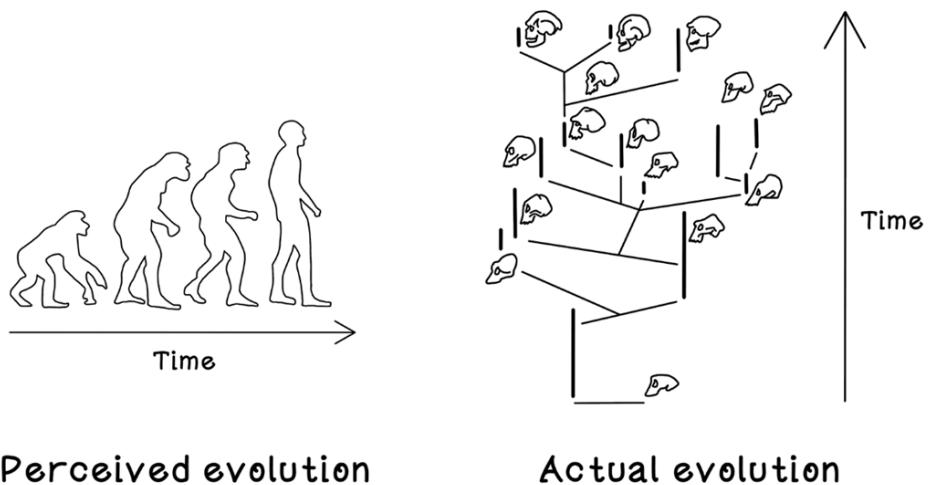
We often make the mistake of thinking evolution is a neat line from “primitive ancestor” to “modern form”. In reality, evolution is messy and branching. Offspring are not perfect clones of their parents; they inherit a mix of genes with small random changes (mutations). At any moment, a species is actually a cloud of variants, not one single clean shape. You only see big, obvious differences when you zoom far out in time and compare averages across thousands of generations.

Figure 4.1 depicts this reality of actual evolution versus the commonly mistaken linear version.

So, evolution is both simple and wild: variation is constantly produced, most variants disappear, and some variants thrive. In doing so, nature essentially “searches” enormous spaces of possibility for the best fit.

Evolutionary algorithms copy exactly this logic to search huge solution spaces in computing: generate diverse candidates, select the better ones, and iterate over generations.

This chapter will explore how to use that same machinery — deliberately and computationally — to solve hard optimization problems.



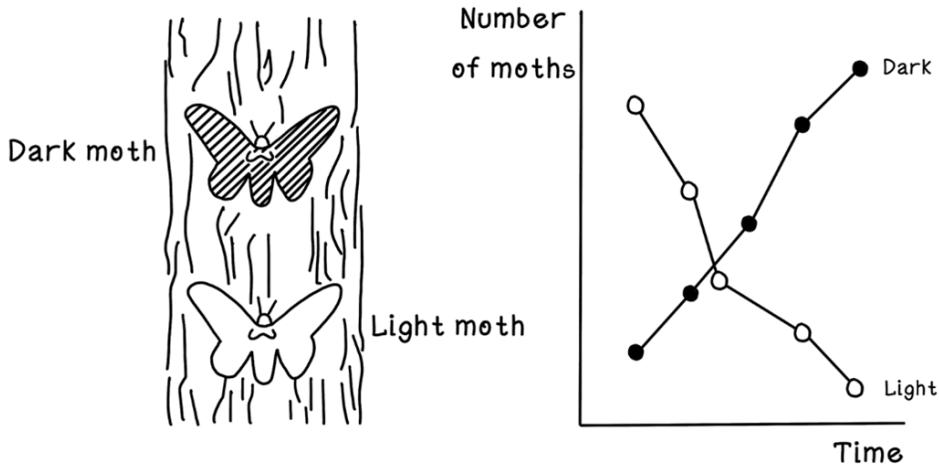
**Figure 4.1** The idea of linear human evolution vs. actual human evolution

Charles Darwin proposed a theory of evolution that centers on natural selection. *Natural selection* is the concept that stronger members of a population are more likely to survive due to being more fit for their environment, which means they reproduce more and, thus, carry traits that are beneficial to survival to future generations—these individuals could potentially perform better than their ancestors.

A classic example of evolution for adaption is the peppered moth. The peppered moth was originally light in color, which made for good camouflage against predators since the moth could blend in with light-colored surfaces in its environment. Only around 2% of the moth population was darker in color. After the Industrial Revolution, around 95% of the species were of the darker color variant. One explanation is that the lighter-colored moths could not blend in with as many surfaces anymore because pollution had darkened surfaces, making lighter-colored moths easier targets for predators because they were more visible. The darker moths had a greater advantage in blending in with the darker surfaces, so they survived longer and reproduced more, and their genetic information was more widely spread to successors (children, grandchildren, etc.).

Among the peppered moths, the attribute that changed on a high level was the color of the moth. This property didn't just magically switch, however. For the change to happen, genes in moths with the darker color had to be carried to successors.

In other examples of natural evolution, we may see dramatic changes in more than simply color between different individuals, but in actuality, these changes are influenced by lower-level genetic differences over many generations (figure 4.2).



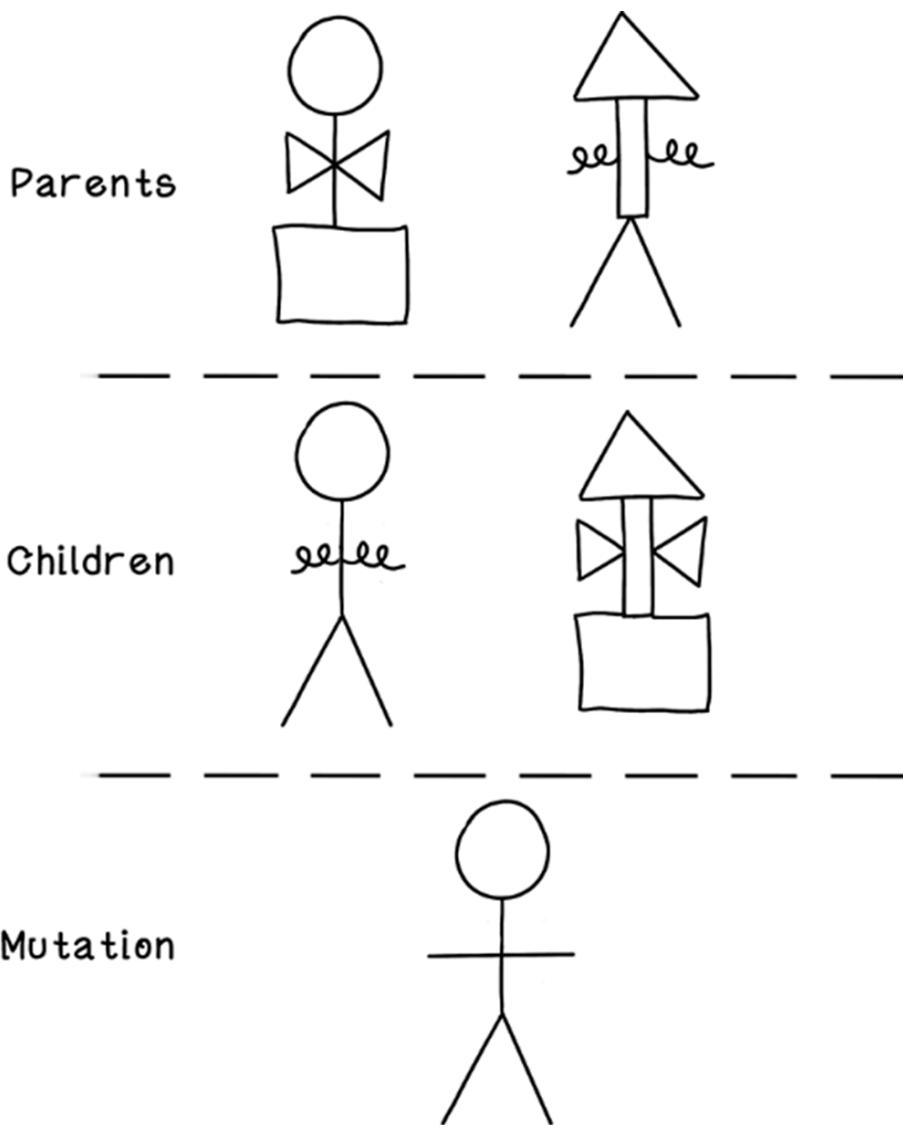
**Figure 4.2 The evolution of the peppered moth**

Evolution encompasses the idea that in a population of a species, pairs of organisms reproduce. The offspring are a combination of the parents' genes, but small changes are made in that offspring through a process called *mutation*. Then the offspring become part of the population. Not all members of a population live on, however. As we know, disease, injury, and consumption by predators cause individuals to die. Individuals that are more adaptive to the environment around them are more likely to live on, a situation that gave rise to the term *survival of the fittest*. Based on Darwinian evolution theory, a population has the following attributes:

- *Variety*—Individuals in the population have different genetic traits.
- *Heredity*—A child inherits genetic properties from its parents.
- *Selection*—A mechanism that measures the fitness of individuals.  
Stronger individuals have the highest likelihood of survival (survival of the fittest).

These properties imply that the following things happen during the process of evolution (figure 4.3):

- *Reproduction*—Usually, two individuals in the population reproduce to create offspring.
- *Crossover and mutation*—The offspring created through reproduction contain a mix of their parents' genes (crossover) and have slight random changes in their genetic code (mutation).



**Figure 4.3 A simple example of reproduction and mutation**

In summary, evolution is a marvelous and chaotic system that produces variations of life forms, some of which are better (fitter) than others for their environments. This theory also applies to evolutionary algorithms; learnings from biological evolution are harnessed for finding optimal solutions to practical problems by generating diverse solutions and converging on better-performing ones over many generations.

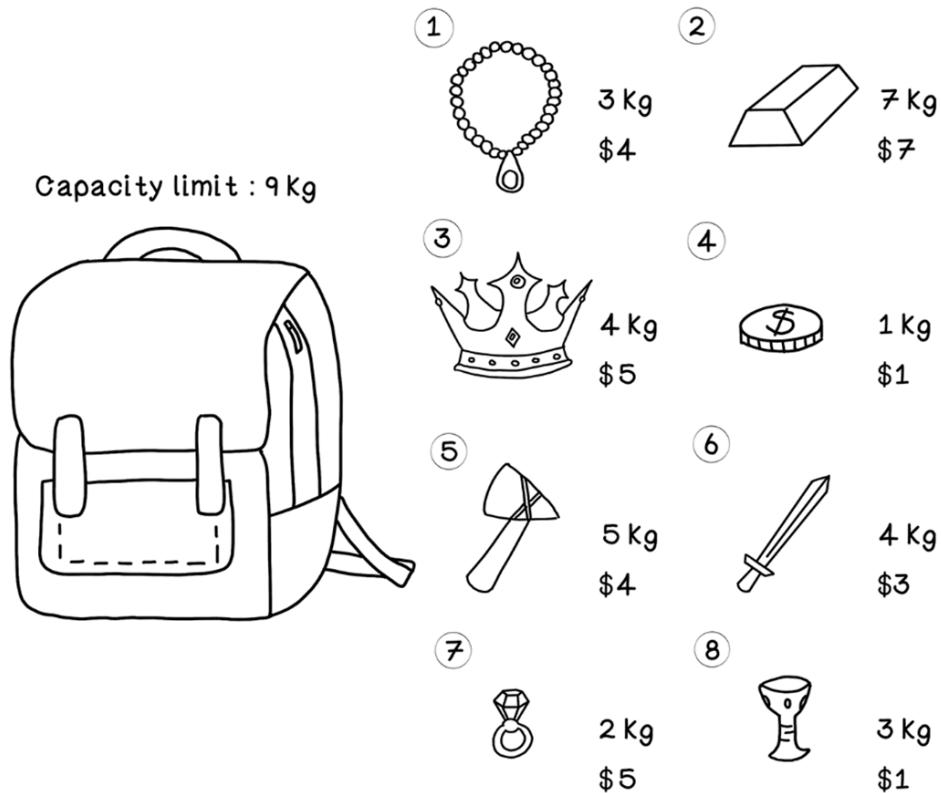
This chapter is dedicated to exploring evolutionary algorithms, which are powerful but underrated approaches to solving hard problems. Evolutionary algorithms can be used in isolation or in conjunction with constructs such as artificial neural networks. Having a solid grasp of this concept opens many possibilities for solving different novel problems.

## 4.2 Problems applicable to evolutionary algorithms

Evolutionary algorithms aren't aimed at solving all problems, but they are powerful for solving *optimization problems* where the solution might be one from a large number of permutations or choices. These problems typically consist of many valid solutions (solutions that work), with some being more optimal than others.

Evolutionary algorithms are powerful, but they are computationally expensive. Use them if your problem has many "traps" (local optima) where a single searcher might get stuck. By using a population, you can explore many areas at once and share the best traits of the survivors.

Consider the Knapsack Problem, a classic problem used in computer science to explore how algorithms work and how efficient they are. In the Knapsack Problem, a knapsack has a specific maximum weight that it can hold. Several items are available to be stored in the knapsack, and each item has a different weight and value. The goal is to fit as many items into the knapsack as possible so that the total value is maximized and the total weight does not exceed the knapsack's limit. The physical size and dimensions of the items are ignored in the simplest variation of the problem (figure 4.4).



**Figure 4.4 A simple Knapsack Problem example**

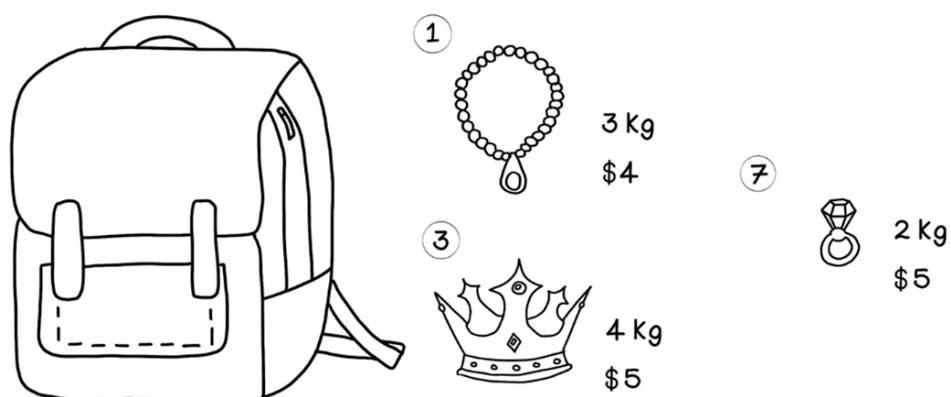
As a trivial example, given the specification of the problem in table 4.1, a knapsack can hold a total weight capacity of 9 kg, and it could contain any of the eight items of varying weight and value.

**Table 4.1 Knapsack weight capacity: 9 kg**

<b>Item ID</b>	<b>Item name</b>	<b>Weight (kg)</b>	<b>Value (\$)</b>
1	Pearls	3	4
2	Gold	7	7
3	Crown	4	5
4	Coin	1	1
5	Axe	5	4
6	Sword	4	3
7	Ring	2	5
8	Cup	3	1

This problem has  $2^8 = 256$  possible solutions, including but not limited to, the following (figure 4.5):

- *Solution 1*—Include Item 1, Item 4, and Item 6. The total weight is 8 kg, and the total value is \$8.
- *Solution 2*—Include Item 1, Item 3, and Item 7. The total weight is 9 kg, and the total value is \$14.
- *Solution 3*—Include Item 2, Item 3, and Item 6. The total weight is 15 kg, which exceeds the knapsack's capacity.

**Figure 4.5 The optimal solution for the simple Knapsack Problem example**

Clearly, the solution with the most value is *Solution 2*. Don't concern yourself too much about how the number of possibilities is calculated, but understand that the possibilities explode as the number of potential items increases.

Although this trivial example can be solved by hand, the Knapsack Problem could have varying weight constraints, a varying number of items, and varying weights and values for each item, making it impossible to solve by hand as the variables grow. It will also be computationally expensive to try to brute-force every combination of items when the variables grow; so, we look for algorithms that are efficient at finding a desirable solution.

Note that we qualify the best solution we can find as a *desirable* solution rather than the *optimal* solution. Although some algorithms attempt to find the one true optimal solution to the Knapsack Problem, an evolutionary algorithm attempts to find the optimal solution but is not guaranteed to find it. The algorithm will find a solution that is acceptable for the use case, however—a subjective opinion of what an acceptable solution is based on the problem at hand. For a mission-critical health system, for example, a “good enough” solution may not cut it; but for a song-recommender system, it may be acceptable.

Now consider the larger dataset (yes, imagine a giant knapsack) in table 4.2, where the number of items and varying weights and values makes the problem difficult to solve by hand. By understanding the complexity of this dataset, you can easily see why many computer science algorithms are measured by their performance in solving such problems. By understanding the complexity of this dataset, you can see why algorithms are judged by two distinct metrics: computational efficiency (how fast they run) and solution fitness (how good the answer is). In the context of Evolutionary Algorithms, we focus heavily on Fitness. In the Knapsack Problem, a solution that yields a higher total dollar value has a higher fitness, regardless of how long it took the computer to find it. Evolutionary algorithms provide one method of finding solutions to the Knapsack Problem.

**Table 4.2 Knapsack capacity: 6,404,180 kg**

<b>Item ID</b>	<b>Item name</b>	<b>Weight (kg)</b>	<b>Value (\$)</b>
1	Axe	32,252	68,674
2	Bronze coin	225,790	471,010
3	Crown	468,164	944,620
4	Diamond statue	489,494	962,094
5	Emerald belt	35,384	78,344
6	Fossil	265,590	579,152
7	Gold coin	497,911	902,698
8	Helmet	800,493	1,686,515
9	Ink	823,576	1,688,691
10	Jewel box	552,202	1,056,157
11	Knife	323,618	677,562
12	Long sword	382,846	833,132
13	Mask	44,676	99,192
14	Necklace	169,738	376,418
15	Opal badge	610,876	1,253,986
16	Pearls	854,190	1,853,562
17	Quiver	671,123	1,320,297
18	Ruby ring	698,180	1,301,637
19	Silver bracelet	446,517	859,835
20	Timepiece	909,620	1,677,534
21	Uniform	904,818	1,910,501
22	Venom potion	730,061	1,528,646
23	Wool scarf	931,932	1,827,477
24	Crossbow	952,360	2,068,204

25	Yesteryear book	926,023	1,746,556
26	Zinc cup	978,724	2,100,851

One way to solve this problem is to use a brute-force approach. This means calculating every possible combination of items and determining the value of each combination that satisfies the knapsack's weight constraint for all possible combinations, then picking the best solution.

Figure 4.6 shows some benchmark analytics for the brute-force approach. Note that the computation is based on the hardware of an average personal computer at the time of writing.

---

Combinations	$2^{26} = 67,108,864$
Iterations	$2^{26} = 67,108,864$
Accuracy	100%
Compute time	~7 minutes

**Figure 4.6 Performance analytics of brute-forcing the Knapsack Problem**

Keep the Knapsack Problem in mind; it will be used throughout this chapter as we attempt to understand, design, and develop a genetic algorithm to find acceptable solutions to this problem.

---

**NOTE** A note about the term *performance*: From the perspective of an individual solution, performance is how well the solution solves the problem. From the perspective of the algorithm, performance may be how well a specific configuration does in finding a solution. Finally, performance may mean computational cycles. Bear in mind that this term is used differently based on the context.

---

The thinking behind using a genetic algorithm to solve the Knapsack Problem can be applied to a range of practical problems. If a logistics company wants to optimize the packing of trucks based on their destinations, for example, a genetic algorithm would be useful. If that same company wanted to find the shortest route between several destinations, a genetic algorithm would be useful as well. If a factory refined items into raw material via a conveyor-belt system, and the order of the items influenced productivity, a genetic algorithm would be useful in determining that order.

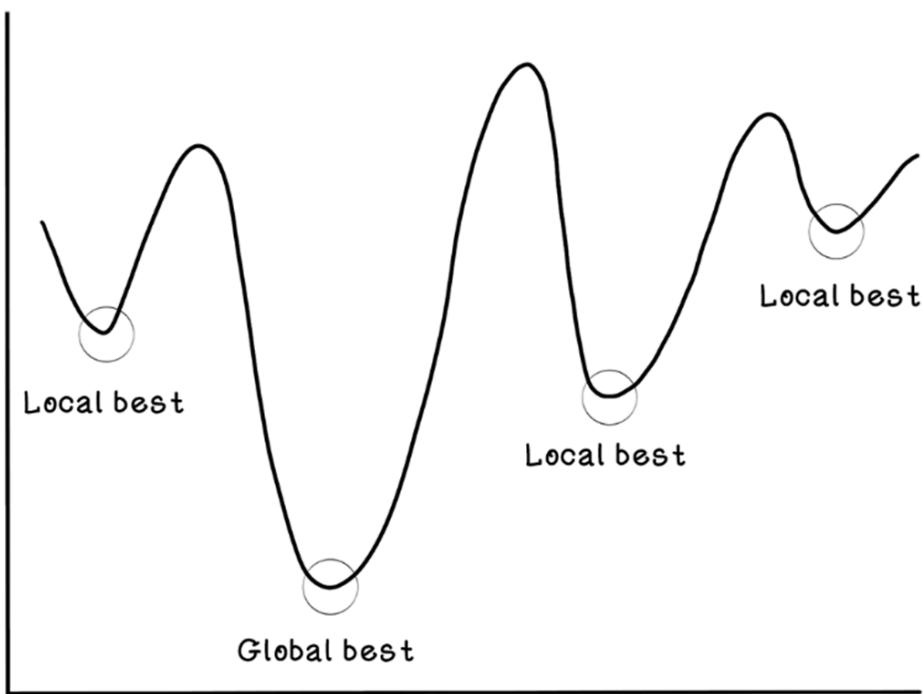
When we dive into the thinking, approach, and life cycle of the genetic algorithm, it should become clear where this algorithm can be applied, and perhaps you will think of new use cases yourself. It is important to keep in mind that a genetic algorithm is stochastic. This means it relies on random probability throughout its lifecycle—such as generating a random initial population, randomly selecting parents, or randomly mutating genes. Consequently, the output is likely to be different each time it is run, even with the same inputs.

### 4.3 Genetic algorithm: Life cycle

The genetic algorithm is a specific algorithm in the family of evolutionary algorithms. Each algorithm works on the same premise of evolution but has small tweaks in the different parts of the life cycle to cater to different problems. We explore some of these parameters in Chapter 5.

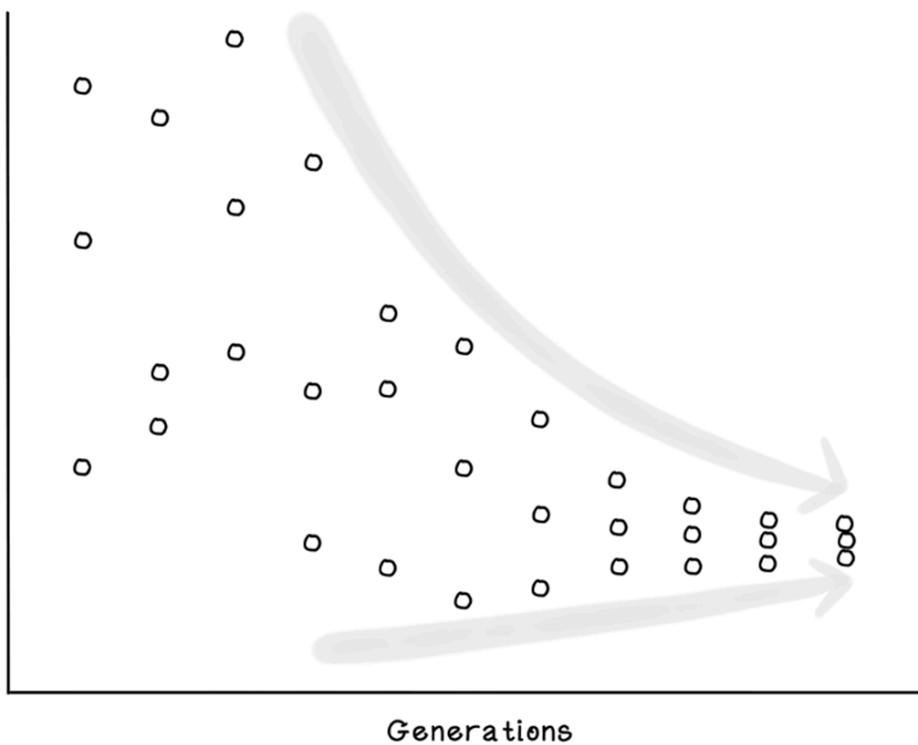
Genetic algorithms are used to evaluate large search spaces for a good solution. It is important to note that a genetic algorithm is not guaranteed to find the absolute best solution; it tries to find the absolute best while avoiding local best solutions, but may settle on a good but not global best one.

A *global best* is the best possible solution, and a *local best* is a solution that is less-optimal. Figure 4.7 represents the possible “best” solutions if the outcome must be minimized—that is, the smaller the value, the better it is. If the goal was to maximize a solution, the larger the value, the better it would be. Optimization algorithms like genetic algorithms aim to incrementally find local best solutions in search of the global best solution.



**Figure 4.7 Local best vs. global best**

Careful attention is needed when configuring the parameters of the algorithm so that it strives for diversity in solutions at the start and gradually gravitates toward better solutions through each generation. At the start, potential solutions should vary widely in individual genetic make-up. Without divergence at the start, the risk of getting stuck in a local best increases (figure 4.8).



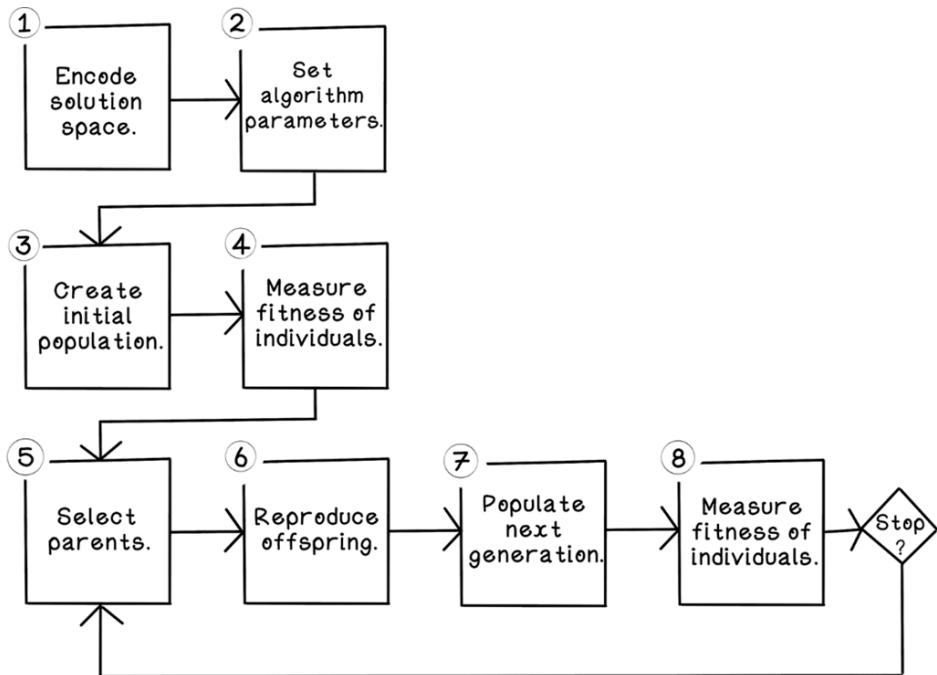
**Figure 4.8 Diversity to convergence**

The configuration for a genetic algorithm is based on the problem space. Each problem has a unique domain where data is represented and evaluated differently.

The general life cycle of a genetic algorithm is making an initial population, measuring fitness, selecting parents, and reproducing:

- *Creating a population*—Creating a random population of potential solutions.
- *Measuring the fitness of individuals in the population*—Determining how good a specific solution is. This task is accomplished by using a fitness function that scores solutions to determine how good they are.
- *Selecting parents based on their fitness*—Selecting pairs of parents that will reproduce offspring.
- *Reproducing individuals from parents*—Creating offspring from their parents by mixing genetic information and applying slight mutations to the offspring.
- *Populating the next generation*—Selecting individuals and offspring from the population that will survive to the next generation.

Several steps are involved in implementing a genetic algorithm. These steps encompass the stages of the algorithm life cycle (figure 4.9).

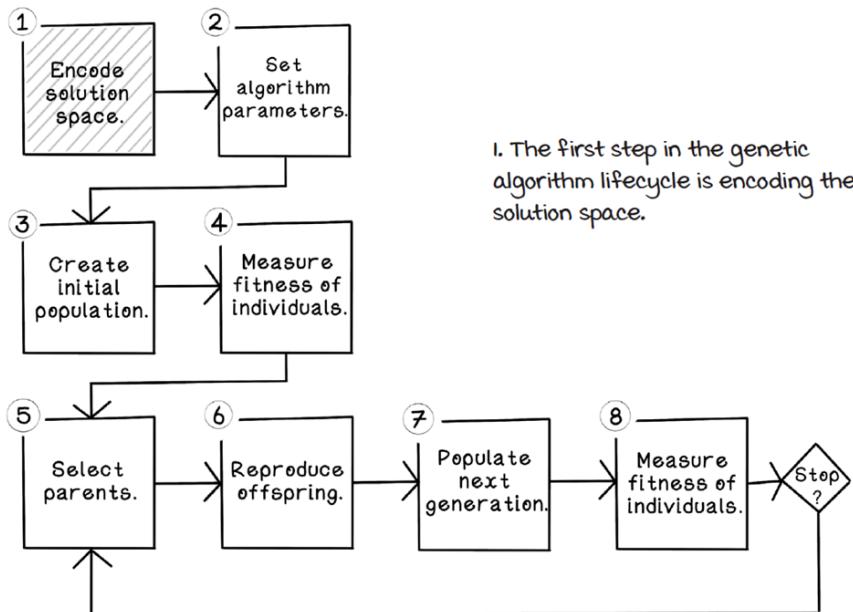


**Figure 4.9** Genetic algorithm life cycle

With the Knapsack Problem in mind, how would we use a genetic algorithm to find solutions to the problem? The next section answers this question.

## 4.4 Encoding the solution spaces

When we use a genetic algorithm, it is paramount to do the encoding step correctly. It requires careful design of the representation of possible states. The *state* is a data structure with specific rules that represents possible solutions to the problem. Furthermore, a collection of states forms the population.



**Figure 4.10** Encode the solution.

### TERMINOLOGY

In evolutionary algorithms, an individual candidate solution is called a chromosome. A chromosome is made up of genes. The gene is the logical type for the unit, and the allele is the actual value stored in that unit. A genotype is a representation of a solution, and a phenotype is a unique solution itself.

Think of it like baking. The Genotype is the recipe card. The Phenotype is the actual cake that comes out of the oven (the result). The Fitness is how good that cake tastes.

Each chromosome always has the same number of genes. A collection of chromosomes forms a population (figure 4.11).

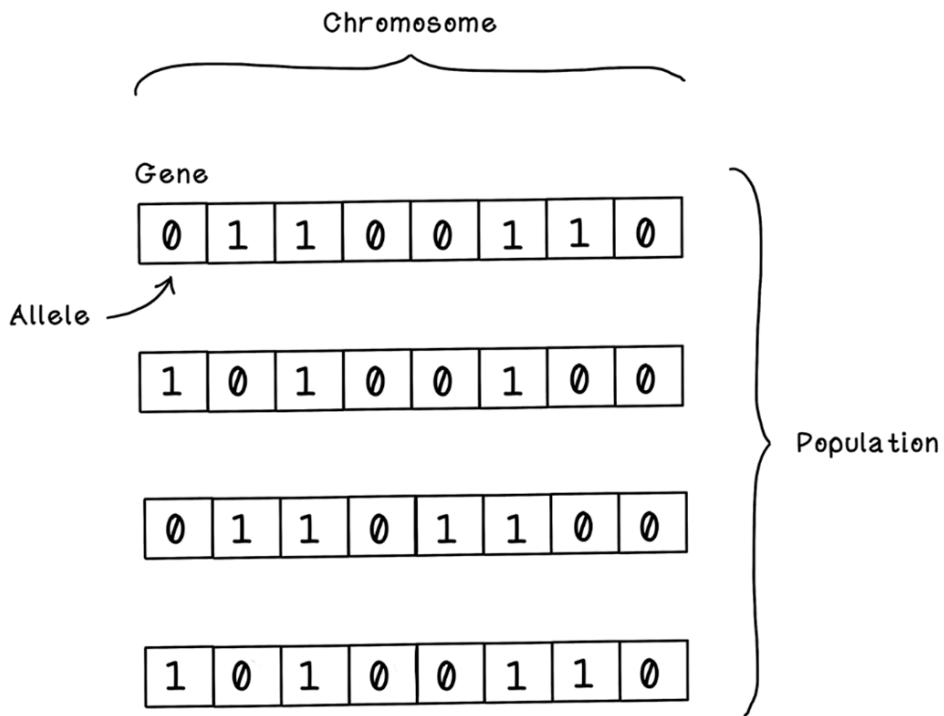


Figure 4.11 Terminology of the data structures representing a population of solutions

In the Knapsack Problem, several items can be placed in the knapsack. A simple way to describe a possible solution that contains some items but not others is binary encoding (figure 4.12). *Binary encoding* represents excluded items with 0s and included items with 1s. If the value at gene index 3 is 1, for example, that item is marked to be included. The complete binary string is always the same size: the number of items available for selection. Several alternative encoding schemes exist, and will be described in Chapter 5.

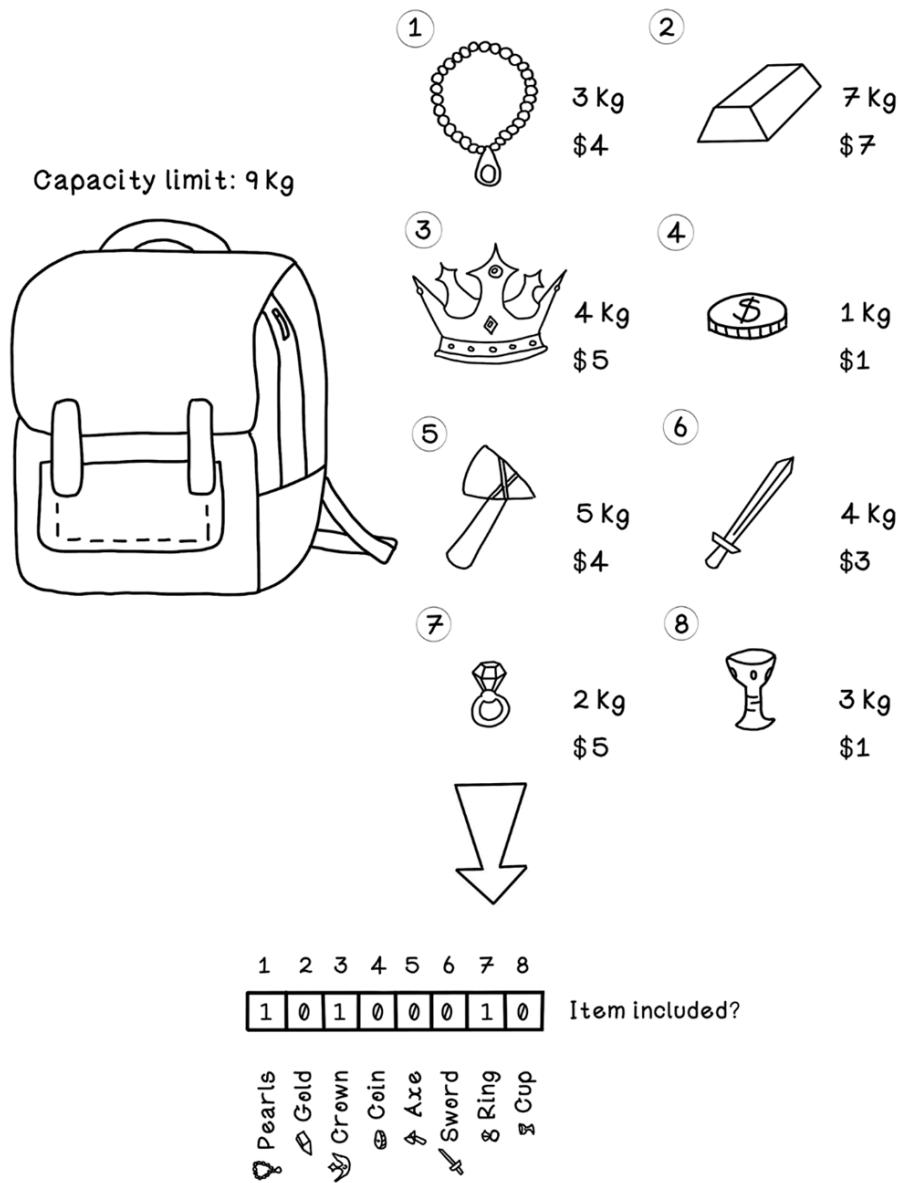


Figure 4.12 Binary-encoding the Knapsack Problem

#### 4.4.1 Binary encoding: Representing possible solutions with zeros and ones

Binary encoding represents a gene as a 0 or 1, so a chromosome is represented by a string of binary bits. Binary encoding can be used in versatile ways to express the presence of a specific element or even encoding numeric values as binary numbers. The advantage of binary encoding is that it is highly efficient for discrete problems (decisions involving "Yes or No" or "On or Off" states). Because it uses primitive types, it places less demand on memory and allows for fast computation. However, binary is not a one-size-fits-all solution. If your problem involves precise decimal numbers (like optimizing a stock price) or specific sequences (like the order of cities in a map), binary encoding can actually make it harder for the algorithm to find a solution. In those cases, Real-Value or Permutation encodings are often better choices. We will explore these in Chapter 5.

Using binary encoding places less demand on working memory, and depending on the programming language used, binary operations are computationally faster. But critical thought must be used to ensure that the encoding makes sense for the problem at hand to represent solutions well; otherwise, the algorithm may perform poorly. Figure 4.13 shows how the different possible items are encoded using binary encoding.

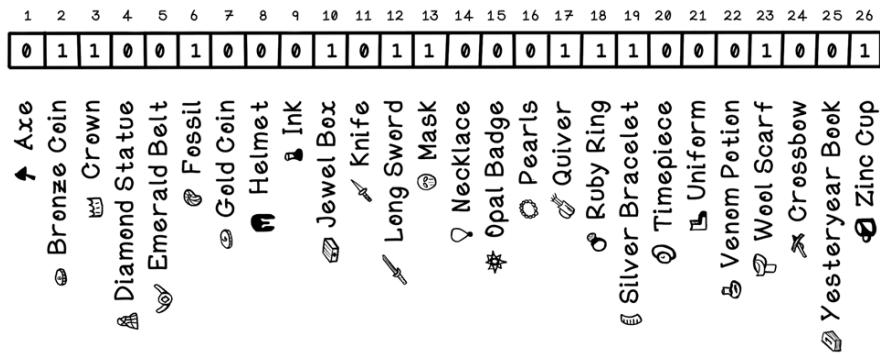


Figure 4.13 Binary-encoding the larger dataset for the Knapsack Problem

Given the Knapsack Problem with a dataset that consists of 26 items of varying weight and value, a binary string can be used to represent the inclusion of each item. The result is a 26-character string in which for each index, 0 means that the respective item is excluded and 1 means that the respective item is included.

Other encoding schemes—including real-value encoding, order encoding, and tree encoding—are discussed in Chapter 5.

### EXERCISE: WHAT IS A POSSIBLE ENCODING FOR THE FOLLOWING PROBLEM?

Suppose we have the following sentence and want to find which words can be excluded or included to maintain a meaningful phrase using a genetic algorithm:

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG

**Incorrect phrases**

THE	BROWN	JUMPS	OVER
QUICK	FOX	OVER	THE
THE	FOX	THE LAZY	

**Correct phrases**

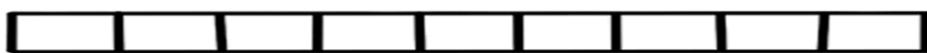
THE	QUICK	FOX						
	QUICK	FOX	JUMPS					
THE	BROWN	FOX					DOG	
THE	BROWN					LAZY	DOG	
THE	QUICK						DOG	
	QUICK			OVER	THE		DOG	
THE	QUICK					LAZY	DOG	

\*Punctuation is excluded.

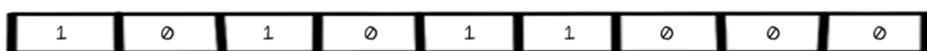
### SOLUTION: WHAT IS A POSSIBLE ENCODING FOR THE FOLLOWING PROBLEM?

Because the number of possible words is always the same, and the words are always in the same position, binary encoding can be used to describe which words are included and which are excluded. The chromosome consists of 9 genes, each gene indicating a word in the phrase.

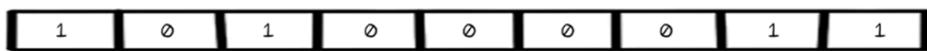
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG



THE BROWN JUMPS OVER

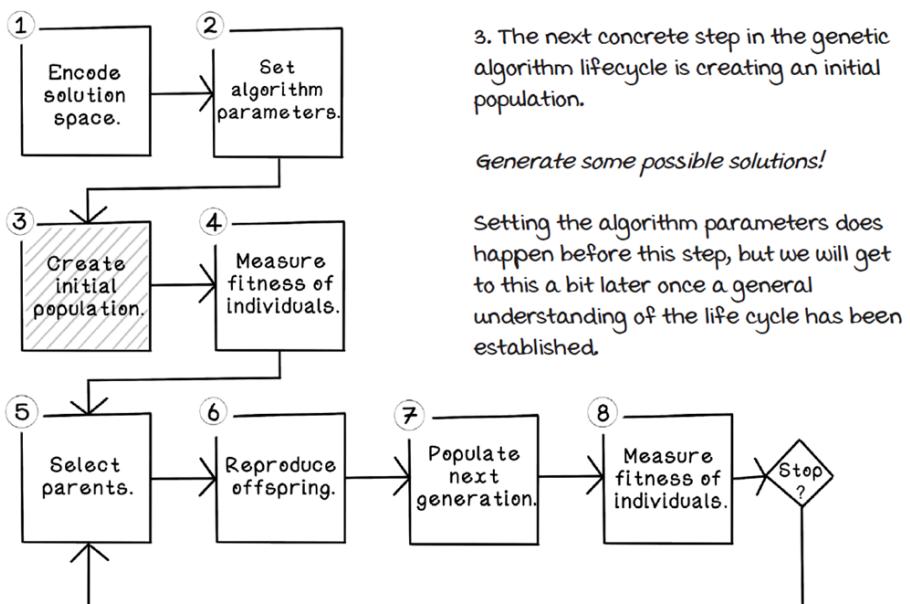


THE BROWN LAZY DOG



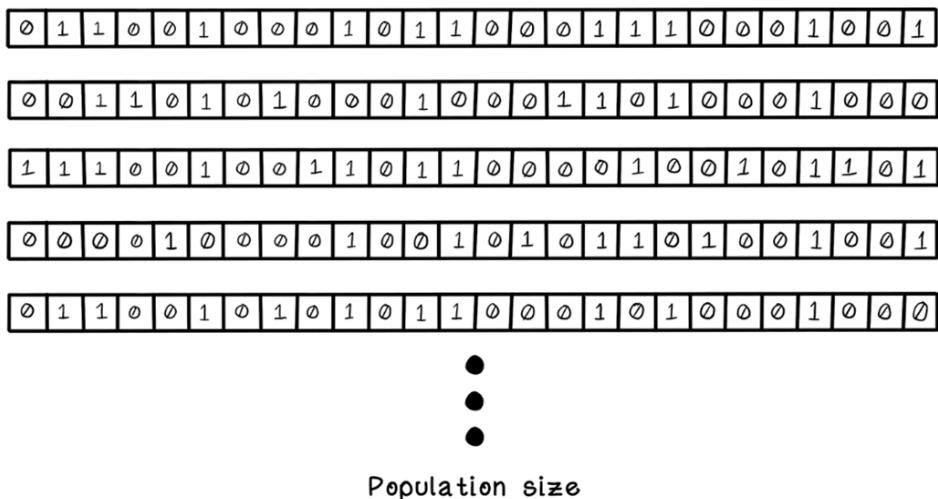
## 4.5 Creating a population of solutions

In the beginning, the population is created. The first step in a genetic algorithm is initializing random potential solutions. In the process of initializing the population, although the chromosomes are generated randomly, the constraints of the problem must be taken into consideration, and the potential solutions should be valid or assigned a terrible fitness score if they violate the constraints (known as a penalty). However, a more effective technique is often to use a repair function—where the algorithm automatically tweaks an invalid solution (like removing one item from an overweight knapsack) to make it valid, rather than just discarding it. The randomly seeded individuals in the population may not solve the problem well, but the solution is valid. As mentioned in the earlier example of packing items into a knapsack, a solution that specifies packing the same item more than once should be an invalid solution and should not form part of the population of possible solutions (figure 4.14).



**Figure 4.14** Create an initial population.

Given how the Knapsack Problem's solution state is represented, this implementation randomly decides whether each item should be included in the bag. That said, only solutions that satisfy the weight-limit constraint should be considered. The problem with simply moving from left to right and randomly choosing whether the item is included is that it creates a bias toward the items on the left end of the chromosome. Similarly, if we start from the right, we will be biased toward items on the right. One possible way to get around this is to generate an entire individual with random genes and then determine whether the solution is valid and does not violate any constraints. Assigning a terrible score to invalid solutions can solve this problem (figure 4.15).



**Figure 4.15 An example of a population of solutions**

## PYTHON CODE SAMPLE

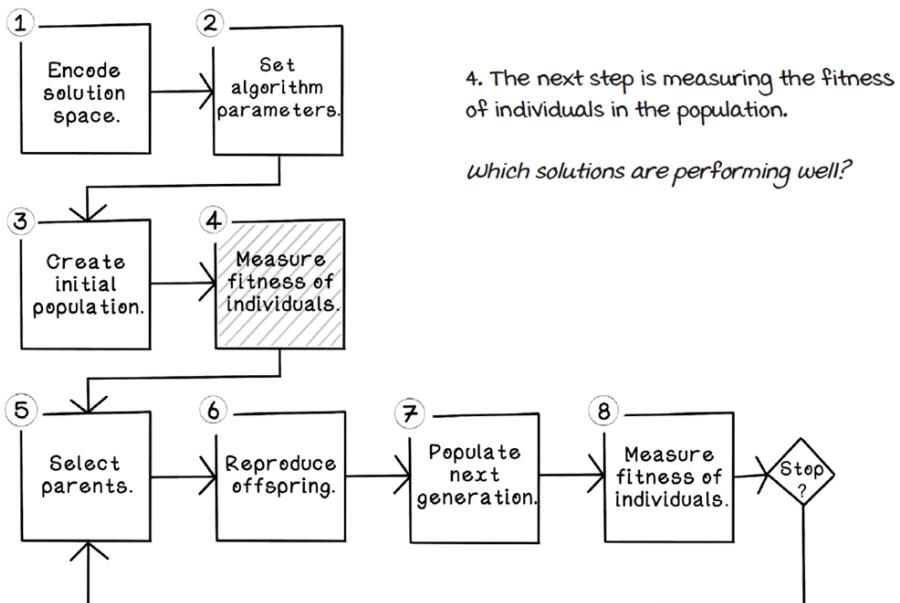
To generate an initial population of possible solutions, an empty array is created to hold the individuals. Then, for each individual in the population, an empty array is created to hold the genes of the individual. Each gene is randomly set to 1 or 0, indicating whether the item at that gene index is included:

```
def generate_initial_population(population_size):
    population = []
    for individual in range(0, population_size):
        individual = ''.join([random.choice('01') for n in range(26)])
        population.append([individual, 0, 0])
    return population
```

## 4.6 Measuring fitness of individuals in a population

When a population has been created, the fitness of each individual in the population needs to be measured. Fitness defines how well a solution performs. The fitness function is critical to the life cycle of a genetic algorithm. If the fitness of the individuals is measured incorrectly or in a way that does not attempt to strive for the optimal solution, the selection process for parents of new individuals and new generations will be influenced; the algorithm will be flawed and cannot strive to find the best possible solution.

Fitness functions are similar to the heuristics that we explored in chapter 3. They are guidelines for finding good solutions (figure 4.16).

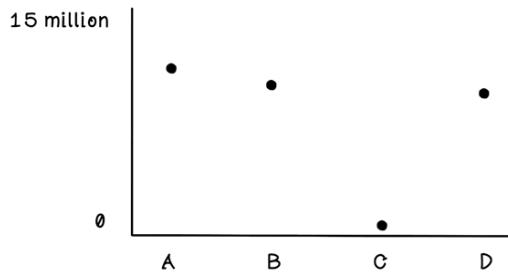


**Figure 4.16 Measure the fitness of individuals.**

In our example, the solution attempts to maximize the value of the items in the knapsack while respecting the weight-limit constraints. The fitness function measures the total value of the items in the knapsack for each individual. The result is that individuals with higher total values are more fit. Note that an invalid individual appears in figure 4.17, to highlight that its fitness score would result in 0—a terrible score—because it exceeds the weight capacity for this instance of the problem, which is 6,404,180.

We deliberately squash this score to 0 to act as a severe penalty. Even though that knapsack might hold a million dollars' worth of items, the fact that it is too heavy to carry makes it effectively worthless. This zero-score ensures the algorithm learns that breaking the rules results in failure, no matter how tempting the contents are.

A	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	1	0	0	1	0	0	0	1	0	1	1	0	0	0	1	1	1	0	0	0	1	0	0	1	11,393,360
0	1	1	0	0	1	0	0	0	1	0	1	1	0	0	0	1	1	1	0	0	0	1	0	0	1			
B	<table border="1"><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	1	1	0	1	0	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	0	0	10,866,684
0	0	1	1	0	1	0	1	0	0	0	1	0	0	0	1	1	0	1	0	0	0	1	0	0	0			
C	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr></table>	1	1	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	1	1	0	1	0 (Overweight)
1	1	1	0	0	1	0	0	1	1	0	1	1	0	0	0	0	1	0	0	1	0	1	1	0	1			
D	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1	0	1	0	0	1	0	0	1	10,715,475		
0	0	0	0	1	0	0	0	0	1	0	0	1	0	1	1	0	1	0	0	1	0	0	1					



**Figure 4.17 Measuring the fitness of individuals**

Depending on the problem being solved, the result of the fitness function may need to be minimized or maximized. In the Knapsack Problem, the contents of the knapsack can be maximized within constraints, or the empty space in the knapsack could be minimized. The approach depends on the interpretation of the problem.

### PYTHON CODE SAMPLE

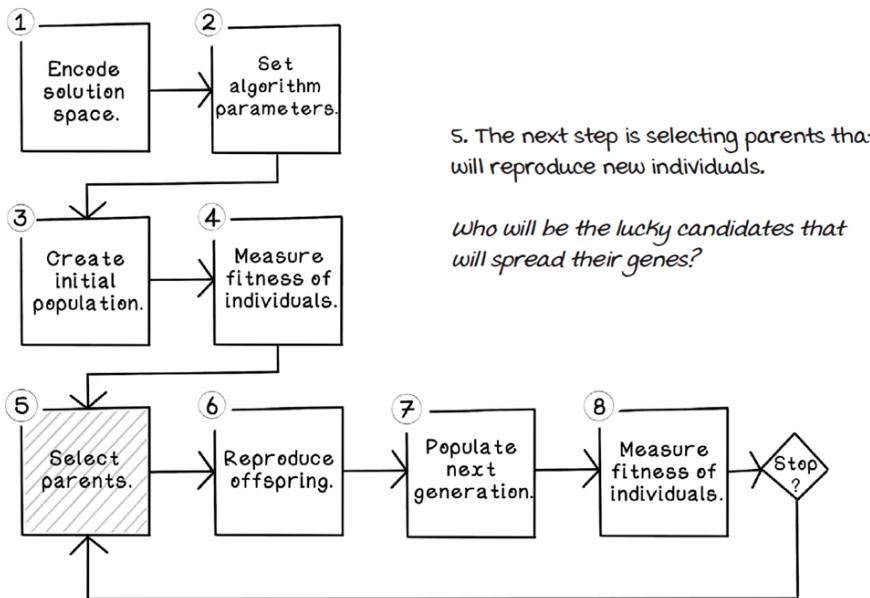
To calculate the fitness of an individual in the Knapsack Problem, the sums of the values of each item that the respective individual includes must be determined. This task is accomplished by setting the total value to 0 and then iterating over each gene to determine whether the item it represents is included. If the item is included, the value of the item represented by that gene is added to the total value. Similarly, the total weight is calculated to ensure that the solution is valid. The concepts of calculating fitness and checking constraints can be split for clearer separation of concerns:

```
def calculate_individual_fitness(individual, maximum_weight):
    total_individual_weight = 0
    total_individual_value = 0
    for gene_index in range(len(individual)):
        gene_switch = individual[gene_index]
        if gene_switch == '1':
            total_individual_weight += knapsack_items[gene_index]
[KNAPSACK_ITEM_WEIGHT_INDEX]
            total_individual_value += knapsack_items[gene_index]
[KNAPSACK_ITEM_VALUE_INDEX]
        if total_individual_weight > maximum_weight:
            return 0
    return total_individual_value
```

## 4.7 Selecting parents based on their fitness

The next step in the genetic algorithm is selecting parents that will produce new individuals. In Darwinian theory, the individuals that are more fit have a higher likelihood of reproduction than others because they typically live longer. Furthermore, these individuals contain desirable attributes for inheritance due to their superior performance in their environment. That said, some individuals are likely to reproduce even if they are not the fittest in the entire group, and these individuals may contain strong traits even though they are not strong in their entirety.

Each individual has a calculated fitness that is used to determine the probability of it being selected to be a parent to a new individual. This attribute makes the genetic algorithm stochastic in nature (figure 4.18).

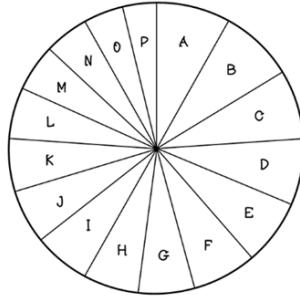


**Figure 4.18 Select parents.**

A popular technique in choosing parents based on their fitness is *roulette-wheel selection*. This strategy gives different individuals portions of a wheel based on their fitness. The wheel is “spun,” and an individual is selected. Higher fitness gives an individual a larger slice of the wheel. This process is repeated until the desired number of parents is reached.

By calculating the probabilities of 16 individuals of varying fitness, the wheel allocates a slice to each. Because many individuals perform similarly, there are many slices of similar size (figure 4.19).

A	<code>1 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1</code>	13,107,019
B	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 0</code>	12,965,145
C	<code>0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1 0 0 0</code>	12,344,873
D	<code>0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 0 0</code>	11,739,363
E	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,711,159
F	<code>1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,611,967
G	<code>1 0 1 0 0 1 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 0</code>	10,042,441
H	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0</code>	9,883,682
I	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 0 0 0 1 0 0 0</code>	9,857,597
J	<code>0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 0 1</code>	9,670,184
K	<code>0 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0</code>	9,277,580
L	<code>1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 0</code>	8,931,719
M	<code>0 1 0 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0</code>	8,324,936
N	<code>1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 0 0</code>	8,018,760
O	<code>0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1</code>	6,900,314
P	<code>0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0</code>	6,056,664



**Figure 4.19 Determining the probability of selection for each individual**

The number of parents selected to be used for reproducing new offspring is determined by the intended total number of offspring required, which is determined by the desired population size for each generation. Two parents are selected, and offspring are created. This process repeats with different parents selected (with a chance of the same individuals being parents more than once) until the desired number of offspring have been generated. Two parents can reproduce a single mixed child or two mixed children. This concept will be made clearer later in this chapter. In our Knapsack Problem example, the individuals with greater fitness are those that fill the bag with the most combined value while respecting the weight-limit constraint.

Population models are ways to control the diversity of the population. Steady state and generational are two population models that have their own advantages and disadvantages.

### 4.7.1 Steady state: Replacing a portion of the population each generation

This high-level approach to population management is not an alternative to the other selection strategies, but a scheme that uses them. The idea is that the majority of the population is retained, and a small group of weaker individuals are removed and replaced with new offspring. This process mimics the cycle of life and death, in which weaker individuals die and new individuals are made through reproduction. If there were 100 individuals in the population, a portion of the population would be existing individuals, and a smaller portion would be new individuals created via reproduction. There may be 80 individuals from the current generation and 20 new individuals.

### 4.7.2 Generational: Replacing the entire population each generation

This high-level approach to population management is similar to the steady-state model but is not an alternative to selection strategies. The generational model creates a number of offspring individuals equal to the population size and replaces the entire population with the new offspring. If there were 100 individuals in the population, each generation would result in 100 new individuals via reproduction. Steady state and generational are overarching ideas for designing the configuration of the algorithm.

### 4.7.3 Roulette wheel: Selecting parents and surviving individuals

Chromosomes with higher fitness scores are more likely to be selected, but chromosomes with lower fitness scores still have a small chance of being selected. The term *roulette-wheel selection* comes from a roulette wheel at a casino, which is divided into slices. Typically, the wheel is spun, and a marble is released into the wheel. The selected slice is the one that the marble lands on when the wheel stops turning.

In this analogy, chromosomes are assigned to slices of the wheel. Chromosomes with higher fitness scores have larger slices of the wheel, and chromosomes with lower fitness scores have smaller slices. A chromosome is selected randomly, much as a ball randomly lands on a slice.

This analogy is an example of probabilistic selection. Each individual has a chance of being selected, whether that chance is small or high. The chance of selection of individuals influences the diversity of the population and convergence rates mentioned earlier in this chapter. Figure 4.19, also earlier in this chapter, illustrates this concept.

## PYTHON CODE SAMPLE

First, the probability of selection for each individual needs to be determined. This probability is calculated for each individual by dividing its fitness by the total fitness of the population. Roulette-wheel selection can be used. The “wheel” is “spun” until the desired number of individuals have been selected. For each selection, a random decimal number between 0 and 1 is calculated. If an individual’s fitness is within that probability, it is selected. Other probabilistic approaches may be used to determine the probability of each individual, including standard deviation, in which an individual’s value is compared with the mean value of the group:

```

def set_probabilities(population):
    population_sum = sum(individual[INDIVIDUAL_FITNESS_INDEX] for individual in
population)
    for individual in population:
        individual[INDIVIDUAL_PROBABILITY_INDEX] =
individual[INDIVIDUAL_FITNESS_INDEX] / population_sum

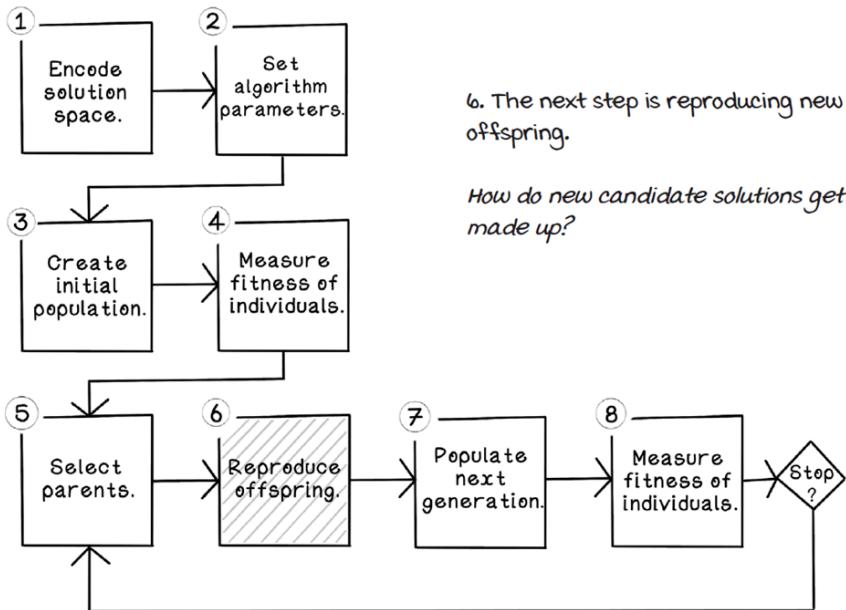

def roulette_wheel_selection(population, number_of_selections): #A
    set_probabilities(population)
    slices = []
    total = 0
    for r in range(0, len(population)):
        individual = population[r]
        slices.append([r, total, total +
individual[INDIVIDUAL_PROBABILITY_INDEX]])
        total += individual[INDIVIDUAL_PROBABILITY_INDEX]
    chosen_ones = []
    for r in range(number_of_selections):
        spin = random.random()
        result = [s[0] for s in slices if s[1] < spin <= s[2]]
        chosen_ones.append(population[result[0]])
    return chosen_ones

#A Roulette wheel selection to select individuals in a population

```

## 4.8 Reproducing individuals from parents

When parents are selected, reproduction needs to happen to create new offspring from the parents. Generally, two steps are related to creating children from two parents. The first concept is *crossover*, which means mixing part of the chromosome of the first parent with part of the chromosome of the second parent, and vice versa. This process results in two offspring that contain inverted mixes of their parents. The second concept is *mutation*, which means randomly changing the offspring slightly to create variation in the population (figure 4.20).



**Figure 4.20 Reproduce offspring.**

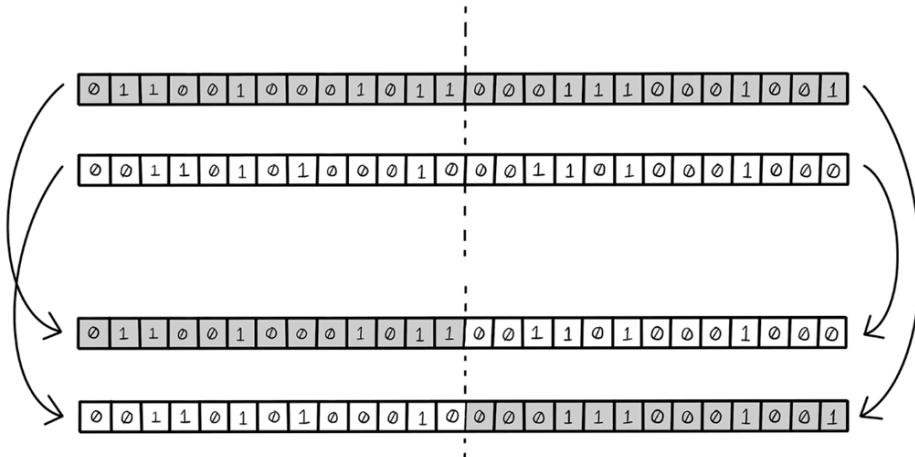
### Crossover

Crossover is mixing genes between two individuals to create one or more offspring individuals. Crossover is inspired by the concept of reproduction. The offspring individuals are parts of their parents, depending on the crossover strategy used. The crossover strategy is highly affected by the encoding used.

#### 4.8.1 Single-point crossover: Inheriting one part from each parent

One point in the chromosome structure is selected. Then, by referencing the two parents in question, the first part of the first parent is used, and the second part of the second parent is used. These two parts combined create a new offspring. A second offspring can be made by using the first part of the second parent and the second part of the first parent.

Single-point crossover is applicable to binary encoding, order/permuation encoding, and real-value encoding (figure 4.21).



**Figure 4.21 Single-point crossover**

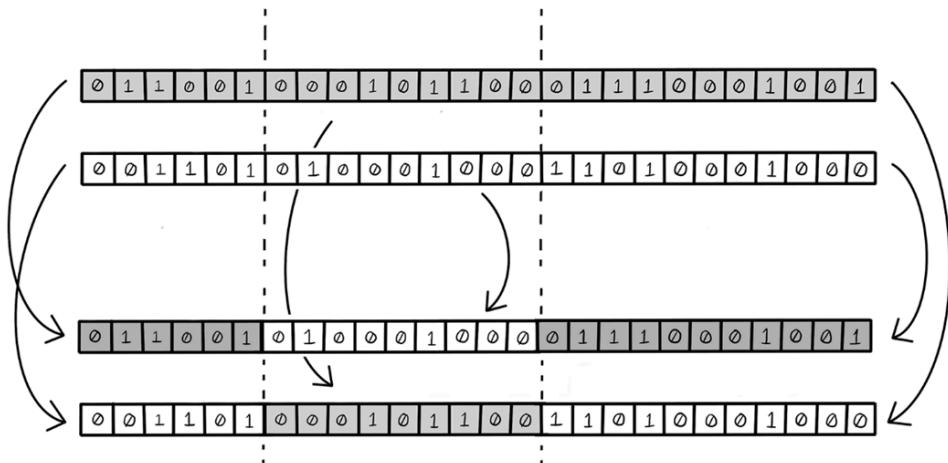
## PYTHON CODE SAMPLE

To create two new offspring individuals, an empty array is created to hold the new individuals. All genes from index 0 to the desired index of parent A are concatenated with all genes from the desired index to the end of the chromosome of parent B, creating one offspring individual. The inverse creates the second offspring individual:

```
def one_point_crossover(parent_a, parent_b, xover_point):
    children = [parent_a[:xover_point] + parent_b[xover_point:],
                parent_b[:xover_point] + parent_a[xover_point:]]
    return children
```

### 4.8.2 Two-point crossover: Inheriting more parts from each parent

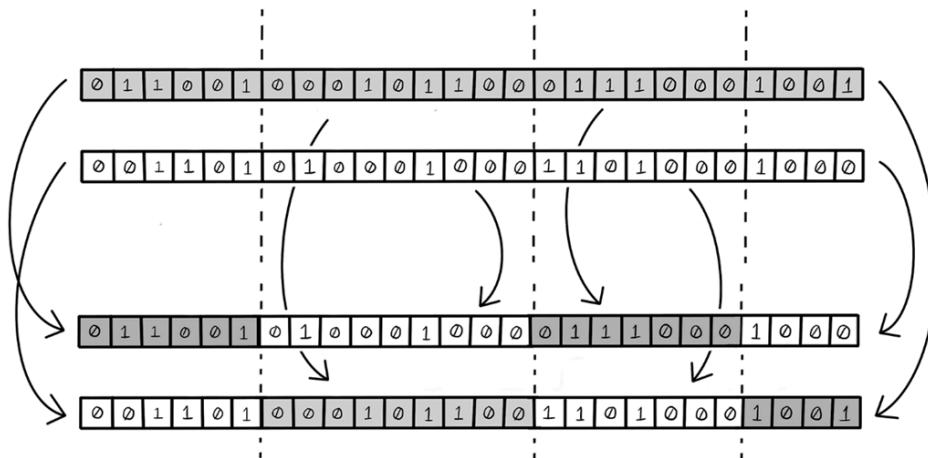
Two points in the chromosome structure are selected; then, referencing the two parents in question, parts are chosen in an alternating manner to make a complete offspring individual. This process is similar to single-point crossover, discussed earlier. To describe the process completely, the offspring consist of the first part of the first parent, the second part of the second parent, and the third part of the first parent. Think about two-point crossover as splicing arrays to create new ones. Again, a second individual can be made by using the inverse parts of each parent. Two-point crossover is applicable to binary encoding and real-value encoding (figure 4.22).



**Figure 4.22 Two-point crossover**

### 4.8.3 Uniform crossover: Inheriting many parts from each parent

Uniform crossover is a step beyond two-point crossover. In uniform crossover, a mask is created that represents which genes from each parent will be used to generate the child offspring. The inverse process can be used to make a second offspring. The mask can be generated randomly each time offspring are created to maximize diversity. Generally speaking, uniform crossover creates more-diverse individuals because the attributes of the offspring are quite different compared with any of their parents. Uniform crossover is applicable to binary encoding and real-value encoding (figure 4.23).



**Figure 4.23 Uniform crossover**

## MUTATION

Mutation is changing offspring individuals slightly to encourage diversity in the population. Several approaches to mutation are used based on the nature of the problem and the encoding method.

One parameter in mutation is the mutation rate—the likelihood that an offspring chromosome will be mutated. Similarly to living organisms, some chromosomes are mutated more than others; an offspring is not an exact combination of its parents' chromosomes but contains minor genetic differences. Mutation can be critical to encouraging diversity in a population and preventing the algorithm from getting stuck in local best solutions.

A high mutation rate means that individuals have a high chance of being selected to be mutated or that genes in the chromosome of an individual have a high chance of being mutated, depending on the mutation strategy. High mutation means more diversity, but too much diversity may result in the deterioration of good solutions.

EXERCISE: WHAT OUTCOME WOULD UNIFORM CROSSOVER GENERATE FOR THESE CHROMOSOMES?

0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1

0 0 1 1 0 1 0 1 0 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 0 0

SOLUTION: WHAT OUTCOME WOULD UNIFORM CROSSOVER GENERATE FOR THESE CHROMOSOMES?

0 1 1 0 0 1 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1

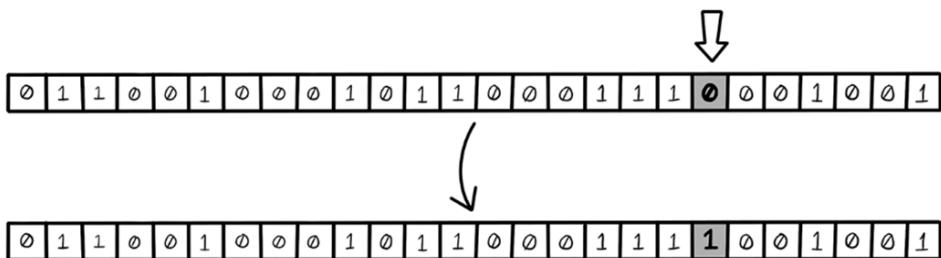
0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 0 1 0 0 0 1 0 0 0

0 1 1 0 0 1 0 0 0 1 0 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1

0 0 1 1 0 1 0 0 0 1 1 1 0 0 0 1 0 1 0 0 0 1 0 0 0

#### 4.8.4 Bit-string mutation for binary encoding

In bit-string mutation, a gene in a binary-encoded chromosome is selected randomly and changed to another valid value (figure 4.24). Other mutation mechanisms are applicable when nonbinary encoding is used.



**Figure 4.24 Bit-string mutation**

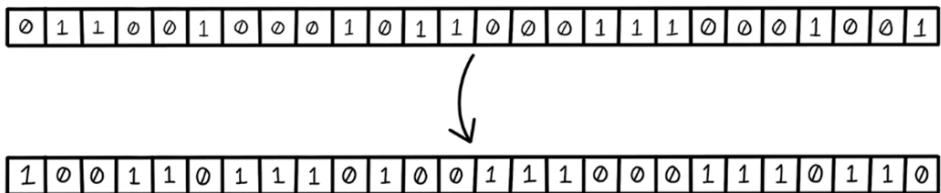
#### PYTHON CODE SAMPLE

To mutate a single gene of an individual's chromosome, a random gene index is selected. If that gene represents 1, change it to represent 0, and vice versa:

```
def mutate_children(children, mutation_rate):
    for child in children:
        random_index = random.randint(0, mutation_rate)
        if child[INDIVIDUAL_CHROMOSOME_INDEX][random_index] == '1':
            mutated_child = list(child[INDIVIDUAL_CHROMOSOME_INDEX])
            mutated_child[random_index] = '0'
            child[INDIVIDUAL_CHROMOSOME_INDEX] = mutated_child
        else:
            mutated_child = list(child[INDIVIDUAL_CHROMOSOME_INDEX])
            mutated_child[random_index] = '1'
            child[INDIVIDUAL_CHROMOSOME_INDEX] = mutated_child
    return children
```

#### 4.8.5 Flip-bit mutation for binary encoding

In flip-bit mutation, all genes in a binary-encoded chromosome are inverted to the opposite value. Where there were 1s are 0s, and where there were 0s are 1s. This type of mutation could degrade good-performing solutions dramatically and usually is used when diversity needs to be introduced into the population constantly (figure 4.25).

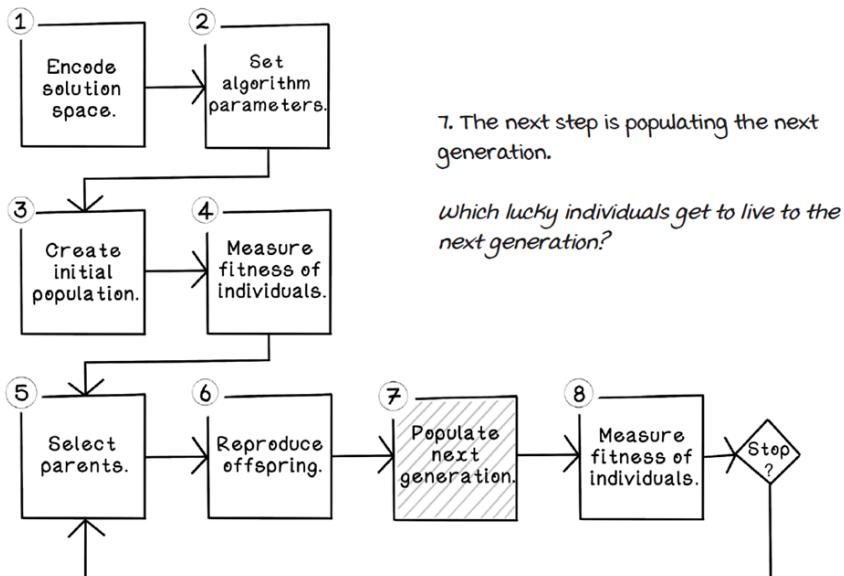


**Figure 4.25 Flip-bit mutation**

## 4.9 Populating the next generation

When the fitness of the individuals in the population has been measured and offspring have been reproduced, the next step is selecting which individuals live on to the next generation. The size of the population is usually fixed, and because more individuals have been introduced through reproduction, some individuals must die off and be removed from the population (figure 4.26).

It may seem like a good idea to take the top individuals that fit into the population size and eliminate the rest. This strategy (called elitism), however, could create stagnation in the diversity of individuals if the individuals that survive are similar in genetic makeup.



**Figure 4.26 Populate the next generation.**

The selection strategies mentioned in this section can be used to determine the individuals that are selected to form part of the population for the next generation.

### 4.9.1 Exploration vs. exploitation

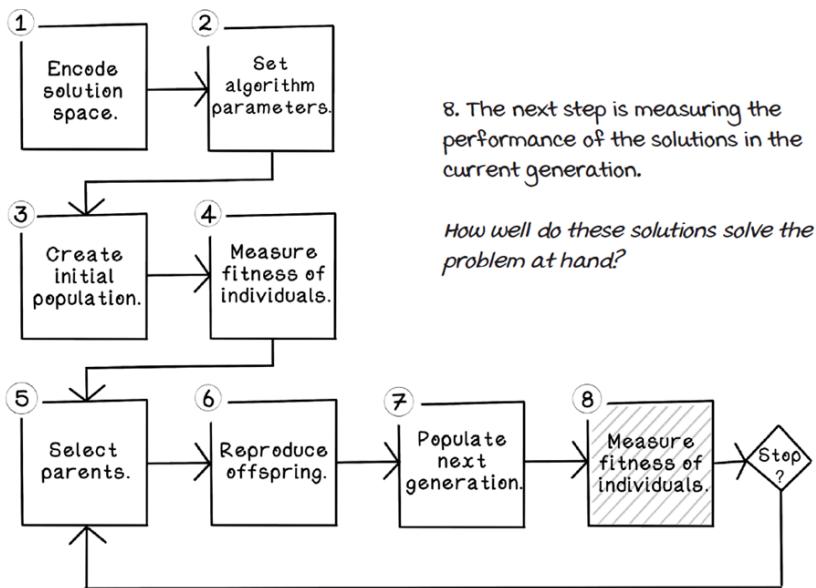
Running a genetic algorithm always involves striking a balance between *exploration* and *exploitation*. The ideal situation is one where there is diversity in individuals and the population as a whole seeks out wildly different potential solutions in the search space; then stronger local solution spaces are exploited to find the most desirable solution.

Imagine you are dropped onto a foggy mountain range and need to find the highest peak.

- *Exploration* (Mutation) is like paragliding to a totally random new spot. You might land in a valley, but you might also find a massive mountain range you couldn't see before.
- *Exploitation* (Crossover/Selection) is like climbing uphill from where you currently stand. You are guaranteed to go higher, but if you don't explore enough, you might get stuck on a small hill (a local maximum) while missing Mount Everest (the global maximum) just a few miles away.

The beauty of this situation is that the algorithm explores as much of the search space as possible while exploiting strong solutions as individuals evolve.

After selecting the individuals and populating the next generation, all individuals' fitness is measured again (figure 4.27).



**Figure 4.27 Measure the fitness of individuals.**

#### 4.9.2 Stopping conditions

Because a genetic algorithm is iterative in finding better solutions through each generation, a stopping condition is needed; otherwise, the algorithm might run forever. A *stopping condition* is the condition that is met where the algorithm ends; the strongest individual of the population at that generation is selected as the best solution.

The simplest stopping condition is a *constant*—a constant value that specifies the number of generations for which the algorithm will run. Another approach is to stop when a certain fitness is achieved. This method is useful when a desired minimum fitness is known but the solution is unknown.

*Stagnation* is a problem in evolutionary algorithms in which the population yields solutions of similar strength for several generations. If a population stagnates, the likelihood of generating strong solutions in future generations is low. A stopping condition could look at the change in the fitness of the best individual in each generation and, if the fitness changes only marginally, choose to stop the algorithm.

#### PYTHON CODE SAMPLE

The various steps of a genetic algorithm are used in a main function that outlines the life cycle in its entirety. The variable parameters include the population size, the number of generations for the algorithm to run, and the knapsack capacity for the fitness function, in addition to the variable crossover position and mutation rate for the crossover and mutation steps:

```

def run_ga():
    best_global_fitness = 0
    global_population = generate_initial_population(INITIAL_POPULATION_SIZE)
    for generation in range(NUMBER_OF_GENERATIONS):
        current_best_fitness = calculate_population_fitness(global_population,
KNAPSACK_WEIGHT_CAPACITY)
        if current_best_fitness > best_global_fitness:
            best_global_fitness = current_best_fitness
        the_chosen = roulette_wheel_selection(global_population, 100)
        the_children = reproduce_children(the_chosen)
        the_children = mutate_children(the_children, MUTATION_RATE)
        global_population = merge_population_and_children(global_population,
the_children)

```

As mentioned at the beginning of this chapter, the Knapsack Problem could be solved using a brute-force approach, which requires more than 60 million combinations to be generated and analyzed. When comparing genetic algorithms that aim to solve the same problem, we can see far more efficiency in computation if the parameters for exploration and exploitation are configured correctly. Remember, in some cases, a genetic algorithm produces a “good enough” solution that is not necessarily the best possible solution but is desirable. Again, using a genetic algorithm for a problem depends on the context (figure 4.28).

	Brute force	Genetic algorithm
Iterations	$2^{26} = 67,108,864$	10,000 - 100,000
Accuracy	100%	100%
Compute time	~7 minutes	~3 seconds
Best value	13,692,887	13,692,887

Figure 4.28 Brute-force performance vs. genetic algorithm performance

## 4.10 Configuring the parameters of a genetic algorithm

In designing and configuring a genetic algorithm, several decisions need to be made that influence the performance of the algorithm. The performance concerns fall into two areas: the algorithm should strive to perform well in finding good solutions to the problem, and the algorithm should perform efficiently from a computation perspective. It would be pointless to design a genetic algorithm to solve a problem if the solution will be more computationally expensive than other traditional techniques. The approach used in encoding, the fitness function used, and the other algorithmic parameters influence both types of performances in achieving a good solution and computation. Here are some parameters to consider:

- *Chromosome encoding*—The chromosome encoding method requires thought to ensure that it is applicable to the problem and that the potential solutions strive for global maxima. The encoding scheme is at the heart of the success of the algorithm.
- *Population size*—The population size is configurable. A larger population encourages more diversity in possible solutions. Larger populations, however, require more computation at each generation. Sometimes, a larger population balances out the need for mutation, which results in diversity at the start but no diversity during generations. A valid approach is to start with a smaller population and grow it based on performance.
- *Population initialization*—Although the individuals in a population are initialized randomly, ensuring that the solutions are valid is important for optimizing the computation of the genetic algorithm and initializing individuals with the right constraints.
- *Number of offspring*—The number of offspring created in each generation can be configured. Given that after reproduction, part of the population is killed off to ensure that the population size is fixed, more offspring means more diversity, but there is a risk that good solutions will be killed off to accommodate those offspring. If the population is dynamic, the population size may change after every generation, but this approach requires more parameters to configure and control.
- *Parent selection method*—The selection method used to choose parents can be configured. The selection method must be based on the problem and the desired explorability versus exploitability.
- *Crossover method*—The crossover method is associated with the encoding method used but can be configured to encourage or discourage diversity in the population. The offspring individuals must still yield a valid solution.

- *Mutation rate*—The mutation rate is another configurable parameter that induces more diversity in offspring and potential solutions. A higher mutation rate means more diversity, but too much diversity may deteriorate good-performing individuals. The mutation rate can change over time to create more diversity in earlier generations and less in later generations. This result can be described as exploration at the start followed by exploitation.
- *Mutation method*—The mutation method is similar to the crossover method in that it is dependent on the encoding method used. An important attribute of the mutation method is that it must still yield a valid solution after the modification or assigned a terrible fitness score.
- *Generation selection methods*—Much like the selection method used to choose parents, a generation selection method must choose the individuals that will survive the generation. Depending on the selection method used, the algorithm may converge too quickly and stagnate or explore too long.
- *Stopping condition*—The stopping condition for the algorithm must make sense based on the problem and desired outcome. Computational complexity and time are the main concerns for the stopping condition.

## 4.11 Use cases for evolutionary algorithms

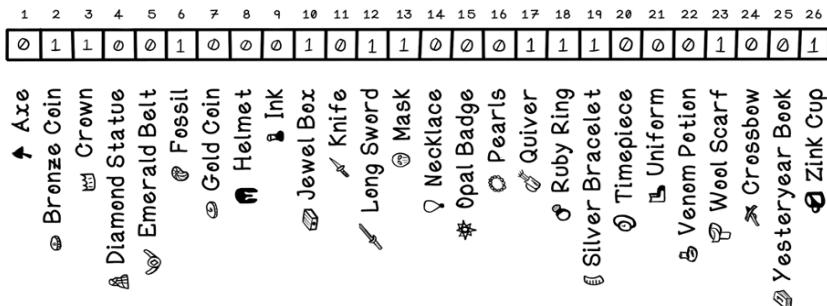
Evolutionary algorithms have a wide variety of uses. Some algorithms address isolated problems; others combine evolutionary algorithms with other techniques to create novel approaches to solving difficult problems, such as the following:

- *Predicting investor behavior in the stock market*—Consumers who invest make decisions every day about whether to buy more of a specific stock, hold on to what they have, or sell stock. Sequences of these actions can be evolved and mapped to outcomes of an investor’s portfolio. Financial institutions can use this insight to proactively provide valuable customer service and guidance.
- *Feature selection in machine learning*—Machine learning is discussed in chapter 8, but a key aspect of machine learning is: given a number of features about something, determining what it is classified as. If we’re looking at houses, we may find many attributes related to houses, such as age, building material, size, color, and location. But to predict market value, perhaps only age, size, and location matter. A genetic algorithm can uncover the isolated features that matter the most.
- *Code breaking and ciphers*—A *cipher* is a message encoded in a certain way to look like something else and is often used to hide information. If the receiver does not know how to decipher the message, it cannot be understood. Evolutionary algorithms can generate many possibilities for changing the ciphered message to uncover the original message.

Chapter 5 dives into advanced concepts of genetic algorithms that adapt them to different problem spaces. We explore different techniques for encoding, crossover, mutation, and selection, as well as uncover effective alternatives.

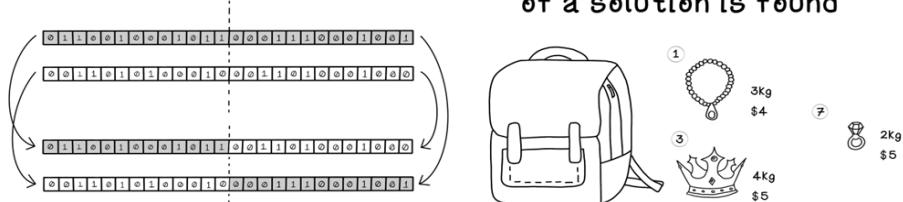
## 4.12 Summary of evolutionary algorithms

**Genetic algorithms use the theory of evolution to find good solutions to optimization problems**

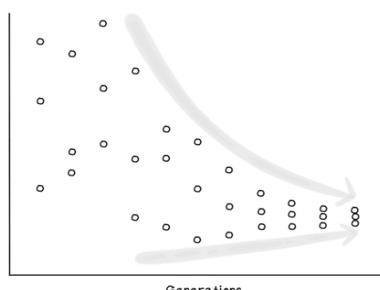


Crossover aims to reproduce better solutions after each generation

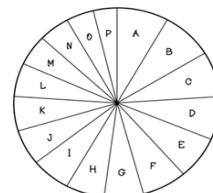
The fitness function directly influences how good of a solution is found



Selection favors stronger individuals but gives weaker ones a chance to reproduce - potentially making good solutions in future generations



Explore at the start, exploit at the end, to produce good solutions



# 5 Advanced evolutionary approaches

## This chapter covers

- Considering options for the various steps in the genetic algorithm life cycle
- Adjusting a genetic algorithm to solve varying problems
- The advanced parameters for configuring a genetic algorithm life cycle based on different scenarios, problems, and datasets

---

**NOTE** Chapter 4 is a prerequisite to this chapter.

---

## 5.1 Evolutionary algorithm life cycle

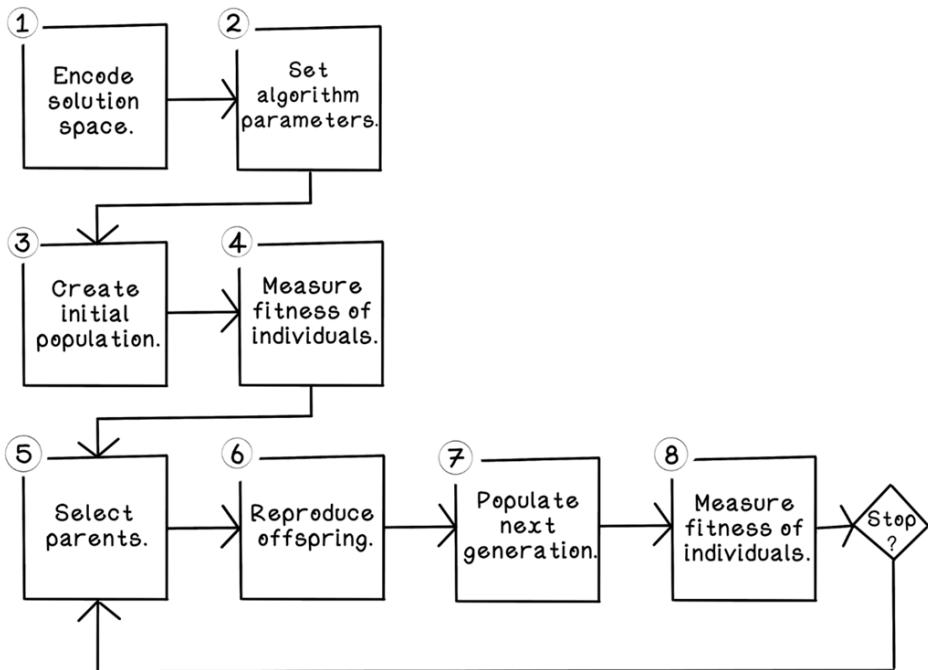
The general life cycle of a genetic algorithm is outlined in chapter 4. In this chapter, we consider other problems that may be suitable to be solved with a genetic algorithm, why some of the approaches demonstrated so far won't work, and alternative approaches.

As a reminder, the general life cycle of a genetic algorithm is:

- *Creating a population*—Creating a random population of potential solutions.
- *Measuring fitness of individuals in the population*—Determining how good a specific solution is. This task is accomplished by using a fitness function that scores solutions to determine how good they are.
- *Selecting parents based on their fitness*—Selecting pairs of parents that will reproduce offspring.

- *Reproducing individuals from parents*—Creating offspring from their parents by mixing genetic information and applying slight mutations to the offspring.
- *Populating the next generation*—Selecting individuals and offspring from the population that will survive to the next generation.

Keep the life cycle flow (depicted in figure 5.1) in mind as we work through this chapter.



**Figure 5.1 Genetic algorithm life cycle**

This chapter starts by exploring alternative selection strategies; these individual approaches can be generically swapped in and out for any genetic algorithm. Then it follows three scenarios that are tweaks of the Knapsack Problem (chapter 4) to highlight the utility of the alternative encoding, crossover, and mutation approaches (figure 5.2).

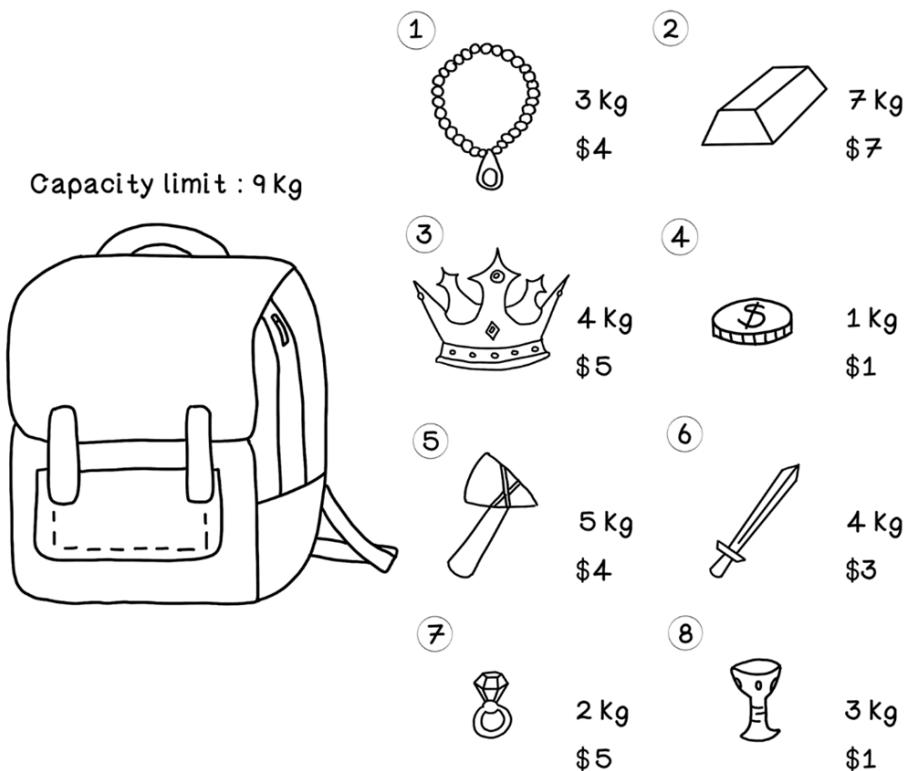


Figure 5.2 The example Knapsack Problem

## 5.2 Alternative selection strategies

In chapter 4, we explored one selection strategy: roulette-wheel selection, which is one of the simplest methods for selecting individuals. The following three selection strategies help mitigate the problems of roulette-wheel selection; each has advantages and disadvantages that affect the diversity of the population, which ultimately affects whether an optimal solution is found.

### 5.2.1 Rank selection: Even the playing field

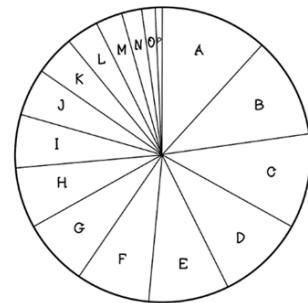
One problem with roulette-wheel selection is the vast differences in the magnitude of fitness between chromosomes. This heavily biases the selection toward choosing individuals with high fitness scores or giving poor-performing individuals a larger chance of selection than desired because their fitness is represented disproportionately. This problem affects the diversity of the population. More diversity means more exploration of the search space, but it can also make finding optimal solutions take too many generations.

Rank selection aims to solve this problem by ranking individuals based on their fitness and then using each individual's rank as the value for calculating the size of its slice on the wheel. In the Knapsack Problem, this value is a number between 1 and 16, because we're choosing among 16 individuals. Although strong individuals are more likely to be selected and weaker ones are less likely to be selected even though they are average, each individual has a fairer chance of being selected based on rank rather than exact fitness.

Think of a marathon. The winner might cross the finish line 1 hour ahead of second place, or just 1 second ahead. In Roulette Wheel selection, that time gap matters—the much faster runner gets a massively bigger slice. In Rank Selection, we ignore the time gap. First place gets \$100, Second gets \$50. It doesn't matter how much better the winner was, only that they were better. This prevents one "super-individual" from dominating the population too early.

When 16 individuals are ranked, the wheel looks slightly different from roulette-wheel selection (figure 5.3).

A	<code>1 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 1</code>	1
B	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 0</code>	2
C	<code>0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1 0 0 0</code>	3
D	<code>0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0</code>	4
E	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	5
F	<code>1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	6
G	<code>1 0 1 0 0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 0</code>	7
H	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 0</code>	8
I	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0</code>	9
J	<code>0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 1 0 0 1</code>	10
K	<code>0 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0</code>	11
L	<code>1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 0</code>	12
M	<code>0 1 0 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0</code>	13
N	<code>1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 0 0 0</code>	14
O	<code>0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1</code>	15
P	<code>0 0 0 1 1 0 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0</code>	16



**Figure 5.3 Example of rank selection**

Figure 5.4 compares roulette-wheel selection and rank selection. It is clear that rank selection gives better-performing solutions a better chance of selection.

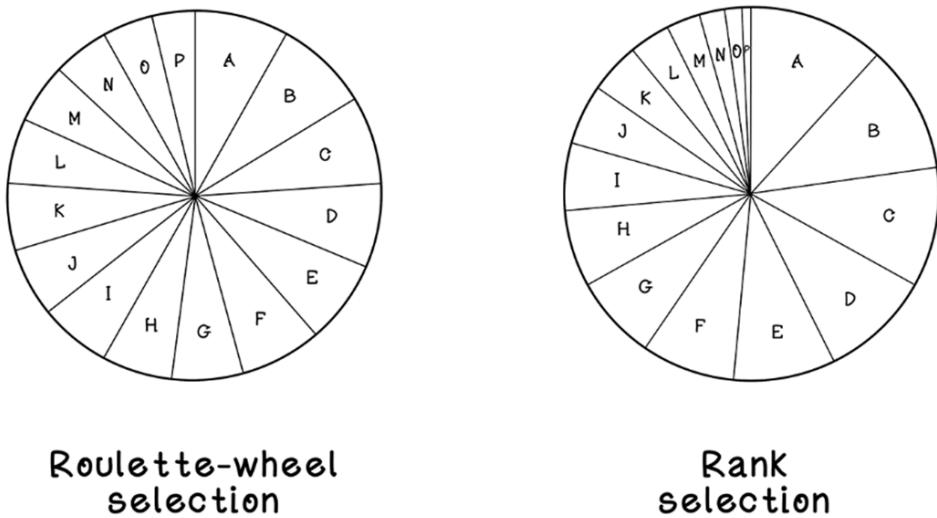


Figure 5.4 Roulette-wheel selection vs. rank selection

### 5.2.2 Tournament selection: Let them fight

Tournament selection plays individual chromosomes against one other. Tournament selection randomly chooses a set number of individuals from the population and places them in a group. This process is performed for a predetermined number of groups. The individual with the highest fitness score in each respective group is selected. The larger the group, the less diverse it is, because only one individual from each group is selected.

Imagine grabbing 4 random people from the population and throwing them into an arena. The single fittest person in that small group wins the right to reproduce. We don't care if they are the strongest in the world, only that they are the strongest in that specific group. This method is fast, efficient, and easily tunable—larger tournaments create stricter pressure to be fit, while smaller tournaments allow for more randomness.

When 16 individuals are allocated to four groups, selecting only 1 individual from each group results in the choice of 4 of the strongest individuals from those groups. Then the 4 winning individuals can be paired to reproduce (figure 5.5).

		Group	
A	<code>1 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1</code>	13,107,019	♠
B	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 0</code>	12,965,145	♠
C	<code>0 0 1 1 0 1 1 0 1 0 1 1 0 0 0 1 0 1 0 1 1 0 0 0</code>	12,344,873	♠
D	<code>0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 0 0</code>	11,739,363	♠
E	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,711,159	♣
F	<code>1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,611,967	♣
G	<code>1 0 1 0 0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 0</code>	10,042,441	♣
H	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0 0</code>	9,883,682	♣
I	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0</code>	9,857,597	♥
J	<code>0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1</code>	9,670,184	♥
K	<code>0 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0</code>	9,277,580	♥
L	<code>1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 0</code>	8,931,719	♥
M	<code>0 1 0 0 0 0 0 1 1 1 0 1 1 1 1 0 0 0 1 0 0 0 0 0 0 0</code>	8,324,936	♦
N	<code>1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0</code>	8,018,760	♦
O	<code>0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1</code>	6,900,314	♦
P	<code>0 0 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0</code>	6,056,664	♦

		Winners	
♠	A	♠	A
♣	E	♣	E
♥	I	♥	I
♦	M	♦	M

Figure 5.5 Example of tournament selection

### 5.2.3 Elitism selection: Choose only the best

Elitism acts like a “VIP Pass” for your best solutions. It allows the top performers to skip the risky process of crossover and mutation and move straight to the next round. This ensures the best solution found so far is never lost. However, relying on it too much is dangerous: if everyone is an elite, no one is exploring new territory, and the algorithm stops learning. The disadvantage of elitism is that the population can fall into a local best solution space and never be diverse enough to find global bests (figure 5.6).

Elitism is often used in conjunction with roulette-wheel selection, rank selection, and tournament selection. The idea is that several elite individuals are selected to reproduce, and the rest of the population is filled with individuals by means of one of the other selection strategies.

A	<code>1 0 1 1 1 0 0 1 1 0 1 1 0 1 0 0 0 1 1 0 0 1 0 0 0 1</code>	13,107,019 *
B	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0</code>	12,965,145 *
C	<code>0 0 1 1 0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 1 0 1 1 0 0 0</code>	12,344,873 *
D	<code>0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 0 1 1 0 0 0 0 0</code>	11,739,363 *
E	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,711,159 *
F	<code>1 1 0 0 0 1 0 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 0</code>	11,611,967 *
G	<code>1 0 1 0 0 1 1 1 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 1 0</code>	10,042,441 *
H	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 0</code>	9,883,682 *
I	<code>1 1 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 0 1 0 0 0 1 0 0 0</code>	9,857,597 !
J	<code>0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1</code>	9,670,184 !
K	<code>0 0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0</code>	9,277,580 !
L	<code>1 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 0 1 0 0</code>	8,931,719 !
M	<code>0 1 0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0</code>	8,324,936 !
N	<code>1 1 1 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0</code>	8,018,760 !
O	<code>0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1</code>	6,900,314 !
P	<code>0 0 0 1 1 0 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0</code>	6,056,664 !

Elitism Survivors

**Figure 5.6 Example of elitism selection**

Chapter 4 explores a problem in which including items in or excluding items from the knapsack was important. A variety of problem spaces require a different encoding because binary encoding won't make sense. The following three sections describe these scenarios.

### 5.3 Real-value encoding: Working with real numbers

Consider that the Knapsack Problem has changed slightly. The problem remains choosing the most valuable items to fill the weight capacity of the knapsack. But the choice involves more than one unit of each item. As shown in table 5.1, the weights and values remain the same as the original dataset, but a quantity of each item is included. With this slight adjustment, a plethora of new solutions are possible, and one or more of those solutions may be more optimal, because a specific item can be selected more than once. Binary encoding is a poor choice in this scenario. Real-value encoding is better suited to representing the state of potential solutions.

**Table 5.1 Knapsack capacity: 6,404,180 kg**

<b>Item ID</b>	<b>Item name</b>	<b>Weight (kg)</b>	<b>Value (\$)</b>	<b>Quantity</b>
1	Axe	32,252	68,674	19
2	Bronze coin	225,790	471,010	14
3	Crown	468,164	944,620	2
4	Diamond statue	489,494	962,094	9
5	Emerald belt	35,384	78,344	11
6	Fossil	265,590	579,152	6
7	Gold coin	497,911	902,698	4
8	Helmet	800,493	1,686,515	10
9	Ink	823,576	1,688,691	7
10	Jewel box	552,202	1,056,157	3
11	Knife	323,618	677,562	5
12	Long sword	382,846	833,132	13
13	Mask	44,676	99,192	15
14	Necklace	169,738	376,418	8
15	Opal badge	610,876	1,253,986	4
16	Pearls	854,190	1,853,562	9
17	Quiver	671,123	1,320,297	12
18	Ruby ring	698,180	1,301,637	17
19	Silver bracelet	446,517	859,835	16
20	Timepiece	909,620	1,677,534	7
21	Uniform	904,818	1,910,501	6
22	Venom potion	730,061	1,528,646	9
23	Wool scarf	931,932	1,827,477	3
24	Crossbow	952,360	2,068,204	1

25	Yesteryear book	926,023	1,746,556	7
26	Zinc cup	978,724	2,100,851	2

### 5.3.1 Real-value encoding at its core

Real-value encoding represents a gene as numeric values, strings, or symbols, and expresses potential solutions in the natural state respective to the problem. This encoding is used when potential solutions contain continuous values that cannot be encoded easily with binary encoding. As an example, because more than one item is available to be carried in the knapsack, each item index cannot indicate only whether the item is included; it must indicate the quantity of that item in the knapsack (figure 5.7).

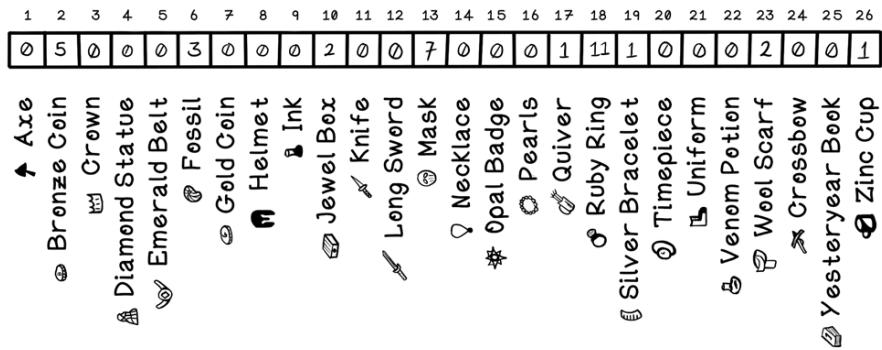


Figure 5.7 Example of real-value encoding

Because the encoding scheme has been changed, new crossover and mutation options become available. The crossover approaches discussed for binary encoding are still valid options to real-value encoding, but mutation should be approached differently.

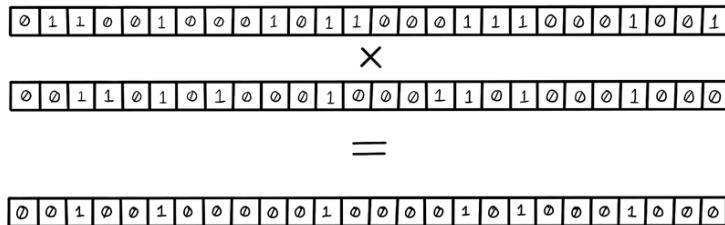
### 5.3.2 Arithmetic crossover: Reproduce with math

Arithmetic crossover involves an arithmetic operation to be computed by using each parent as variables in the expression. The result of applying an arithmetic operation using both parents is the new offspring. When we use this strategy with binary encoding, it is important to ensure that the result of the operation is still a valid chromosome. This technique is primarily designed for real-value encoding. While it is technically possible to adapt it for binary encoding (by rounding the result), it is standard practice to use it when genes represent continuous numbers, such as weights or coordinates. Figure 5.8 illustrates a multiplication operation for crossover.

---

**NOTE** Be wary: this approach can create offspring that are simply the “average” of their parents. In the early stages, this exploration is good. However, if the parents have found a precise, high-performing peak, mathematically averaging them might push the offspring away from that optimal solution, effectively “un-solving” the progress made by the parents.

---



**Figure 5.8 Example of arithmetic crossover**

### 5.3.3 Boundary mutation

In boundary mutation, a gene randomly selected from a real-value encoded chromosome is set randomly to a lower bound value or upper bound value. Given 26 genes in a chromosome, a random index is selected, and the value is set to either a minimum value or a maximum value. In figure 5.9, the original value happens to be 0 and will be adjusted to 6, which is the maximum for that item. The minimum and maximum can be the same for all indexes or set uniquely for each index if knowledge of the problem informs the decision. This approach attempts to evaluate the impact of individual genes on the chromosome.

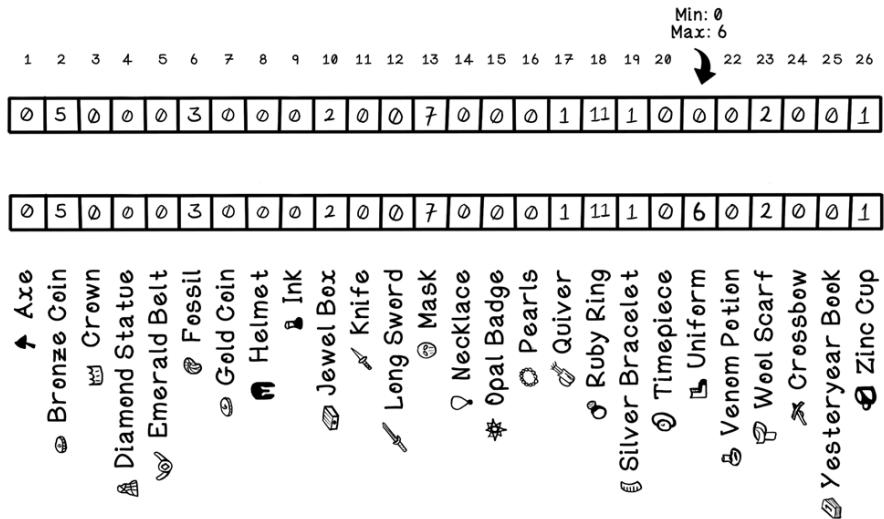


Figure 5.9 Example of boundary mutation

### 5.3.4 Arithmetic mutation

In arithmetic mutation, a randomly selected gene in a real-value-encoded chromosome is changed by adding or subtracting a small number. Note that although the example in figure 5.10 includes whole numbers, the numbers could be decimal numbers, including fractions.

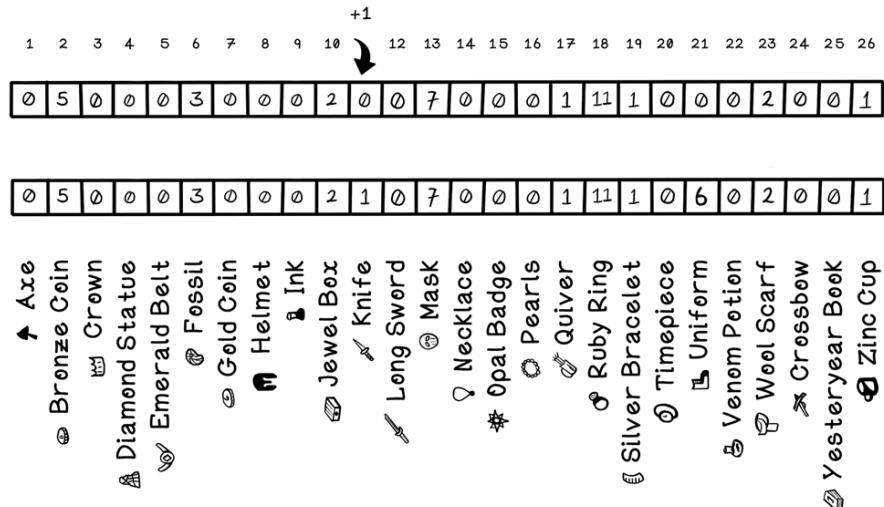


Figure 5.10 Example of arithmetic mutation

## 5.4 Order encoding: Working with sequences

We still have the same items as in the Knapsack Problem. We won't be determining the items that will fit into a knapsack; instead, all the items need to be processed in a refinery where each item is broken down to extract its source material. Perhaps the gold coin, silver bracelet, and other items are smelted to extract their source compounds. In this scenario, items are not selected to be included, they are all included.

To make things interesting, the refinery requires a steady rate of extraction, given the extraction time and the value of the item. We assume that the value of the refined material is more or less the same as the value of the item in its original state. The problem becomes an ordering problem. In what order should the items be processed to maintain a constant rate of value? Table 5.2 describes the items with their respective extraction times.

**Table 5.2 Factory value per hour: 600,000**

<b>Item ID</b>	<b>Item name</b>	<b>Weight (kg)</b>	<b>Value (\$)</b>	<b>Extraction time</b>
1	Axe	32,252	68,674	60
2	Bronze coin	225,790	471,010	30
3	Crown	468,164	944,620	45
4	Diamond statue	489,494	962,094	90
5	Emerald belt	35,384	78,344	70
6	Fossil	265,590	579,152	20
7	Gold coin	497,911	902,698	15
8	Helmet	800,493	1,686,515	20
9	Ink	823,576	1,688,691	10
10	Jewel box	552,202	1,056,157	40
11	Knife	323,618	677,562	15
12	Long sword	382,846	833,132	60
13	Mask	44,676	99,192	10
14	Necklace	169,738	376,418	20
15	Opal badge	610,876	1,253,986	60
16	Pearls	854,190	1,853,562	25
17	Quiver	671,123	1,320,297	30
18	Ruby ring	698,180	1,301,637	70
19	Silver bracelet	446,517	859,835	50
20	Timepiece	909,620	1,677,534	45
21	Uniform	904,818	1,910,501	5
22	Venom potion	730,061	1,528,646	5
23	Wool scarf	931,932	1,827,477	5
24	Crossbow	952,360	2,068,204	25

25	Yesteryear book	926,023	1,746,556	5
26	Zinc cup	978,724	2,100,851	10

### 5.4.1 Importance of the fitness function

With the change in the Knapsack Problem to the Refinery Problem, a key difference is the measurement of successful solutions. Because the factory requires a constant minimum rate of value per hour, the accuracy of the fitness function used becomes paramount to finding optimal solutions. In the Knapsack Problem, the fitness of a solution is trivial to compute, as it involves only two things: ensuring that the knapsack's weight limit is respected and summing the selected items' value. In the Refinery Problem, the fitness function must calculate the rate of value provided, given the extraction time for each item as well as the value of each item. This calculation is more complex, and an error in the logic of this fitness function directly influences the quality of solutions.

### 5.4.2 Order encoding at its core

Order encoding, also known as permutation encoding, represents a chromosome as a sequence of elements. Order encoding usually requires all elements to be present in the chromosome, which implies that standard crossover methods cannot be used, as they would create invalid solutions with duplicate or missing items.

Imagine a band planning a world tour. They need to visit London, Tokyo, and New York exactly once. Binary encoding doesn't work here—you can't visit "London" twice or skip "Tokyo" (1 or 0). You need to visit all of them, but the order changes the cost (flight prices). This is why we use Order Encoding: the sequence [London, Tokyo, New York] is different from [New York, London, Tokyo].

Specialized operators must be used to preserve the permutation constraints. Common techniques include Partially Mapped Crossover (PMX), Order Crossover (OX), and Cycle Crossover (CX), which are designed specifically to shuffle the order without breaking the list. Figure 5.11 depicts how a chromosome represents the order of processing of the available items.

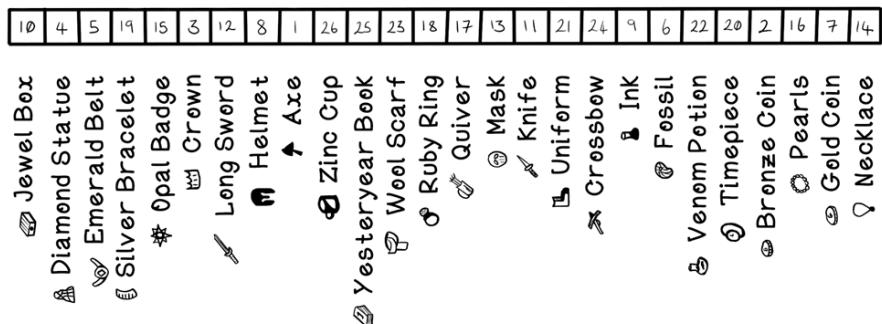


Figure 5.11 Example of order encoding

Another example in which order encoding is sensible is representing potential solutions to route optimization problems. Given a certain number of destinations, each of which must be visited at least once while minimizing the total distance traveled, the route can be represented as a string of the destinations in the order in which they are visited. We will use this example when covering swarm intelligence in chapter 6.

### 5.4.3 Order mutation: Order / permutation encoding

In order mutation, two randomly selected genes in an order-encoded chromosome swap positions, ensuring that all items remain in the chromosome while introducing diversity (figure 5.12).

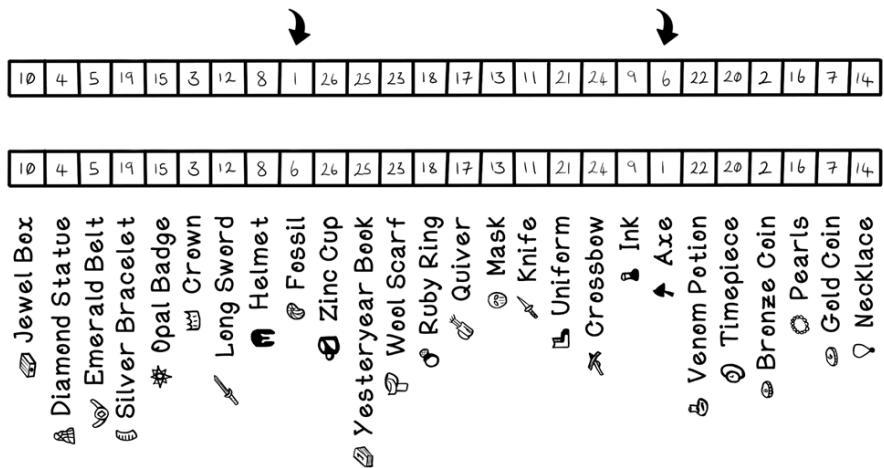


Figure 5.12 Example of order mutation

## 5.5 Tree encoding: Working with hierarchies

The last sections show that binary encoding is useful for selecting items from a set, real-value encoding is useful when real numbers are important to the solution, and order encoding is useful for determining priority and sequences. Suppose that the items in the Knapsack Problem are placed in packages to be shipped to homes around the town. Each delivery wagon can hold a specific volume. The requirement is to determine the optimal positioning of packages to minimize empty space in each wagon (table 5.3).

**Table 5.3 Wagon capacity: 1000 wide × 1000 high**

<b>Item ID</b>	<b>Item name</b>	<b>Weight (kg)</b>	<b>Value (\$)</b>	<b>W</b>	<b>H</b>
1	Axe	32,252	68,674	20	60
2	Bronze coin	225,790	471,010	10	10
3	Crown	468,164	944,620	20	20
4	Diamond statue	489,494	962,094	30	70
5	Emerald belt	35,384	78,344	30	20
6	Fossil	265,590	579,152	15	15
7	Gold coin	497,911	902,698	10	10
8	Helmet	800,493	1,686,515	40	50
9	Ink	823,576	1,688,691	5	10
10	Jewel box	552,202	1,056,157	40	30
11	Knife	323,618	677,562	10	30
12	Long sword	382,846	833,132	15	50
13	Mask	44,676	99,192	20	30
14	Necklace	169,738	376,418	15	20
15	Opal badge	610,876	1,253,986	5	5
16	Pearls	854,190	1,853,562	10	5
17	Quiver	671,123	1,320,297	30	70
18	Ruby ring	698,180	1,301,637	5	10
19	Silver bracelet	446,517	859,835	10	20
20	Timepiece	909,620	1,677,534	15	20
21	Uniform	904,818	1,910,501	30	40
22	Venom potion	730,061	1,528,646	15	15

23	Wool scarf	931,932	1,827,477	20	30
24	Crossbow	952,360	2,068,204	50	70
25	Yesteryear book	926,023	1,746,556	25	30
26	Zinc cup	978,724	2,100,851	15	25

In the interest of simplicity, let's assume that the wagon's volume is a two-dimensional rectangle and that the packages are rectangular rather than 3D boxes.

### 5.5.1 Tree encoding at its core

Tree encoding represents a chromosome as a tree of elements. Tree encoding is versatile for representing potential solutions in which the hierarchy of elements is important and/or required.

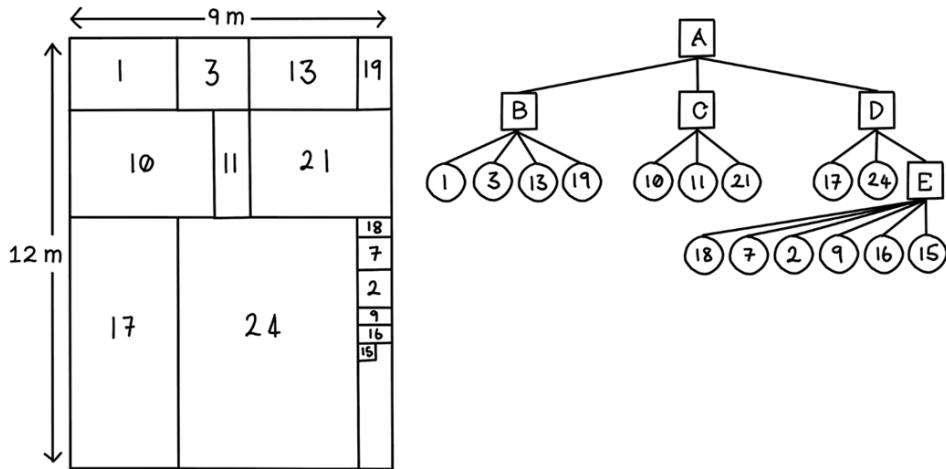
Think of this like a corporate organizational chart. The CEO (Root) makes a high-level decision. That decision branches down to VPs, then Managers, then Interns. You cannot simply swap a VP with an Intern without breaking the chain of command. Tree encoding preserves these strict parent-child relationships.

Tree encoding can even represent functions, which consist of a tree of expressions. This concept is the foundation of Genetic Programming (GP), a specialized field where the algorithm evolves executable programs or formulas rather than just static variables. As a result, tree encoding could be used to evolve program functions where the function itself solves a specific problem; the solution may work but look bizarre.

However, note that tree encoding is computationally costly. It is best suited for problems where the hierarchical structure or logic needs to evolve, rather than simple numeric optimization.

Here is an example in which tree encoding makes sense. We have a wagon with a specific height and width, and a certain number of packages must fit in the wagon. The goal is to fit the packages in the wagon so that empty space is minimized. A tree-encoding approach would work well in representing potential solutions to this problem.

In figure 5.13, the root node, node A, represents the packing of the wagon from top to bottom. Node B represents all packages horizontally, similarly to node C and node D. Node E represents packages packed vertically in its slice of the wagon.



**Figure 5.13 Example of a tree used to represent the Wagon Packing Problem**

### 5.5.2 Tree crossover: Inheriting portions of a tree

Tree crossover is similar to single-point crossover (chapter 4) in that a single point in the tree structure is selected and then the parts are exchanged and combined with copies of the parent individuals to create an offspring individual. The inverse process can be used to make a second offspring. The resulting children must be verified to be valid solutions that obey the constraints of the problem. More than one point can be used for crossover if using multiple points makes sense in solving the problem (figure 5.14).

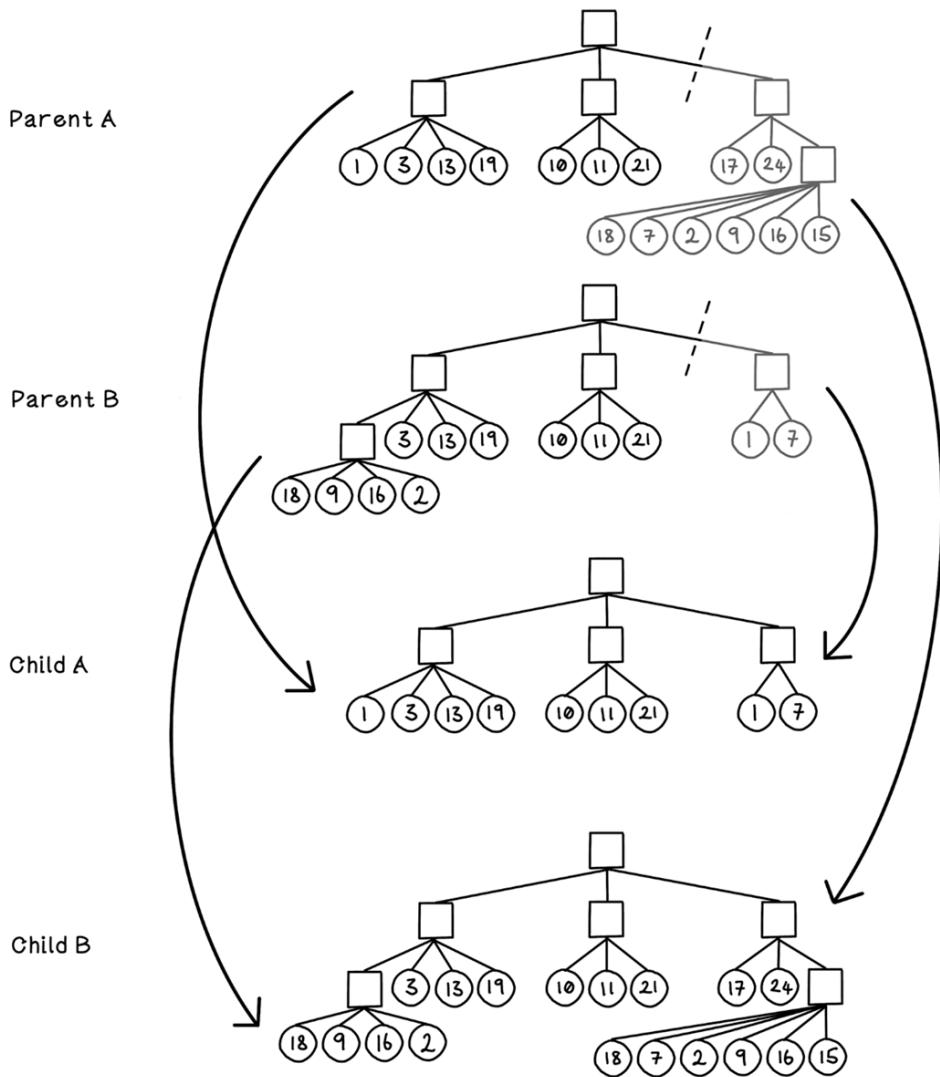
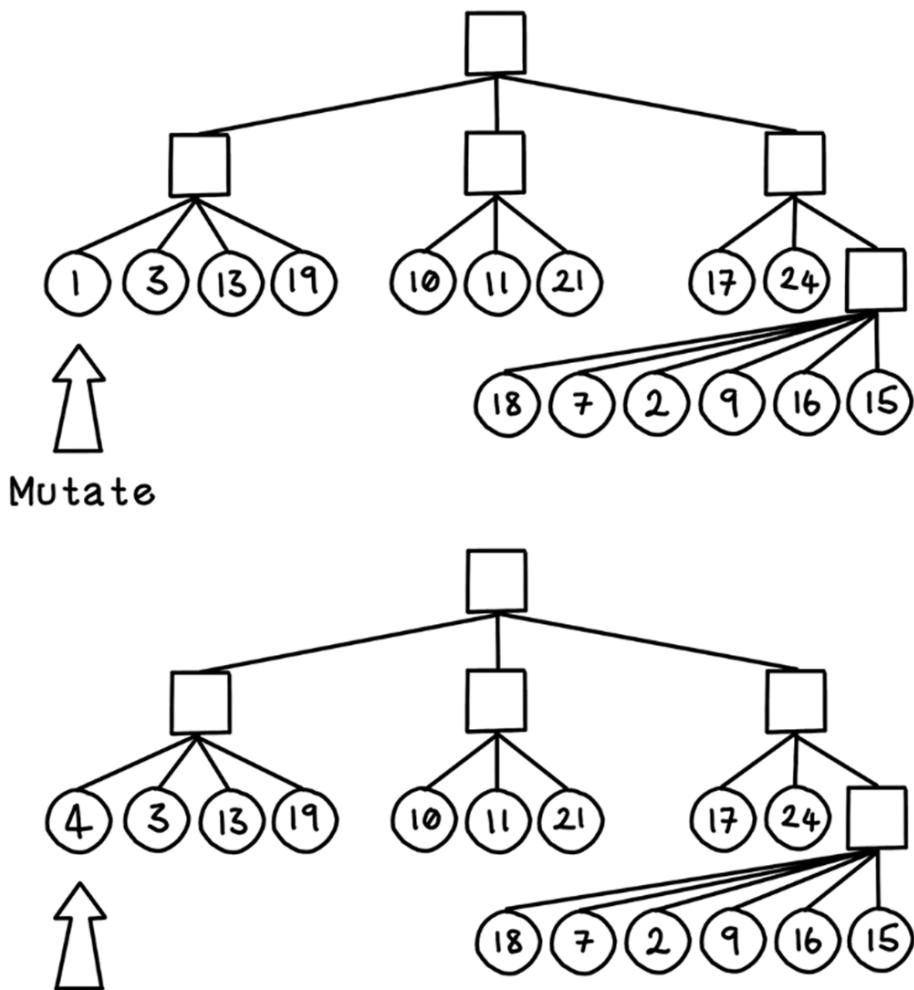


Figure 5.14 Example of tree crossover

### 5.5.3 Change node mutation: Changing the value of a node

In change node mutation, a randomly selected node in a tree-encoded chromosome is changed to a randomly selected valid object for that node. Given a tree representing an organization of items, we can change an item to another valid item (figure 5.15).



**Figure 5.15 Change node mutation in a tree**

This chapter and chapter 4 cover several encoding schemes, crossover schemes, and selection strategies. You could substitute your own approaches for these steps in your genetic algorithms if doing so makes sense for the problem you're solving.

## 5.6 Common types of evolutionary algorithms

This chapter focuses on the life cycle and alternative approaches for a genetic algorithm. Variations of the algorithm can be useful for solving different problems. Now that we have a grounding in how a genetic algorithm works, we'll look at these variations and possible use cases for them.

### 5.6.1 Genetic programming

Genetic programming follows a process similar to that of genetic algorithms but is used primarily to generate computer programs to solve problems. The process described in the previous section also applies here. The fitness of potential solutions in a genetic programming algorithm is how well the generated program solves a computational problem. With this in mind, we see that the tree-encoding method would work well here, because most computer programs are graphs consisting of nodes that indicate operations and processes. These trees of logic can be evolved, so the computer program will be evolved to solve a specific problem. One thing to note: these computer programs usually evolve to look like a mess of code that's difficult for people to understand and debug.

### 5.6.2 Evolutionary programming

Evolutionary programming is similar to genetic programming, but the potential solution is parameters for a predefined fixed computer program, not a generated computer program. If a program requires finely tuned inputs, and determining a good combination of inputs is difficult, a genetic algorithm can be used to evolve these inputs. The fitness of potential solutions in an evolutionary programming algorithm is determined by how well the fixed computer program performs based on the parameters encoded in an individual. Perhaps an evolutionary programming approach could be used to find the optimal architecture or hyperparameters (like the number of layers or learning rate) for an artificial neural network (Chapter 9), a technique known as Neuroevolution.

## 5.7 Glossary of evolutionary algorithm terms

Here is a useful glossary of evolutionary algorithms terms for future research and learning:

- *Allele*—The value of a specific gene in a chromosome
- *Chromosome*—A collection of genes that represents a possible solution
- *Individual*—A single chromosome in a population
- *Population*—A collection of individuals
- *Genotype*—The artificial representation of the potential solution population in the computation space
- *Phenotype*—The actual representation of the potential solution population in the real world
- *Generation*—A single iteration of the algorithm
- *Exploration*—The process of finding a variety of possible solutions, some of which may be good and some of which may be bad

- *Exploitation*—The process of honing in on good solutions and iteratively refining them
- *Fitness function*—A particular type of objective function
- *Objective function*—A function that attempts to maximize or minimize

## 5.8 More use cases for evolutionary algorithms

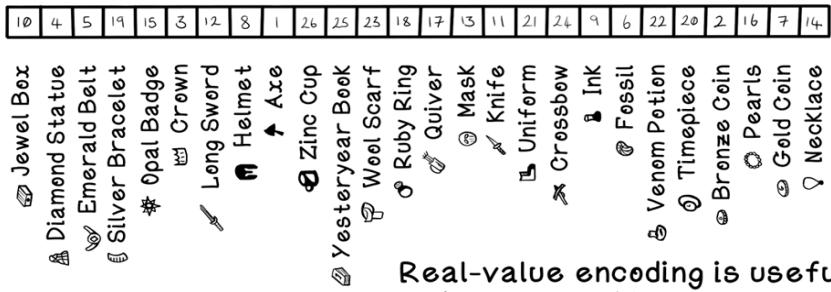
Some of the use cases for evolutionary algorithms are listed in chapter 4, but many more exist. The following use cases are particularly interesting because they use one or more of the concepts discussed in this chapter:

- *Adjusting weights in artificial neural networks*—Artificial neural networks are discussed later, in chapter 9, but a key concept is adjusting weights in the network to learn patterns and relationships in data. Several mathematical techniques adjust weights, but evolutionary algorithms are more efficient alternatives in the right scenarios.
- *Electronic circuit design*—Electronic circuits with the same components can be designed in many configurations. Some configurations are more efficient than others. If two components that work together often are closer together, this configuration may improve efficiency. Evolutionary algorithms can be used to evolve different circuit configurations to find the most optimal design.
- *Molecular structure simulation and design*—As in electronic circuit design, different molecules behave differently and have their own advantages and disadvantages. Evolutionary algorithms can be used to generate different molecular structures to be simulated and studied to determine their behavioral properties.

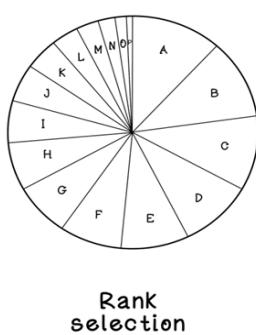
Now that we've been through the general genetic algorithm life cycle in chapter 4 and some advanced approaches in this chapter, you should be equipped to apply evolutionary algorithms in your contexts and solutions.

## 5.9 Summary of advanced evolutionary approaches

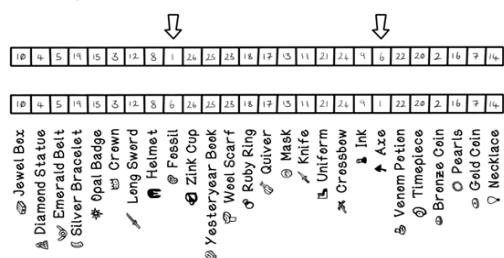
Genetic algorithms can be used to solve a multitude of optimization problems by leveraging different strategies



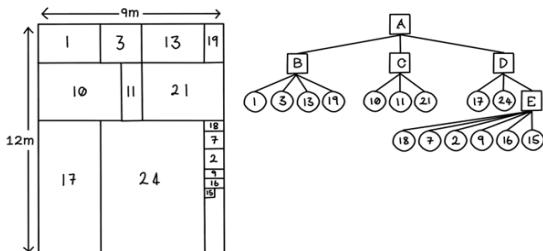
Order encoding is useful when the priority of sequence is important



Tweaking the parameters of the GA are useful to find good solutions efficiently based on the problem at hand



Tree encoding is useful when relationships and hierarchy are important



# 6 *Swarm intelligence: Ants*

## This chapter covers

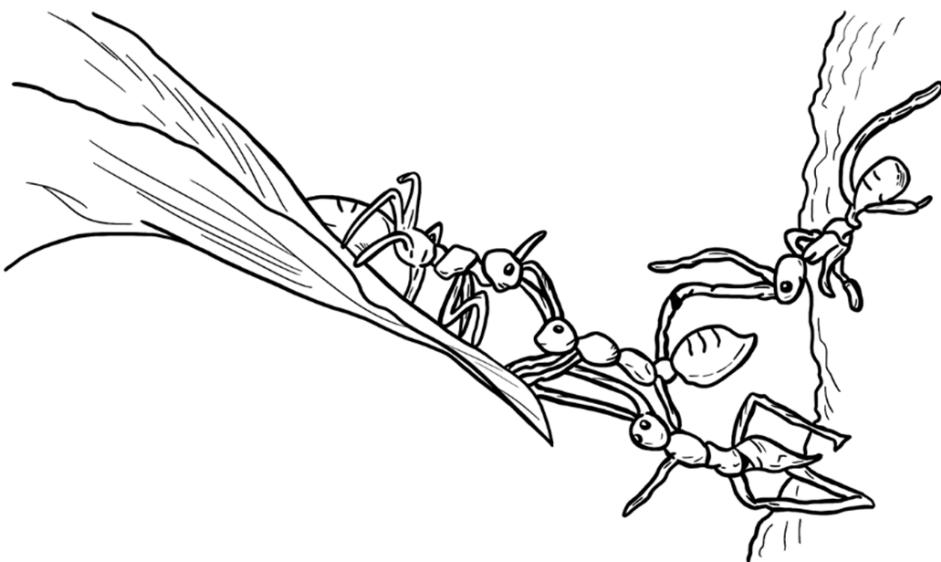
- Seeing and understanding what inspired swarm intelligence algorithms
- Solving problems with swarm intelligence algorithms
- Designing and implementing an ant colony optimization algorithm

## 6.1 What is swarm intelligence?

Swarm intelligence algorithms, much like the evolutionary algorithms discussed in Chapter 5, are a family of nature-inspired algorithms. However, while evolutionary algorithms mimic genetic reproduction, swarm intelligence mimics the collective behavior of animals. When we observe the world around us, we see many life forms that are seemingly primitive and unintelligent as individuals, yet exhibit intelligent emergent behavior when acting in groups.

An example of these life forms is ants. A single ant can carry 10 to 50 times its own body weight and run 700 times its body length per minute. These are impressive qualities; however, when acting in a group, that single ant can accomplish much more. In a group, ants are able to build colonies; find and retrieve food; warn other ants, show recognition to other ants, and use peer pressure to influence others in the colony. They achieve this through *pheromones*—essentially, dropping perfumes that signal other ants as they move. Other ants can sense these perfumes and change their behavior based on them. Ants have access to between 10 and 20 types of pheromones that can be used to communicate different intentions. Because individual ants use pheromones to indicate their intentions and needs, we observe complex emergent intelligent behavior when they are in groups.

Figure 6.1 shows an example of ants working as a team to create a bridge between two points to enable other ants to carry out tasks. These tasks may be to retrieve food or materials for their colony.

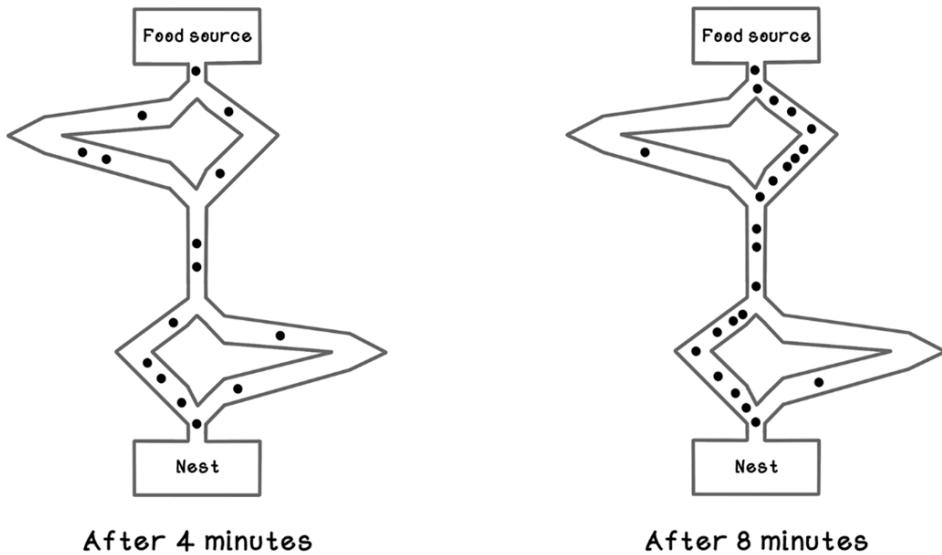


**Figure 6.1 A group of ants working together to cross a chasm**

An experiment based on real-life harvesting ants showed that they always converged to the shortest path between the nest and the food source. Figure 6.2 depicts the difference in the colony movement from the start to when ants have walked their paths and increased the pheromone intensity on those paths.

This outcome was observed in a classical asymmetric bridge experiment with real ants. Notice that the ants converge to the shortest path after just eight minutes.

Why does the shorter bridge win? It isn't magic; it is speed. An ant taking the short bridge can make two trips in the time it takes an ant on the long bridge to make just one. Because they make twice as many trips, they lay down twice as much pheromone per minute. This creates a snowball effect that pulls the rest of the colony toward the short path.



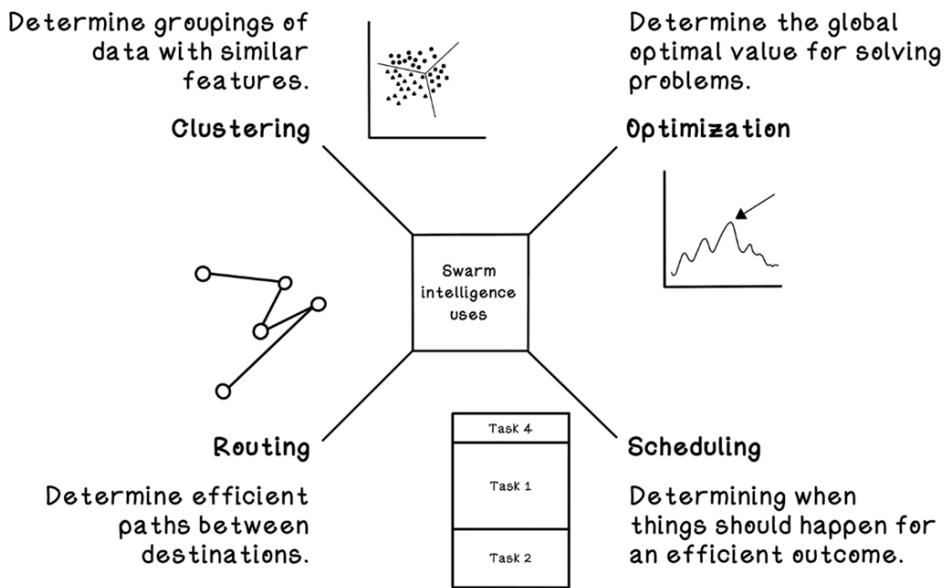
**Figure 6.2 Asymmetric bridge experiment**

Ant colony optimization (ACO) algorithms simulate the emergent behavior shown in this experiment. In the case of finding the shortest path, the algorithm converges to a similar state, as observed with real ants.

Swarm intelligence algorithms are powerful tools for solving optimization problems where the search space is vast, rugged, and finding an absolute best solution is mathematically difficult. These problems belong to the same broad class that genetic algorithms aim to solve, but the choice between the two often comes down to how the problem is encoded.

- Genetic Algorithms are typically better for discrete choices (like choosing which items to pack in a knapsack).
- Swarm Algorithms (like Particle Swarm Optimization) excel at continuous numeric problems (like tuning the exact floating-point weights of a neural network).

We dive into the technicalities of optimization problems in particle swarm optimization in chapter 7. Swarm intelligence is useful in several real-world contexts, some of which are represented in figure 6.3.



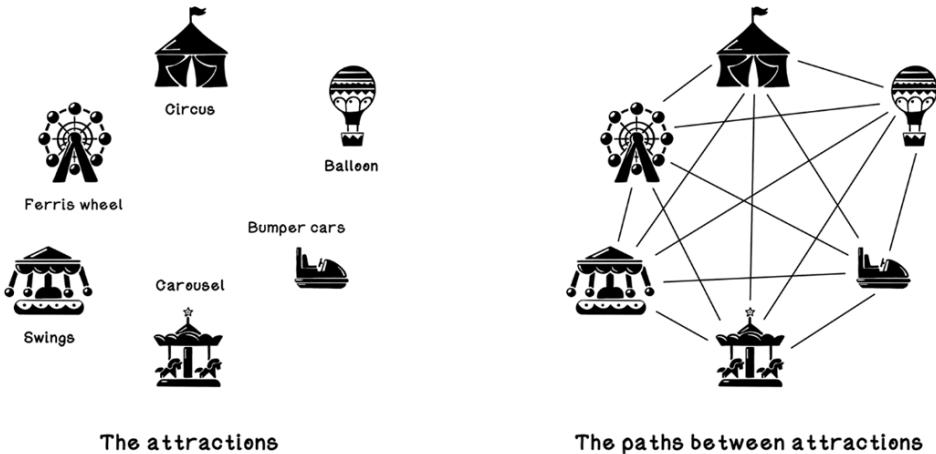
**Figure 6.3 Problems addressed by swarm optimization**

Given a general understanding of swarm intelligence in ants, the following sections explore specific implementations that are inspired by these concepts. The ant colony optimization algorithm is inspired by the behavior of ants moving between destinations, dropping pheromones, and acting on pheromones that they come across. The emergent behavior is ants converging to paths of least resistance.

## 6.2 Problems applicable to ant colony optimization

Imagine that we are visiting a carnival that has many attractions. Each attraction is located in a different area, with varying distances between them. Because we don't feel like wasting time and walking too much, we will attempt to find the shortest paths between all the attractions.

Figure 6.4 illustrates the attractions at a small carnival and the distances between them. Notice that taking different paths to the attractions have different total lengths of travel.

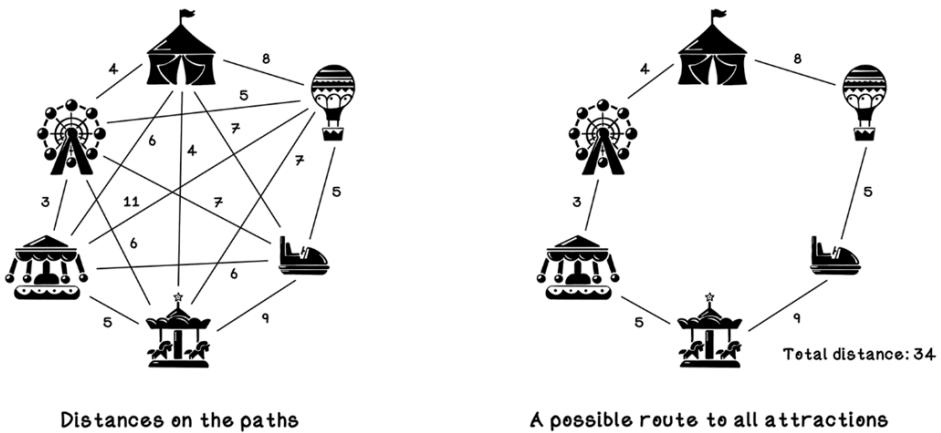


**Figure 6.4 Carnival attractions and paths between them**

The figure shows six attractions to visit, with 15 paths between them. This diagram should look familiar. This problem is represented by a fully connected graph, as described in chapter 2. The attractions are nodes or vertices, and the paths between attractions are edges. The following formula is used to calculate the number of edges in a fully connected graph. As the number of attractions gets larger, the number of edges explodes:

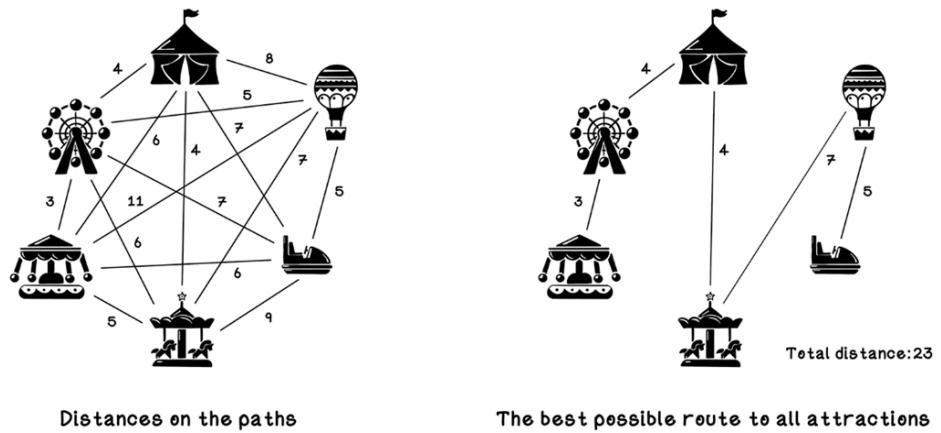
$$n(n-1)/2$$

Attractions have different distances between them. Figure 6.5 depicts the distance on each path between every attraction; it also shows a possible path between all attractions. Note that the lines in figure 6.5 showing the distances between the attractions are not drawn to scale.



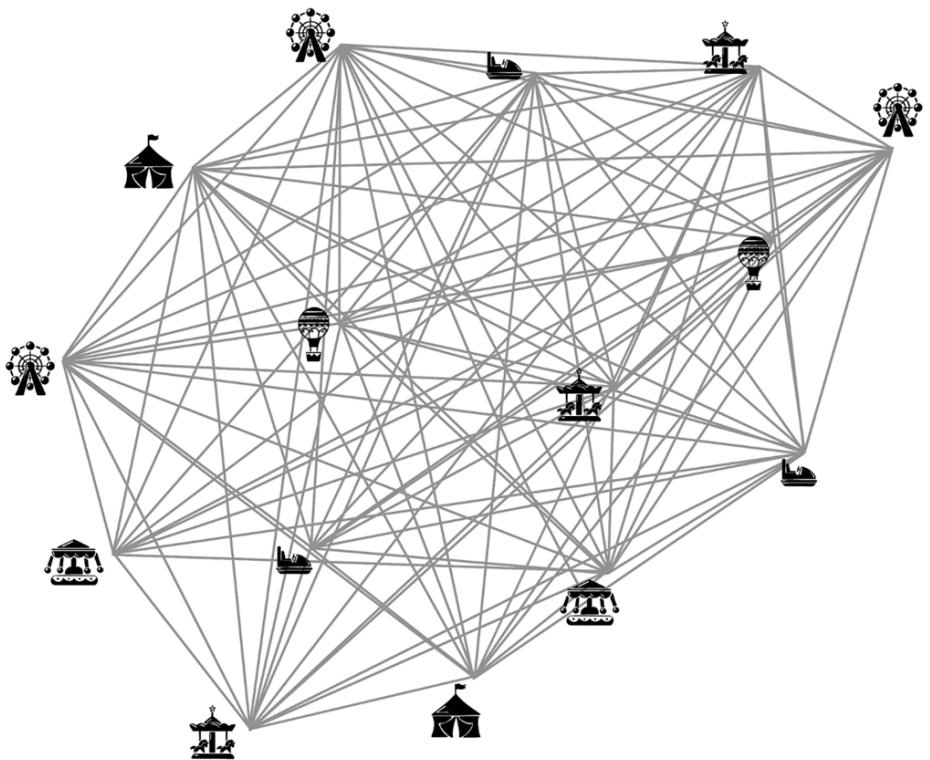
**Figure 6.5 Distances between attractions and a possible path**

If we spend some time analyzing the distances between all the attractions, we will find that figure 6.6 shows an optimal path between all the attractions. We visit the attractions in this sequence: Swings, Ferris wheel, Circus, Carousel, Balloons, and Bumper cars.



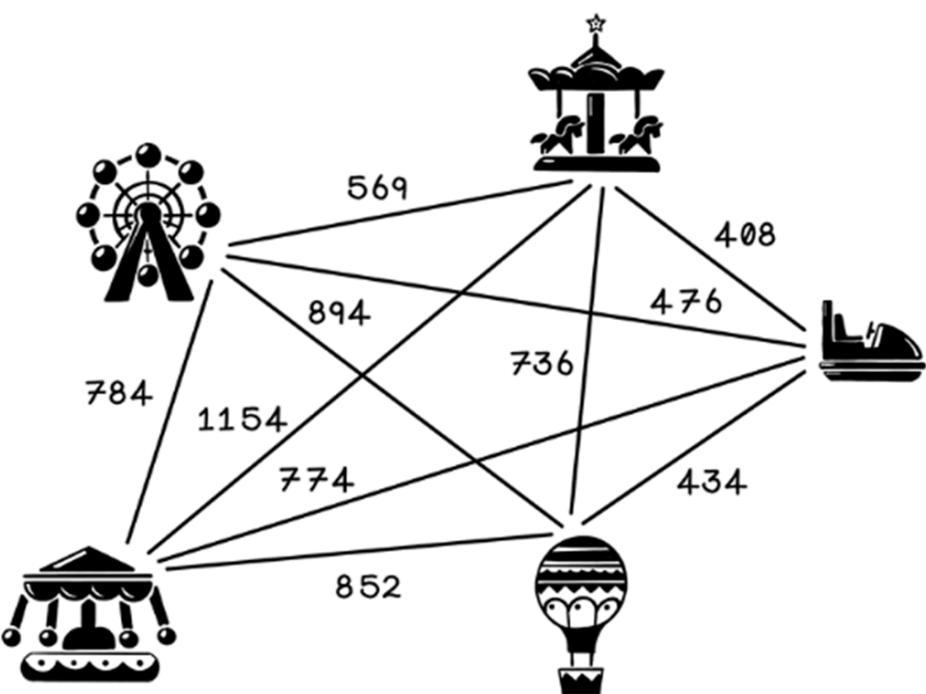
**Figure 6.6 Distances between attractions and an optimal path**

The small dataset with six attractions is trivial to solve by hand, but if we increase the number of attractions to 15, the number of possibilities increase substantially (figure 6.7). Suppose that the attractions are servers, and the paths are network connections. We need intelligent algorithms to solve these types of problems.

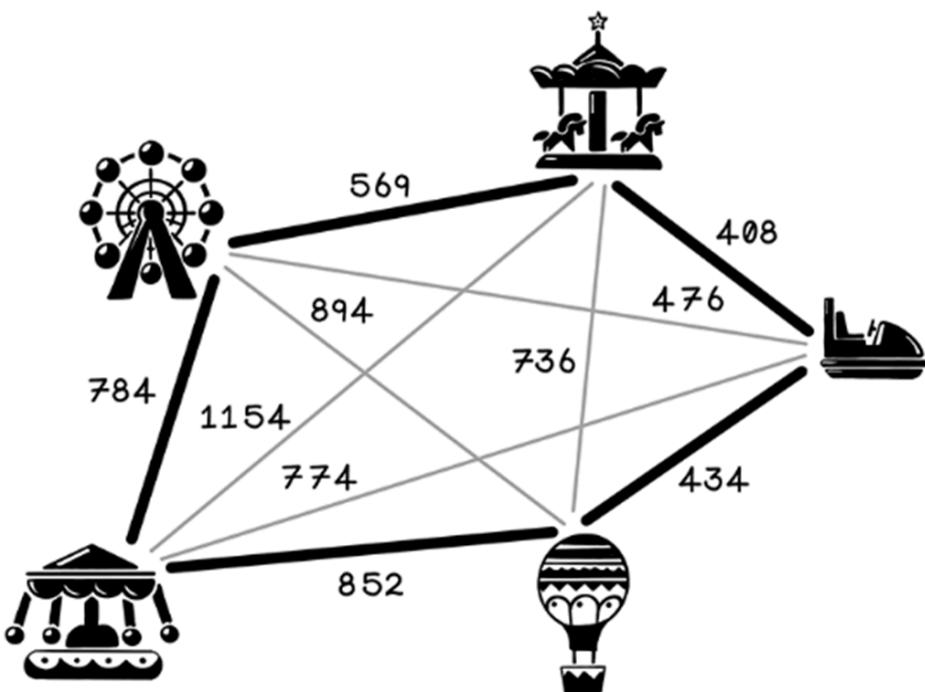


**Figure 6.7 A larger dataset of attractions and paths between them**

EXERCISE: FIND THE SHORTEST PATH IN THIS CARNIVAL CONFIGURATION BY HAND



## SOLUTION: FIND THE SHORTEST PATH IN THIS CARNIVAL CONFIGURATION BY HAND



One way to solve this problem computationally is to brute-force: every combination of tours (a tour is a sequence of visits in which every attraction is visited once) of the attractions is generated and evaluated, then the best tour is picked. Again, this solution may seem to be reasonable, but in a large dataset, this computation is expensive and time-consuming. A brute-force approach with 48 attractions runs for tens of hours before completing.

### 6.3 Representing state: What do paths and ants look like?

Given the Carnival Problem, we need to represent the data in a way that is suitable to be processed by the ant colony optimization algorithm. Because we have several attractions and all the distances between them, we can use a distance matrix to represent the problem space accurately and simply.

A *distance matrix* is a 2D array in which every index represents an entity; the related set is the distance between that entity and another entity. Similarly, each index in the list denotes a unique entity. This matrix is similar to the adjacency matrix that we dived into in chapter 2 (figure 6.8 and table 6.1).

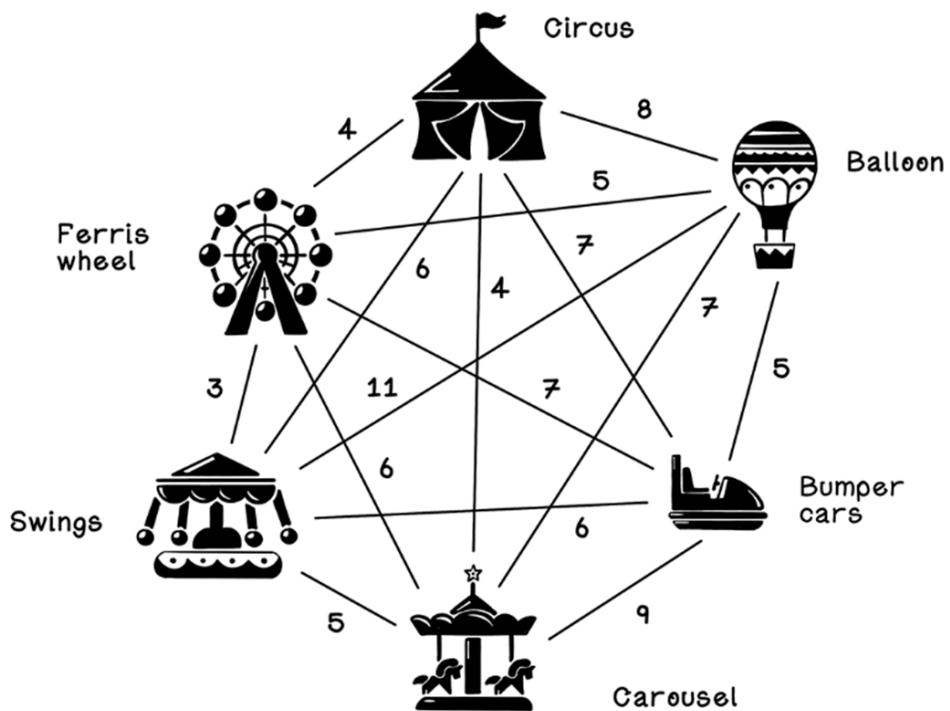


Figure 6.8 An example of the Carnival Problem

**Table 6.1 Distances between attractions**

	<b>Circus</b>	<b>Balloons</b>	<b>Bumper cars</b>	<b>Carousel</b>	<b>Swings</b>	<b>Ferris wheel</b>
Circus	0	8	7	4	6	4
Balloon	8	0	5	7	11	5
Bumper cars	7	5	0	9	6	7
Carousel	4	7	9	0	5	6
Swings	6	11	6	5	0	3
Ferris wheel	4	5	7	6	3	0

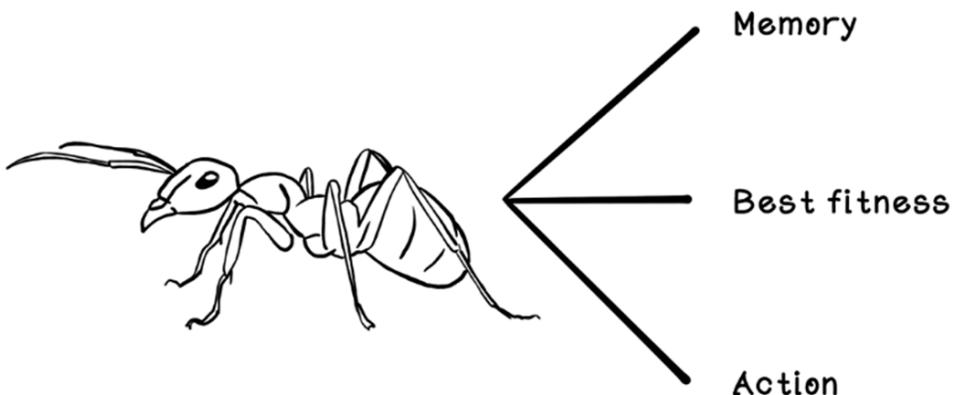
### PYTHON CODE SAMPLE

The distances between attractions can be represented as a distance matrix, an array of arrays in which a reference to  $x, y$  in the array references the distance between attractions  $x$  and  $y$ . Notice that the distance between the same attraction will be 0 because it's in the same position. This array can also be created programmatically by iterating through data from a file and creating each element:

```
attraction_distances = [
    [0, 8, 7, 4, 6, 4],
    [8, 0, 5, 7, 11, 5],
    [7, 5, 0, 9, 6, 7],
    [4, 7, 9, 0, 5, 6],
    [6, 11, 6, 5, 0, 3],
    [4, 5, 7, 6, 3, 0]
]
```

The next element to represent are the ants. Ants move to different attractions and leave pheromones behind. Ants also make a judgment about which attraction to visit next. Finally, ants have knowledge about their own total distance traveled. Here are the basic properties of an ant (figure 6.9):

- *Memory*—In the ACO algorithm, this is the list of attractions already visited.
- *Best fitness*—This is the shortest total distance traveled across all attractions.
- *Action*—Choose the next destination to visit, and drop pheromones along the way.



**Figure 6.9 Properties of an ant**

### PYTHON CODE SAMPLE

Although the abstract concept of an ant entails memory, best fitness, and action, specific data and functions are required to solve the Carnival Problem. To encapsulate the logic for an ant, we can use a class. When an instance of the ant class is initialized, an empty array is initialized to represent a list of attractions that the ant will visit. Furthermore, a random attraction will be selected to be the starting point for that specific ant:

```
class Ant:
    def __init__(self, attraction_count):
        self.visited_attractions = []

        start_node = random.randint(0, attraction_count - 1)
        self.visited_attractions.append(start_node)
```

The ant class also contains several functions used for ant movement. The `visit_*` functions are used to determine to which attraction the ant moves to next. The `visit_attraction` function generates a random chance of visiting a random attraction. In this case, `visit_random_attraction` is called; otherwise, `roulette_wheel_selection` is used with a calculated list of probabilities. More details are coming up in the next section:

```

class Ant:
    def visit_attraction(self, pheromone_trails):
        pass

    def visit_random_attraction(self):
        pass

    def visit_probabilistic_attraction(self, pheromone_trails):
        pass

    def roulette_wheel_selection(self, probabilities):
        pass

    def get_distance_travelled(self):
        pass

```

Last, the `get_distance_traveled` function is used to calculate the total distance traveled by a specific ant, using its list of visited attractions. This distance must be minimized to find the shortest path and is used as the fitness for the ants:

```

def get_distance_travelled(self):
    total_distance = 0

    for i in range(1, len(self.visited_attractions)):
        previous_attraction = self.visited_attractions[i - 1]
        current_attraction = self.visited_attractions[i]

        dist = attraction_distances[previous_attraction][current_attraction]
#A
        total_distance += dist

    return total_distance

```

#A Retrieve distance from global matrix

The final data structure to design is the concept of pheromone trails. Similarly to the distances between attractions, pheromone intensity on each path can be represented as a distance matrix, but instead of containing distances, the matrix contains pheromone intensities. In figure 6.10, thicker lines indicate more-intense pheromone trails. Table 6.2 describes the pheromone trails between attractions.

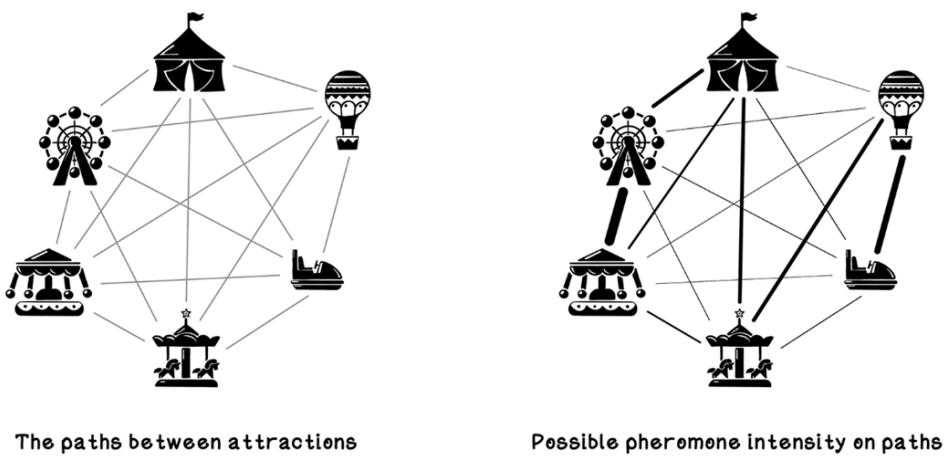


Figure 6.10 Example pheromone intensity on paths

Table 6.2 Pheromone intensity between attractions

	Circus	Balloons	Bumper cars	Carousel	Swings	Ferris wheel
Circus	0	2	0	8	6	8
Balloon	2	0	10	8	2	2
Bumper cars	2	10	0	0	2	2
Carousel	8	8	2	0	2	2
Swings	6	2	2	2	0	10
Ferris wheel	8	2	2	2	10	0

## 6.4 The ant colony optimization algorithm life cycle

Now that we understand the data structures required, we can dive into the workings of the ant colony optimization algorithm. The approach in designing an ACO algorithm is based on the problem space being addressed. Each problem has a unique context and a different domain in which data is represented, but the principles remain the same.

That said, let's look into how an ant colony optimization algorithm can be configured to solve the Carnival Problem. The general life cycle of such an algorithm is as follows:

- *Initialize the pheromone trails*—Create the concept of pheromone trails between attractions, and initialize their intensity values.
- *Set up the population of ants*—Create a population of ants in which each ant starts at a different attraction.
- *Choose the next visit for each ant*—Choose the next attraction to visit for each ant until each ant has visited all attractions once.
- *Update the pheromone trails*—Update the intensity of pheromone trails based on the ants' movements on them, as well as factor in evaporation of pheromones.
- *Update the best solution*—Update the best solution, given the total distance covered by each ant.
- *Determine the stopping criteria*—The process of ants visiting attractions repeats for several iterations. One iteration is every ant visiting all attractions once. The stopping criterion determines the total number of iterations to run. More iterations allow ants to make better decisions based on the pheromone trails.

Figure 6.11 describes the general life cycle of the ant colony optimization algorithm.

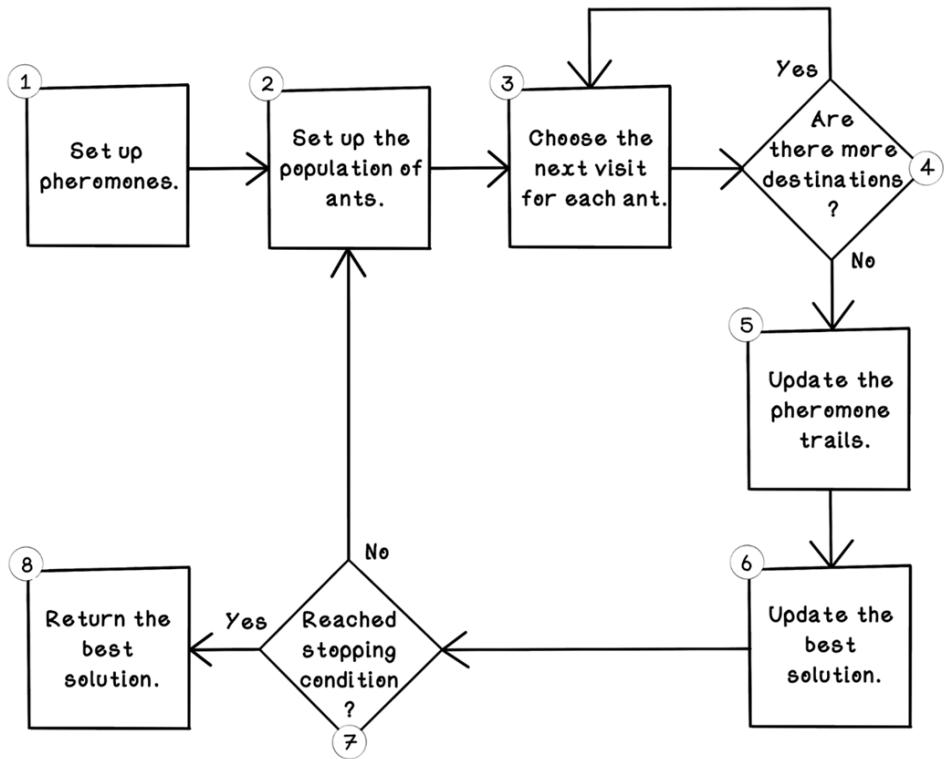
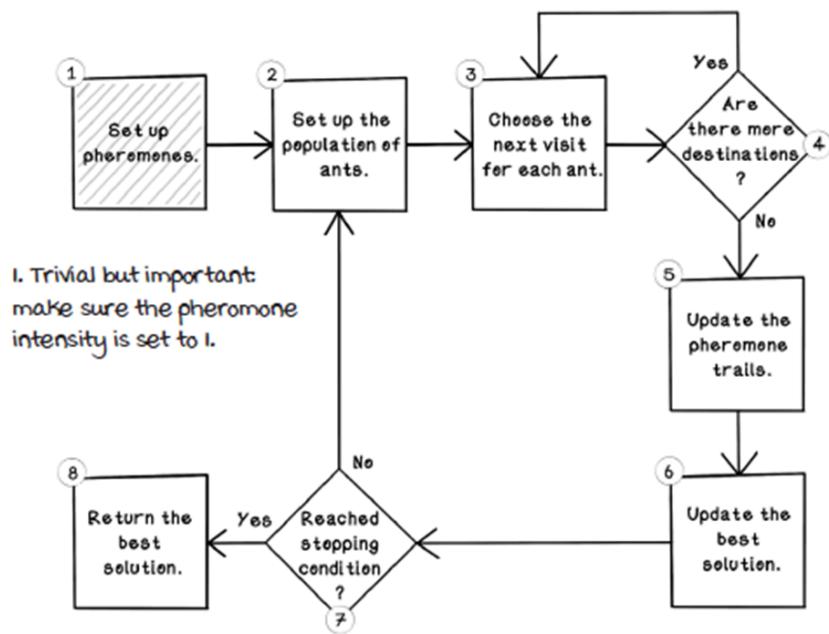


Figure 6.11 The ant colony optimization algorithm life cycle

#### 6.4.1 Initialize the pheromone trails

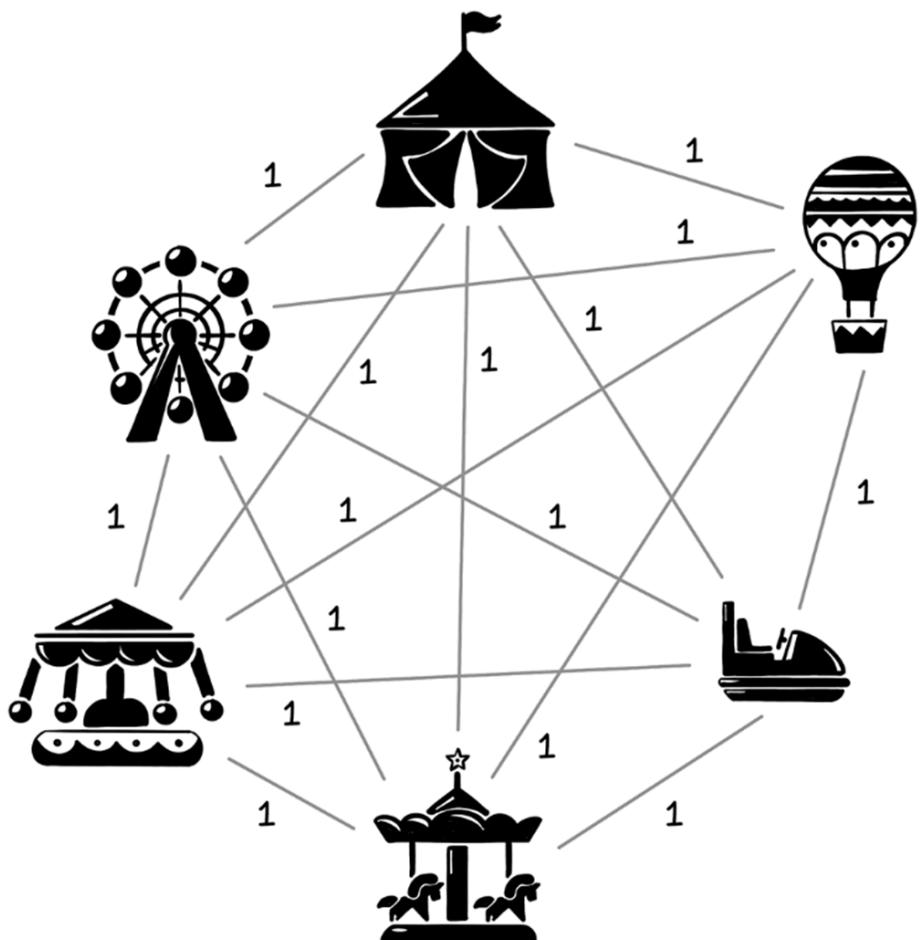
The first step in the ant colony optimization algorithm is to initialize the pheromone trails. Because no ants have walked on the paths between attractions yet, the pheromone trails will be initialized to 1. When we set all pheromone trails to 1, no trail has any advantage over the others. The important aspect is defining a reliable data structure to contain the pheromone trails, which we look at next (figure 6.12).



**Figure 6.12 Set up the pheromones.**

This concept can be applied to other problems in which instead of distances between locations, the pheromone intensity is defined by another heuristic.

In figure 6.13, the heuristic is the distance between two destinations.



**Pheromones initialize at 1**

Figure 6.13 Initialization of pheromones

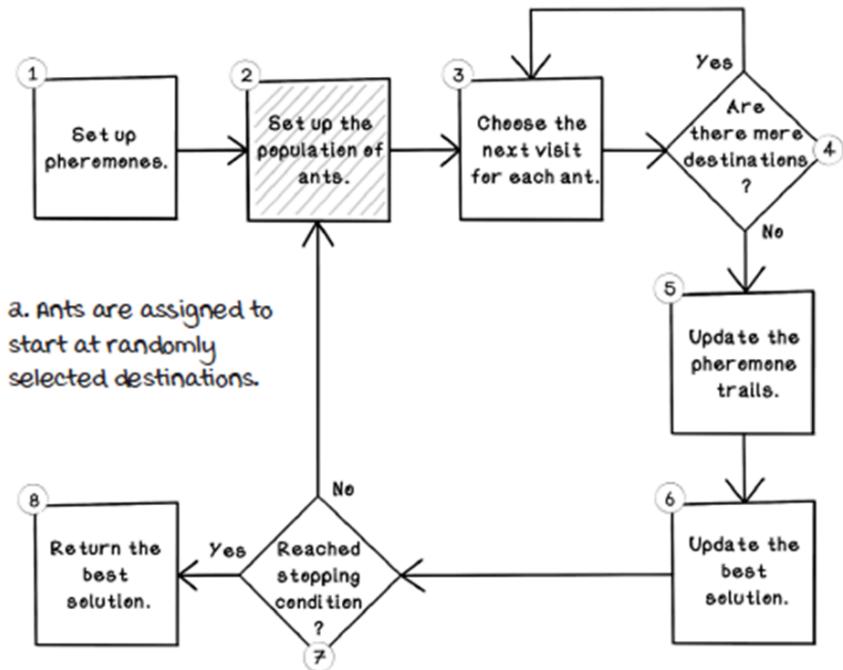
## PYTHON CODE SAMPLE

Similarly to the attraction distances, the pheromone trails can be represented by a distance matrix, but referencing  $x$ ,  $y$  in this array provides the pheromone intensity on the path between attractions  $x$  and  $y$ . The initial pheromone intensity on every path is initialized to 1. Values for all paths should initialize with the same number to prevent biasing any paths from the start:

```
pheromone_trails = [  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 1],  
    [1, 1, 1, 1, 1, 1]  
]
```

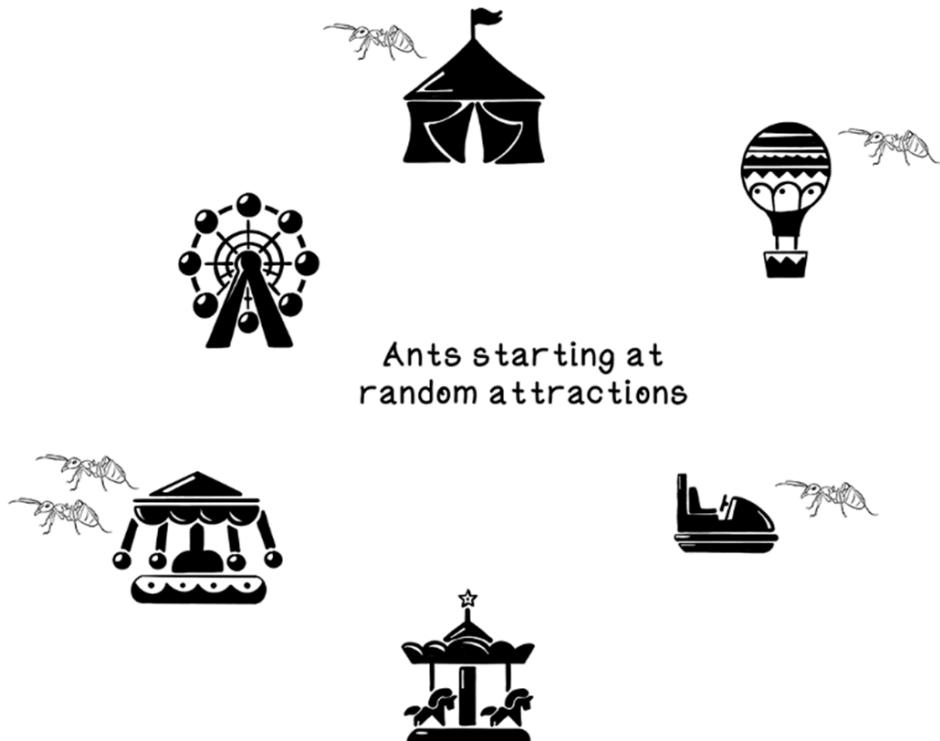
### 6.4.2 Set up the population of ants

The next step of the ACO algorithm is creating a population of ants that will move between the attractions and leave pheromone trails between them (figure 6.14).



**Figure 6.14** Set up the population of ants.

Ants will start at randomly assigned attractions (figure 6.15)—at a random point in a potential sequence because the ant colony optimization algorithm can be applied to problems in which actual distance doesn't exist. After touring all the destinations, ants are set to their respective starting points.



**Figure 6.15 Ants start at random attractions.**

We can adapt this principle to a different problem. In a task-scheduling problem, each ant starts at a different task.

### PYTHON CODE SAMPLE

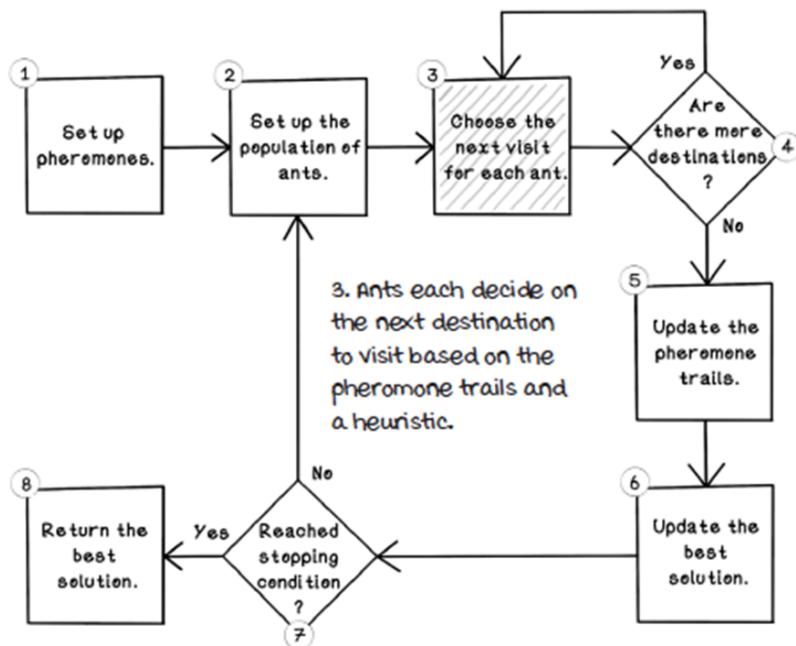
Setting up the colony of ants includes initializing several ants and appending them to a list where they can be referenced later. Remember that the initialization function of the ant class chooses a random attraction to start at:

```
def setup_ants(self, number_of_ants_factor):
    number_of_ants = round(ATTRACTION_COUNT * number_of_ants_factor)
    self.ant_colony.clear()
    for i in range(0, number_of_ants):
        self.ant_colony.append(Ant())
```

### 6.4.3 Choose the next visit for each ant

Ants need to select the next attraction to visit. They visit new attractions until they have visited all attractions once; this is a tour. Ants choose the next destination based on two factors (figure 6.16):

- *Pheromone intensities*—The pheromone intensity on all available paths
- *Heuristic value*—A result from a defined heuristic for all available paths, which is the distance of the path between attractions in the carnival example



**Figure 6.16 Choose the next visit for each ant.**

Ants will not travel to destinations they have already visited. If an ant has already visited the bumper cars, it will not travel to that attraction again in the current tour.

## THE “RANDOM” NATURE OF ANTS

The ant colony optimization algorithm has an element of randomness. The intention is to allow ants the possibility of exploring less-optimal immediate paths, which might result in a better overall tour distance.

In standard Ant Colony Optimization, ants don't simply flip a coin to decide whether to explore. Instead, they make a probabilistic decision based on two factors:

- *Pheromone Strength*: How “popular” the path is.
- Distance Heuristic: How short the path looks.

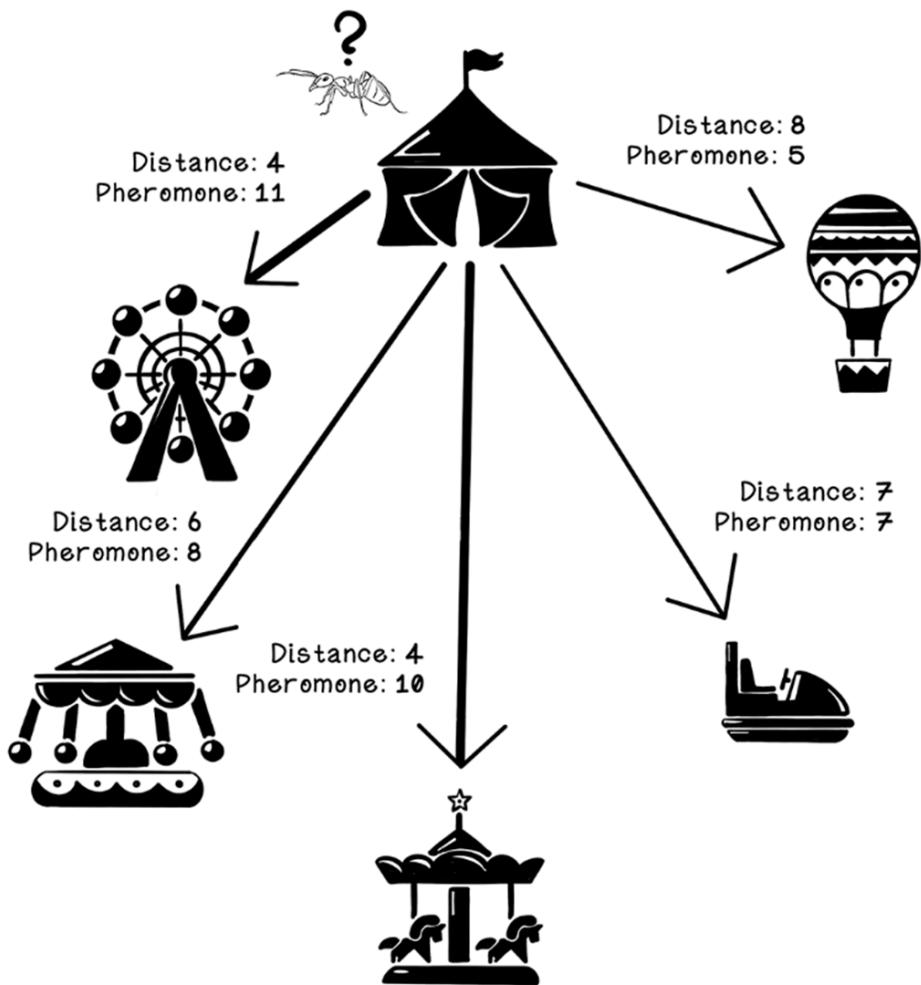
Think of this like a Roulette Wheel. A path with strong pheromones and a short distance gets a huge slice of the wheel. A path with weak pheromones gets a tiny slice. The ant spins the wheel to choose its next step. It is likely to pick the best path, but sometimes the ball lands on the tiny slice, allowing the ant to explore new routes naturally.

## **SELECTING DESTINATION BASED ON A HEURISTIC**

When an ant faces the decision of choosing the next destination that is not random, it determines the pheromone intensity on that path and the heuristic value by using the following formula:

$$\frac{(\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b}{\sum_{\substack{\text{available} \\ \text{destinations}}} ((\text{pheromones on path } n)^a * (1 / \text{heuristic for path } n)^b)}$$

After it applies this function to every possible path toward its respective destination, the ant selects the destination with the best overall value to travel to. Figure 6.17 illustrates the possible paths from the circus with their respective distances and pheromone intensities.



**Figure 6.17 Example of possible paths from the circus**

Let's work through the formula to demystify the calculations that are happening and how the results affect decision-making (figure 6.18).

$$\frac{(\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b}{\text{Pheromone influence} \quad \text{Heuristic influence}}$$

**Figure 6.18 The pheromone influence and heuristic influence of the formula**

The variables *alpha* (*a*) and *beta* (*b*) are used to give greater weight to either the pheromone influence or the heuristic influence. These variables can be adjusted to balance the ant's judgment between making a move based on what it knows versus pheromone trails, which represent what the colony knows about that path. These parameters are defined up front and are usually not adjusted while the algorithm runs.

- Pheromone (Alpha) is social pressure: "Look at that long line of people! That ride must be amazing". This is trusting the collective wisdom of the colony.
- Heuristic (Beta) is greedy logic: "But that other ride is right next door and I'm bored right now". This is trusting the immediate distance.

If Alpha is too high: The ants blindly follow the crowd even if it's long. If Beta is too high: The ants become "greedy loners" and ignore the collective wisdom, turning the algorithm into a simple greedy search.

The following example works through each path starting at the circus and calculates the probabilities of moving to each respective attraction.

- *a* (*alpha*) is set to 1.
- *b* (*beta*) is set to 2.

Because *b* is greater than *a*, the heuristic influence is favored in this example.

Let's work through an example of the calculations used to determine the probability of choosing a specific path (figure 6.19).

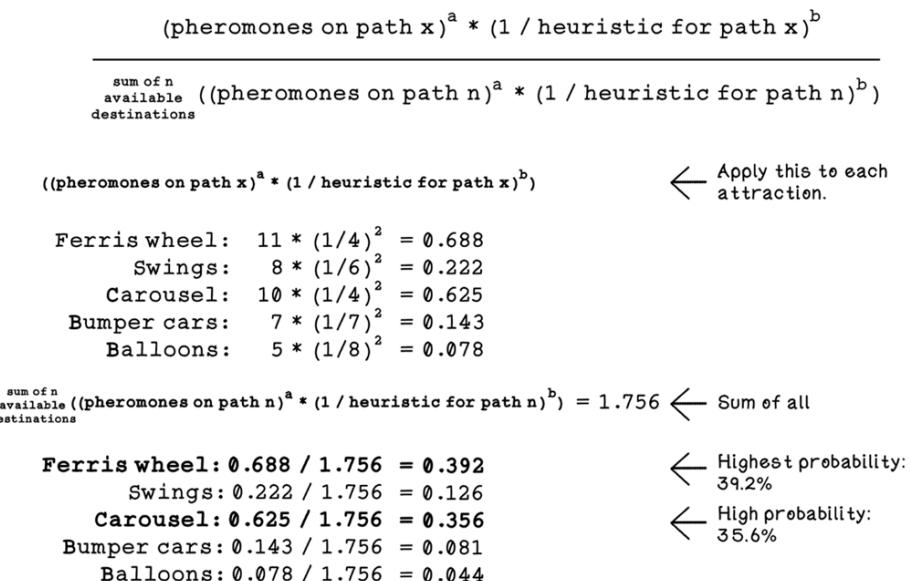


Figure 6.19 Probability calculations for paths

After applying this calculation, given all the available destinations, the ant is left with the options shown in figure 6.20.

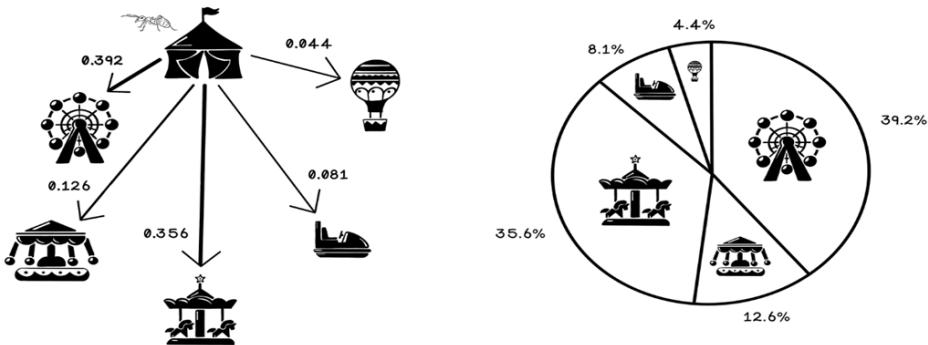


Figure 6.20 The final probability of each attraction being selected

Remember that only the available paths are considered; these paths have not been explored yet. Figure 6.21 illustrates the possible paths from the circus, excluding the Ferris wheel, because it's been visited already. Figure 6.22 shows probability calculations for paths.

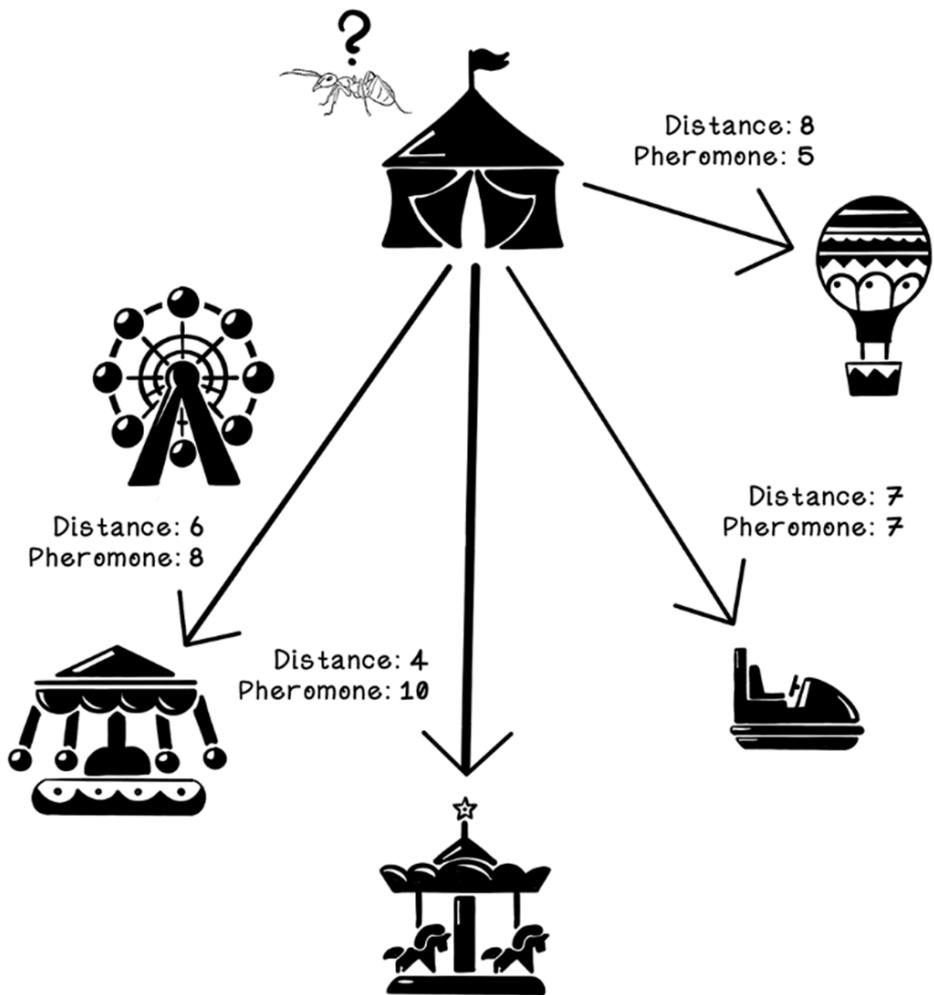


Figure 6.21 Example of possible paths from the circus, excluding visited attractions

$$\frac{(\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b}{\sum_{\text{available destinations}} ((\text{pheromones on path } n)^a * (1 / \text{heuristic for path } n)^b)}$$

Swings:  $8 * (1/6)^2 = 0.222$   
 Carousel:  $10 * (1/4)^2 = 0.625$   
 Bumper cars:  $7 * (1/7)^2 = 0.143$   
 Balloons:  $5 * (1/8)^2 = 0.078$

$\sum_{\text{available destinations}} ((\text{pheromones on path } n)^a * (1 / \text{heuristic for path } n)^b) = 1.068 \leftarrow \text{Sum of all}$

Swings:  $0.222 / 1.068 = 0.208$   
**Carousel:  $0.625 / 1.068 = 0.585$**   
 Bumper cars:  $0.143 / 1.068 = 0.134$   
 Balloons:  $0.078 / 1.068 = 0.073$

$\leftarrow$  Highest probability:  
 58.5%

Figure 6.22 Probability calculations for paths

The ant's decision now looks like figure 6.23.

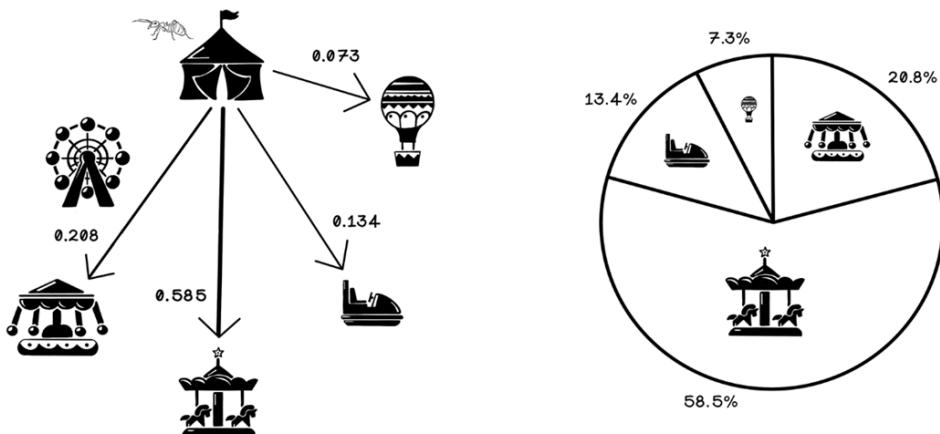


Figure 6.23 The final probability of each attraction being selected

## PYTHON CODE SAMPLE

The code for calculating the probabilities of visiting the possible attractions is closely aligned with the mathematical functions that we have worked through. Some interesting aspects of this implementation include:

- *Determining the available attractions to visit*—Because the ant would have visited several attractions, it should not return to those attractions. The `possible_attractions` array stores this value by removing `visited_attractions` from the complete list of attractions: `all_attractions`.
- *Using three variables to store the outcome of the probability calculations*—`possible_indexes` stores the attraction indexes; `possible_probabilities` stores the probabilities for the respective index; and `total_probabilities` stores the sum of all probabilities, which should equal 1 when the function is complete. These three data structures could be represented by a class for a cleaner code convention.

```
def visit_probabilistic_attraction(self, pheromone_trails):
    current_attraction = self.visited_attractions[-1]
    all_attractions = set(range(0, ATTRACTION_COUNT))
    possible_attractions = all_attractions - set(self.visited_attractions)
    possible_indexes = []
    possible_probabilities = []
    total_probabilities = 0
    for attraction in possible_attractions:
        possible_indexes.append(attraction)
        pheromones_on_path = math.pow(pheromone_trails[current_attraction]
[attraction], ALPHA)
        heuristic_for_path = math.pow(1 /
attraction_distances[current_attraction][attraction], BETA)
        probability = pheromones_on_path * heuristic_for_path
        possible_probabilities.append(probability)
        total_probabilities += probability
    possible_probabilities = [probability / total_probabilities for
probability in possible_probabilities]
    return [possible_indexes, possible_probabilities,
len(possible_attractions)]
```

We meet roulette-wheel selection again. The roulette-wheel selection function takes the possible probabilities and attraction indexes as input. It generates a list of slices, each of which includes the index of the attraction in element 0, the start of the slice in index 1, and the end of the slice in index 2. All slices contain a start and end between 0 and 1. A random number between 0 and 1 is generated, and the slice that it falls into is selected as the winner:

```
def roulette_wheel_selection(possible_indexes, possible_probabilities):
    slices = []
    total = 0

    for i in range(len(possible_indexes)): #A
        start = total #B
        end = total + possible_probabilities[i] #B
        #B
        slices.append([possible_indexes[i], start, end]) #B

        total += possible_probabilities[i]

    spin = random.random()

    for slice_data in slices: #C
        attraction_index = slice_data[0]
        lower_bound = slice_data[1]
        upper_bound = slice_data[2]

        if lower_bound <= spin < upper_bound: #D
            return attraction_index

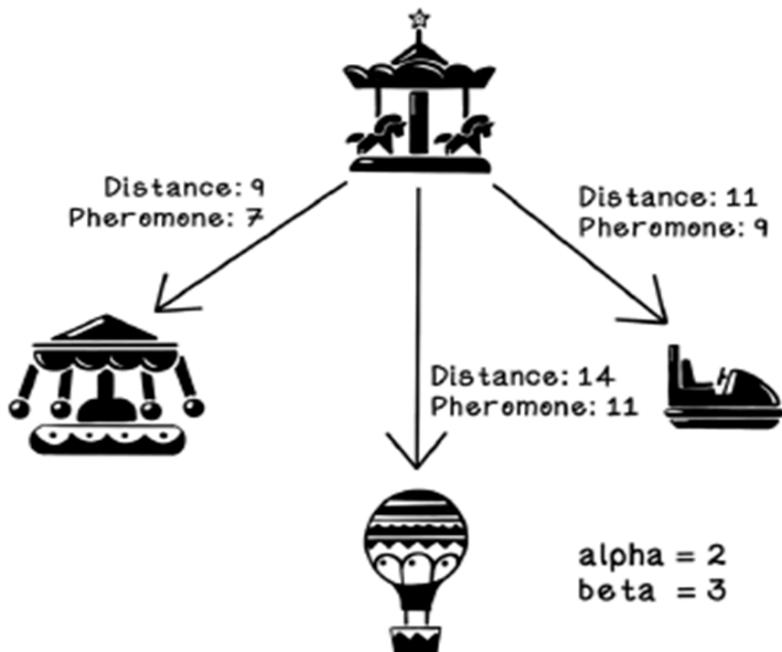
    return possible_indexes[-1] #E

#A Build the cumulative probability wheel
#B Create a slice: [attraction_index, range_start, range_end]
#C Find which slice the random spin lands in
#D Check if spin is within the bounds of this slice
#E Fallback for rounding errors (returns the last element)
```

Now that we have probabilities of selecting the different attractions to visit, we will use roulette-wheel selection.

To recap, roulette-wheel selection (from chapters 3 and 4) gives different possibilities portions of a wheel based on their fitness. Then the wheel is “spun,” and an individual is selected. A higher fitness gives an individual a larger slice of the wheel, as shown in figure 6.23 earlier in this chapter. The process of choosing an attraction and visiting it continues for every ant until each one has visited all the attractions once.

EXERCISE: DETERMINE THE PROBABILITIES OF VISITING THE ATTRACTIONS WITH THE FOLLOWING INFORMATION



**SOLUTION: DETERMINE THE PROBABILITIES OF VISITING THE ATTRACTIONS WITH THE FOLLOWING INFORMATION**

$$(\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b$$

$$\frac{\sum_{\substack{\text{available} \\ \text{destinations}}} ((\text{pheromones on path } n)^a * (1 / \text{heuristic for path } n)^b)}{\sum_{\substack{\text{available} \\ \text{destinations}}} ((\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b)}$$

$$((\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b)$$

$$\text{Swings: } 7^2 * (1/9)^3 = 0.067$$

$$\text{Bumper cars: } 9^2 * (1/11)^3 = 0.061$$

$$\text{Balloons: } 11^2 * (1/14)^3 = 0.044$$

$$\frac{\sum_{\substack{\text{available} \\ \text{destinations}}} ((\text{pheromones on path } n)^a * (1 / \text{heuristic for path } n)^b)}{\sum_{\substack{\text{available} \\ \text{destinations}}} ((\text{pheromones on path } x)^a * (1 / \text{heuristic for path } x)^b)} = 0.172$$

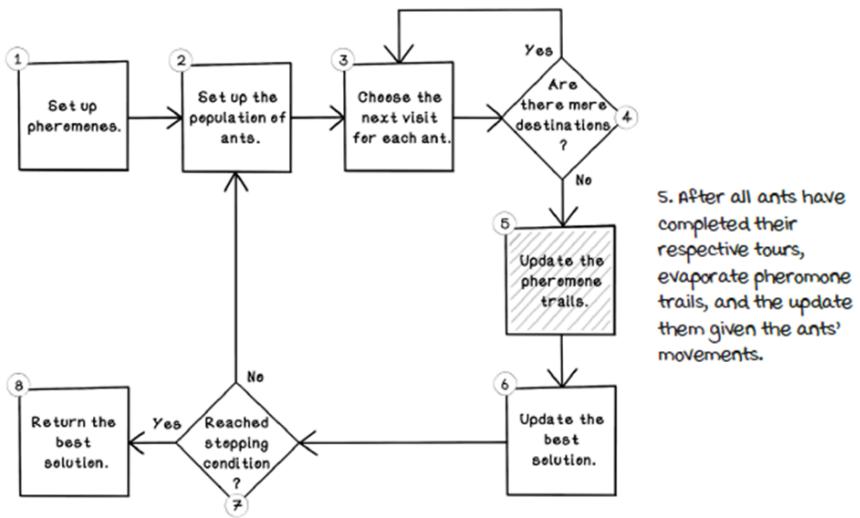
$$\text{Swings: } 0.067 / 0.172 = 0.39$$

$$\text{Bumper cars: } 0.061 / 0.172 = 0.355$$

$$\text{Balloons: } 0.044 / 0.172 = 0.256$$

#### 6.4.4 Update the pheromone trails

Now that the ants have completed a tour of all the attractions, they have all left pheromones behind, which changes the pheromone trails between the attractions (figure 6.24).



**Figure 6.24 Update the pheromone trails.**

Two steps are involved in updating the pheromone trails: evaporation and depositing new pheromones.

### UPDATING PHEROMONES DUE TO EVAPORATION

The concept of evaporation is also inspired by nature. Over time, the pheromone trails lose their intensity. Pheromones are updated by multiplying their respective current values by an evaporation factor—a parameter that can be adjusted to tweak the performance of the algorithm in terms of exploration and exploitation.

Why do pheromones need to evaporate? Imagine if they didn't. The very first path found—even if it was a terrible, long route—would accumulate pheromones. Without evaporation, that initial “bad advice” would stay forever, confusing future ants. Evaporation acts like a “validity timer”. It ensures that old information disappears unless it is constantly reinforced by new ants verifying that “Yes, this path is still the best”.

Figure 6.25 illustrates the updated pheromone trails due to evaporation.

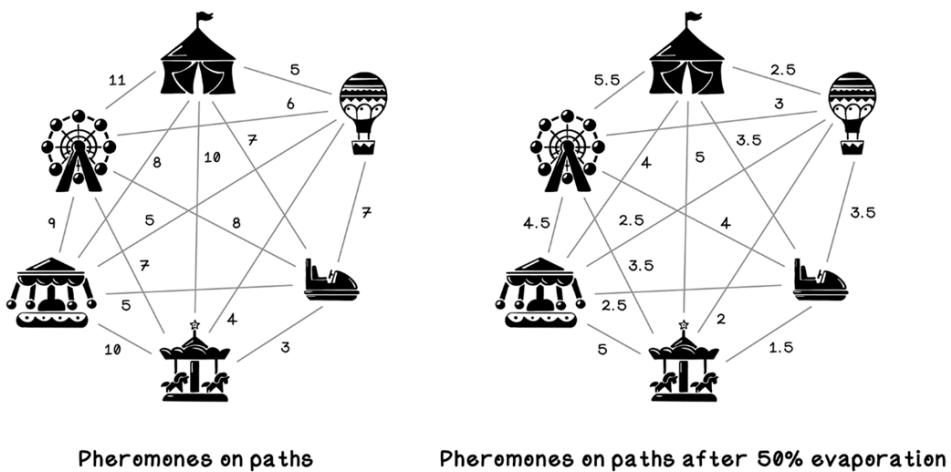


Figure 6.25 Example of updating pheromone trails for evaporation

## UPDATING PHEROMONES BASED ON ANT TOURS

Pheromones are updated based on the ants that have moved along the paths. If more ants move on a specific path, there will be more pheromones on that path.

Each ant contributes its fitness value to the pheromones on every path it has moved on. The effect is that ants with better solutions have a greater influence on the best paths. Figure 6.26 illustrates the updated pheromone trails based on ant movements on the paths.

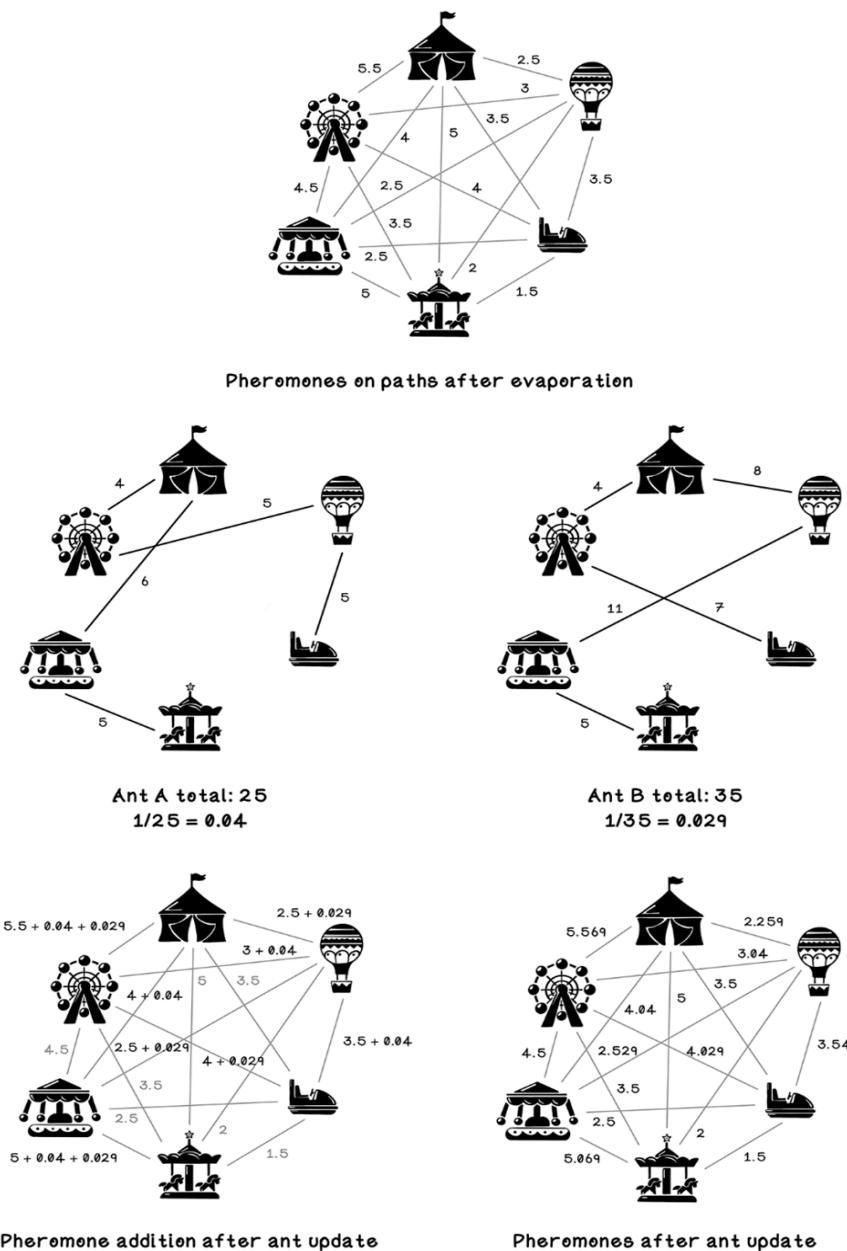
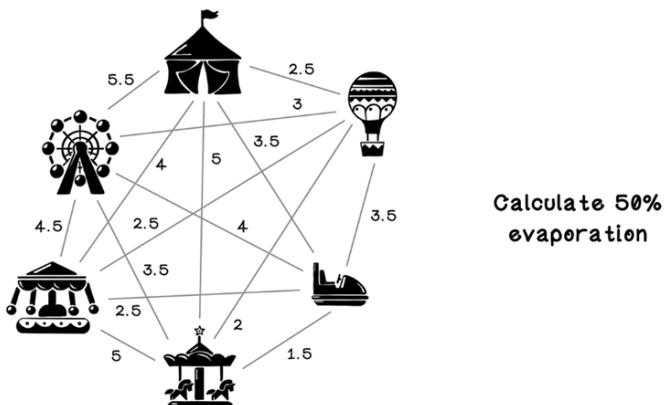
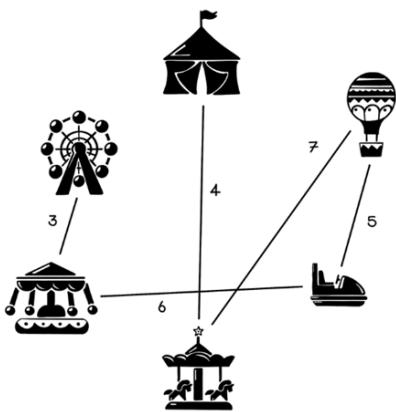


Figure 6.26 Pheromone updates based on ant movements

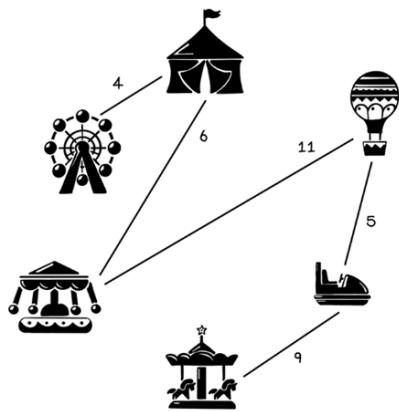
**EXERCISE: CALCULATE THE PHEROMONE UPDATE GIVEN THE FOLLOWING SCENARIO**



Pheromones on paths

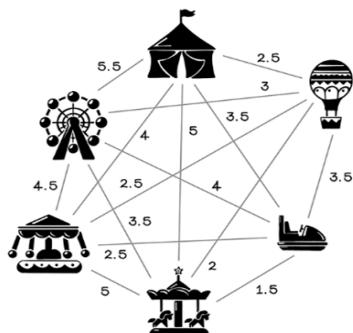


Ant A path

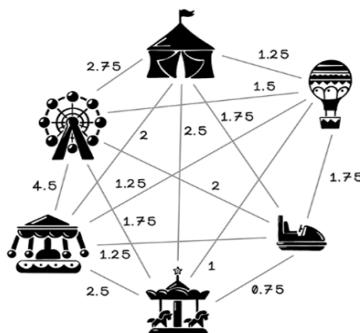


Ant B path

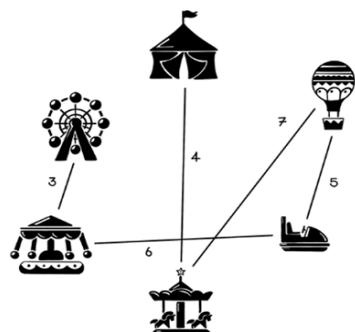
**SOLUTION: CALCULATE THE PHEROMONE UPDATE GIVEN THE FOLLOWING SCENARIO**



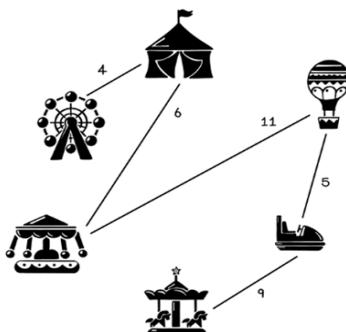
## Pheromones on paths



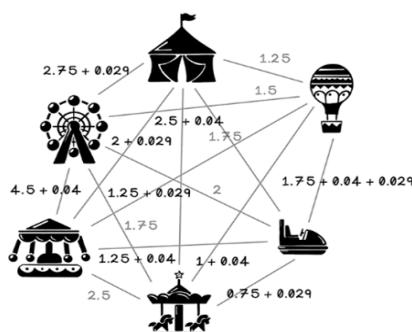
#### Pheromones on paths after 50% evaporation



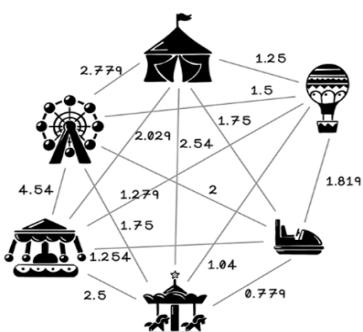
$$\text{Ant A total: } 25$$
$$1/25 = 0.04$$



Ant B total: 35  
 $1/35 = 0.029$



### Pheromone addition after ant update



### Pheromones after ant update

## PYTHON CODE SAMPLE

The `update_pheromones` function applies two important concepts to the pheromone trails. First, the current pheromone intensity is evaporated based on the evaporation rate. If the evaporation rate is 0.5, for example, the intensity decreases by half. The second operation adds pheromones based on ant movements on that path. The amount of pheromones contributed by each ant is determined by the ant's fitness, which in this case is each respective ant's total distance traveled:

```
def update_pheromones(self, evaporation_rate):
    for x in range(0, ATTRACTION_COUNT):
        for y in range(0, ATTRACTION_COUNT):
            self.pheromone_trails[x][y] = self.pheromone_trails[x][y] *
evaporation_rate
            for ant in self.ant_colony:
                self.pheromone_trails[x][y] += 1 /
ant.get_distance_travelled()
```

### 6.4.5 Update the best solution

The best solution is described by the sequence of attraction visits that has the lowest total distance (figure 6.27).

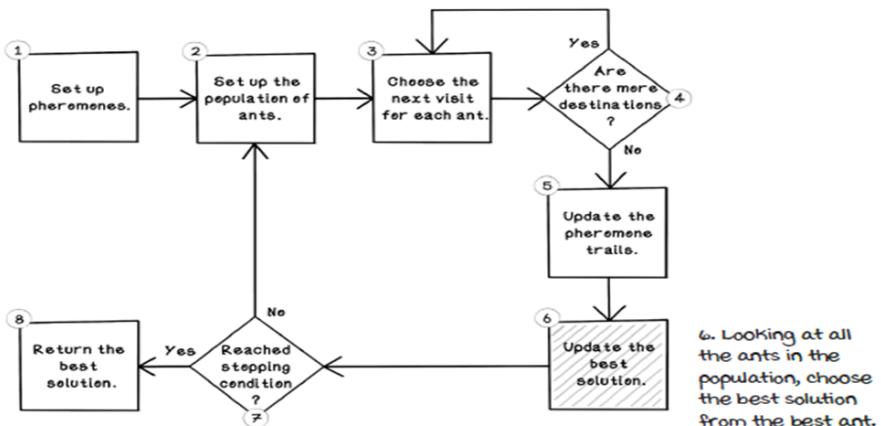


Figure 6.27 Update the best solution.

## PYTHON CODE SAMPLE

After an iteration, after every ant has completed a tour (a tour is complete when an ant visits every attraction), the best ant in the colony must be determined. To make this determination, we find the ant that has the lowest total distance traveled and set it as the new best ant in the colony:

```
def get_best(self, ant_population):
    for ant in ant_population:
        distance_travelled = ant.get_distance_traveled()
        if distance_travelled < self.best_distance:
            self.best_distance = distance_travelled
            self.best_ant = ant
    return self.best_ant
```

### 6.4.6 Determine the stopping criteria

The algorithm stops after several iterations: conceptually, the number of tours that the group of ants concludes. Ten iterations means that each ant does 10 tours; each ant would visit each attraction once and do that 10 times (figure 6.28).

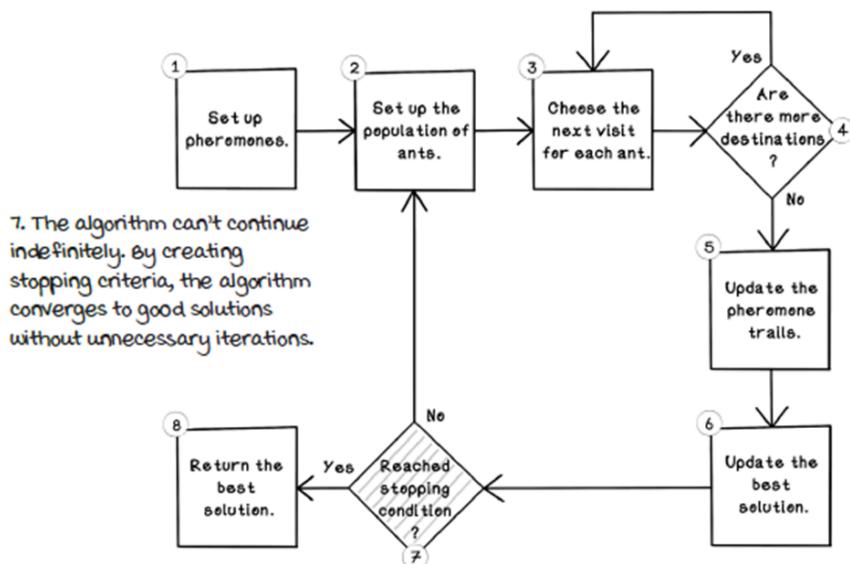


Figure 6.28 Reached stopping condition?

The stopping criteria for the ant colony optimization algorithm can differ based on the domain of the problem being solved. In some cases, realistic limits are known, and when they're unknown, the following options are available:

- *Stop when a predefined number of iterations is reached*—In this scenario, we define a total number of iterations for which the algorithm will always run. If 100 iterations are defined, each ant completes 100 tours before the algorithm terminates.
- *Stop when the best solution stagnates*—In this scenario, the best solution after each iteration is compared with the previous best solution. If the solution doesn't improve after a defined number of iterations, the algorithm terminates. If iteration 20 resulted in a solution with fitness 100, and that iteration is repeated up until iteration 30, it is likely (but not guaranteed) that no better solution exists.

## PYTHON CODE SAMPLE

The `solve` function ties everything together and should give you a better idea of the sequence of operations and the overall life cycle of the algorithm. Notice that the algorithm runs for several defined total iterations. The ant colony is also initialized to its starting point at the beginning of each iteration, and a new best ant is determined after each iteration:

```
def solve(self, total_iterations, evaporation_rate):
    self.setup_pheromones()
    for i in range(0, TOTAL_ITERATIONS):
        self.setup_ants(NUMBER_OF_ANTS_FACTOR)
        for r in range(0, ATTRACTION_COUNT - 1):
            self.move_ants(self.ant_colony)
        self.update_pheromones(evaporation_rate)
        self.best_ant = self.get_best(self.ant_colony)
        print(i, ' Best distance: ', self.best_ant.get_distance_travelled())

def move_ants(self, ant_population):
    for ant in ant_population:
        ant.visit_attraction(self.pheromone_trails)
```

We can tweak several parameters to alter the exploration and exploitation of the ant colony optimization algorithm. These parameters influence how long the algorithm will take to find a good solution. Some randomness is good for exploring. Balancing the weighting between heuristics and pheromones influences whether ants attempt a greedy search (when favoring heuristics) or trust pheromones more. The evaporation rate also influences this balance. The number of ants and the total number of iterations they have influences the quality of a solution. When we add more ants and more iterations, more computation is required. Based on the problem at hand, time to compute may influence these parameters (figure 6.29):

**Set the probability of ants choosing a random attraction to visit (0.0 - 1.0) (0% - 100%).**

`RANDOM_ATTRACTION_FACTOR = 0.3`

**Set the weight for pheromones on path for selection by ants.**

`ALPHA = 4`

**Set the weight for heuristic of path for selection by ants.**

`BETA = 7`

**Set the percentage of ants in the colony based on the total number of attractions.**

`NUMBER_OF_ANTS_FACTOR = 0.5`

**Set the number of tours ants must complete.**

`TOTAL_ITERATIONS = 1000`

**Set the rate of pheromone evaporation (0.0 - 1.0) (0% - 100%).**

`EVAPORATION_RATE = 0.4`

**Figure 6.29 Parameters that can be tweaked in the ant colony optimization algorithm**

Now you have insight into how ant colony optimization algorithms work and how they can be used to solve the Carnival Problem. The following section describes some other possible use cases. Perhaps these examples may help you find uses for the algorithm in your work.

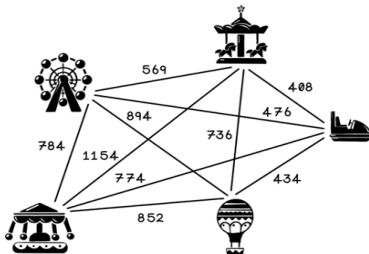
## 6.5 Use cases for ant colony optimization algorithms

Ant colony optimization algorithms are versatile and useful in several real-world applications. These applications usually center on complex optimization problems such as the following:

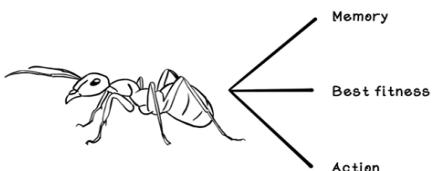
- *Route optimization*—Routing problems usually include several destinations that need to be visited with several constraints. In a logistics example, perhaps the distance between destinations, traffic conditions, types of packages being delivered, and times of day are important constraints that need to be considered to optimize the operations of the business. Ant colony optimization algorithms can be used to address this problem. The problem is similar to the carnival problem explored in this chapter, but the heuristic function is likely to be more complex and context specific.
- *Job scheduling*—Job scheduling is present in almost any industry. Nurse shifts are important to ensure that good health care can be provided. Computational jobs on servers must be scheduled in an optimal manner to maximize the use of the hardware without waste. Ant colony optimization algorithms can be used to solve these problems. Instead of looking at the entities that ants visit as locations, we see that ants visit tasks in different sequences. The heuristic function includes constraints and desired rules specific to the context of the jobs being scheduled. Nurses, for example, need days off to prevent fatigue, and jobs with high priorities on a server should be favored.
- *Image processing*—The ant colony optimization algorithm can be used for edge detection in image processing. An image is composed of several adjacent pixels, and the ants move from pixel to pixel, leaving behind pheromone trails. Ants drop stronger pheromones based on the pixel colors' intensity, resulting in pheromone trails along the edges of objects containing the highest density of pheromones. This algorithm essentially traces the outline of the image by performing edge detection. The images may require preprocessing to decolorize the image to grayscale so that the pixel-color values can be compared consistently.

## 6.6 Summary of ant colony optimization

Ant Colony Optimization uses the concept of pheromones and heuristics to solve optimization problems

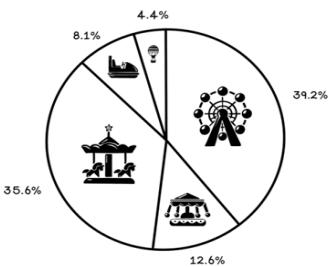
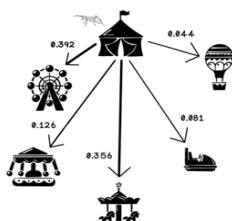


ACO is useful for problems like finding shortest paths or optimal task schedules

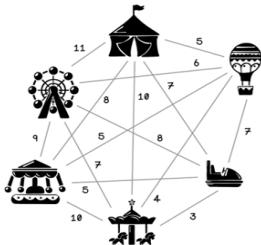


Weightings between a heuristic and the pheromones on paths are used to calculate a probability of selection

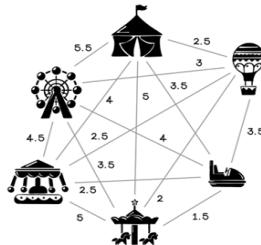
Ants have a concept of memory, their fitness, and what actions they can perform



Pheromones are contributed by each ant proportional to its respective fitness - they also evaporate



Pheromones on paths



Pheromones on paths after 50% evaporation

# 7 *Swarm intelligence: Particles*

## This chapter covers

- Understanding the inspiration for particle swarm intelligence algorithms
- Understanding and solving optimization problems
- Designing and implementing a particle swarm optimization algorithm

### 7.1 What is particle swarm optimization?

*Particle swarm optimization* (PSO) is another swarm algorithm. Swarm intelligence relies on emergent behavior of many individuals to solve difficult problems as a collective. We saw in Chapter 6 how ants can find the shortest paths between destinations through their use of pheromones.

Bird flocks are another ideal example of swarm intelligence in nature. When a single bird is flying, it might attempt several maneuvers and techniques to preserve energy, like jumping and gliding through the air or leveraging wind currents to carry it in the direction it wants to travel. This behavior indicates some primitive level of intelligence in a single individual. But birds also have the need to migrate between different seasons. In winter, there is less availability of insects and other food, and suitable nesting locations also become scarce. Birds tend to flock to warmer areas to take advantage of better weather conditions, which improves their likelihood of survival. Migration is usually not a short trip. It takes thousands of miles of movement to arrive at an area with suitable conditions. When birds travel these long distances, they tend to flock. Birds flock because there is strength in numbers when facing predators; additionally, it saves energy. The formation that we observe in bird flocks has many advantages. A large, strong bird will take the lead, and when it flaps its wings, it creates uplift for the birds behind it. These birds can fly while using significantly less energy. Flocks can change leaders if the direction changes or if the leader becomes fatigued. When a specific bird moves out of formation, it experiences more difficulty in flying via air resistance and corrects its movement to get back into formation. Figure 7.1 illustrates a bird flock formation; you may have seen something similar.



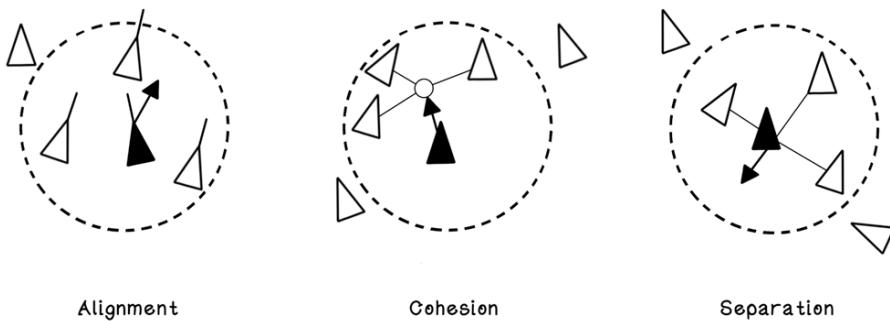
**Figure 7.1 An example bird flock formation**

Craig Reynolds developed a simulator program in 1987 to understand the attributes of emergent behavior in bird flocks and used the following rules to guide the group. These rules are extracted from observation of bird flocks:

- *Alignment*—An individual should steer in the average heading of its neighbors to ensure that the group travels in a similar direction.
- *Cohesion*—An individual should move toward the average position of its neighbors to maintain the formation of the group.

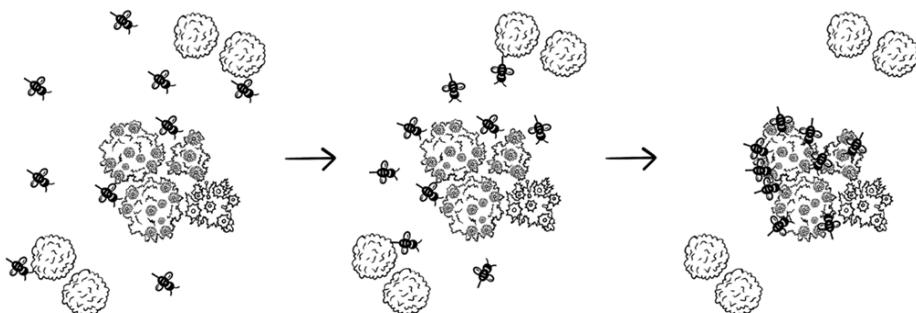
- *Separation*—An individual should avoid crowding or colliding with its neighbors to ensure that individuals do not collide, disrupting the group.

Additional rules are used in different variants attempting to simulate swarm behavior. Figure 7.2 illustrates the behavior of an individual in different scenarios, as well as the direction in which it is influenced to move to obey a rule. Adjusting movement is a balance of these three principles shown in the figure.



**Figure 7.2 Rules that guide a swarm**

PSOs involve a group of individuals at different points in the solution space, all using real-life swarm concepts to find an optimal solution in the space. This chapter dives into the workings of the PSO algorithm and shows how it can be used to solve problems. Imagine a swarm of bees that spreads out looking for flowers and gradually converges on an area that has the most density of flowers. As more bees find the flowers, more are attracted to the flowers. At its core, this example is what particle swarm optimization entails (figure 7.3).

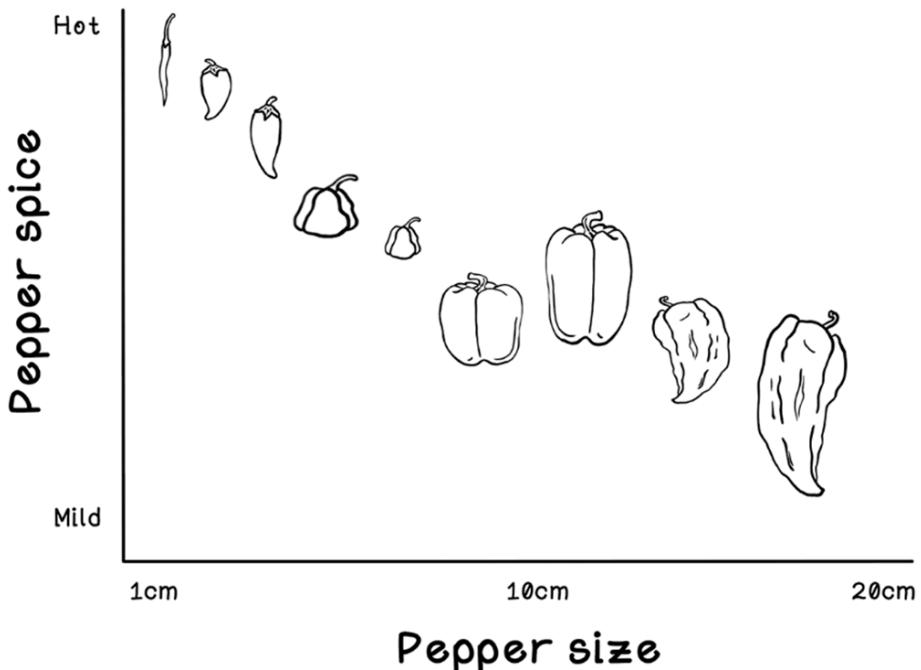


**Figure 7.3 A bee swarm converging on its goal**

Optimization problems have been mentioned in several chapters. Finding the optimal path through a maze, determining the optimal items for a knapsack, and finding the optimal path between attractions in a carnival are examples of optimization problems. We worked through them without diving into the details behind them. From this chapter on, however, a deeper understanding of optimization problems is important. The next section works through some of the intuition to be able to spot optimization problems when they occur.

## 7.2 Optimization problems: A slightly more technical perspective

Suppose that we have several peppers of different sizes. Usually, small peppers tend to be spicier than large peppers. If we plot all the peppers on a chart based on size and spiciness, it may look like figure 7.4.



**Figure 7.4 Pepper spice vs. pepper size**

The figure depicts the size of each pepper and how spicy it is. Now, by removing the imagery of the peppers, plotting the data points, and drawing a possible curve between them, we are left with figure 7.5. If we had more peppers, we would have more data points, and the curve would be more accurate.

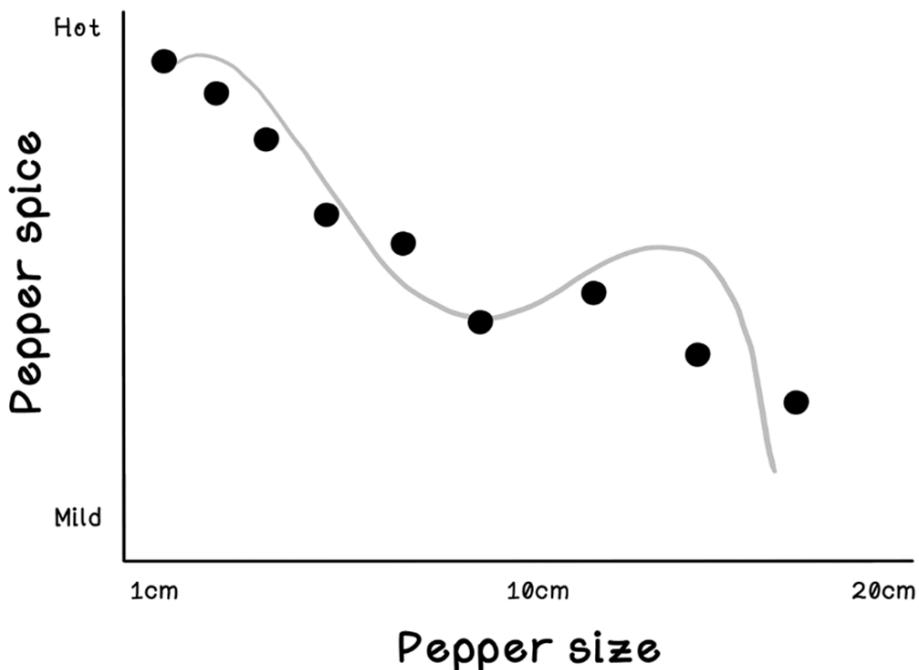


Figure 7.5 Pepper spice vs. pepper size trend

This example could potentially be an optimization problem. If we searched for a minimum from left to right, we would come across several points less than the previous ones, but in the middle, we encounter one that is higher. Should we stop? If we did, we would be missing the actual minimum, which is the last data point, known as the *global minimum*.

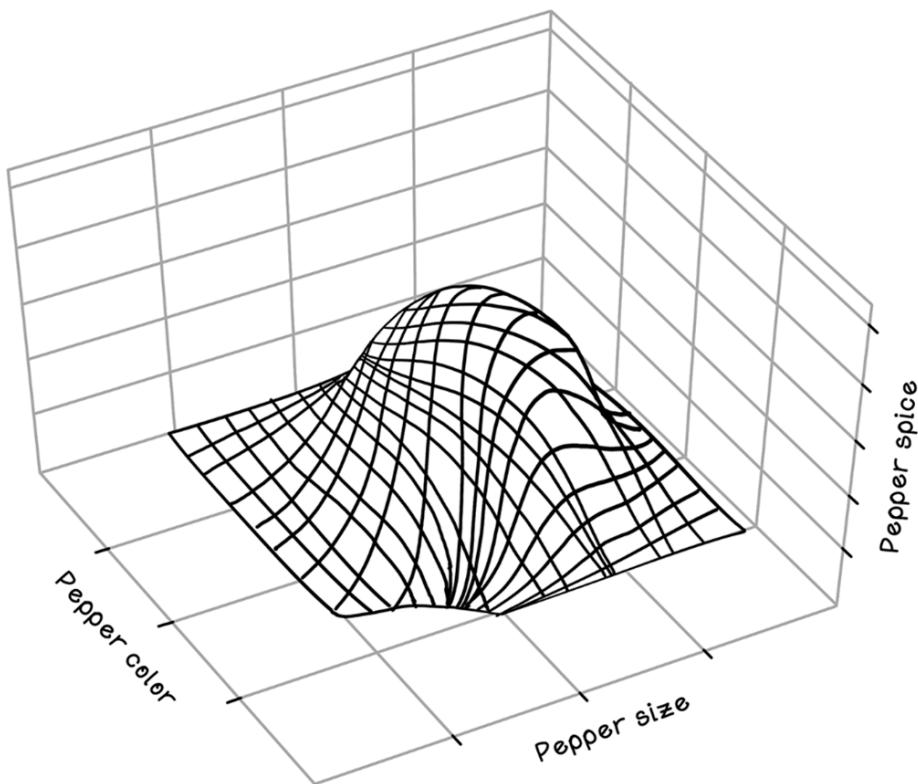
The trend line/curve that is approximated can be represented by a function, such as the one shown in figure 7.6. This function can be interpreted as the spiciness of the pepper being equal to the result of this function where the size of the pepper is represented by  $x$ .

$$f(x) = -(x - 4)(x - 0.2)(x - 3) + 5$$

Figure 7.6 An example function for pepper spice vs. pepper size

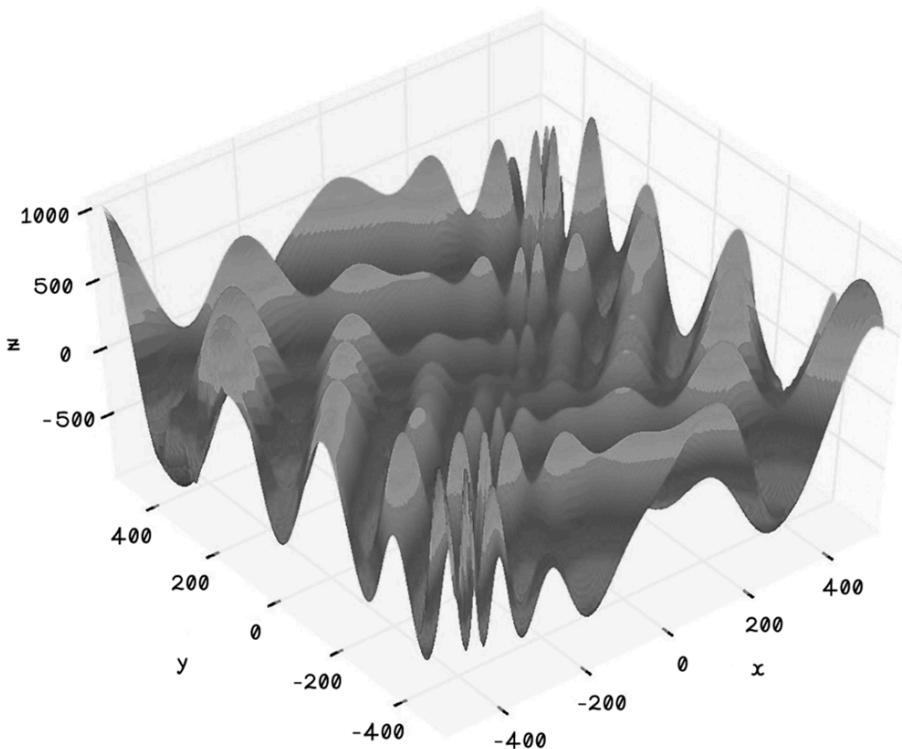
Real-world problems typically have thousands of data points, and the minimum output of the function is not as clear as in this example. The search spaces are massive and difficult to solve by hand.

Notice that we have used only two properties of the pepper to create the data points, which resulted in a simple curve. If we consider another property of the pepper, such as color, the representation of the data changes significantly. Now the chart has to be represented in 3D, and the trend becomes a surface instead of a curve. A surface is like a warped blanket in three dimensions (figure 7.7). This surface is also represented as a function but is more complex.



**Figure 7.7 Pepper spice vs. pepper size vs. pepper color**

Furthermore, a 3D search space could look fairly simple, like figure 7.7, or be so complex that attempting to inspect it visually to find the minimum would be almost impossible (figure 7.8).



**Figure 7.8 A function visualized in the 3D space as a plane**

Figure 7.9 shows the function that represents this plane.

$$f(x,y) = -(y + 4\pi)\sin \sqrt{\left| \frac{x}{2} + (y + 4\pi) \right|} - x \sin \sqrt{\left| x - (y + 4\pi) \right|}$$

**Figure 7.9 The function that represents the surface in figure 7.8**

It gets more interesting! We have looked at three attributes of a pepper: its size, its color, and how spicy it is. As a result, we're searching in three dimensions. What if we want to include the location of growth? This attribute would make it even more difficult to visualize and understand the data, because we are searching in four dimensions. If we add the pepper's age and the amount of fertilizer used while growing it, we are left with a massive search space in six dimensions, and we can't imagine what this search might look like. This search too is represented by a function, but again, it is too complex and difficult for a person to solve.

Particle swarm optimization algorithms are particularly good at solving difficult optimization problems. Particles are distributed over the multidimensional search space and work together to find good maximums or minimums.

Particle swarm optimization algorithms are particularly useful in the following scenarios:

- *Large search spaces*—There are many data points and possibilities of combinations.
- *Search spaces with high dimensions*—There is complexity in high dimensions. Many dimensions of a problem are required to find a good solution.

#### **EXERCISE: HOW MANY DIMENSIONS WILL THE SEARCH SPACE FOR THE FOLLOWING SCENARIO BE?**

In this scenario, we need to determine a good city to live in based on the average minimum temperature during the year, because we don't like the cold. It is also important that the population be less than 700,000 people, because crowded areas can be inconvenient. The average property price should be as little as possible, and the more trains in the city, the better.

#### **SOLUTION: HOW MANY DIMENSIONS WILL THE SEARCH SPACE FOR THE FOLLOWING SCENARIO BE?**

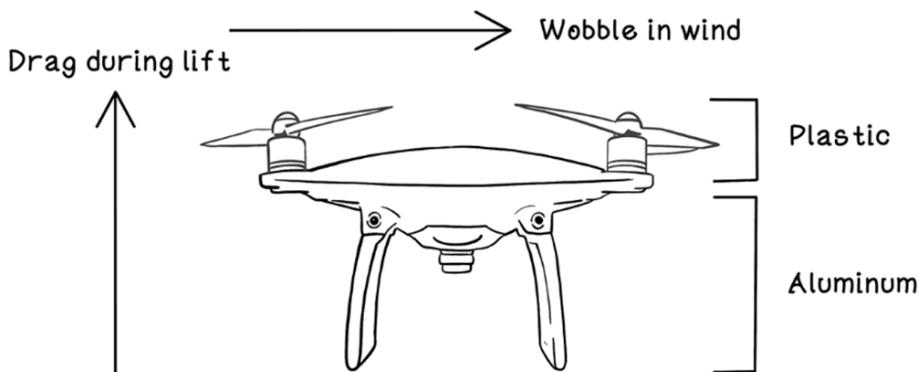
The problem in this scenario consists of four dimensions:

- Average temperature
- Size of population
- Average price of property
- Number of trains

### 7.3 Problems applicable to particle swarm optimization

Imagine that we are developing a drone, and several materials are used to create its body and propeller wings (the blades that make it fly). Through many research trials, we have found that different amounts of two specific materials yield different results in terms of optimal performance for lifting the drone and resisting strong winds. These two materials are aluminum, for the chassis, and plastic, for the blades. Too much or too little of either material will result in a poor-performing drone. But several combinations yield a good-performing drone, and only one combination results in an exceptionally well-performing drone.

Figure 7.10 illustrates the components made of plastic and the components made of aluminum. The arrows illustrate the forces that influence the performance of the drone. In simple terms, we want to find a good ratio of plastic to aluminum for a version of the drone that reduces drag during lift and decreases wobble in the wind. So plastic and aluminum are the inputs, and the output is the resulting stability of the drone. Let's describe ideal stability as reducing drag during liftoff and wobble in the wind.



**Good performance = Low drag and less wobble in the wind**

**Figure 7.10 The drone optimization example**

Precision in the ratio of aluminum and plastic is important, and the range of possibilities is large. In this scenario, researchers have found the function for the ratio of aluminum and plastic. We will use this function in a simulated virtual environment that tests the drag and wobble to find the best values for each material before we manufacture another prototype drone. We also know that the maximum and minimum ratios for the materials are 10 and -10, respectively. This fitness function is similar to a heuristic.

Note that negative numbers for aluminum and plastic are bizarre in reality; however, we're using them in this example to demonstrate the fitness function used to optimize these values.

Figure 7.11 shows the fitness function for the ratio between aluminum ( $x$ ) and plastic ( $y$ ). The result is a performance score based on drag and wobble, given the input values for  $x$  and  $y$ .

$$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

**Figure 7.11** The example function for optimizing aluminum ( $x$ ) and plastic ( $y$ )

How can we find the amount of aluminum and the amount of plastic required to create a good drone? One possibility is to try every combination of values for aluminum and plastic until we find the best ratio of materials for our drone. Take a step back and imagine the amount of computation required to find this ratio. If we treat the materials as continuous values (meaning the amount could be 10.0, 10.01, or 10.0001), the search space becomes effectively infinite. Trying to brute-force every tiny decimal increment is computationally impossible if we want to solve this efficiently. We need a smarter way to navigate this vast landscape. We need to compute the result for the items in table 7.1.

**Table 7.1 Possible values for aluminum and plastic compositions**

<b>How many parts aluminum? (x)</b>	<b>How many parts plastic? (y)</b>
-0.1	1.34
-0.134	0.575
-1.1	0.24
-1.1645	1.432
-2.034	-0.65
-2.12	-0.874
0.743	-1.1645
0.3623	-1.87
1.75	-2.7756
...	...
$-10 \geq \text{Aluminum} \geq 10$	$-10 \geq \text{Plastic} \geq 10$

This computation will go on for every possible number between the constraints and is computationally expensive, so it is realistically impossible to brute-force this problem. A better approach is needed.

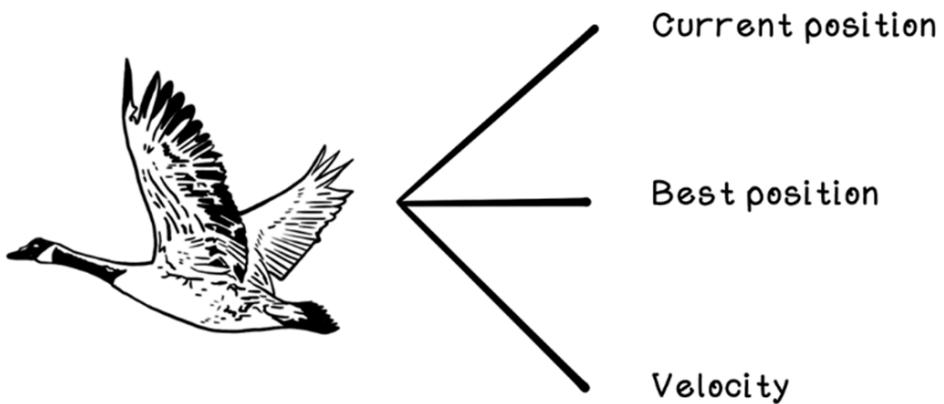
Particle swarm optimization provides a means to search a large search space without checking every value in each dimension. In the drone problem, aluminum is one dimension of the problem, plastic is the second dimension. Together, these form a 2-dimensional search space. The resulting performance of the drone is not a dimension we search through, but rather the fitness score (or height) associated with each point in that 2D space.

Think of it like a hiker on a mountain: The hiker moves North or East (2 dimensions of movement) to try and find the highest peak (Maximum fitness for the problem).

Next, we determine the data structures required to represent a particle, including the data about the problem that it will contain.

## 7.4 Representing state: What do particles look like?

Because particles move across the search space, the concept of a particle must be defined (figure 7.12).



**Figure 7.12 Properties of a particle**

The following represent the concept of a particle:

- *Position*—The position of the particle in all dimensions
- *Best position*—The best position found using the fitness function
- *Velocity*—The current velocity of the particle’s movement

### PYTHON CODE SAMPLE

To fulfill the three attributes of a particle—position, best position, and velocity—the following properties are required in a constructor of the particle for the various operations of the particle swarm optimization algorithm. Don’t worry about the inertia, cognitive component, and social component right now; they will be explained in upcoming sections:

```

class Particle:

    def __init__(self, x, y, inertia, cognitive_constant, social_constant):
        self.x = x
        self.y = y
        self.fitness = math.inf
        self.velocity = (0.0, 0.0)
        self.best_x = x
        self.best_y = y
        self.best_fitness = math.inf
        self.inertia = inertia
        self.cognitive_constant = cognitive_constant
        self.social_constant = social_constant

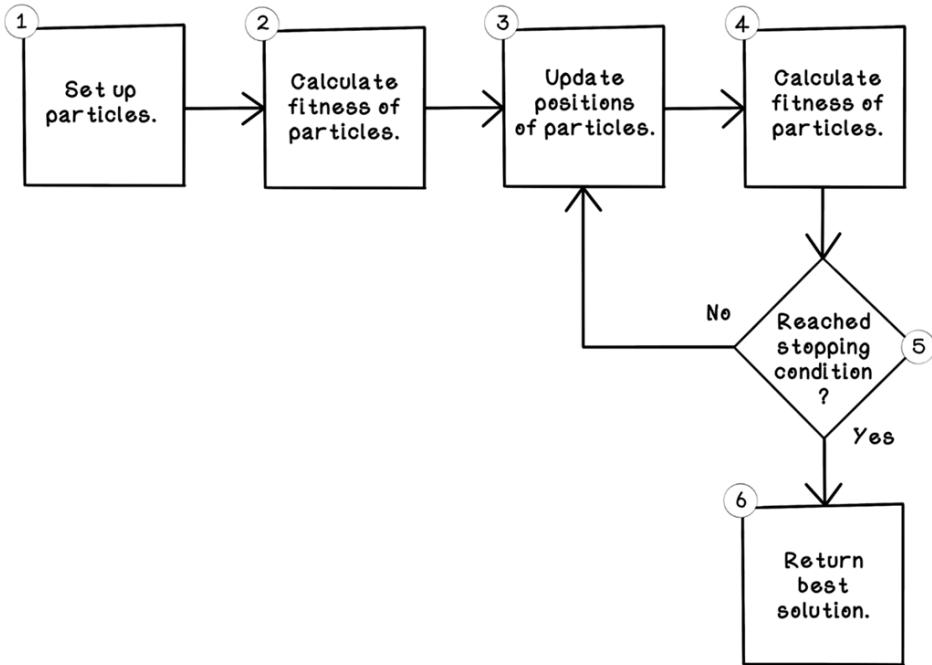
```

## 7.5 Particle swarm optimization life cycle

The approach to designing a particle swarm optimization algorithm is based on the problem space being addressed. Each problem has a unique context and a different domain in which data is represented. Solutions to different problems are also measured differently. Let's dive into how a particle swarm optimization can be designed to solve the drone construction problem.

The general life cycle of a particle swarm optimization algorithm is as follows (figure 7.13):

1. *Initialize the population of particles*—Determine the number of particles to be used, and initialize each particle to a random position in the search space.
2. *Calculate the fitness of each particle*—Given the position of each particle, determine the fitness of that particle at that position.
3. *Update the position of each particle*—Repetitively update the position of all the particles, using principles of swarm intelligence. Particles will explore the search space and then converge to good solutions.
4. *Determine the stopping criteria*—Determine when the particles stop updating and the algorithm stops.

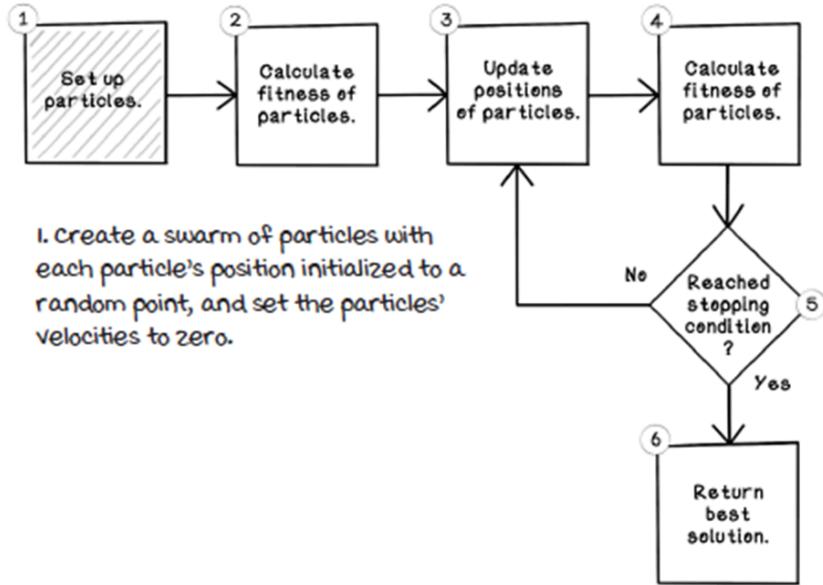


**Figure 7.13 The life cycle of a particle swarm optimization algorithm**

The particle swarm optimization algorithm is fairly simple, but the details of step 3 are particularly intricate. Let's look at each step in isolation and uncover the details that make the algorithm work.

### 7.5.1 Initialize the population of particles

The algorithm starts by creating a specific number of particles, which will remain the same for the lifetime of the algorithm (figure 7.14).

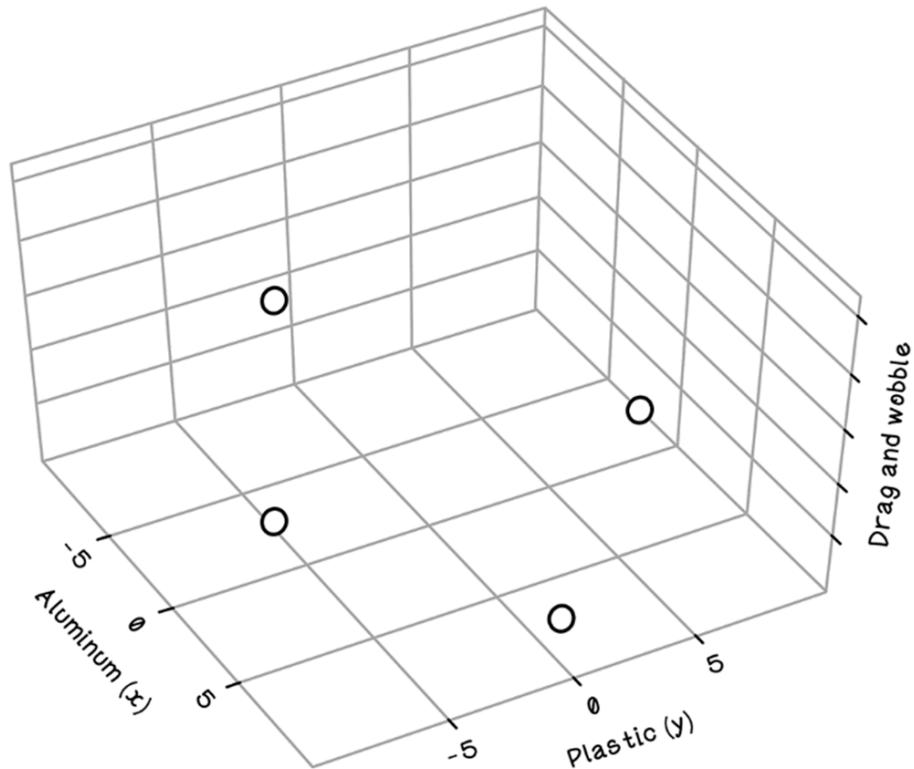


**Figure 7.14 Set up the particles.**

The three factors that are important in initializing the particles are (figure 7.15):

- *Number of particles*—The number of particles influences computation. The more particles that exist, the more computation is required. Additionally, more particles will likely mean that converging on a global best solution will take longer because more particles are attracted to their local best solutions. The constraints of the problem also affect the number of particles. A larger search space may need more particles to explore it. There could be as many as 1,000 particles or as few as 4. Usually, 50 to 100 particles produce good solutions without being too computationally expensive.
- *Starting position for each particle*—The starting position for each particle should be a random position in all the respective dimensions. It is important that the particles are distributed evenly across the search space. If most of the particles are in a specific region of the search space, they will struggle to find solutions outside that area.

- *Starting velocity for each particle*—While it is possible to start at 0, standard practice is to initialize velocity to small random values. This random “kick” ensures that particles immediately explore different directions in the first iteration, rather than waiting for the swarm to find a leader. Think of releasing fireflies from a jar. They don't drop to the ground and wait; they immediately scatter in random directions to cover the most ground.



**Figure 7.15 A visualization of the initial positions of four particles in a 3D plane**

Table 7.2 describes the data encapsulated by each particle at the initialization step of the algorithm. Notice that the velocity is 0; the current fitness and best fitness values are 0 because they have not been calculated yet.

**Table 7.2 Data attributes for each particle**

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum (x)</b>	<b>Current plastic (y)</b>	<b>Current fitness</b>	<b>Best aluminum (x)</b>	<b>Best plastic (y)</b>	<b>Best fitness</b>
1	0	7	1	0	7	1	0
2	0	-1	9	0	-1	9	0
3	0	-10	1	0	-10	1	0
4	0	-2	-5	0	-2	-5	0

## PYTHON CODE SAMPLE

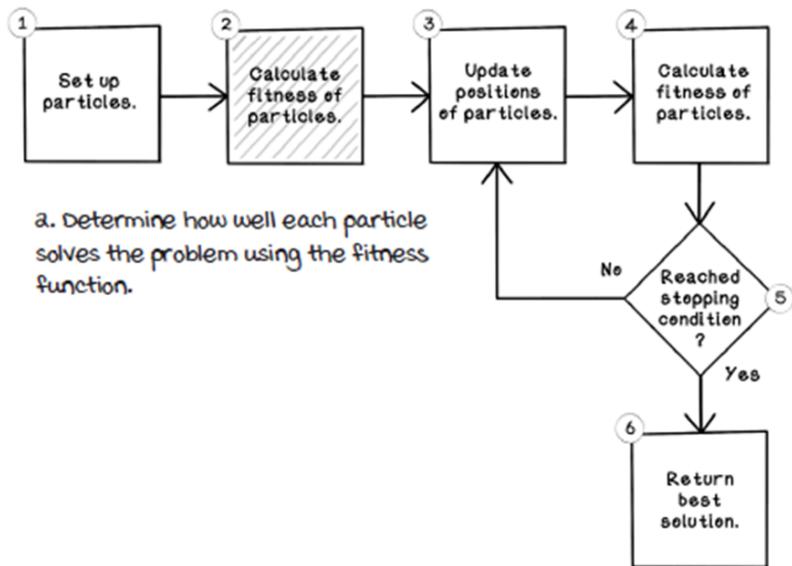
The method to generate a swarm consists of creating an empty list and appending new particles to it. The key factors are:

- Ensuring that the number of particles is configurable.
- Ensuring that the random number generation is done uniformly; numbers are distributed across the search space within the constraints. This implementation depends on the features of the random number generator used.
- Ensuring that the constraints of the search space are specified: in this case, -10 and 10 for both x and y of the particle.

```
def get_random_swarm(number_of_particles):
    particles = []
    for p in range(number_of_particles):
        particles.append(Particle(random.randint(-10, 10),
                                  random.randint(-10, 10),
                                  INERTIA, COGNITIVE_CONSTANT,
                                  SOCIAL_CONSTANT))
    return particles
```

### 7.5.2 Calculate the fitness of each particle

The next step is calculating the fitness of each particle at its current position. The fitness of particles is calculated every time the entire swarm changes position (figure 7.16).



**Figure 7.16 Calculate the fitness of the particles.**

In the drone scenario, the scientists provided a function in which the result is the amount of drag and wobble given a specific number of aluminum and plastic components. This function is used as the fitness function in the particle swarm optimization algorithm in this example (figure 7.17).

$$f(x,y) = (x + 2y - 7)^2 + (2x + y - 5)^2$$

**Figure 7.17 The example function for optimizing aluminum ( $x$ ) and plastic ( $y$ )**

If  $x$  is aluminum and  $y$  is plastic, the calculations in figure 7.18 can be made for each particle to determine its fitness by substituting  $x$  and  $y$  for the values of aluminum and plastic.

$$f(7,1) = (7 + 2(1) - 7)^2 + (2(7) + 1 - 5)^2 = 104$$

$$f(-1,9) = (-1 + 2(9) - 7)^2 + (2(-1) + 9 - 5)^2 = 104$$

$$f(-10,1) = (-10 + 2(1) - 7)^2 + (2(-10) + 1 - 5)^2 = 801$$

$$f(-2,-5) = (-2 + 2(-5) - 7)^2 + (2(-2) - 5 - 5)^2 = 557$$

**Figure 7.18 Fitness calculations for each particle**

Now the table of particles represents the calculated fitness for each particle (table 7.3). It is also set as the best fitness for each particle because it is the only known fitness in the first iteration. After the first iteration, the best fitness for each particle is the best fitness in each specific particle's history.

**Table 7.3 Data attributes for each particle**

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum (x)</b>	<b>Current plastic (y)</b>	<b>Current fitness</b>	<b>Best aluminum (x)</b>	<b>Best plastic (y)</b>	<b>Best fitness</b>
1	0	7	1	104	7	1	104
2	0	-1	9	104	-1	9	104
3	0	-10	1	801	-10	1	801
4	0	-2	-5	557	-2	-5	557

**EXERCISE: WHAT WOULD THE FITNESS BE FOR THE FOLLOWING INPUTS GIVEN THE DRONE FITNESS FUNCTION?**

Particle	Velocity	Current aluminum (x)	Current plastic (y)	Current fitness	Best aluminum (x)	Best plastic (y)	Best fitness
1	0	5	-3	0	5	-3	0
2	0	-6	-1	0	-6	-1	0
3	0	7	3	0	7	3	0
4	0	-1	9	0	-1	9	0

**SOLUTION: WHAT WOULD THE FITNESS BE FOR THE FOLLOWING INPUTS GIVEN THE DRONE FITNESS FUNCTION?**

$$f(5, -3) = (5 + 2(-3) - 7)^2 + (2(5) - 3 - 5)^2 = 68$$

$$f(-6, -1) = (-6 + 2(-1) - 7)^2 + (2(-6) - 1 - 5)^2 = 549$$

$$f(7, 3) = (7 + 2(3) - 7)^2 + (2(7) + 3 - 5)^2 = 180$$

$$f(-1, 9) = (-1 + 2(9) - 7)^2 + (2(-1) + 9 - 5)^2 = 104$$

## PYTHON CODE SAMPLE

The fitness function is representing the mathematical function in code. Any math library will contain the operations required, such as a power function and a square-root function:

```
def calculate_booth(x, y):
    return math.pow(x + 2 * y - 7, 2) + math.pow(2 * x + y - 5, 2)
```

The function for updating the fitness of a particle is also trivial, in that it determines whether the new fitness is better than a past best and then stores that information:

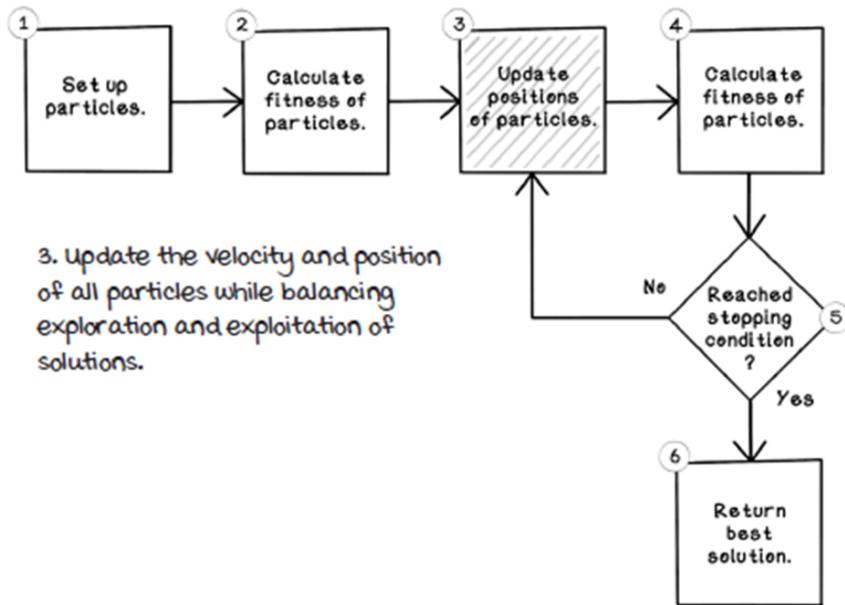
```
def update_fitness(self):
    self.fitness = calculate_booth(self.x, self.y)
    if self.fitness < self.best_fitness:
        self.best_fitness = self.fitness
        self.best_x = self.x
        self.best_y = self.y
```

The function to determine the best particle in the swarm iterates through all particles, updates their fitness based on their new positions, and finds the particle that yields the smallest value for the fitness function. In this case, we are minimizing, so a smaller value is better:

```
def get_best_in_swarm(self):
    best = math.inf
    best_particle = None
    for p in self.swarm:
        p.update_fitness()
        if p.fitness < best:
            best = p.fitness
            best_particle = p
    return best_particle
```

### 7.5.3 Update the position of each particle

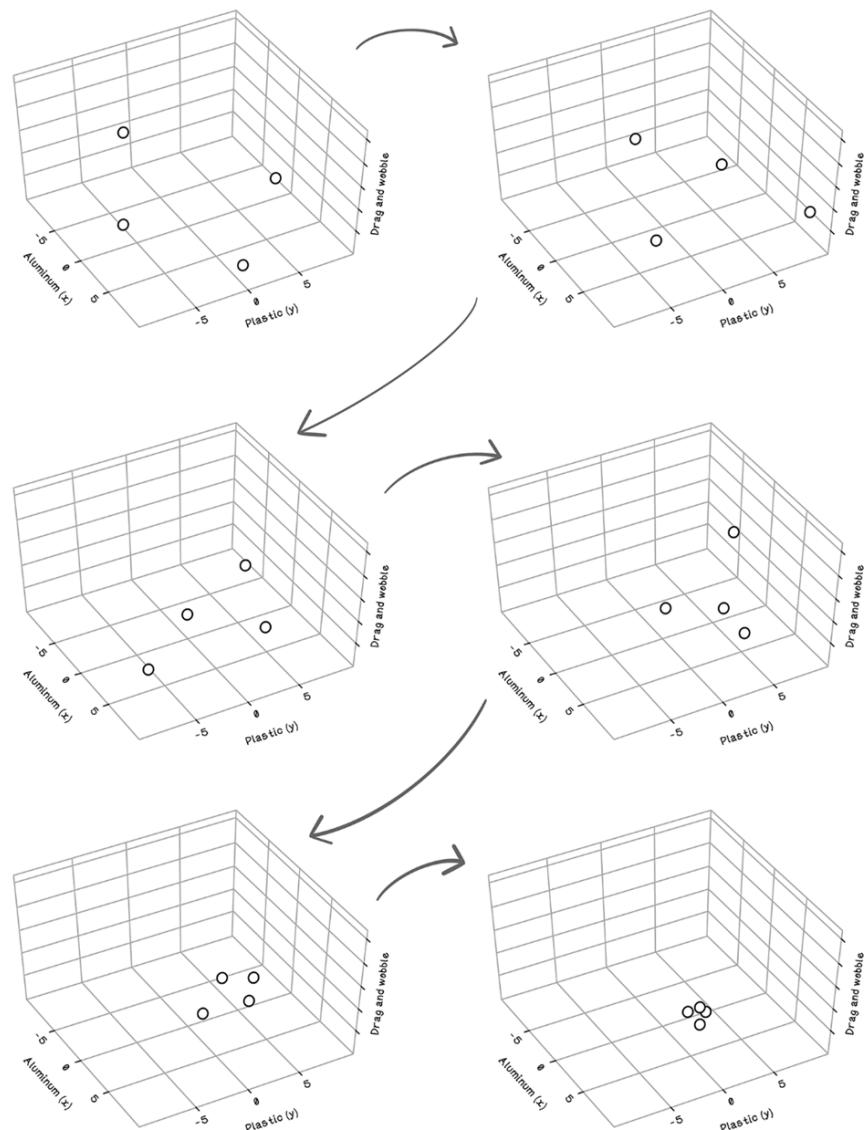
The update step of the algorithm is the most intricate, because it is where the magic happens. The update step encompasses the properties of swarm intelligence in nature into a mathematical model that allows the search space to be explored while honing in on good solutions (figure 7.19).



**Figure 7.19 Update the positions of the particles.**

Particles in the swarm update their position given a cognitive ability and factors in the environment around them, such as inertia and what the swarm is doing. These factors influence the velocity and position of each particle. The first step is understanding how velocity is updated. The velocity determines the direction and speed of movement of the particle.

The particles in the swarm move to different points in the search space to find better solutions. Each particle relies on its memory of a good solution and the knowledge of the swarm's best solution. Figure 7.20 illustrates the movement of the particles in the swarm as their positions are updated.



**Figure 7.20 The movement of particles over five iterations**

## THE COMPONENTS OF UPDATING VELOCITY

Three components are used to calculate the new velocity of each particle: inertia, cognitive, and social. Each component influences the movement of the particle. We will look at each of the components in isolation before diving into how they are combined to update the velocity and, ultimately, the position of a particle:

- *Inertia*—The inertia component represents the resistance to movement or change in direction for a specific particle that influences its velocity. The inertia component consists of two values: the inertia magnitude and the current velocity of the particle. The inertia value is a number between 0 and 1.

## Inertia component:

**inertia \* current velocity**

- *High Inertia (closer to 1.0)*: Translates to Exploration. The particle maintains its previous momentum. Like a heavy speeding freight truck, it has so much momentum that it can't turn easily, allowing it to fly through the search space and discover new regions.
- *Low Inertia (closer to 0)*: Translates to Exploitation. The particle slows down quickly. Like a house fly, it has almost no momentum and can change direction quickly, allowing it to hover and fine-tune its search around the best solutions found so far.
- *Cognitive*—The cognitive constant is a number greater than 0 and less than 2. A greater cognitive constant encourages individual independence (or personal exploration), preventing the particle from blindly following the swarm and ensuring it thoroughly checks the area around its own discoveries.

### Cognitive component:

**cognitive acceleration \* (particle best position - current position)**

↓

**cognitive acceleration = cognitive constant \* random cognitive number**

- *Social*—The social component represents the ability of a particle to interact with the swarm. A particle knows the best position in the swarm and uses this information to influence its movement. Social acceleration is determined by using a constant and scaling it with a random number. The social constant remains the same for the lifetime of the algorithm, and the random factor encourages diversity in favoring the social factor.

### **Social component:**

```
social acceleration * (swarm best position - current position)
    ↘
social acceleration = social constant * random social number
```

The greater the social constant, the more exploitation (or convergence) there will be, because the particle favors the swarm's best findings over its own. You can think of these constants as defining the personality of the swarm:

- *High Cognitive, Low Social*: The particles become “Nostalgic Loners”. They roam around but keep getting pulled back to their own personal victories, ignoring the group. The swarm spreads out.
- *Low Cognitive, High Social*: The particles become “Trend Followers”. They rush blindly toward the current leader. The swarm collapses quickly into a single point (which might be a trap or local optimum).

## **UPDATING VELOCITY**

Now that we understand the inertia component, cognitive component, and social component, let's look at how they can be combined to update a new velocity for the particles (figure 7.21).

Imagine a particle is a tourist deciding where to go next. Its direction is determined by three arguing voices:

- *Inertia (The Habit)*: “Let's keep moving in the direction we are already going. It takes effort to turn”.
- *Cognitive (The Memory)*: “Hey, remember that great spot we found yesterday? Let's go back there”.
- *Social (The Peer Pressure)*: “Look! Everyone else is heading towards that hill. There must be something interesting happening there”.

The final velocity is simply the sum of these three “urges”.

**New velocity:**

inertia component + social component + cognitive component

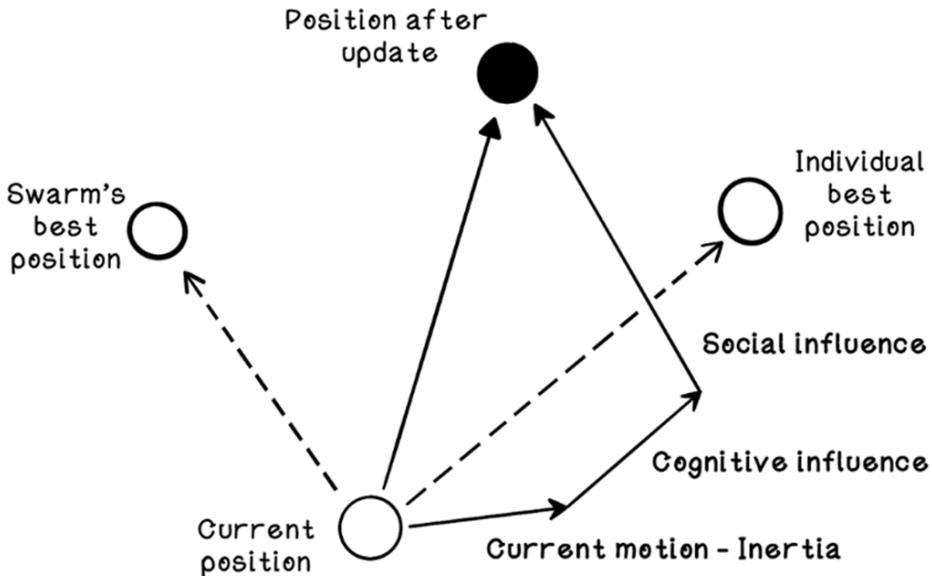
(inertia \* current velocity)

(social acceleration \* (swarm best position - current position))

(cognitive acceleration \* (particle best position - current position))

**Figure 7.21 Formula to calculate velocity**

By looking at the math, we may find it difficult to understand how the different components in the function affect the velocity of the particles. Figure 7.22 depicts how the different factors influence a particle.



**Figure 7.22 The intuition of the factors influencing velocity updates**

Table 7.4 shows the attributes of each particle after the fitness of each is calculated.

**Table 7.4 Data attributes for each particle**

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum</b>	<b>Current plastic</b>	<b>Current fitness</b>	<b>Best aluminum</b>	<b>Best plastic</b>	<b>Best fitness</b>
1	0	7	1	104	2	4	104
2	0	-1	9	104	-1	9	104
3	0	-10	1	801	-10	1	801
4	0	-2	-5	557	-2	-5	557

Next, we will dive into the velocity update calculations for a particle, given the formulas that we have worked through.

Here are the constant configurations that have been set for this scenario:

- *Inertia is set to 0.2.* This setting favors slower exploration.
- *Cognitive constant is set to 0.35.* Because this constant is less than the social constant, the social component is favored over an individual particle's cognitive component.
- *Social constant is set to 0.45.* Because this constant is more than the cognitive constant, the social component is favored. Particles put more weight on the best values found by the swarm.

Figure 7.23 describes the particle 4 calculations of the inertia component, cognitive component, and social component for the velocity update formula.

**Inertia component:**

```
inertia * current velocity
= 0.2 * 0
= 0
```

---

**Cognitive component:**

```
cognitive acceleration = cognitive constant * random cognitive number
= 0.35 * 0.2
= 0.07

cognitive acceleration * (particle best position - current position)
= 0.07 * ([ -2, -5] - [ -2, -5])
= 0.07 * 0
= 0
```

---

**Social component:**

```
social acceleration = social constant * random social number
= 0.45 * 0.3
= 0.135

social acceleration * (swarm best position - current position)
= 0.135 * ([ 7, 1] - [ -2, -5])
= 0.135 * sqrt( (7 - (-2))^2 + (1 - (-5))^2)    Distance formula: sqrt((x1 - x2)^2 + (y1 - y2)^2)
= 0.135 * 10.817
= 1.46
```

---

**New velocity:**

```
inertia component + cognitive component + social component
= 0 + 0 + 1.46
= 1.46
```

**Figure 7.23 Particle velocity calculation walkthrough**

After these calculations have been completed for all particles, the velocity of each particle is updated, as represented in table 7.5.

**Table 7.5 Data attributes for each particle**

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum</b>	<b>Current plastic</b>	<b>Current fitness</b>	<b>Best aluminum</b>	<b>Best plastic</b>	<b>Best fitness</b>
1	0	7	1	104	7	1	104
2	1.52	-1	9	104	-1	9	104
3	2.295	-10	1	801	-10	1	801
4	1.46	-2	-5	557	-2	-5	557

**POSITION UPDATE**

Now that we understand how velocity is updated, we can update the current position of each particle, using the new velocity (figure 7.24).

**Position:**

current position + new velocity

**New position:**

current position + new velocity

$$= ([-2, -5]) + 1.46$$

$$= [-0.54, -3.54]$$

**Figure 7.24 Calculating the new position of a particle**

By adding the current position and new velocity, we can determine the new position of each particle and update the table of particle attributes with the new velocities. Then the fitness of each particle is calculated again, given its new position, and its best position is remembered (table 7.6).

**Table 7.6 Data attributes for each particle**

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum</b>	<b>Current plastic</b>	<b>Current fitness</b>	<b>Best aluminum</b>	<b>Best plastic</b>	<b>Best fitness</b>
1	0	7	1	104	7	1	104
2	1.52	0. 52	10.52	255.3	-1	9	104
3	2.295	-7.71	3.3	358.8	- 7.71	3.3	358.8
4	1. 46	-0.54	-3. 54	306.3	-0. 54	-3. 54	306.3

Calculating the initial velocity for each particle in the first iteration is fairly simple because there was no previous best position for each particle—only a swarm best position that affected only the social component.

Let's examine what the velocity update calculation will look like with the new information for each particle's best position and the swarm's new best position. Figure 7.25 describes the calculation for particle 4 in the list.

**Inertia component:**

```

inertia * current velocity
= 0.2 * 1.46
= 0.292

```

---

**Cognitive component:**

```

cognitive acceleration = cognitive constant * random cognitive number
= 0.35 * 0.2
= 0.07

cognitive acceleration * (particle best position - current position)
= 0.07 * ([-0.54, -3.54] - [-0.54, -3.54])
= 0.07 * 0
= 0

```

---

**Social component:**

```

social acceleration = social constant * random social number
= 0.45 * 0.3
= 0.135

social acceleration * (swarm best position - current position)
= 0.135 * ([7, 1] - [-0.54, -3.54])
= 0.135 * sqrt((7 - (-0.54))^2 + (1 - (-3.54))^2)
= 0.135 * 8.80
= 1.188

```

---

**New velocity:**

```

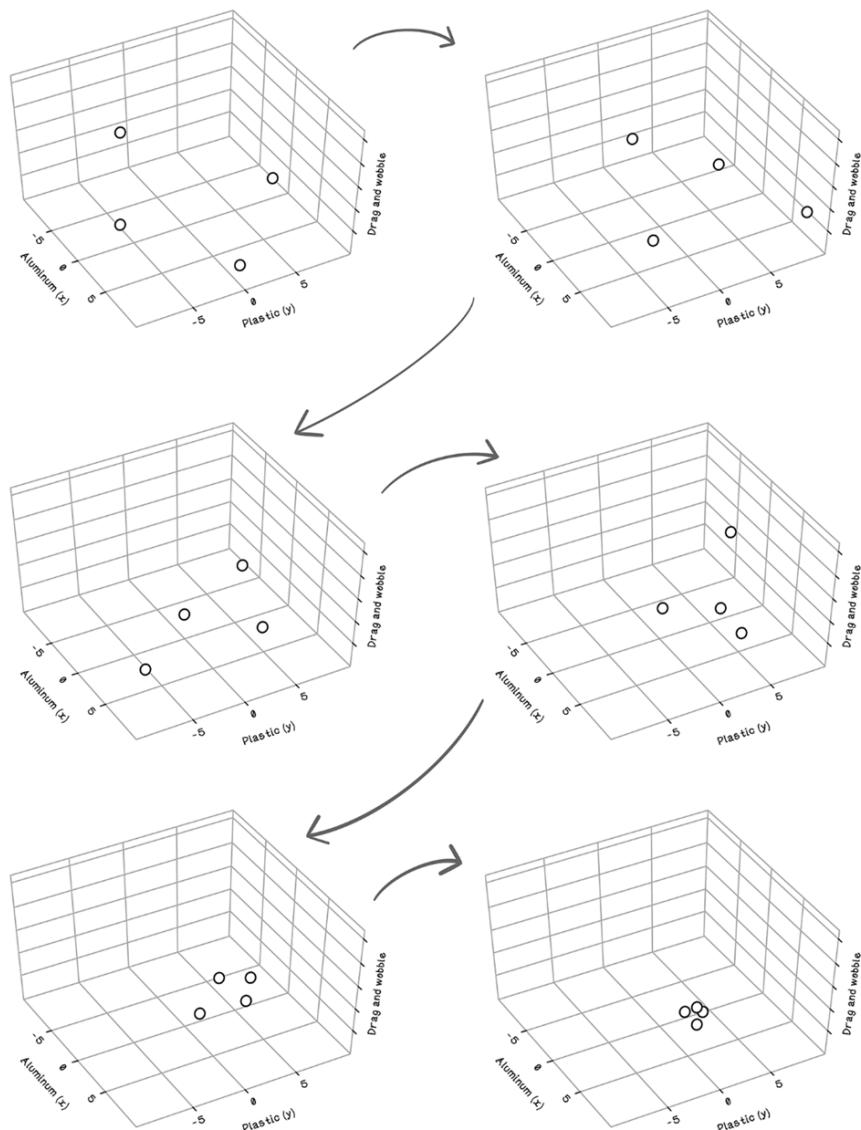
inertia component + cognitive component + social component
= 0.292 + 0 + 1.188
= 1.48

```

**Figure 7.25 Particle velocity calculation walkthrough**

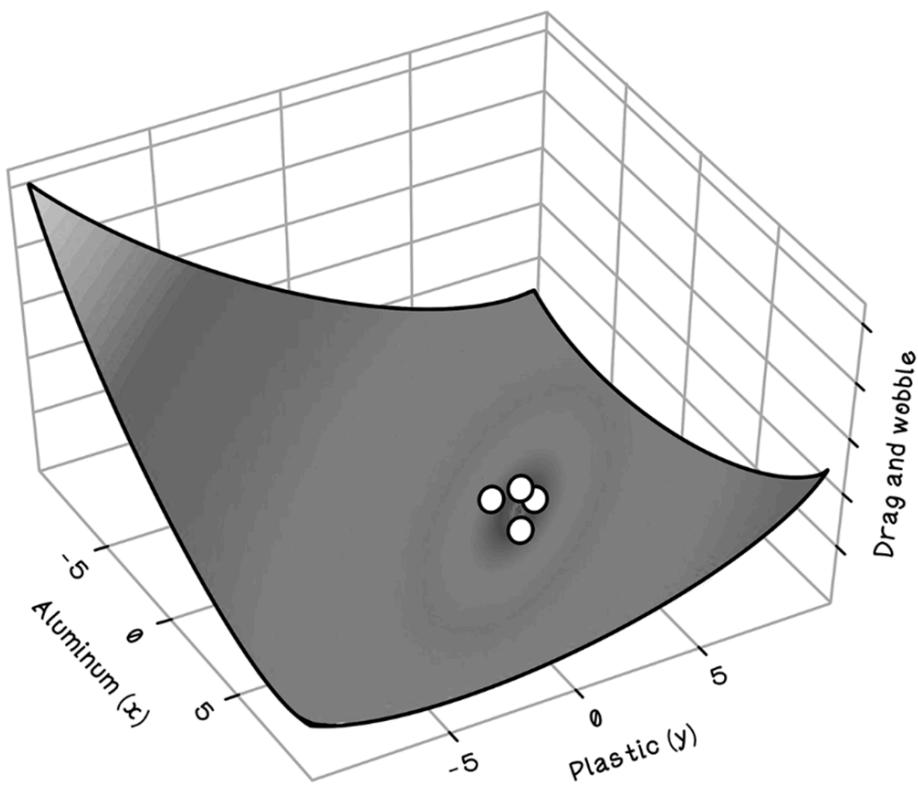
In this scenario, the cognitive component and the social component both play a role in updating the velocity, whereas the scenario described in figure 7.23 is influenced by the social component, due to it being the first iteration.

Particles move to different positions over several iterations. Figure 7.26 depicts the particles' movement and their convergence on a solution.



**Figure 7.26 A visualization of the movement of particles in the search space**

In the last frame of figure 7.26, all the particles have converged in a specific region in the search space. The best solution from the swarm will be used as the final solution. In real-world optimization problems, it is not possible to visualize the entire search space (which would make optimization algorithms unnecessary). To demonstrate this, we used a standard mathematical benchmark known as the Booth Function. Why use a test function? In real-world problems, we rarely know the perfect answer ahead of time. However, to test if an algorithm like PSO works, scientists use specific mathematical shapes (like the Booth Function) where the global minimum is already known (in this case, at  $x=1$ ,  $y=3$ ). This allows us to verify that our swarm actually converged on the correct target.



**Figure 7.27 Visualization of convergence of particles and a known surface**

After using the particle swarm optimization algorithm for the drone example, we find that the optimal ratio of aluminum and plastic to minimize drag and wobble is 1:3—that is, 1 part aluminum and 3 parts plastic. When we feed these values into the fitness function, the result is 0, which is the minimum value for the function.

## PYTHON CODE SAMPLE

The update step can seem to be daunting, but if the components are broken into simple focused functions, the code becomes simpler and easier to write, use, and understand. The first functions are the inertia calculation function, the cognitive acceleration function, and the social acceleration function.

```
def calculate_inertia(inertia, current_velocity):
    return current_velocity[0] * inertia, current_velocity[1] * inertia

def calculate_acceleration(constant, random_factor):
    return constant * random_factor
```

The cognitive component is calculated by finding the cognitive acceleration, using the function that we defined in an earlier section, and the distance between the particle's best position and its current position:

```
def calculate_cognitive(self,
                      cognitive_constant,
                      cognitive_random,
                      particle_best_position_x,
                      particle_best_position_y,
                      particle_current_position_x,
                      particle_current_position_y):
    cognitive_acceleration = self.calculate_acceleration(cognitive_constant,
cognitive_random)
    return (
        cognitive_acceleration * (particle_best_position_x -
particle_current_position_x),
        cognitive_acceleration * (particle_best_position_y -
particle_current_position_y),
    )
```

The social component is calculated by finding the social acceleration, using the function that we defined earlier, and the distance between the swarm's best position and the particle's current position:

```

def calculate_social(self,
                     social_constant,
                     social_random,
                     swarm_best_position_x,
                     swarm_best_position_y,
                     particle_current_position_x,
                     particle_current_position_y):
    social_acceleration = self.calculate_acceleration(social_constant,
social_random)
    return (
        social_acceleration * (swarm_best_position_x -
particle_current_position_x),
        social_acceleration * (swarm_best_position_y -
particle_current_position_y),
    )

```

The update function wraps everything that we have defined to carry out the actual update of a particle's velocity and position. The velocity is calculated by using the inertia component, cognitive component, and social component. The position is calculated by adding the new velocity to the particle's current position:

```

def update(self, swarm_best_x, swarm_best_y):
    i = self.calculate_inertia(self.inertia, self.velocity)
    c = self.calculate_cognitive(self.cognitive_constant, random.random(),
self.x, self.y, self.best_x, self.best_y)
    s = self.calculate_social(self.social_constant, random.random(), self.x,
self.y, swarm_best_x, swarm_best_y)
    v = self.calculate_updated_velocity(i, c, s)
    self.velocity = v
    p = self.calculate_position(self.x, self.y, v)
    self.x = p[0]
    self.y = p[1]

```

**Exercise: Calculate the new velocity and position for particle 1 given the following information about the particles**

- Inertia is set to 0.1.
- The cognitive constant is set to 0.5, and the cognitive random number is 0.2.

- The social constant is set to 0.5, and the social random number is 0.5.

<b>Particle</b>	<b>Velocity</b>	<b>Current aluminum</b>	<b>Current plastic</b>	<b>Current fitness</b>	<b>Best aluminum</b>	<b>Best plastic</b>	<b>Best fitness</b>
1	3	4	8	721.286	7	1	296
2	4	3	3	73.538	0.626	10	73.538
3	1	6	2	302.214	-10	1	80
4	2	2	5	179.105	-0.65	-3.65	179.105

**SOLUTION: CALCULATE THE NEW VELOCITY AND POSITION FOR PARTICLE 1 GIVEN THE FOLLOWING INFORMATION ABOUT THE PARTICLES**

Inertia component:

$$\begin{aligned} \text{inertia * current velocity} \\ = 0.1 * 3 \\ = 0.3 \end{aligned}$$

---

Cognitive component:

$$\begin{aligned} \text{cognitive acceleration} &= \text{cognitive constant} * \text{random cognitive number} \\ &= 0.5 * 0.2 \\ &= 0.1 \\ \\ \text{cognitive acceleration} &* (\text{particle best position} - \text{current position}) \\ &= 0.1 * ([7,1] - [4,8]) \\ &= 0.1 * \sqrt{((7 - 4)^2 + (1 - 8)^2)} \\ &= 0.1 * 7.616 \\ &= 0.7616 \end{aligned}$$

---

Social component:

$$\begin{aligned} \text{social acceleration} &= \text{social constant} * \text{random social number} \\ &= 0.5 * 0.5 \\ &= 0.25 \\ \\ \text{social acceleration} &* (\text{swarm best position} - \text{current position}) \\ &= 0.25 * ([0.626,10] - [4,8]) \\ &= 0.25 * \sqrt{((0.626 - 4)^2 + (10 - 8)^2)} \\ &= 0.25 * 3.922 \\ &= 0.981 \end{aligned}$$

---

New velocity:

$$\begin{aligned} \text{inertia component} &+ \text{cognitive component} + \text{social component} \\ &= 0.3 + 0.7616 + 0.981 \\ &= 2.0426 \end{aligned}$$

New position:

current position + new velocity

$$= [4, 8] + 2.0426$$

$$= [6.0426, 10.0426]$$

#### 7.5.4 Determine the stopping criteria

The particles in the swarm cannot keep updating and searching indefinitely. A stopping criterion needs to be determined to allow the algorithm to run for a reasonable number of iterations to find a suitable solution (figure 7.28).

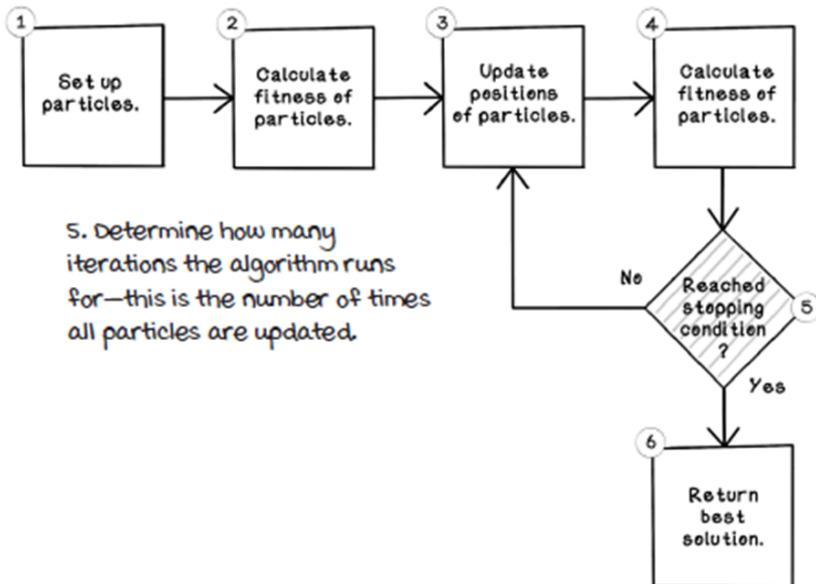
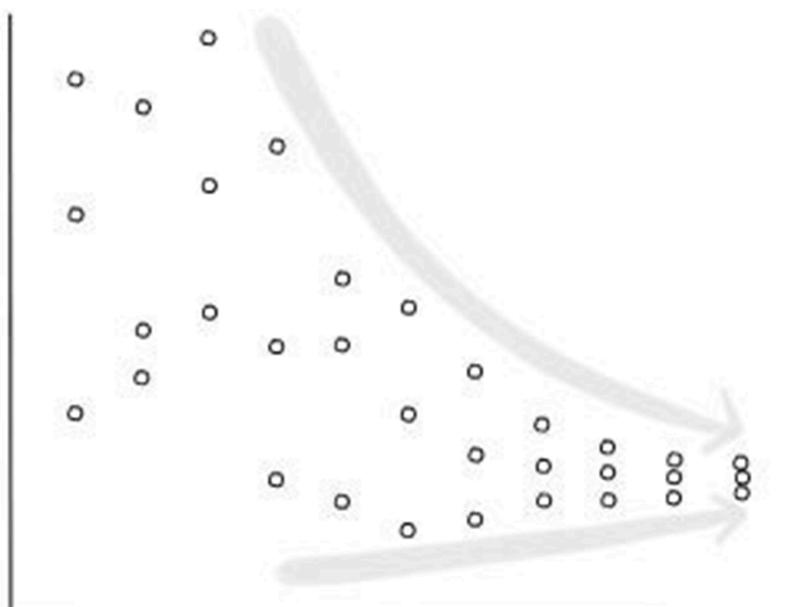


Figure 7.28 Has the algorithm reached a stopping condition?

The number of iterations influences several aspects of finding solutions, including:

- *Exploration*—Particles require time to explore the search space to find areas with better solutions. Exploration is also influenced by the constants defined in the update velocity function.
- *Exploitation*—Particles should converge on a good solution after reasonable exploration occurs.

A strategy to stop the algorithm is to examine the best solution in the swarm and determine whether it is stagnating. Stagnation occurs when the value of the best solution doesn't change or doesn't change by a significant amount. Running more iterations in this scenario will not help find better solutions. When the best solution stagnates, the parameters in the update function can be adjusted to favor more exploration. If more exploration is desired, this adjustment usually means more iterations. Stagnation could mean that a good solution was found or that the swarm is stuck on a local best solution. If enough exploration occurred at the start, and the swarm gradually stagnates, the swarm has converged on a good solution (figure 7.29).



**Figure 7.29 Exploration converging and exploiting**

## 7.6 Use cases for particle swarm optimization algorithms

Particle swarm optimization algorithms are interesting because they simulate a natural phenomenon, which makes them easier to understand, but they can be applied to a range of problems at different levels of abstraction. This chapter looked at an optimization problem for drone manufacturing, but particle swarm optimization algorithms can be used in conjunction with other algorithms, such as artificial neural networks, playing a small but critical role in finding good solutions.

One interesting application of a particle swarm optimization algorithm is deep brain stimulation. The concept involves installing probes with electrodes into the human brain to stimulate it to treat conditions such as Parkinson's disease. Each probe contains electrodes that can be configured in different directions to treat the condition correctly per patient. Researchers at the University of Minnesota have developed a particle swarm optimization algorithm to optimize the direction of each electrode to maximize the region of interest, minimize the region of avoidance, and minimize energy use. Because particles are effective in searching these multidimensional problem spaces, the particle swarm optimization algorithm is effective for finding optimal configurations for electrodes on the probes (figure 7.30).

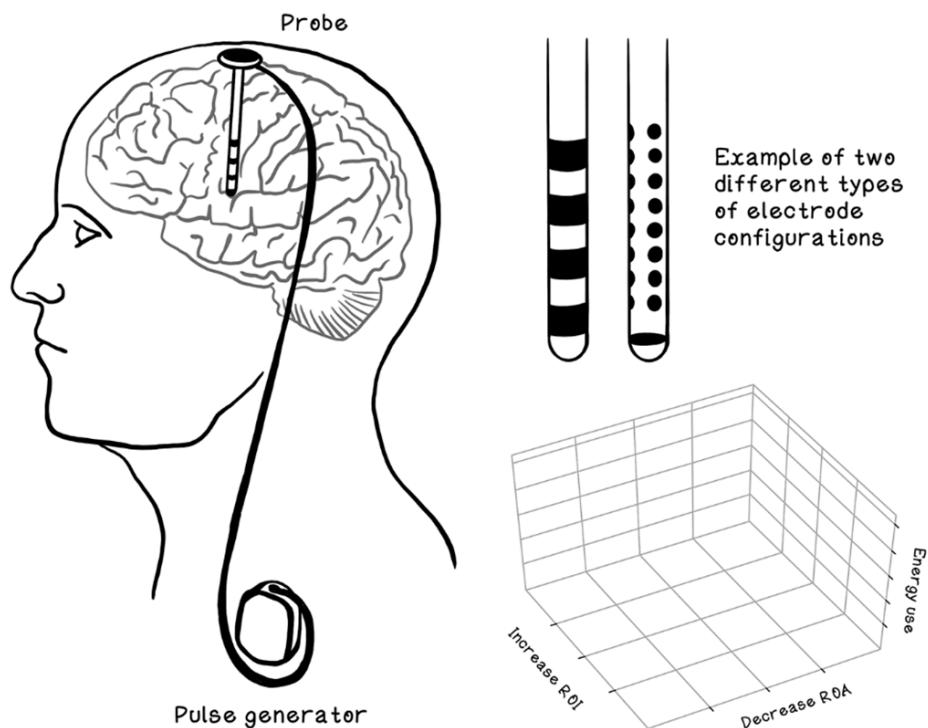


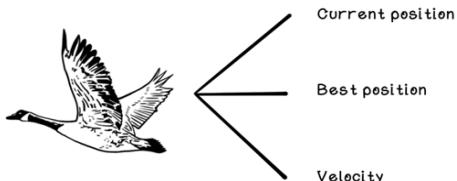
Figure 7.30 Example of factors involved for probes in deep brain stimulation

Here are some other real-world applications of particle swarm optimization algorithms:

- *Optimizing weights in an artificial neural network*—Artificial neural networks are modeled on an idea of how the human brain works. Neurons pass signals to other neurons, and each neuron adjusts the signal before passing it on. An artificial neural network uses weights to adjust each signal. The power of the network is finding the right balance of weights to form patterns in relationships of the data. Adjusting weights is computationally expensive, as the search space is massive. Imagine having to brute-force every possible decimal number combination for 10 weights. That process would take years.
- Don't panic if this concept sounds confusing. We explore how artificial neural networks operate in chapter 9. Particle swarm optimization can be used to adjust the weights of neural networks faster, because it seeks optimal values in the search space without exhaustively attempting each one.
- *Motion tracking in videos*—Motion tracking of people is a challenging task in computer vision. The goal is to identify the poses of people and imply a motion by using the information from the images in the video alone. People move differently, even though their joints move similarly. Because the images contain many aspects, the search space becomes large, with many dimensions to predict the motion for a person. Particle swarm optimization works well in high-dimension search spaces and can be used to improve the performance of motion tracking and prediction.
- *Speech enhancement in audio*—Audio recordings are nuanced. There is always background noise that may interfere with what someone is saying in the recording. A solution is to remove the noise from recorded speech audio clips. A technique used for this purpose is filtering the audio clip with noise and comparing similar sounds to remove the noise in the audio clip. This solution is still complex, as reduction of certain frequencies may be good for parts of the audio clip but may deteriorate other parts of it. Fine searching and matching must be done for good noise removal. Traditional methods are slow, as the search space is large. Particle swarm optimization works well in large search spaces and can be used to speed the process of removing noise from audio clips.

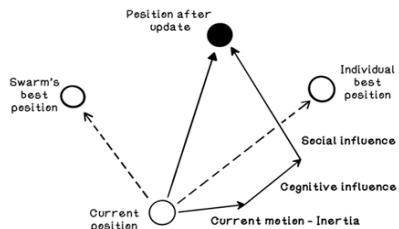
## 7.7 Summary of particle swarm optimization

Particle Swarm Optimization finds good solutions in very large search spaces



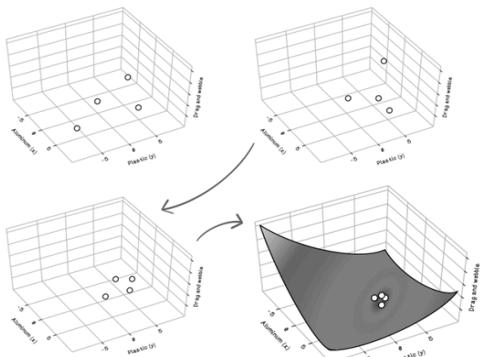
Adjusting the particles' velocity is the critical step of the PSO algorithm using inertia, cognitive influence, and social influence

Particles use their best position and the swarm's best position to move through the search space



New velocity:

inertia component + social component + cognitive component  
 $\downarrow$   
 (inertia \* current velocity)  
 $\downarrow$   
 (social acceleration \* (swarm best position - current position))  
 $\downarrow$   
 (cognitive acceleration \* (particle best position - current position))



Particles move through the search space while finding different good solutions and ideally converging on a global best solution

# 8 Machine learning

## This chapter covers

- Solving problems with machine learning algorithms
- Grasping a machine learning life cycle, preparing data, and selecting algorithms
- Understanding and implementing a linear-regression algorithm for predictions
- Understanding and implementing a decision-tree learning algorithm for classification
- Gaining intuition about other machine learning algorithms and their usefulness

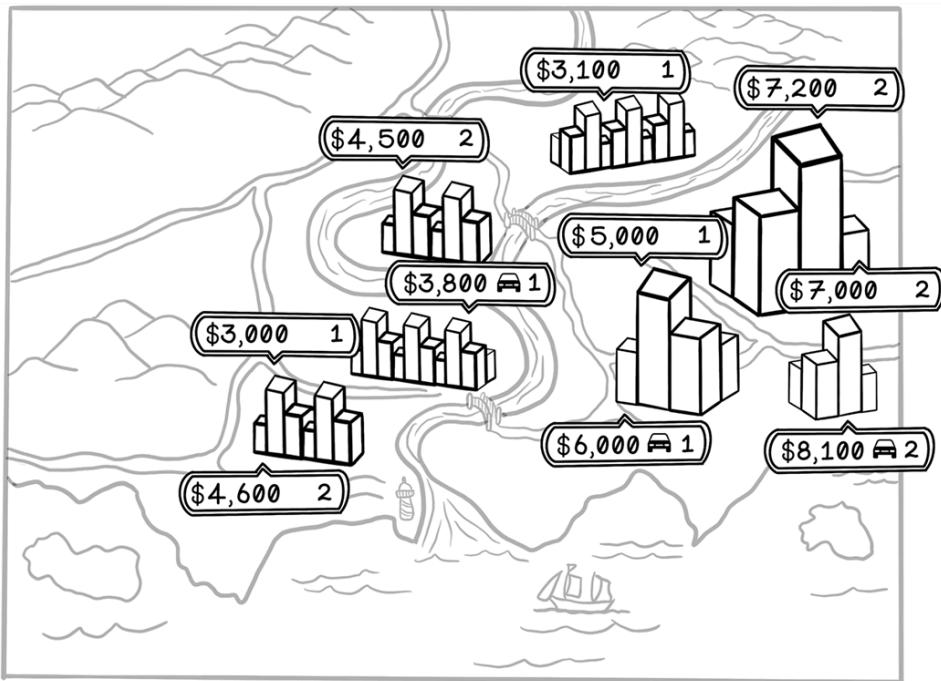
## 8.1 What is machine learning?

Machine learning can seem like a daunting concept to learn and apply, but with the right framing and understanding of the process and algorithms, it can be interesting and fun.

Suppose that you're looking for a new apartment. You speak to friends and family, and do some online searches for apartments in the city. You notice that apartments in different areas are priced differently. Here are some of your observations from all your research:

- A one-bedroom apartment in the city center (close to work) costs \$5,000 per month.
- A two-bedroom apartment in the city center costs \$7,000 per month.
- A one-bedroom apartment in the city center with a garage costs \$6,000 per month.
- A one-bedroom apartment outside the city center, where you will need to commute to work, costs \$3,000 per month.
- A two-bedroom apartment outside the city center costs \$4,500 per month.
- A one-bedroom apartment outside the city center with a garage costs \$3,800 per month.

You notice some patterns. Apartments in the city center are most expensive and are usually between \$5,000 and \$7,000 per month. Apartments outside the city are cheaper. Increasing the number of rooms adds between \$1,500 and \$2,000 per month, and access to a garage adds between \$800 and \$1,000 per month (figure 8.1).

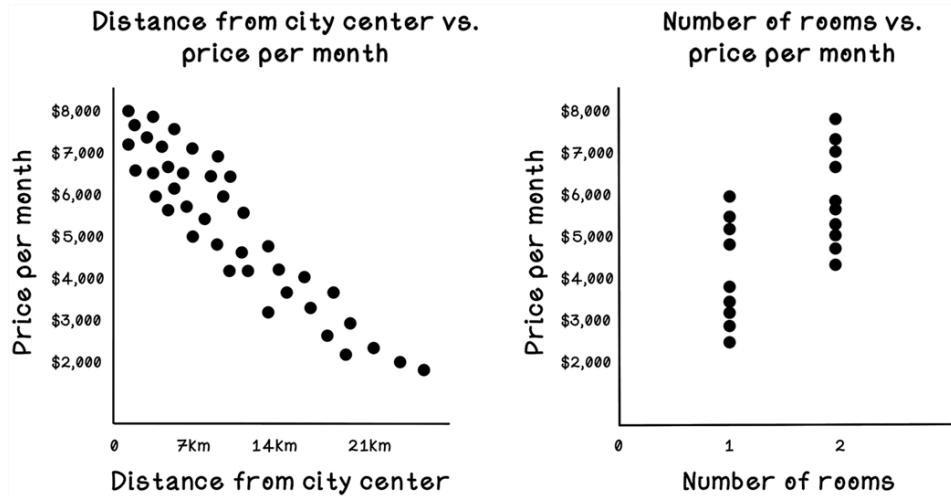


**Figure 8.1 An illustration of property prices and features in different regions**

This example shows how we use data to find patterns and make decisions. If you encounter a two-bedroom apartment in the city center with a garage, it's reasonable to assume that the price would be approximately \$8,000 per month.

*Machine learning* aims to find patterns in data for useful applications in the real world. We could spot the pattern in this small dataset, but machine learning spots them for us in large, complex datasets. Figure 8.2 depicts the relationships among different attributes of the data. Each dot represents an individual property.

Notice that there are more dots closer to the city center and that there is a clear pattern related to price per month: the price gradually drops as distance to the city center increases. There is also a pattern in the price per month related to the number of rooms; the gap between the bottom cluster of dots and the top cluster shows that the price jumps significantly. We could naïvely assume that this effect may be related to the distance from the city center. Machine learning algorithms can help us validate or invalidate this assumption. We dive into how this process works throughout this chapter.

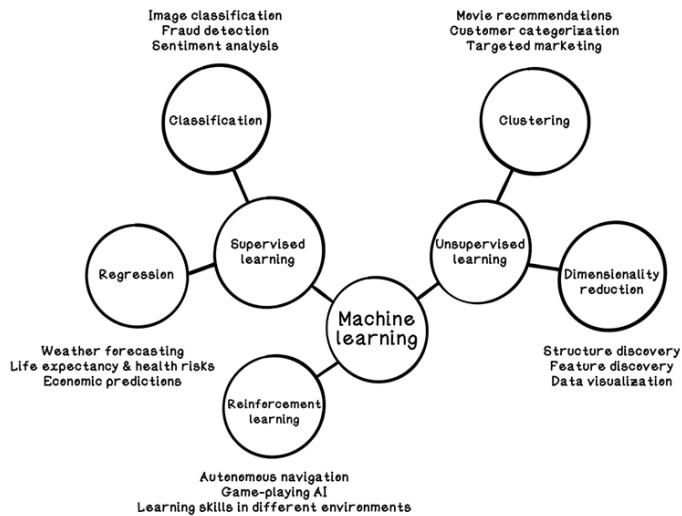


**Figure 8.2 Example visualization of relationships among data**

Typically, data is represented in tables. The columns are referred to as *features* of the data, and the rows are referred to as *examples*. When we compare two features, the feature being measured is sometimes represented as  $y$ , and the features being changed are grouped as  $x$ . We will gain a better intuition for this terminology as we work through some problems.

## 8.2 Problems applicable to machine learning

Machine learning is useful only if you have data and have questions to ask that the data might answer. Machine learning algorithms find patterns in data but cannot do useful things magically. Different categories of machine learning algorithms use different approaches to answer different questions. These broad categories are supervised learning, unsupervised learning, and reinforcement learning (figure 8.3).



**Figure 8.3 Categorization of machine learning and uses**

### 8.2.1 Supervised learning

One of the most common techniques in traditional machine learning is *supervised learning*. We want to look at data, understand the patterns and relationships among the data, and predict the results if we are given new examples of different data in the same format. The apartment-finding problem is an example of supervised learning to find the pattern. We also see this example in action when we type a search that autocompletes or when music applications suggest new songs to listen to based on our activity and preference. Supervised learning has two subcategories: regression and classification.

*Regression* involves drawing a line through a set of data points to most closely fit the overall shape of the data. Regression can be used for applications such as trends between marketing initiatives and sales. (Is there a direct relationship between marketing through online ads and actual sales of a product?) It can also be used to determine factors that affect something. (Is there a direct relationship between time and the value of cryptocurrency, and will cryptocurrency increase exponentially in value as time passes?)

*Classification* aims to predict categories of examples based on their features. (Can we determine whether something is a car or a truck based on its number of wheels, weight, and top speed?)

### 8.2.2 Unsupervised learning

*Unsupervised learning* involves finding underlying patterns in data that may be difficult to find by inspecting the data manually. Unsupervised learning is useful for clustering data that has similar features and uncovering features that are important in the data. On an e-commerce site, for example, products might be clustered based on customer purchase behavior. If many customers purchase soap, sponges, and towels together, it is likely that more customers would want that combination of products, so soap, sponges, and towels would be clustered and recommended to new customers.

### 8.2.3 Reinforcement learning

*Reinforcement learning* is inspired by behavioral psychology and operates by rewarding or punishing an algorithm based on its actions in an environment. It has similarities to supervised learning and unsupervised learning, as well as many differences. Reinforcement learning aims to train an agent in an environment based on rewards and penalties. Imagine rewarding a pet for good behavior with treats; the more it is rewarded for a specific behavior, the more it will exhibit that behavior. We discuss reinforcement learning in chapter 10.

## 8.3 A machine learning workflow

Machine learning isn't just about algorithms. In fact, it is often about the context of the data, the preparation of the data, and the questions that are asked.

We can find questions in two ways:

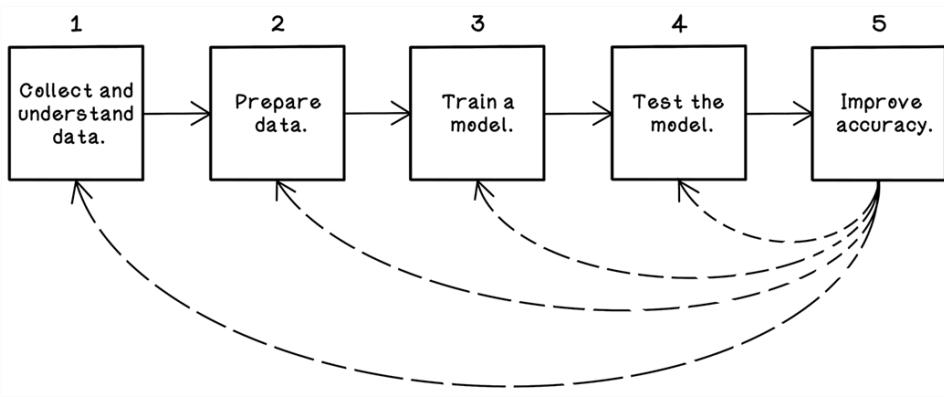
- A problem can be solved with machine learning, and the right data needed can be collected to help solve it. Suppose that a bank has a vast amount of transaction data of legitimate and fraudulent transactions, and it wants to train a model with this question: "Can we detect fraudulent transactions in real time?"
- We have data in a specific context and want to determine how it can be used to solve several problems. An agriculture company, for example, might have data about the weather in different locations, nutrition required for different plants, and the soil content in different locations. The question might be "What correlations and relationships can we find among the different types of data?" These relationships may inform a more concrete question, such as "Can we determine the best location for growing a specific plant based on the weather and soil in that location?"

Think of the machine learning lifecycle like learning to cook a new dish:

1. *Collect Data:* Going to the market to buy ingredients. You need fresh, high-quality produce (data) to make a good meal.

2. *Prepare Data*: Washing vegetables, peeling potatoes, and chopping onions. You can't just throw raw, dirty ingredients into the pot; you have to clean and format them first.
3. *Train Model*: The actual cooking process. You combine ingredients, apply heat, and taste the stew.
4. *Test Model*: Serving it to a customer. Do they like it? Is it too salty?
5. *Improve Accuracy*: Adjusting the recipe based on feedback. Next time, you add less salt or cook it longer.

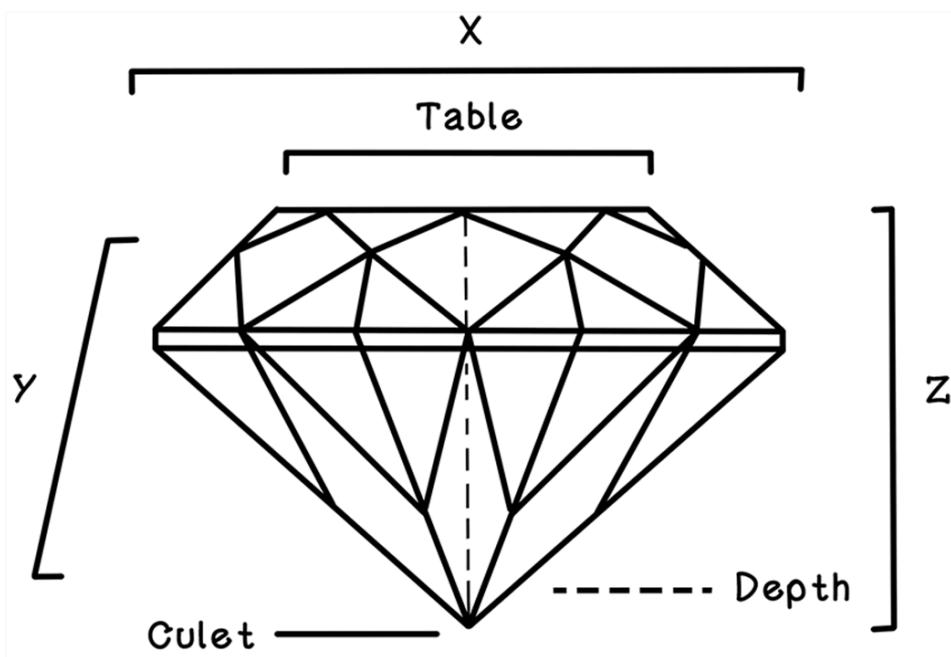
Figure 8.4 is a simplified view of the steps involved in a typical machine learning endeavor.



**Figure 8.4 A workflow for machine learning experiments and projects**

### 8.3.1 Collecting and understanding data: Know your context

Collecting and understanding the data you're working with is paramount to a successful machine learning endeavor. If you're working in a specific area in the finance industry, knowledge of the terminology and workings of the processes and data in that area is important for sourcing data that is best to help answer questions for the goal you're trying to achieve. If you want to build a fraud detection system, understanding what data is stored about transactions and what it means is critical to identifying fraudulent transactions. Data may also need to be sourced from various systems and combined to be effective. Sometimes, the data we use is augmented with data from outside the organization to enhance the accuracy. In this section, we use an example dataset about diamond measurements to understand the machine learning workflow and explore various algorithms (figure 8.5).



**Figure 8.5 Terminology of diamond measurements**

Table 8.1 describes several diamonds and their properties. X, Y, and Z describe the size of a diamond in the three spatial dimensions. Only a subset of data is used in the examples.

**Table 8.1 The diamond dataset**

	<b>Carat</b>	<b>Cut</b>	<b>Color</b>	<b>Clarity</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.30	Good	J	SI1	64.0	55	339	4.25	4.28	2.73
2	0.41	Ideal	I	SI1	61.7	55	561	4.77	4.80	2.95
3	0.75	Very Good	D	SI1	63.2	56	2,760	5.80	5.75	3.65
4	0.91	Fair	H	SI2	65.7	60	2,763	6.03	5.99	3.95
5	1.20	Fair	F	I1	64.6	56	2,809	6.73	6.66	4.33
6	1.31	Premium	J	SI2	59.7	59	3,697	7.06	7.01	4.20
7	1.50	Premium	H	I1	62.9	60	4,022	7.31	7.22	4.57
8	1.74	Very Good	H	I1	63.2	55	4,677	7.62	7.59	4.80
9	1.96	Fair	I	I1	66.8	55	6,147	7.62	7.60	5.08
10	2.21	Premium	H	I1	62.2	58	6,535	8.31	8.27	5.16

The diamond dataset consists of 10 columns of data, which are referred to as *features*. The full dataset has more than 50,000 rows. Here's what each feature means:

- *Carat*—The weight of the diamond. Out of interest: 1 carat equals 200 mg.
- *Cut*—The quality of the diamond, by increasing quality: fair, good, very good, premium, and ideal.
- *Color*—The color of the diamond, ranging from D to J, where D is the best color and J is the worst color. D indicates a clear diamond, and J indicates a foggy one.
- *Clarity*—The imperfections of the diamond, by decreasing quality: FL, IF, VVS1, VVS2, VS1, VS2, SI1, SI2, I1, I2, and I3. (Don't worry about understanding these code names; they simply represent different levels of perfection.)
- *Depth*—The percentage of depth, which is measured from the culet to the table of the diamond. Typically, the table-to-depth ratio is important for the "sparkle" aesthetic of a diamond.
- *Table*—The percentage of the flat end of the diamond relative to the X dimension.
- *Price*—The price of the diamond when it was sold.
- *X*—The x dimension of the diamond, in millimeters.
- *Y*—The y dimension of the diamond, in millimeters.
- *Z*—The z dimension of the diamond, in millimeters.

Keep this dataset in mind; we will be using it to see how data is prepared and processed by machine learning algorithms.

### 8.3.2 Preparing data: Clean and wrangle

Real-world data is never ideal to work with. Data might be sourced from different systems and different organizations, which may have different standards and rules for data integrity. There is always missing data, inconsistent data, and data in a format that is difficult to work with for the algorithms that we want to use.

In the sample diamond dataset in table 8.2, again, it is important to understand that the columns are referred to as the *features* of the data and that each row is an *example*.

**Table 8.2 The diamond dataset with missing or inconsistent data**

	<b>Carat</b>	<b>Cut</b>	<b>Color</b>	<b>Clarity</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.30	Good	J	SI1	64.0	55	339	4.25	4.28	2.73
2	0.41	Ideal	I	si1	61.7	55	561	4.77	4.80	2.95
3	0.75	Very Good	D	SI1	63.2	56	2,760	5.80	5.75	3.65
4	0.91	-	H	SI2	-	60	2,763	6.03	5.99	3.95
5	1.20	Fair	F	I1	64.6	56	2,809	6.73	6.66	4.33
6	1.21	Good	E	I1	57.2	62	3,144	7.01	6.96	3.99
7	1.31	Premium	J	SI2	59.7	59	3,697	7.06	7.01	4.20
8	1.50	Premium	H	I1	62.9	60	4,022	7.31	7.22	4.57
9	1.74	Very Good	H	i1	63.2	55	4,677	7.62	7.59	4.80
10	1.83	fair	J	I1	70.0	58	5,083	7.34	7.28	5.12
11	1.96	Fair	I	I1	66.8	55	6,147	7.62	7.60	5.08
12	-	Premium	H	i1	62.2	-	6,535	8.31	-	5.16

### MISSING DATA

In table 8.2, example 4 is missing values for the Cut and Depth features, and example 12 is missing values for Carat, Table, and Y. To compare examples, we need complete understanding of the data, and missing values make this difficult. A goal for a machine learning project might be to estimate these values; we cover estimations in the upcoming material. Assume that missing data will be problematic in our goal to use it for something useful. Here are some ways to deal with missing data:

- *Remove*—Remove the examples that have missing values for features—in this case, examples 4 and 12 (table 8.3). The benefit of this approach is that the data is more reliable because nothing is assumed; however, the removed examples may have been important to the goal we’re trying to achieve.

**Table 8.3 The diamond dataset with missing data: removing examples**

	<b>Carat</b>	<b>Cut</b>	<b>Color</b>	<b>Clarity</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.30	Good	J	SI1	64.0	55	339	4.25	4.28	2.73
2	0.41	Ideal	I	si1	61.7	55	561	4.77	4.80	2.95
3	0.75	Very Good	D	SI1	63.2	56	2,760	5.80	5.75	3.65
4	0.91	-	H	SI2	-	60	2,763	6.03	5.99	3.95
5	1.20	Fair	F	I1	64.6	56	2,809	6.73	6.66	4.33
6	1.21	Good	E	I1	57.2	62	3,144	7.01	6.96	3.99
7	1.31	Premium	J	SI2	59.7	59	3,697	7.06	7.01	4.20
8	1.50	Premium	H	I1	62.9	60	4,022	7.31	7.22	4.57
9	1.74	Very Good	H	i1	63.2	55	4,677	7.62	7.59	4.80
10	1.83	fair	J	I1	70.0	58	5,083	7.34	7.28	5.12
11	1.96	Fair	I	I1	66.8	55	6,147	7.62	7.60	5.08
12	-	Premium	H	i1	62.2	-	6,535	8.31	-	5.16

- *Mean or median*—Another option is to replace the missing values with the mean or median for the respective feature.  
The *mean* is the average calculated by adding all the values and dividing by the number of examples. The *median* is calculated by ordering the examples by value ascending and choosing the value in the middle.  
Using the mean is easy and efficient to do but doesn’t take into account possible correlations between features. This approach cannot be used with categorical features such as the Cut, Clarity, and Depth features in the diamond dataset (table 8.4).

**Table 8.4 The diamond dataset with missing data: using mean values**

	<b>Carat</b>	<b>Cut</b>	<b>Color</b>	<b>Clarity</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.30	Good	J	SI1	64.0	55	339	4.25	4.28	2.73
2	0.41	Ideal	I	si1	61.7	55	561	4.77	4.80	2.95
3	0.75	Very Good	D	SI1	63.2	56	2,760	5.80	5.75	3.65
4	0.91	-	H	SI2	-	60	2,763	6.03	5.99	3.95
5	1.20	Fair	F	I1	64.6	56	2,809	6.73	6.66	4.33
6	1.21	Good	E	I1	57.2	62	3,144	7.01	6.96	3.99
7	1.31	Premium	J	SI2	59.7	59	3,697	7.06	7.01	4.20
8	1.50	Premium	H	I1	62.9	60	4,022	7.31	7.22	4.57
9	1.74	Very Good	H	i1	63.2	55	4,677	7.62	7.59	4.80
10	1.83	fair	J	I1	70.0	58	5,083	7.34	7.28	5.12
11	1.96	Fair	I	I1	66.8	55	6,147	7.62	7.60	5.08
12	1.19	Premium	H	i1	62.2	57	6,535	8.31	-	5.16

To calculate the mean of the Table feature, we add every available value and divide the total by the number of values used:

```
Table mean = (55 + 55 + 56 + 60 + 56 + 62 + 59 + 60 + 55 + 58 + 55) / 11
Table mean = 631 / 11
Table mean = 57.364
```

Using the Table mean for the missing values seems to make sense, because the table size doesn't seem to differ radically among different examples of data. But there could be correlations that we do not see, such as the relationship between the table size and the width of the diamond (X dimension).

On the other hand, using the Carat mean does not make sense, because we can see a correlation between the Carat feature and the Price feature if we plot the data on a graph. The price seems to increase as the Carat value increases.

- *Most frequent*—Replace the missing values with the value that occurs most often for that feature, which is known as the *mode* of the data. This approach works well with categorical features but doesn't take into account possible correlations among features, and it can introduce bias by using the most frequent values.
- *(Advanced) Statistical approaches*—Use k-nearest neighbor, or neural networks. K-nearest neighbor uses many features of the data to find an estimated value. Similar to k-nearest neighbor, a neural network can predict the missing values accurately, given enough data. Both algorithms are computationally expensive for the purpose of handling missing data.
- *(Advanced) Do nothing*—Some algorithms handle missing data without any preparation, such as XGBoost, but the algorithms that we will be exploring will fail.

## AMBIGUOUS VALUES

Another problem is values that mean the same thing but are represented differently. Examples in the diamond dataset are rows 2, 9, 10, and 12. The values for the Cut and Clarity features are lowercase instead of uppercase. Note that we know this only because we understand these features and the possible values for them. Without this knowledge, we might see Fair and fair as different categories. To fix this problem, we can standardize these values to uppercase or lowercase to maintain consistency (table 8.5).

**Table 8.5 The diamond dataset with ambiguous data: standardizing values**

	<b>Carat</b>	<b>Cut</b>	<b>Color</b>	<b>Clarity</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.30	Good	J	SI1	64.0	55	339	4.25	4.28	2.73
2	0.41	Ideal	I	si1	61.7	55	561	4.77	4.80	2.95
3	0.75	Very Good	D	SI1	63.2	56	2,760	5.80	5.75	3.65
4	0.91	-	H	SI2	-	60	2,763	6.03	5.99	3.95
5	1.20	Fair	F	I1	64.6	56	2,809	6.73	6.66	4.33
6	1.21	Good	E	I1	57.2	62	3,144	7.01	6.96	3.99
7	1.31	Premium	J	SI2	59.7	59	3,697	7.06	7.01	4.20
8	1.50	Premium	H	I1	62.9	60	4,022	7.31	7.22	4.57
9	1.74	Very Good	H	i1	63.2	55	4,677	7.62	7.59	4.80
10	1.83	fair	J	I1	70.0	58	5,083	7.34	7.28	5.12
11	1.96	Fair	I	I1	66.8	55	6,147	7.62	7.60	5.08
12	1.19	Premium	H	i1	62.2	57	6,535	8.31	-	5.16

## ENCODING CATEGORICAL DATA

Because computers and statistical models work with numeric values, there will be a problem with modeling string values and categorical values such as Fair, Good, SI1, and I1. We need to represent these categorical values as numerical values. Here are ways to accomplish this task:

- *One-hot encoding*—Think about one-hot encoding as switches, all of which are off except one. The one that is on represents the presence of the feature at that position. If we were to represent Cut with one-hot encoding, the Cut feature becomes five different features, and each value is 0 except for the one that represents the Cut value for each respective example. Note that the other features have been removed in the interest of space in table 8.6.

**Table 8.6 The diamond dataset with encoded values**

	<b>Carat</b>	<b>Cut: Fair</b>	<b>Cut: Good</b>	<b>Cut: Very Good</b>	<b>Cut: Premium</b>	<b>Cut: Ideal</b>
1	0.30	0	1	0	0	0
2	0.41	0	0	0	0	1
3	0.75	0	0	1	0	0
4	0.91	0	0	0	0	0
5	1.20	1	0	0	0	0
6	1.21	0	1	0	0	0
7	1.31	0	0	0	1	0
8	1.50	0	0	0	1	0
9	1.74	0	0	1	0	0
10	1.83	1	0	0	0	0
11	1.96	1	0	0	0	0
12	1.19	0	0	0	1	0

- *Label encoding*—Represent each category as a number between 0 and the number of categories. This approach should be used only for ratings or rating-related labels; otherwise, the model that we will be training will assume that the number carries weight for the example and can introduce unintended bias.

### **EXERCISE: IDENTIFY AND FIX THE PROBLEM DATA IN THIS EXAMPLE**

Decide which data preparation techniques can be used to fix the following dataset. Decide which rows to delete, what values to use the mean for, and how categorical values will be encoded. Note that the dataset is slightly different from what we've been working with thus far.

	<b>Carat</b>	<b>Origin</b>	<b>Depth</b>	<b>Table</b>	<b>Price</b>	<b>X</b>	<b>Y</b>	<b>Z</b>
1	0.35	South Africa	64.0	55	450	4.25		2.73
2	0.42	Canada	61.7	55	680		4.80	2.95
3	0.87	Canada	63.2	56	2,689	5.80	5.75	3.65
4	0.99	Botswana	65.7		2,734	6.03	5.99	3.95
5	1.34	Botswana	64.6	56	2,901	6.73	6.66	
6	1.45	South Africa	59.7	59	3,723	7.06	7.01	4.20
7	1.65	Botswana	62.9	60	4,245	7.31	7.22	4.57
8	1.79		63.2	55	4,734	7.62	7.59	4.80
9	1.81	Botswana	66.8	55	6,093	7.62	7.60	5.08
10	2.01	South Africa	62.2	58	7,452	8.31	8.27	5.16

## SOLUTION: IDENTIFY AND FIX THE PROBLEM DATA IN THIS EXAMPLE

One approach for fixing this dataset involves the following three tasks:

- *Remove row 8 due to missing Origin.* We don't know what the dataset will be used for. If the Origin feature is important, and this row is removed after cleanup, it may cause issues. Alternatively, the value for this feature could be estimated if it has a relationship with other features.
- *Use one-hot encoding to encode the Origin column value.* In the example explored thus far in the chapter, we used label encoding to convert string values to numeric values. This approach worked because the values indicated more superior cut, clarity, or color. In the case of Origin, the value identifies where the diamond was sourced. By using label encoding, we introduce bias to the dataset, because no Origin location is better than another in this dataset.
- *Find the mean for missing values.* Row 1, 2, 4, and 5 are missing values for Y, X, Table, and Z, respectively. Using a mean value should be a good technique because, as we know about diamonds, the dimensions and table features are related.

## TESTING AND TRAINING DATA

Before we jump into training a linear regression model, we need to ensure that we have data to teach (or train) the model, as well as some data to test how well it does in predicting new examples. Think back to the property-price example. After gaining a feel for the attributes that affect price, we could make a price prediction by looking at the distance and number of rooms. For this example, we will use table 8.7 as the training data because we have more real-world data to use for testing later.

### 8.3.3 Training a model: Predict with linear regression

Choosing an algorithm to use is based largely on two factors: the question that is being asked and the nature of the data that is available. If the question is to make a prediction about the price of a diamond with a specific carat weight, regression algorithms can be useful. The algorithm choice also depends on the number of features in the dataset and the relationships among those features. If the data has many dimensions (there are many features to consider to make a prediction), we can consider several algorithms and approaches.

Regression means predicting a continuous value, such as the Price or Carat of the diamond. Continuous means that the values can be any number in a range. The price of \$2,271, for example, is a continuous value between 0 and the maximum price of any diamond that regression can help predict.

Linear regression is one of the simplest machine learning algorithms: it finds relationships between two variables and allows us to predict one variable given the other. An example is predicting the price of a diamond based on its carat value. By looking at many examples of known diamonds, including their Price and Carat values, we can teach a model the relationship and ask it to make predictions.

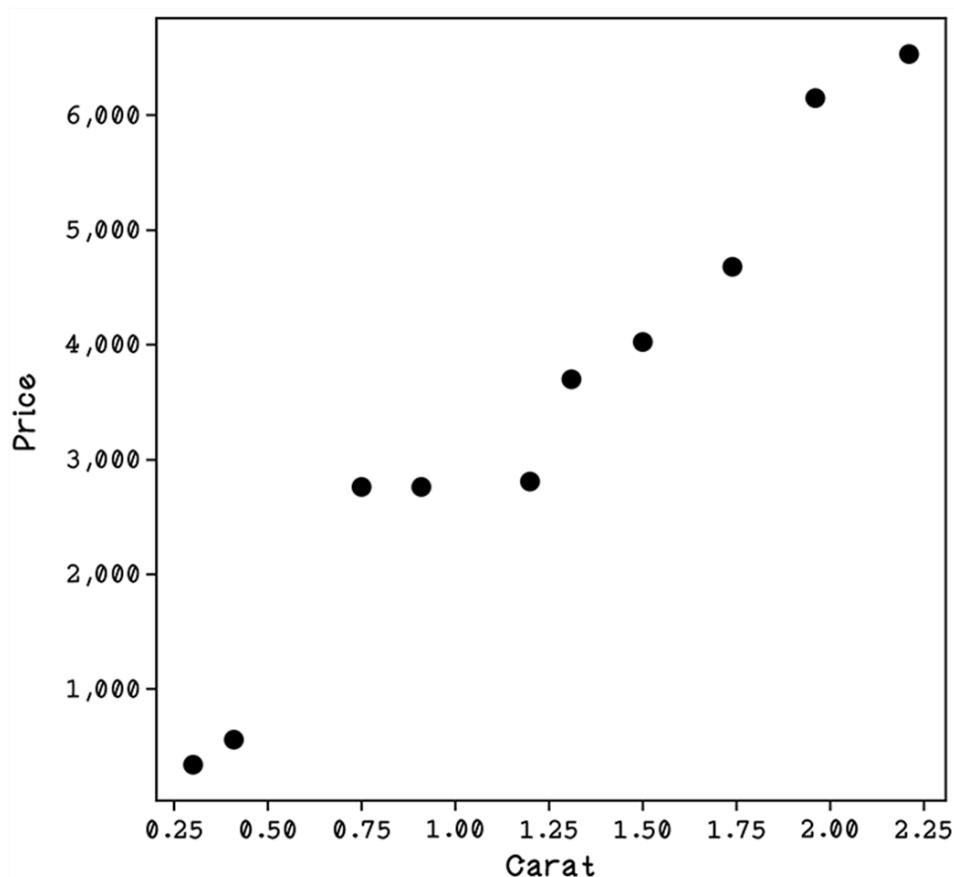
## FITTING A LINE TO THE DATA

Let's start trying to find a trend in the data and attempt to make some predictions. For exploring linear regression, the question we're asking is "Is there a correlation between the carats of a diamond and its price, and if there is, can we make accurate predictions?"

We start by isolating the Carat and Price features and plotting the data on a graph. Because we want to find the price based on Carat value, we will treat carats as  $x$  and price as  $y$ . Why did we choose this approach?

- *Carat as the independent variable ( $x$ )*—An *independent variable* is one that is changed in an experiment to determine the effect on a dependent variable. In this example, the value for carats will be adjusted to determine the price of a diamond with that value.
- *Price as the dependent variable ( $y$ )*—A *dependent variable* is one that is being tested. It is affected by the independent variable and changes based on the independent variable value changes. In our example, we are interested in the price given a specific carat value.

Figure 8.6 shows the carat and price data plotted on a graph, and table 8.7 describes the actual data.



**Figure 8.6 A scatterplot of carat and price data**

**Table 8.7 Carat and price data**

	<b>Carat (x)</b>	<b>Price (y)</b>
1	0.30	339
2	0.41	561
3	0.75	2,760
4	0.91	2,763
5	1.20	2,809
6	1.31	3,697
7	1.50	4,022
8	1.74	4,677
9	1.96	6,147
10	2.21	6,535

Notice that compared with Price, the Carat values are tiny. The price goes into the thousands, and carats are in the range of decimals. To make the calculations easier to understand for the purposes of learning in this chapter, we can scale the Carat values to be comparable to the Price values. By multiplying every Carat value by 1,000, we get numbers that are easier to compute by hand in the upcoming walkthroughs. Note that by scaling all the rows, we are not affecting the relationships in the data, because every example has the same operation applied to it. The resulting data (figure 8.7) is represented in table 8.8.

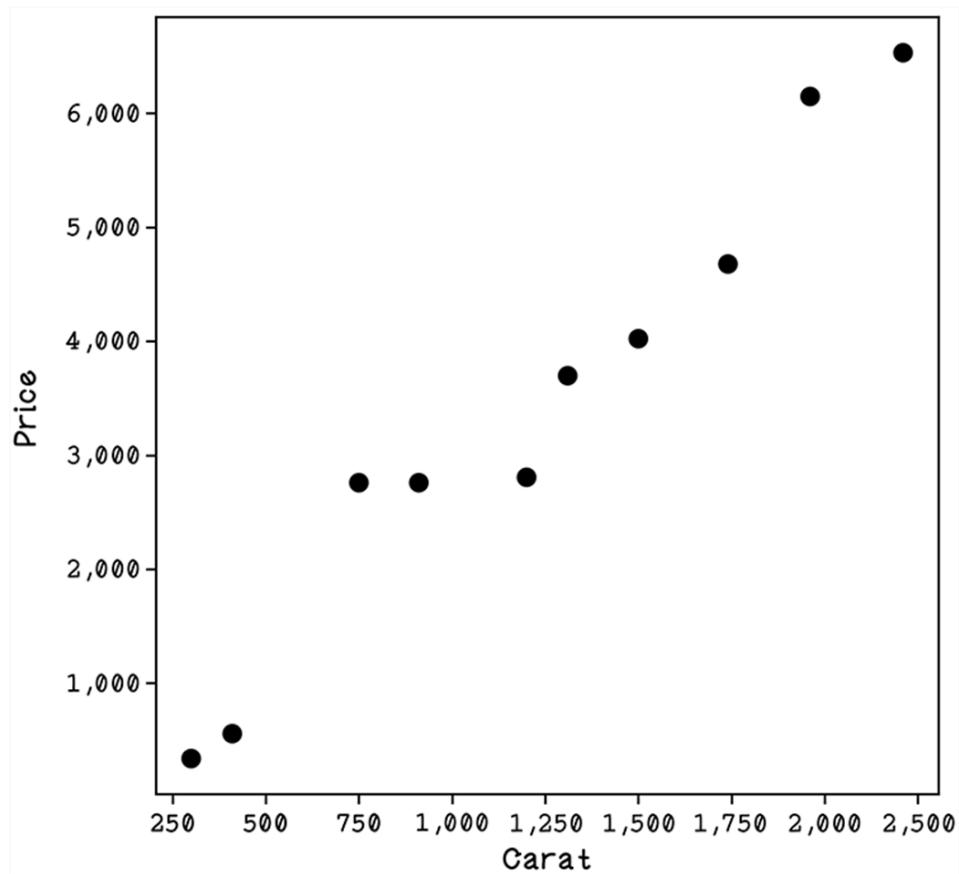


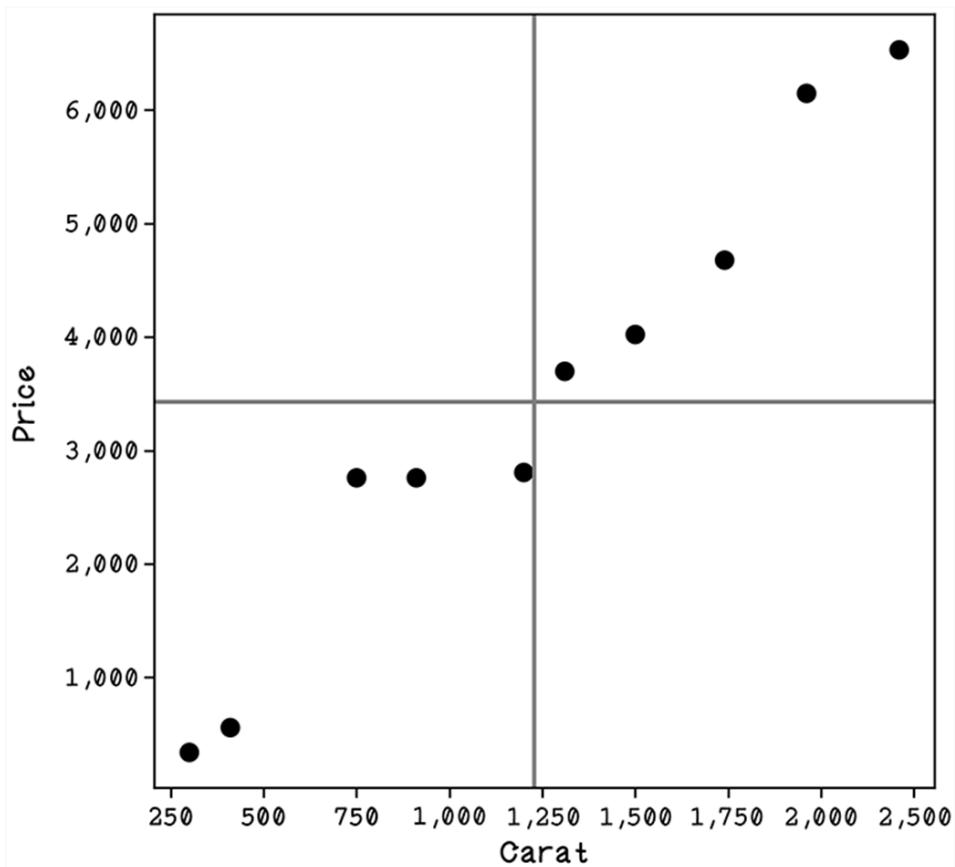
Figure 8.7 A scatterplot of carat and price data

**Table 8.8 Data with adjusted carat values**

	<b>Carat (x)</b>	<b>Price (y)</b>
1	300	339
2	410	561
3	750	2,760
4	910	2,763
5	1,200	2,809
6	1,310	3,697
7	1,500	4,022
8	1,740	4,677
9	1,960	6,147
10	2,210	6,535

## FINDING THE MEAN OF THE FEATURES

The first thing we need to do to find a regression line is find the mean for each feature. The mean is the sum of all values divided by the number of values. The mean is 1,229 for carats, represented by the vertical line on the x axis. The mean is \$3,431 for price, represented by the horizontal line on the y axis (figure 8.8).



**Figure 8.8 The means of  $x$  and  $y$  represented by vertical and horizontal lines**

The mean is important because mathematically, any regression line we find will pass through the intersection of the mean of  $x$  and the mean of  $y$ . Many lines may pass through this point. Some regression lines might be better than others at fitting the data. The *method of least squares* aims to create a line that minimizes the distances between the line and among all the points in the dataset. The method of least squares is a popular method for finding regression lines. Figure 8.9 illustrates examples of regression lines.

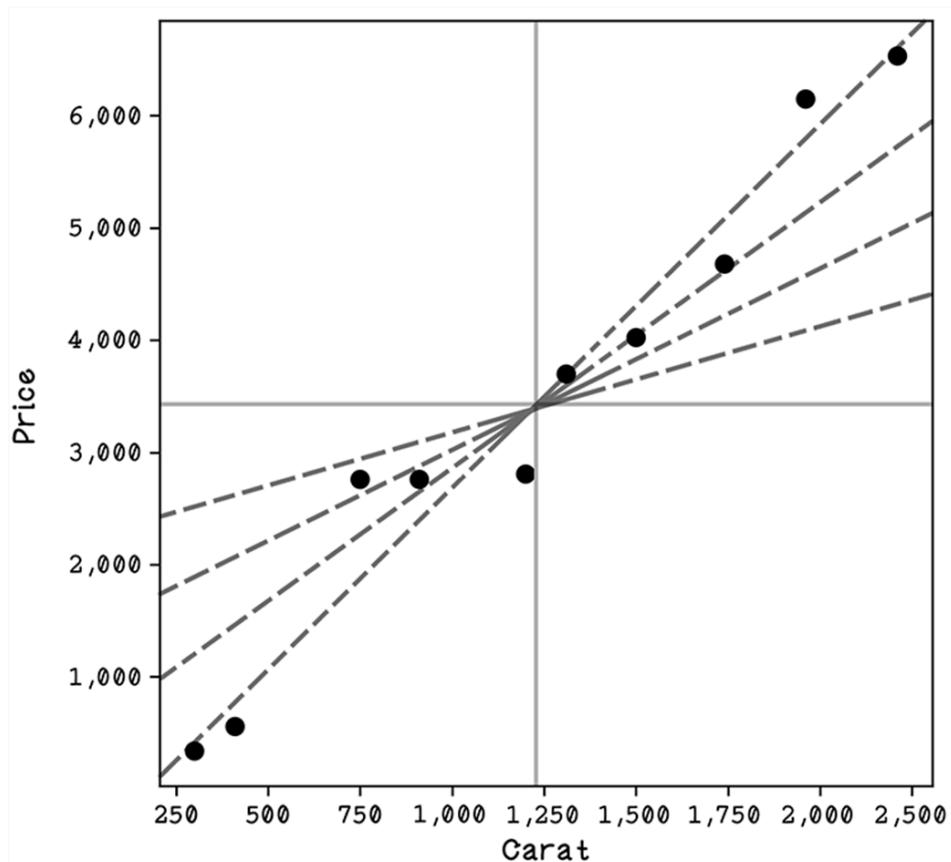
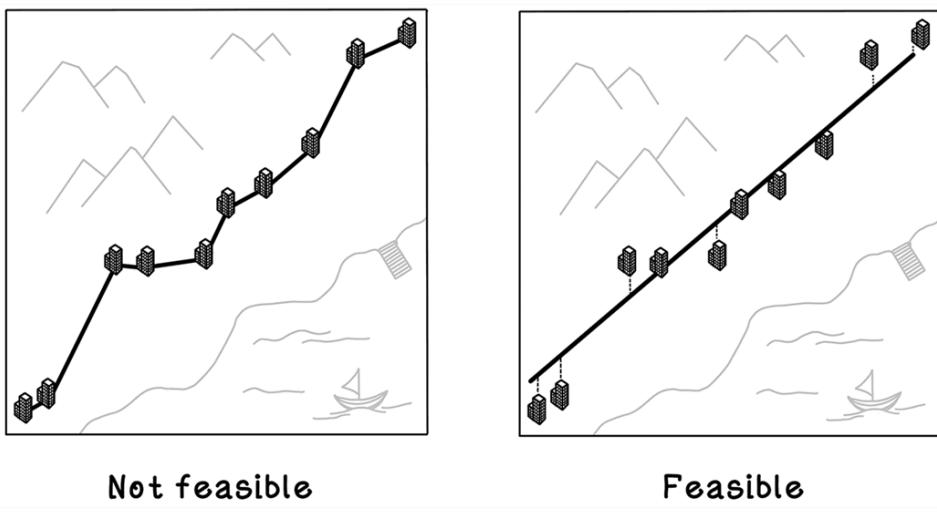


Figure 8.9 Possible regression lines

## FINDING REGRESSION LINES WITH THE LEAST-SQUARES METHOD

But what is the regression line's purpose? Suppose that we're building a subway that tries to be as close as possible to all major office buildings. It will not be feasible to have a subway line that visits every building; there will be too many stations and it will cost a lot. So, we will try to create a straight-line route that minimizes the distance to each building. Some commuters may have to walk farther than others, but the straight line is optimized for everyone's office. This goal is exactly what a regression line aims to achieve; the buildings are data points, and the line is the straight subway path (figure 8.10).

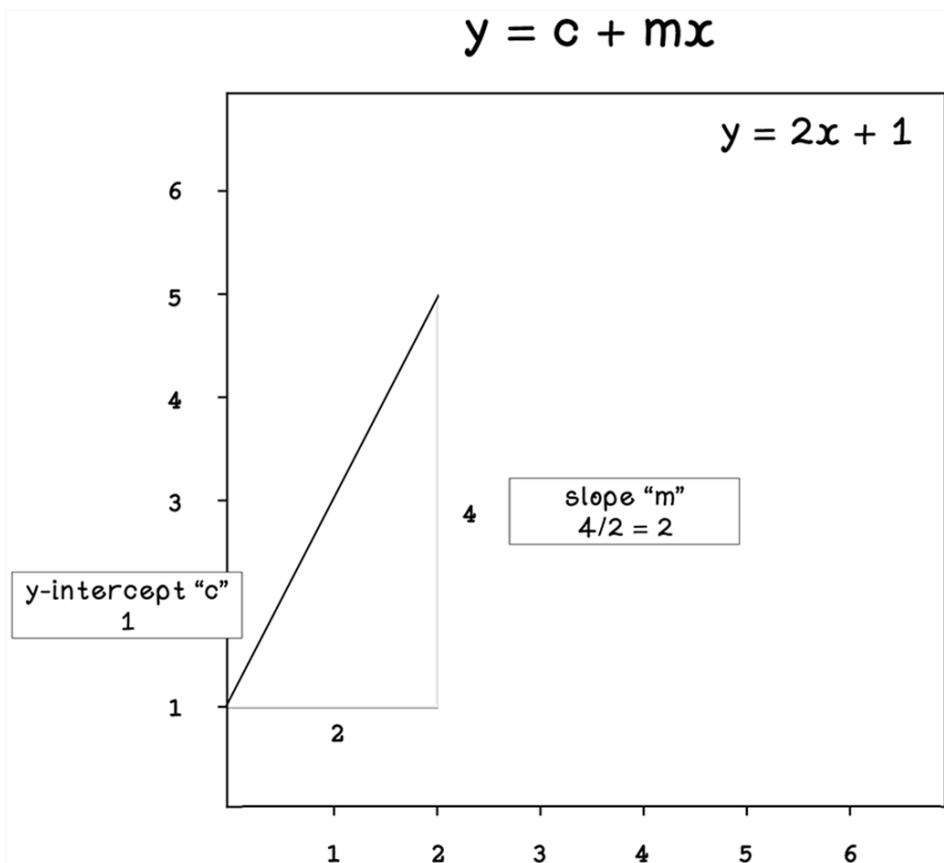


**Figure 8.10 Intuition of regression lines**

Linear regression will always find a straight line that fits the data to minimize distance among points overall. Understanding the equation for a line is important because we will be learning how to find the values for the variables that describe a line.

A straight line is represented by the equation  $y = c + mx$  (figure 8.11):

- $y$ : The dependent variable
- $x$ : The independent variable
- $m$ : The slope of the line
- $c$ : The  $y$ -value where the line intercepts the  $y$  axis



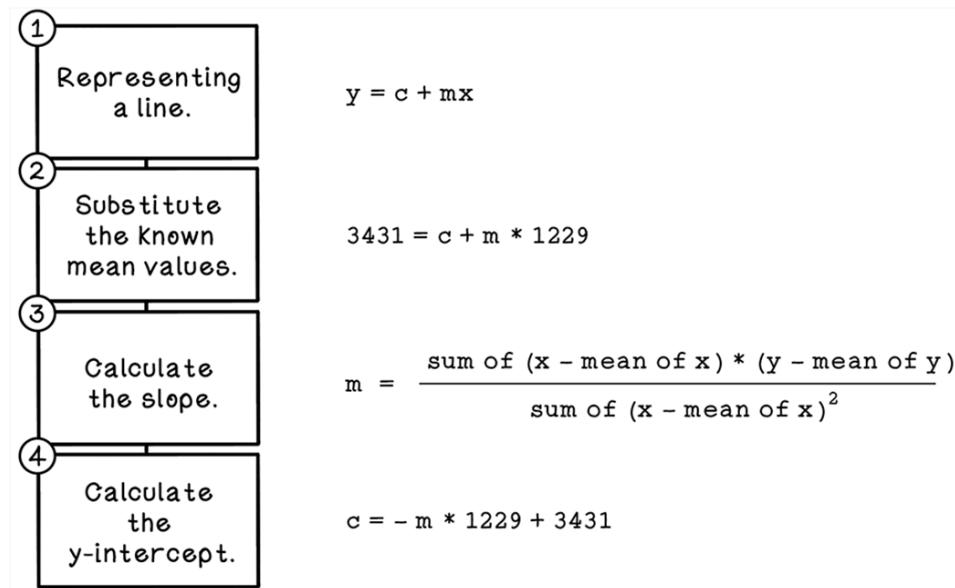
**Figure 8.11** Intuition of the equation that represents a line

The method of least squares is used to find the regression line. At a high level, the process involves the steps depicted in figure 8.12. To find the line that's closest to the data, we find the difference between the actual data values and the predicted data values. The differences for data points will vary. Some differences will be large, and some will be small. Some differences will be negative values, and some will be positive values. By squaring the differences and summing them, we take into consideration all differences for all data points. Minimizing the total difference is getting the least square difference to achieve a good regression line.

Imagine drawing a line through your data points. Now, for every point, draw a perfect square that connects the point to the line. Some squares will be tiny; others might be huge.

The algorithm's goal is to minimize the total area of all these squares combined. This is why it is called "Least Squares". It also explains why a single outlier is so dangerous: a point that is 10 units away doesn't just add 10 to the error; it creates a massive square of 100, forcing the line to move drastically to shrink it.

Don't worry if figure 8.12 looks a bit daunting; we will work through each step.



**Figure 8.12** The basic workflow for calculating a regression line

Thus far, our line has some known variables. We know that an  $x$  value is 1,229 and a  $y$  value is 3,431, as shown in step 2.

Next, we calculate the difference between every Carat value and the Carat mean, as well as the difference between every Price value and the Price mean, to find  $(x - \text{mean of } x)$  and  $(y - \text{mean of } y)$ , which is used in step 3 (table 8.9).

**Table 8.9 The diamond dataset and calculations**

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>x – mean of x</b>		<b>y – mean of y</b>	
1	300	339	300 – 1,229	-929	339 – 3,431	-3,092
2	410	561	410 – 1,229	-819	561 – 3,431	-2,870
3	750	2,760	750 – 1,229	-479	2,760 – 3,431	-671
4	910	2,763	910 – 1,229	-319	2,763 – 3,431	-668
5	1,200	2,809	1,200 – 1,229	-29	2,809 – 3,431	-622
6	1,310	3,697	1,310 – 1,229	81	3,697 – 3,431	266
7	1,500	4,022	1,500 – 1,229	271	4,022 – 3,431	591
8	1,740	4,677	1,740 – 1,229	511	4,677 – 3,431	1,246
9	1,960	6,147	1,960 – 1,229	731	6,147 – 3,431	2,716
10	2,210	6,535	2,210 – 1,229	981	6,535 – 3,431	3,104
	1,229	3,431				
	Means					

For step 3, we also need to calculate the square of the difference between every Carat and the Carat mean to find  $(x - \text{mean of } x)^2$ . We also need to sum these values to minimize, which equals 3,703,690 (table 8.10).

**Table 8.10 The diamond dataset and calculations, part 2**

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>x – mean of x</b>		<b>y – mean of y</b>		<b>(x – mean of x)<sup>2</sup></b>
1	300	339	300 – 1,229	-929	339 – 3,431	-3,092	863,041
2	410	561	410 – 1,229	-819	561 – 3,431	-2,870	670,761
3	750	2,760	750 – 1,229	-479	2,760 – 3,431	-671	229,441
4	910	2,763	910 – 1,229	-319	2,763 – 3,431	-668	101,761
5	1,200	2,809	2,100 – 1,229	-29	2,809 – 3,431	-622	841
6	1,310	3,697	1,310 – 1,229	81	3,697 – 3,431	266	6,561
7	1,500	4,022	1,500 – 1,229	271	4,022 – 3,431	591	73,441
8	1,740	4,677	1,740 – 1,229	511	4,677 – 3,431	1,246	261,121
9	1,960	6,147	1,960 – 1,229	731	6,147 – 3,431	2,716	534,361
10	2,210	6,535	2,210 – 1,229	981	6,535 – 3,431	3,104	962,361
	1,229	3,431					3,703,690
	Means						Sums

The last missing value for the equation in step 3 is the value for  $(x - \text{mean of } x) * (y - \text{mean of } y)$ . Again, the sum of the values is required. The sum equals 11,624,370 (table 8.11).

**Table 8.11 The diamond dataset and calculations, part 3**

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>x – mean of x</b>		<b>y – mean of y</b>		<b>(x – mean of x)<sup>2</sup></b>	<b>(x – mean of x) * (y – mean of y)</b>
1	300	339	300 – 1,229	-929	339 – 3,431	-3,092	863,041	2,872,468
2	410	561	410 – 1,229	-819	561 – 3,431	-2,870	670,761	2,350,530
3	750	2,760	750 – 1,229	-479	2,760 – 3,431	-671	229,441	321,409
4	910	2,763	910 – 1,229	-319	2,763 – 3,431	-668	101,761	213,092
5	1,200	2,809	2,100 – 1,229	-29	2,809 – 3,431	-622	841	18,038
6	1,310	3,697	1,310 – 1,229	81	3,697 – 3,431	266	6,561	21,546
7	1,500	4,022	1,500 – 1,229	271	4,022 – 3,431	591	73,441	160,161
8	1,740	4,677	1,740 – 1,229	511	4,677 – 3,431	1,246	261,121	636,706
9	1,960	6,147	1,960 – 1,229	731	6,147 – 3,431	2,716	534,361	1,985,396
10	2,210	6,535	2,210 – 1,229	981	6,535 – 3,431	3,104	962,361	3,045,024
	1,229	3,431					3,703,690	11,624,370
	Means						Sums	

Now we can plug in the calculated values to the least-squares equation to calculate  $m$ :

$$\begin{aligned} m &= 11624370 / 3703690 \\ m &= 3.139 \end{aligned}$$

Now that we have a value for  $m$ , we can calculate  $c$  by substituting the mean values for  $x$  and  $y$ . Remember that all regression lines will pass this point, so it is a known point within the regression line:

$y = c + mx$

```
3431 = c + 3.139x
3431 = c + 391.5594
3431 - 391.5594 = c
c = 3,039.4406
```

Complete regression line:

$y = 3039.4406 + 0.3186x$

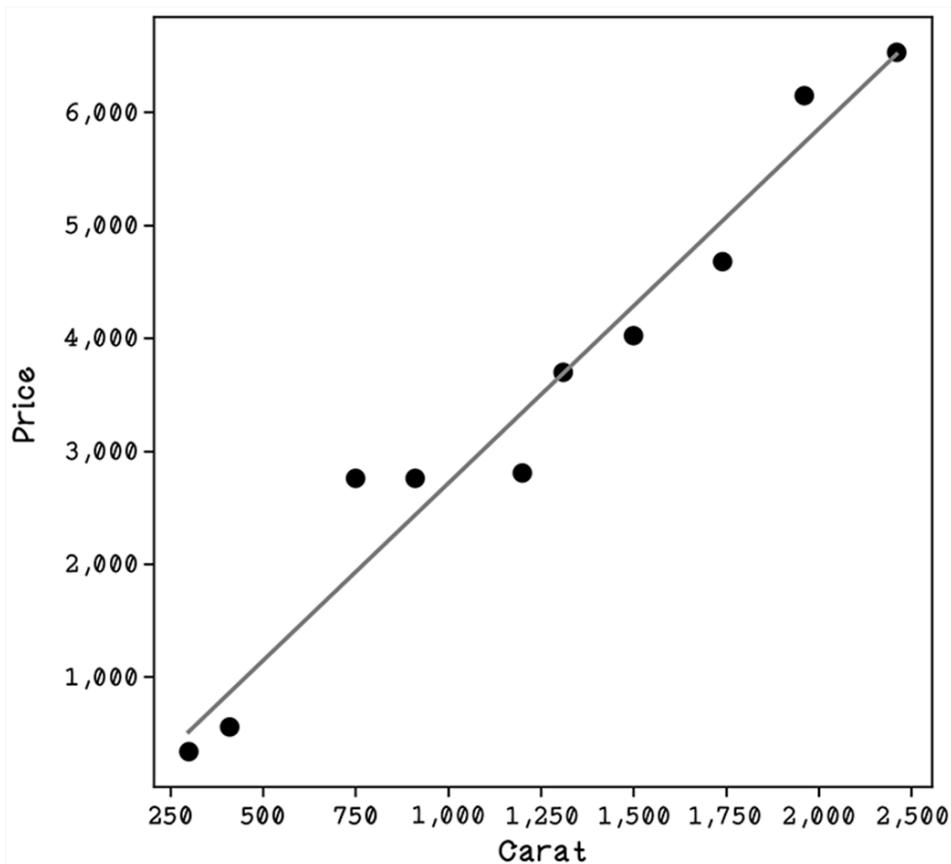
Finally, we can plot the line by generating some values for carats between the minimum value and maximum value, plugging them into the equation that represents the regression line, and then plotting it (figure 8.13):

```
x (Carat) minimum = 300
x (Carat) maximum = 2210
```

```
Sample between the minimum and maximum at intervals of 500:
x = [300, 2210]
```

```
Plug the values for x into the regression line:
y = [-426 + 3.139(300) = 515.7,
      -426 + 3.139(2210) = 6511.19]
```

```
Complete x and y samples:
x = [300, 2210]
y = [3981, 9975]
```



**Figure 8.13** A regression line plotted with the data points

We've trained a linear regression line based on our dataset that accurately fits the data, so we've actually done some machine learning by hand!

### EXERCISE: CALCULATE A REGRESSION LINE USING THE LEAST-SQUARES METHOD

Following the steps described and using the following dataset, calculate the regression line with the least-squares method.

	<b>Carat (x)</b>	<b>Price (y)</b>
1	320	350
2	460	560
3	800	2,760
4	910	2,800
5	1,350	2,900
6	1,390	3,600
7	1,650	4,000
8	1,700	4,650
9	1,950	6,100
10	2,000	6,500

### SOLUTION: CALCULATE A REGRESSION LINE USING THE LEAST-SQUARES METHOD

The means for each dimension need to be calculated. The means are 1,253 for  $x$  and 3,422 for  $y$ . The next step is calculating the difference between each value and its mean. Next, the square of the difference between  $x$  and the mean of  $x$  is calculated and summed, which results in 3,251,610. Finally, the difference between  $x$  and the mean of  $x$  is multiplied by the difference between  $y$  and the mean of  $y$  and summed, resulting in 10,566,940.

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>x – mean of x</b>	<b>y – mean of y</b>	<b>(x – mean of x)<sup>2</sup></b>	<b>(x – mean of x) * (y – mean of y)</b>
1	320	350	-933	-3,072	870,489	2,866,176
2	460	560	-793	-2,862	628,849	2,269,566
3	800	2,760	-453	-662	205,209	299,886
4	910	2,800	-343	-622	117,649	213,346
5	1,350	2,900	97	-522	9,409	-50,634
6	1,390	3,600	137	178	18,769	24,386
7	1,650	4,000	397	578	157,609	229,466
8	1,700	4,650	447	1,228	199,809	548,916
9	1,950	6,100	697	2,678	485,809	1,866,566
10	2,000	6,500	747	3,078	558,009	2,299,266
	1,253	3,422			3,251,610	10,566,940

The values can be used to calculate the slope,  $m$ :

$$m = 10566940 / 3251610$$

$$m = 3.25$$

Remember the equation for a line:

$$y = c + mx$$

Substitute the mean values for  $x$  and  $y$  and the newly calculated  $m$ :

$$3422 = c + 3.35 * 1253$$

$$c = -775.55$$

Substitute the minimum and maximum values for  $x$  to calculate points to plot a line:

```

Point 1, we use the minimum value for Carat: x = 320
y = 775.55 + 3.25 * 320
y = 1 815.55
Point 2, we use the maximum value for Carat: x = 2000
y = 775.55 + 3.25 * 2000
y = 7 275.55

```

Now that we have an intuition about how to use linear regression and how regression lines are calculated, let's take a look at the code.

## PYTHON CODE SAMPLE

The code is similar to the steps that we walked through. The only interesting aspects are the two `for` loops used to calculate summed values by iterating over every element in the dataset:

```

def fit_regression_line(carats, prices):
    n = len(carats)

    mean_X = sum(carats) / n
    mean_Y = sum(prices) / n

    sum_X_squared = 0
    for x_i in carats:
        ans = (x_i - mean_X) ** 2
        sum_X_squared += ans

    sum_multiple = 0
    for i in range(n):
        ans = (carats[i] - mean_X) * (prices[i] - mean_Y)
        sum_multiple += ans

    b1 = sum_multiple / sum_X_squared

    b0 = mean_Y - (b1 * mean_X)

    min_x = min(carats)
    max_x = max(carats)

    y1 = b0 + b1 * min_x
    y2 = b0 + b1 * max_x

    return b0, b1, (min_x, y1), (max_x, y2)

```

### 8.3.4 Testing the model: Determine the accuracy of the model

Now that we have determined a regression line, we can use it to make price predictions for other Carat values. We can measure the performance of the regression line with new examples in which we know the actual price and determine how accurate the linear regression model is.

We can't test the model with the same data that we used to train it. This approach would result in high accuracy and be meaningless. The trained model must be tested with real data that the model hasn't seen before.

## SEPARATING TRAINING AND TESTING DATA

Training and testing data are usually split 80/20, with 80% of the available data used as training data and 20% used to test the model. Percentages are used because the number of examples needed to train a model accurately is difficult to know; different contexts and questions being asked may need more or less data.

Figure 8.14 and table 8.12 represent a set of testing data for the diamond example. Remember that we scaled the Carat values to be similar-size numbers to the Price values (all Carat values have been multiplied by 1,000) to make them easier to read and work with. The dots represent the testing data points, and the line represents the trained regression line.

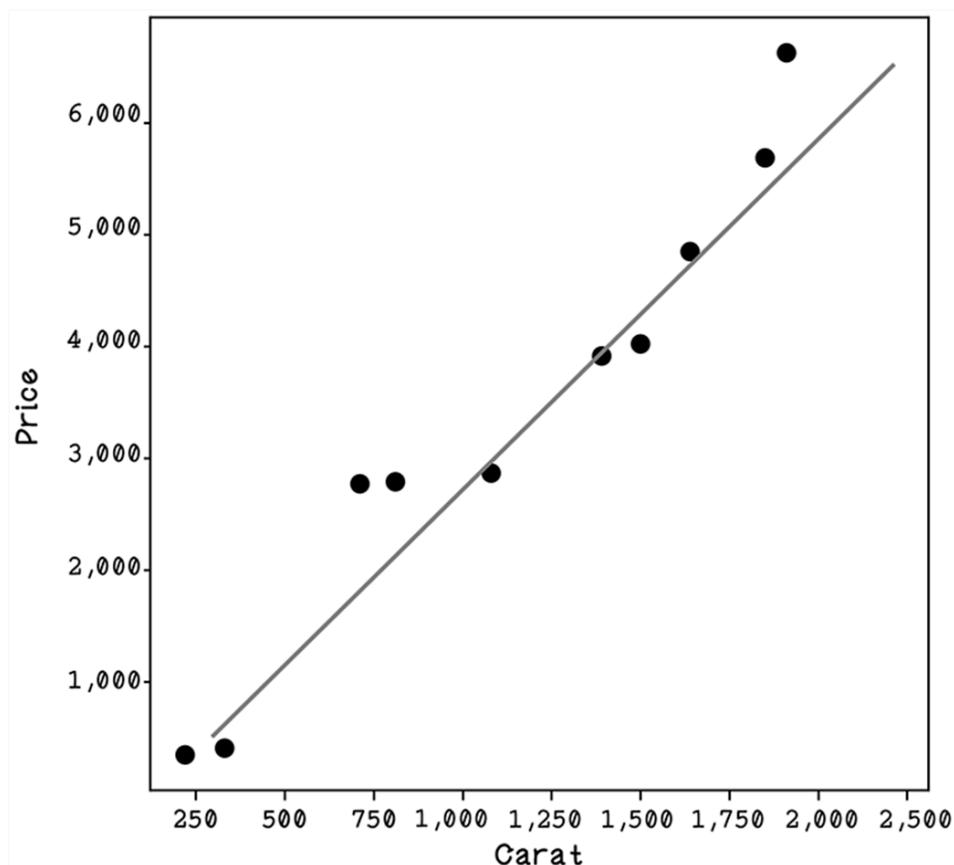


Figure 8.14 A regression line plotted with the data points

**Table 8.12 The carat and price data**

	<b>Carat (x)</b>	<b>Price (y)</b>
1	220	342
2	330	403
3	710	2,772
4	810	2,789
5	1,080	2,869
6	1,390	3,914
7	1,500	4,022
8	1,640	4,849
9	1,850	5,688
10	1,910	6,632

Testing a model involves making predictions with unseen training data and then comparing the accuracy of the model's prediction with the actual values. In the diamond example, we have the actual Price values, so we will determine what the model predicts and compare the difference.

## MEASURING THE PERFORMANCE OF THE LINE

In linear regression, a common method of measuring the accuracy of the model is calculating R<sup>2</sup> (R squared). R<sup>2</sup> is used to determine the variance between the actual value and a predicted value. The following equation is used to calculate the R<sup>2</sup> score:

$$R^2 = \frac{\text{sum of } (\text{predicted } y - \text{mean of actual } y)^2}{\text{sum of } (actual \ y - \text{mean of actual } y)^2}$$

The first things we need to do, similar to the training step, are calculate the mean of the actual Price values, calculate the distances between the actual Price values and the mean of the prices, and then calculate the square of those values. We are using the values plotted as dots in figure 8.14 (table 8.13).

**Table 8.13 The diamond dataset and calculations**

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>y – mean of y</b>	<b>(y – mean of y)<sup>2</sup></b>
1	220	342	-3,086	9,523,396
2	330	403	-3,025	9,150,625
3	710	2,772	-656	430,336
4	810	2,789	-639	408,321
5	1,080	2,869	-559	312,481
6	1,390	3,914	486	236,196
7	1,500	4,022	594	352,836
8	1,640	4,849	1,421	2,019,241
9	1,850	5,688	2,260	5,107,600
10	1,910	6,632	3,204	10,265,616
		3,428		37,806,648
		Mean		Sum

The next step is calculating the predicted Price value for every Carat value, squaring the values, and calculating the sum of all those values (table 8.14).

**Table 8.14 The diamond dataset and calculations, part 2**

	<b>Carat (x)</b>	<b>Price (y)</b>	<b>y – mean of y</b>	<b>(y – mean of y)<sup>2</sup></b>		<b>Predicted y</b>	<b>Predicted y – mean of y</b>	<b>(Predicted y – mean of y)<sup>2</sup></b>
1	220	342	-3,086	9,523,396	264	-3,164	10,009,876	
2	330	403	-3,025	9,150,625	609	-2,819	7,944,471	
3	710	2,772	-656	430,336	1,802	-1,626	2,643,645	
4	810	2,789	-639	408,321	2,116	-1,312	1,721,527	
5	1,080	2,869	-559	312,481	2,963	-465	215,900	
6	1,390	3,914	486	236,196	3,936	508	258,382	
7	1,500	4,022	594	352,836	4,282	854	728,562	
8	1,640	4,849	1,421	2,019,241	4,721	1,293	1,671,748	
9	1,850	5,688	2,260	5,107,600	5,380	1,952	3,810,559	
10	1,910	6,632	3,204	10,265,616	5,568	2,140	4,581,230	
		3,428		37,806,648			33,585,901	
		Mean		Sum			Sum	

Using the sum of the square of the difference between the predicted price and mean, and the sum of the square of the difference between the actual price and mean, we can calculate the R<sup>2</sup> score:

$$R^2 = \frac{\text{sum of } (\text{predicted } y - \text{mean of actual } y)^2}{\text{sum of } (\text{actual } y - \text{mean of actual } y)^2}$$

$$R^2 = 33585901 / 37806648$$

$$R^2 = 0.88$$

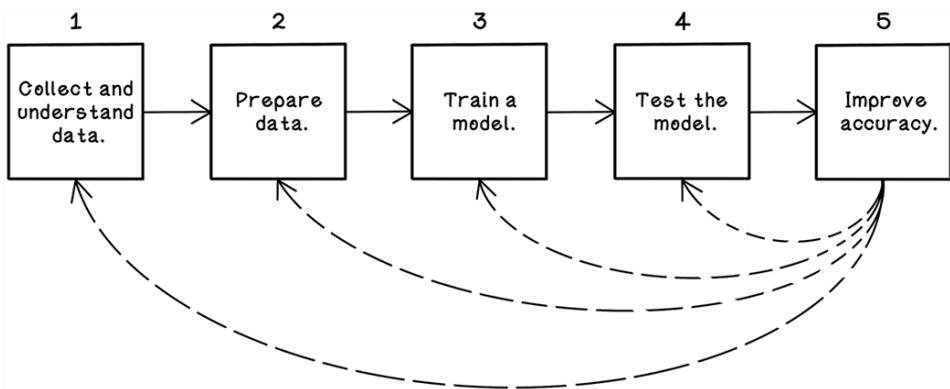
The result—an  $R^2$  score of 0.88—means that our model explains 88% of the variance in the target variable (diamond price). In plain English: 88% of the difference in diamond prices can be explained by the features we included (like weight and cut), while the remaining 12% is due to other factors or random noise. This indicates a strong fit to the data.

This result is a fairly good one, showing that the linear regression model is fairly accurate. For the diamond example, this result is satisfactory. Determining whether the accuracy is satisfactory for the problem we’re trying to solve depends on the domain of the problem. We will be exploring the performance of machine learning models in the next section.

Additional information: For a gentle introduction to fitting lines to data, reference <http://mng.bz/Ed5q> —a chapter from *Math for Programmers* by Manning Publications. Linear regression can be applied to more dimensions. We can determine the relationship among Carat values, Prices, and Cut of diamonds, for example, through a process called *multiple regression*. This process adds some complexity to the calculations, but the fundamental principles remain the same.

### 8.3.5 Improving accuracy

After training a model on data and measuring how well it performs on new testing data, we have an idea of how well the model performs. Often, models don’t perform as well as desired, and additional work needs to be done to improve the model, if possible. This improvement involves iterating on the various steps in the machine learning life cycle (figure 8.15).



**Figure 8.15 A refresher on the machine learning life cycle**

The results may require us to pay attention to one or more of the following areas. Machine learning is experimental work in which different tactics at different stages are tested before settling on the best-performing approach. In the diamond example, if the model that used Carat values to predict Price performed poorly, we might use the dimensions of the diamond that indicate size, coupled with the Carat value, to try to predict the price more accurately. Here are some ways to improve the accuracy of the model:

- *Collect more data.* One solution may be to collect more data related to the dataset that is being explored, perhaps augmenting the data with relevant external data or including data that was previously not considered.
- *Prepare the data differently.* The data used for training may need to be prepared in a different way. Referring to the techniques used to fix data earlier in this chapter, there may be errors in the selected approach. We may need to use different techniques to find values for missing data, replace ambiguous data, and encode categorical data.
- *Choose different features in the data.* Other features in the dataset may be better suited to predicting the dependent variable. The X dimension value might be a good choice to predict the Table value, for example, because it has a physical relationship with it, as shown in the diamond terminology figure (figure 8.5), whereas predicting Clarity with the X dimension is meaningless.
- *Use a different algorithm to train the model.* Sometimes, the selected algorithm is not suited to the problem being solved or the nature of the data. We can use a different algorithm to accomplish different goals, as discussed in the next section.
- *Dealing with false-positive tests.* When checking for generalization, evaluation metrics can be deceiving. A model might achieve a high accuracy score on training data, but when presented with new, unseen data, it might perform poorly. This discrepancy is usually due to *Overfitting*.

*Overfitting* is when the model is too closely aligned with the training data and is not flexible for dealing with new data with more variance.

Think of it like studying for an exam: Overfitting is memorizing the specific answers to the practice test. You get 100% on the practice, but fail the real exam because the questions are worded slightly differently. Underfitting is not studying at all. You fail both tests. A *good fit* is understanding the concepts. You might miss a few distinct details, but you can adapt to solve new problems you haven't seen before.

If linear regression didn't provide useful results, or if we have a different question to ask, we can try a range of other algorithms. The next two sections will explore algorithms to use when the question is different in its nature.

## 8.4 Classification with decision trees

Simply put, classification problems involve assigning a label to an example based on its attributes. These problems are different from regression, in which a value is estimated. Let's dive into classification problems and see how to solve them.

### 8.4.1 Classification problems: Either this or that

We have learned that regression involves predicting a value based on one or more other variables, such as predicting the Price of a diamond given its Carat value. Classification is similar in that it aims to predict a value but it predicts *discrete classes* instead of *continuous values*. Discrete values are categorical features of a dataset such as Cut, Color, or Clarity in the diamond dataset, as opposed to continuous values such as Price or Depth.

Here's another example. Suppose that we have several vehicles that are cars and trucks. We will measure the weight of each vehicle and the number of wheels of each vehicle. We also forget for now that cars and trucks look very different. Almost all cars have four wheels, and many large trucks have more than four wheels. Trucks are usually heavier than cars, but a large sport-utility vehicle may be as heavy as a small truck. We could find relationships between the weight and number of wheels of vehicles to predict whether a vehicle is a car or a truck (figure 8.16).

Truck	Truck	Car	Car	Car
				
4 wheels 8 tons	6 wheels 10 tons	4 wheels 4 tons	4 wheels 2 tons	4 wheels 1 ton

Figure 8.16 Example vehicles for potential classification based on the number of wheels and weight

### EXERCISE: REGRESSION VS. CLASSIFICATION

Consider the following scenarios, and determine whether each one is a regression or classification problem:

- Based on data about rats, we have a life-expectancy feature and an obesity feature. We're trying to find a correlation between the two features.
- Based on data about animals, we have the weight of each animal and whether or not it has wings. We're trying to determine which animals are birds.
- Based on data about computing devices, we have the screen size, weight, and operating system of several devices. We want to determine which devices are tablets, laptops, or phones.

4. Based on data about weather, we have the amount of rainfall and a humidity value. We want to determine the humidity in different rainfall seasons.

## SOLUTION: REGRESSION VS. CLASSIFICATION

1. *Regression*—The relationship between two variables is being explored. Life expectancy is the dependent variable, and obesity is the independent variable.
2. *Classification*—We are classifying an example as a bird or not a bird, using the weight and the wing characteristic of the examples.
3. *Classification*—An example is being classified as a tablet, laptop, or phone by using its other characteristics.
4. *Regression*—The relationship between rainfall and humidity is being explored. Humidity is the dependent variable, and rainfall is the independent variable.

### 8.4.2 The basics of decision trees

Different algorithms are used for regression and classification problems. Some popular algorithms include support vector machines, decision trees, and random forests. In this section, we will be looking at a decision-tree algorithm to learn classification.

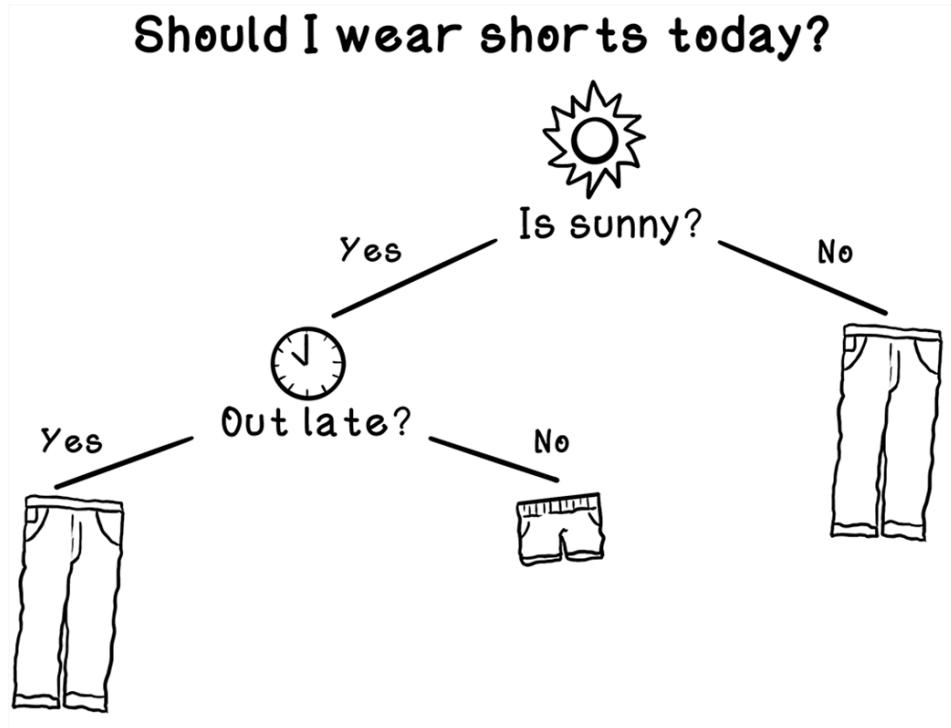
*Decision trees* are structures that describe a series of decisions that are made to find a solution to a problem (figure 8.17).

Think of a Decision Tree as playing the guessing game “21 Questions” where someone thinks of something, and the player asks questions to narrow their thinking and guess correctly.

- *Bad question*: “Is it an object?” (Splits the data poorly; almost everything is an object).
- *Good question*: “Is it alive?” (Splits the data perfectly into *living* vs. *non-living* things).

The algorithm’s goal is to win the game in as few questions as possible by picking the specific question that eliminates the most wrong answers at once.

If we’re deciding whether to wear shorts for the day, we might make a series of decisions to inform the outcome. Will it be cold during the day? If not, will we be out late in the evening, when it does get cold? We might decide to wear shorts on a warm day, but not if we will be out when it gets cold.



**Figure 8.17 Example of a basic decision tree**

For the diamond example, we will try to predict the cut of a diamond based on the Carat and Price values by using a decision tree. To simplify this example, assume that we're a diamond dealer who doesn't care about each specific cut. We will group the different cuts into two broader categories. Fair and Good cuts will be grouped into a category called Okay, and Very Good, Premium, and Ideal cuts will be grouped into a category called Perfect.

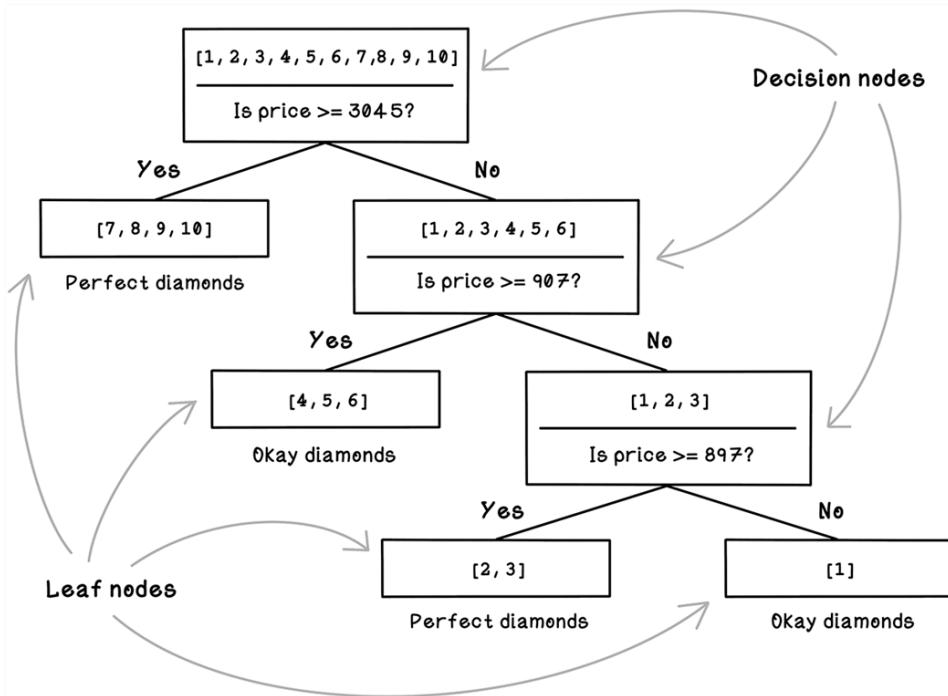
<b>1</b>	Fair	<b>1</b>	Okay
<b>2</b>	Good		
<b>3</b>	Very Good	<b>2</b>	Perfect
<b>4</b>	Premium		
<b>5</b>	Ideal		

Our sample dataset now looks like table 8.15.

**Table 8.15 The dataset used for the classification example**

	<b>Carat</b>	<b>Price</b>	<b>Cut</b>
1	0.21	327	Okay
2	0.39	897	Perfect
3	0.50	1,122	Perfect
4	0.76	907	Okay
5	0.87	2,757	Okay
6	0.98	2,865	Okay
7	1.13	3,045	Perfect
8	1.34	3,914	Perfect
9	1.67	4,849	Perfect
10	1.81	5,688	Perfect

By looking at the values in this small example and intuitively looking for patterns, we might notice something. The price seems to spike significantly after 0.98 carats, and the increased price seems to correlate with the diamonds that are Perfect, whereas diamonds with smaller Carat values tend to be Average. But example 3, which is Perfect, has a small Carat value. Figure 8.18 shows what would happen if we were to create questions to filter the data and categorize it by hand. Notice that decision nodes contain our questions, and leaf nodes contain examples that have been categorized.



**Figure 8.18 Example of a decision tree designed through human intuition**

With the small dataset, we could easily categorize the diamonds by hand. In real-world datasets, however, there are thousands of examples to work through, with possibly thousands of features, making it close to impossible for a person to create a decision tree by hand. This is where decision tree algorithms come in. Decision trees can create the questions that filter the examples. A decision tree finds complex patterns that we might miss and processes data at a scale humans cannot. However, raw power doesn't always mean better results; decision trees are so eager to find patterns that they can easily "overfit" (memorize) the data unless we apply specific constraints to keep them generalized.

### 8.4.3 Training decision trees

To create a tree that is intelligent in making the right decisions to classify diamonds, we need a training algorithm to learn from the data. There is a family of algorithms for decision tree learning, and we will use a specific one named CART (Classification and Regression Tree). The foundation of CART and the other tree learning algorithms is this: decide what questions to ask and when to ask those questions to best filter the examples into their respective categories. In the diamond example, the algorithm must learn the best questions to ask about the Carat and Price values, and when to ask them, to best segment Average and Perfect diamonds.

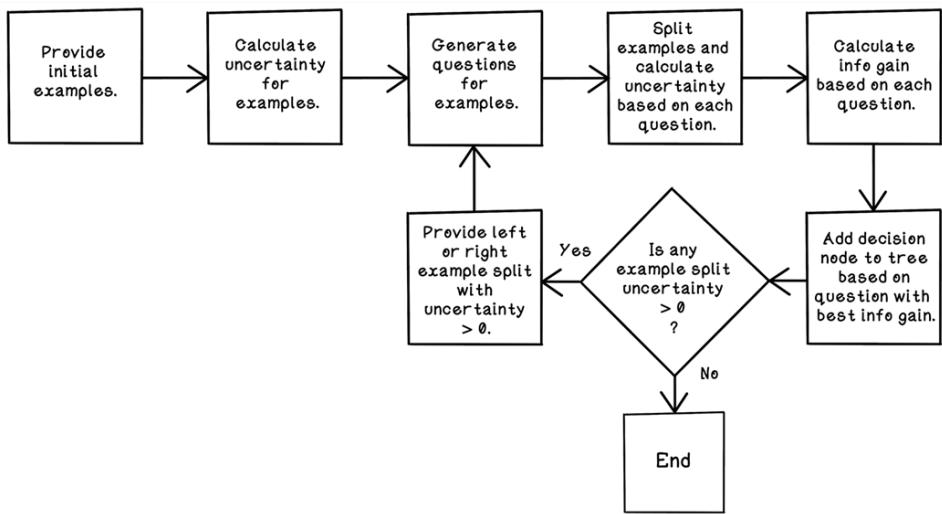
## DATA STRUCTURES FOR DECISION TREES

To help us understand how the decisions of the tree will be structured, we can review the following data structures, which organize logic and data in a way that's suitable for the decision tree learning algorithm:

- *Map of classes/label groupings*—A *map* is a key-value pair of elements that cannot have two keys that are the same. This structure is useful for storing the number of examples that match a specific label and will be useful to store the values required for calculating entropy, also known as *uncertainty*. We'll learn about entropy soon.
- *Tree of nodes*—As depicted in the previous tree figure (figure 8.18), several nodes are linked to compose a tree. This example may be familiar from some of the earlier chapters. The nodes in the tree are important for filtering/partitioning the examples into categories:
  - *Decision node*—A node in which the dataset is being split or filtered.  
Question: What question is being asked? (See the Question point coming up).  
True examples: The examples that satisfy the question.  
False examples: The examples that don't satisfy the question.
  - *Examples node/leaf node*—A node containing a list of examples only. All examples in this list would have been categorized correctly.
- *Question*—A question can be represented differently depending on how flexible it can be. We could ask, "Is the Carat value  $> 0.5$  and  $< 1.13$ ?" To keep this example simple to understand, the question is a variable feature, a variable value, and the  $\geq$  operator: "Is Carat  $\geq 0.5$ ?" or "Is Price  $\geq 3,045$ ?"
  - *Feature*—The feature that is being interrogated
  - *Value*—The constant value that the comparing value must be greater than or equal to

## DECISION-TREE LEARNING LIFE CYCLE

This section discusses how a decision-tree algorithm filters data with decisions to classify a dataset correctly. Figure 8.19 shows the steps involved in training a decision tree. The flow described in figure 8.19 is covered throughout the rest of this section.

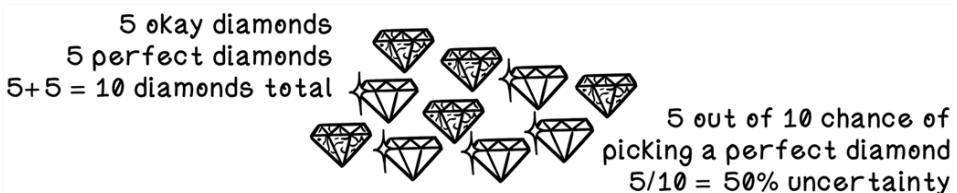


**Figure 8.19 A basic flow for building a decision tree**

In building a decision tree, we test all possible questions to determine which one is the best question to ask at a specific point in the decision tree. To test a question, we use the concept of *entropy*—the measurement of uncertainty or “impurity” in a dataset.

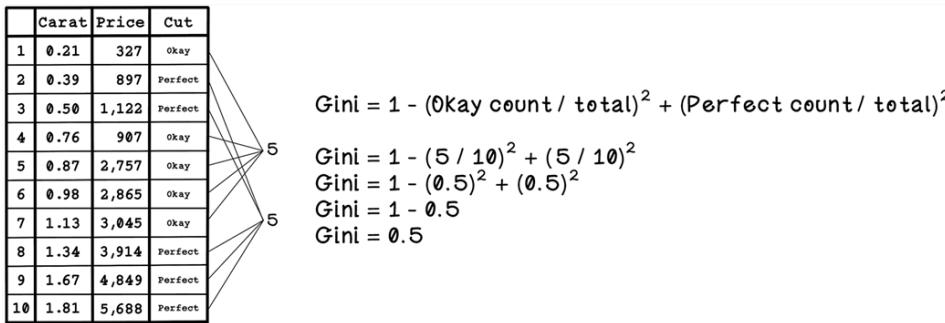
Imagine we have 5 Perfect diamonds and 5 Okay diamonds in a bag. If you reached in, you'd be totally unsure of what you'd get. This 50/50 split represents High Entropy (maximum disorder).

Now imagine a pile of 10 Perfect diamonds and 0 Okay diamonds. There is zero surprise; you know exactly what you are getting. This is Low Entropy (zero disorder). The goal of a decision tree is to ask questions that split the data to reduce entropy, moving us from a messy, mixed state to a clean, pure state.



**Figure 8.20 Example of uncertainty**

Given an initial dataset of diamonds with the Carat, Price, and Cut features, we can determine the uncertainty of the dataset by using the Gini index. A Gini index of 0 means that the dataset has no uncertainty and is pure; it might have 10 Perfect diamonds, for example. Figure 8.21 describes how the Gini index is calculated.



**Figure 8.21** The Gini index calculation

The Gini index is 0.5, so there's a 50% chance of choosing an incorrectly labeled example if one is randomly selected, as shown in figure 8.20 earlier.

The next step is creating a decision node to split the data. The decision node includes a question that can be used to split the data in a sensible way and decrease the uncertainty. Remember that 0 means no uncertainty. We aim to partition the dataset into subsets with zero uncertainty.

Many questions are generated based on every feature of each example to split the data and determine the best split outcome. Because we have 2 features and 10 examples, the total number of questions generated would be 20. Figure 8.22 depicts some of the questions asked—simple questions about whether the value of a feature is greater than or equal to a specific value.

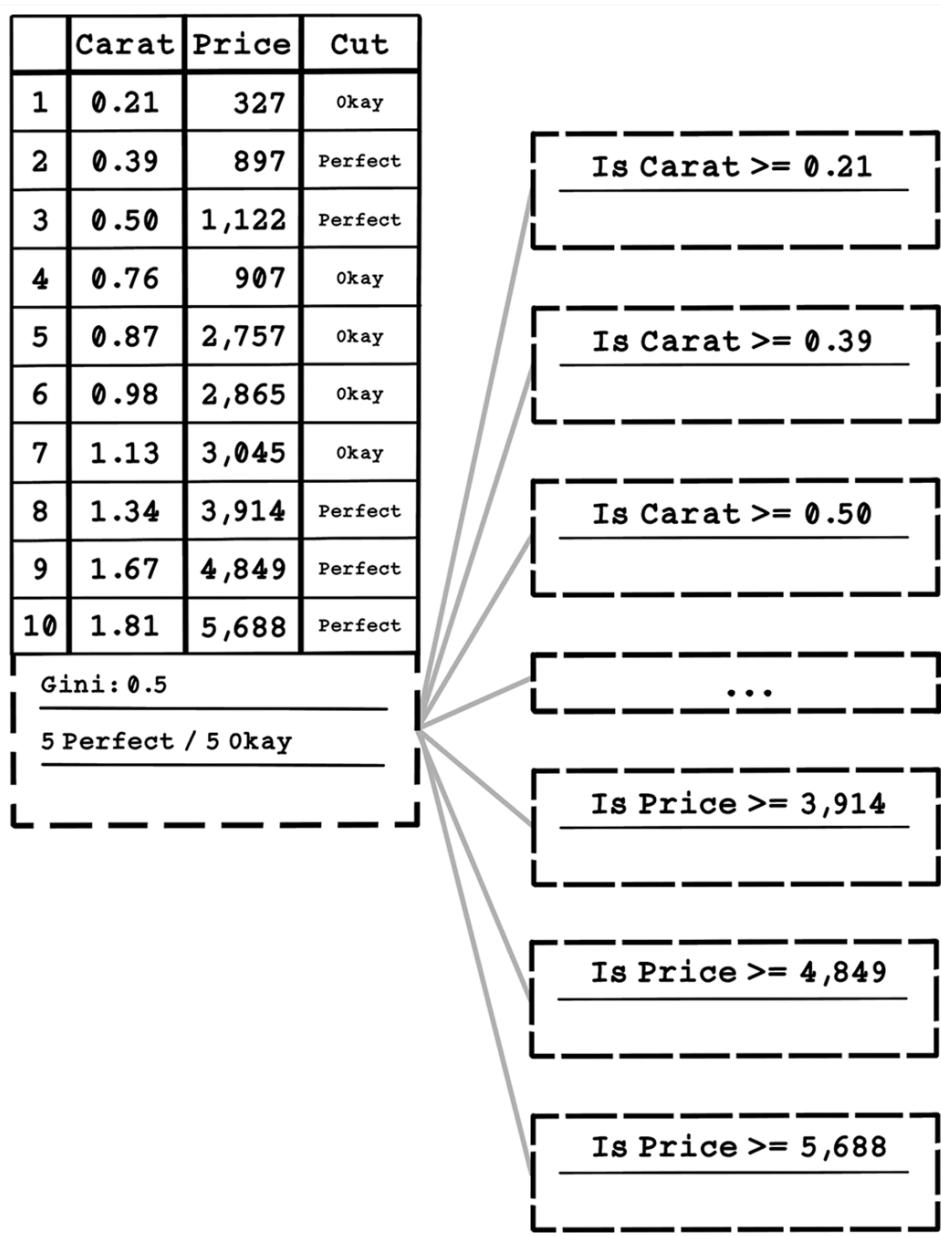


Figure 8.22 An example of questions asked to split the data with a decision node

Uncertainty in a dataset is determined by the *Gini index*, and questions aim to reduce uncertainty. *Entropy* is another concept that measures disorder using the Gini index for a specific split of data based on a question asked. We must have a way to determine how well a question reduced uncertainty, and we accomplish this task by measuring information gain. *Information gain* describes the amount of information gained by asking a specific question. If a lot of information is gained, the uncertainty is smaller.

Information gain is calculated by the subtracting entropy before the question is asked by the entropy after the question is asked, following these steps:

1. Split the dataset by asking a question.
2. Measure the Gini index for the left split.
3. Measure the entropy for the left split compared with the dataset before the split.
4. Measure the Gini index for the right split.
5. Measure the entropy for the right split compared with the dataset before the split.
6. Calculate the total entropy after by adding the left entropy and right entropy.
7. Calculate the information gain by subtracting the total entropy after from the total entropy before.

Figure 8.23 illustrates the data split and information gain for the question “Is Price  $\geq$  3914?”

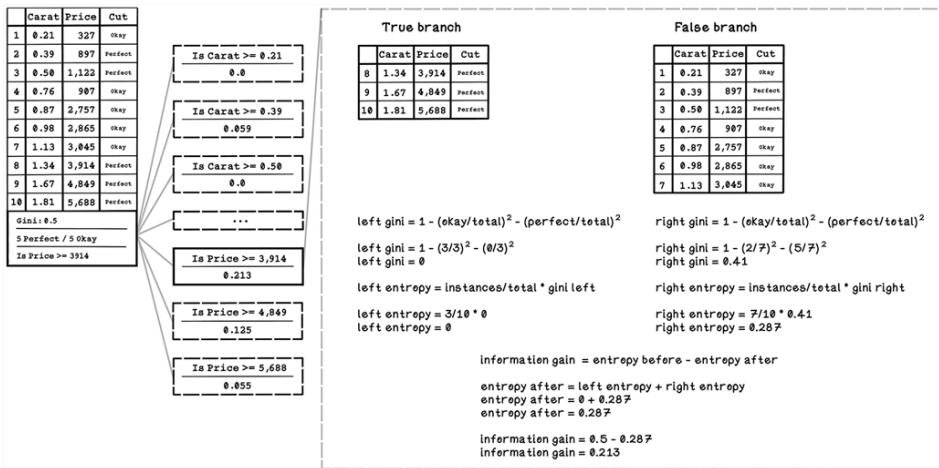


Figure 8.23 Illustration of data split and information gain based on a question

In the example in figure 8.23, the information gain for all questions is calculated, and the question with the highest information gain is selected as the best question to ask at that point in the tree. Then the original dataset is split based on the decision node with the question "Is Price  $\geq 3,914$ ?" A decision node containing this question is added to the decision tree, and the left and right splits stem from that node.

In figure 8.24, after the dataset is split, the left side contains a pure dataset of Perfect diamonds only, and the right side contains a dataset with mixed diamond classifications, including two Perfect diamonds and five Okay diamonds. Another question must be asked on the right side of the dataset to split the dataset further. Again, several questions are generated by using the features of each example in the dataset.

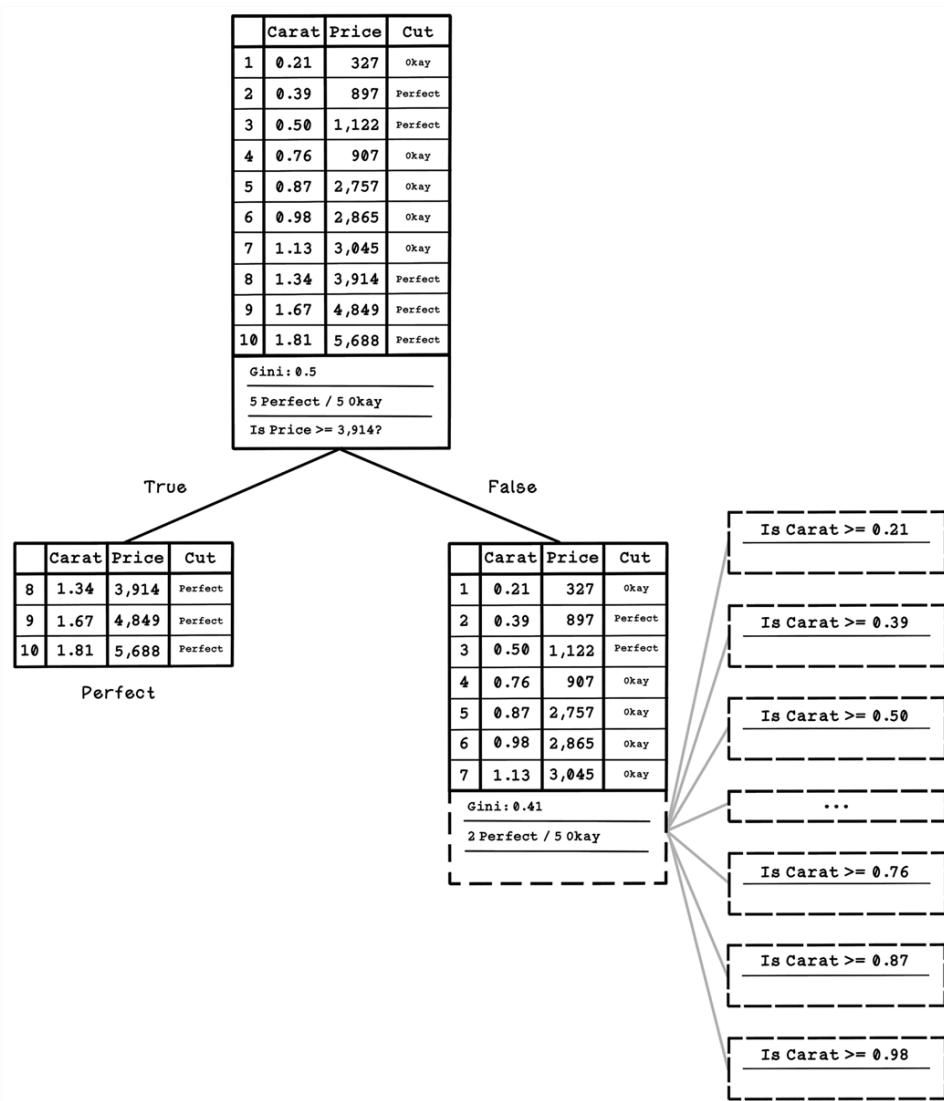


Figure 8.24 The resulting decision tree after the first decision node and possible questions

## EXERCISE: CALCULATING UNCERTAINTY AND INFORMATION GAIN FOR A QUESTION

Using the knowledge gained and figure 8.23 as a guide, calculate the information gain for the question "Is Carat  $\geq 0.76$ ?"

## SOLUTION: CALCULATING UNCERTAINTY AND INFORMATION GAIN FOR A QUESTION

The solution depicted in figure 8.25 highlights the reuse of the pattern of calculations that determine the entropy and information gain, given a question. Feel free to practice more questions and compare the results with the information-gain values in the figure.

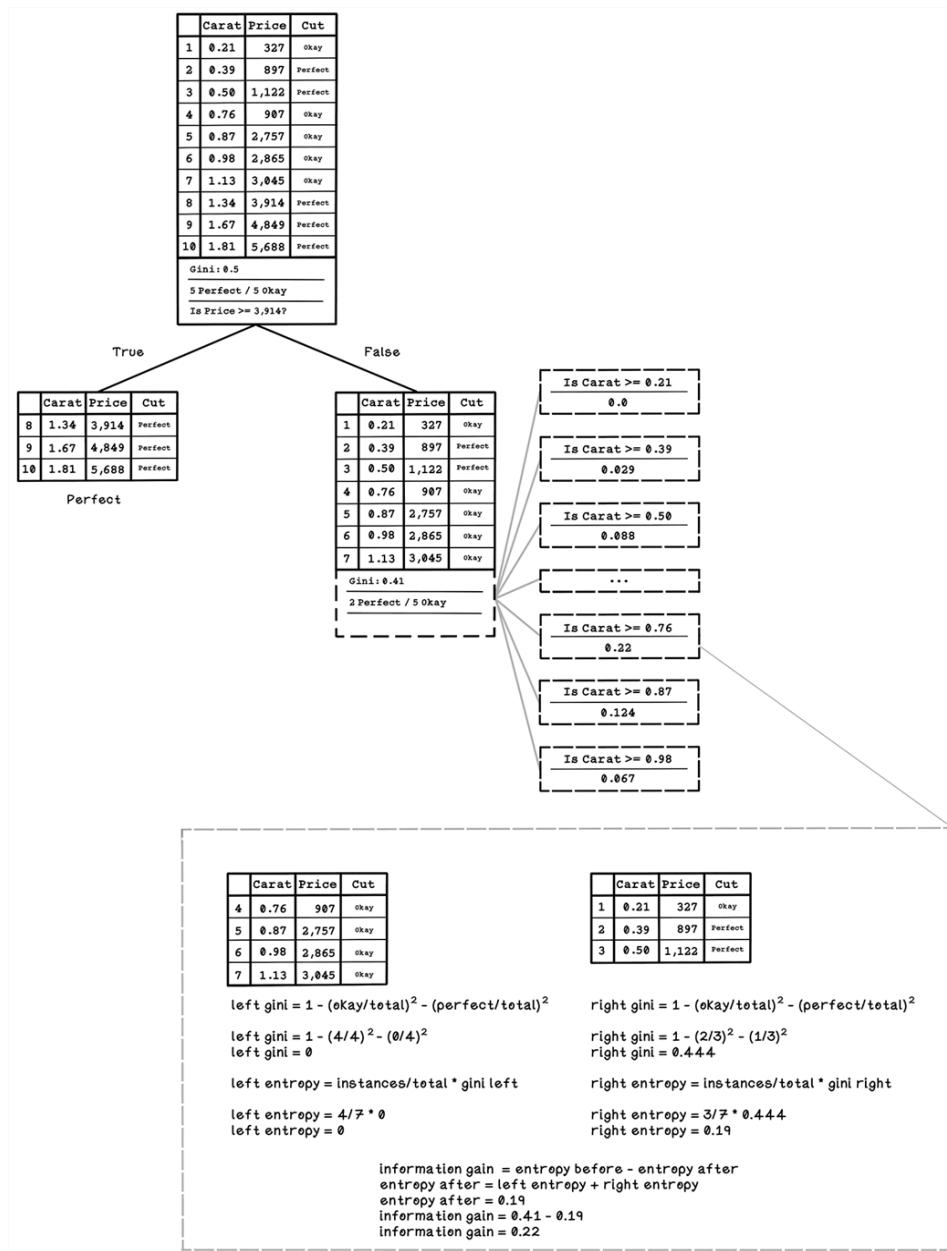


Figure 8.25 Illustration of data split and information gain based on a question at the second level

The process of splitting, generating questions, and determining information gained happens recursively until the dataset is completely categorized by questions. Figure 8.26 shows the complete decision tree, including all the questions asked and the resulting splits.

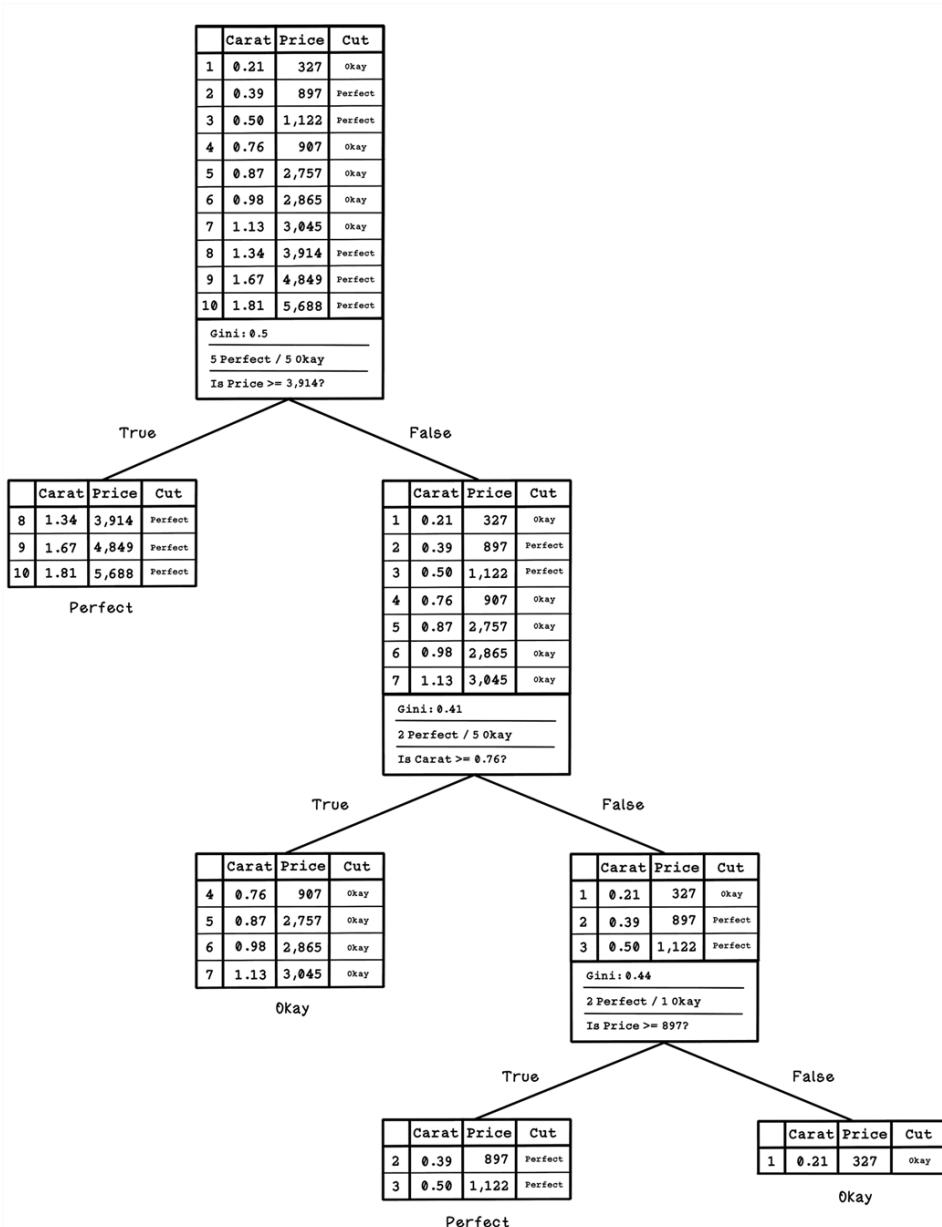


Figure 8.26 The complete trained decision tree

It is important to note that decision trees are usually trained with a much larger sample of data. The questions asked need to be more general to accommodate a wider variety of data and, thus, would need a variety of examples to learn from.

## PYTHON CODE SAMPLE

When programming a decision tree from scratch, the first step is counting the number of examples of each class—in this case, the number of Okay diamonds and the number of Perfect diamonds:

```
def find_unique_label_counts(examples):
    class_count = {}

    for example in examples:
        label = example['quality']

        if label not in class_count:
            class_count[label] = 0

        class_count[label] += 1

    return class_count
```

Next, examples are split based on a question. Examples that satisfy the question are stored in `examples_true`, and the rest are stored in `examples_false`:

```

class Question:
    def __init__(self, feature_name, value):
        self.feature_name = feature_name
        self.value = value

    def filter(self, example):
        return example[self.feature_name] >= self.value

def split_examples(examples, question):
    examples_true = []
    examples_false = []

    for example in examples:
        if question.filter(example):
            examples_true.append(example)
        else:
            examples_false.append(example)

    return examples_true, examples_false

```

We need a function that calculates the Gini index for a set of examples. The next function calculates the Gini index by using the method described in figure 8.23:

```

def calculate_gini(examples):
    label_counts = find_unique_label_counts(examples)
    total_examples = len(examples)

    uncertainty = 1

    for label in label_counts:
        count = label_counts[label]

        probability_of_label = count / total_examples

        uncertainty -= probability_of_label ** 2

    return uncertainty

```

`information_gain` uses the left and right splits and the current uncertainty to determine the information gain:

```
def calculate_information_gain(left, right, current_uncertainty):
    total = len(left) + len(right)

    left_gini = calculate_gini(left)
    left_entropy = (len(left) / total) * left_gini

    right_gini = calculate_gini(right)
    right_entropy = (len(right) / total) * right_gini

    uncertainty_after = left_entropy + right_entropy

    information_gain = current_uncertainty - uncertainty_after

    return information_gain
```

The next function may look daunting, but it's iterating over all the features and their values in the dataset, and finding the best information gain to determine the best question to ask:

```
def find_best_split(examples, feature_names):
    best_gain = 0
    best_question = None

    current_uncertainty = calculate_gini(examples)

    for feature_name in feature_names:

        values = set(example[feature_name] for example in examples)

        for value in values:

            question = Question(feature_name, value)

            true_examples, false_examples = split_examples(examples, question)

            if len(true_examples) != 0 and len(false_examples) != 0:

                gain = calculate_information_gain(
                    true_examples, false_examples, current_uncertainty
                )

                if gain >= best_gain:
                    best_gain = gain
                    best_question = question

    return best_gain, best_question
```

The next function ties everything together, using the functions defined previously to build a decision tree:

```

class ExamplesNode:
    def __init__(self, examples):
        self.predictions = find_unique_label_counts(examples)

    def __repr__(self):
        return f"Leaf: {self.predictions}"

class DecisionNode:
    def __init__(self, question, true_branch, false_branch):
        self.question = question
        self.true_branch = true_branch
        self.false_branch = false_branch

    def __repr__(self):
        return f"DecisionNode: {self.question}"

def build_tree(examples, feature_names):
    gain, question = find_best_split(examples, feature_names)

    if gain == 0:
        return ExamplesNode(examples)

    true_examples, false_examples = split_examples(examples, question)

    true_branch = build_tree(true_examples, feature_names)
    false_branch = build_tree(false_examples, feature_names)

    return DecisionNode(question, true_branch, false_branch)

```

Note that this function is recursive. It splits the data and recursively splits the resulting dataset until there is no information gain, indicating that the examples cannot be split any further. As a reminder, *decision nodes* are used to split the examples, and *example nodes* are used to store split sets of examples.

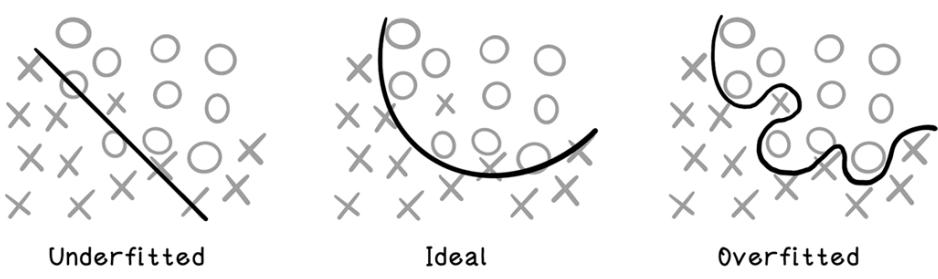
We've now learned how to build a decision-tree classifier. Remember that the trained decision-tree model will be tested with unseen data, similar to the linear regression approach explored earlier.

One problem with decision trees is overfitting, which occurs when the model is trained too well on several examples but performs poorly for new examples. Overfitting happens when the model learns the patterns of the training data but new real-world data is slightly different and doesn't meet the splitting criteria of the trained model. While achieving 100% accuracy on training data sounds ideal, it is often a red flag. True overfitting is identified not just by a high training score, but by the gap between high training accuracy and poor performance on new, unseen data.

To prevent this in Decision Trees, we use a technique called Pruning. Just as a gardener trims a tree to keep it healthy, we remove branches that are too deep or too specific, forcing the model to learn general patterns rather than memorizing every single data point.

Overfitting can happen with any machine learning model, not just decision trees.

Figure 8.27 illustrates the concept of overfitting. Underfitting includes too many incorrect classifications, and overfitting includes too few or no incorrect classifications; the ideal is somewhere in between.



**Figure 8.27 Underfitting, ideal, and overfitting**

#### 8.4.4 Classifying examples with decision trees

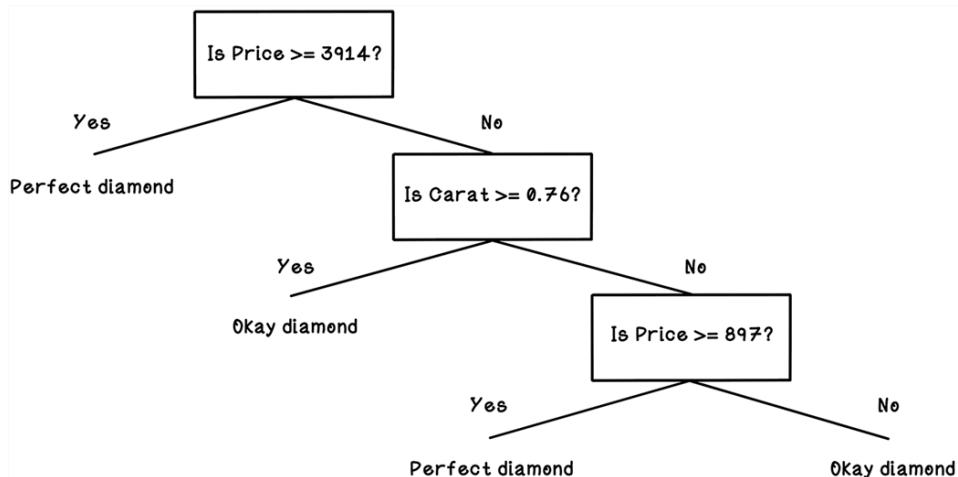
Now that a decision tree has been trained and the right questions have been determined, we can test it by providing it with new data to classify. The model that we're referring to is the decision tree of questions that was created by the training step.

To test the model, we provide several new examples of data and measure whether they have been classified correctly, so we need to know the labeling of the testing data. In the diamond example, we need more diamond data, including the Cut feature, to test the decision tree (table 8.16).

**Table 8.16 The diamond dataset for classification**

	<b>Carat</b>	<b>Price</b>	<b>Cut</b>
1	0.26	689	Perfect
2	0.41	967	Perfect
3	0.52	1,012	Perfect
4	0.76	907	Okay
5	0.81	2,650	Okay
6	0.90	2,634	Okay
7	1.24	2,999	Perfect
8	1.42	3850	Perfect
9	1.61	4,345	Perfect
10	1.78	3,100	Okay

Figure 8.28 illustrates the decision-tree model that we trained, which will be used to process the new examples. Each example is fed through the tree and classified.

**Figure 8.28 The decision tree model that will process new examples**

The resulting predicted classifications are detailed in table 8.17. Assume that we're trying to predict Okay diamonds. Notice that three examples are incorrect. That result is 3 of 10, which means that the model predicted 7 of 10, or 70% of the testing data correctly. This performance isn't terrible, but it illustrates how examples can be misclassified.

**Table 8.17 The diamond dataset for classification and predictions**

	<b>Carat</b>	<b>Price</b>	<b>Cut</b>	<b>Prediction</b>	
1	0.26	689	Okay	Okay	✓
2	0.41	880	Perfect	Perfect	✓
3	0.52	1,012	Perfect	Perfect	✓
4	0.76	907	Okay	Okay	✓
5	0.81	2,650	Okay	Okay	✓
6	0.90	2,634	Okay	Okay	✓
7	1.24	2,999	Perfect	Okay	✗
8	1.42	3,850	Perfect	Okay	✗
9	1.61	4,345	Perfect	Perfect	✓
10	1.78	3,100	Okay	Perfect	✗

A confusion matrix is often used to measure the performance of a model with testing data. A *confusion matrix* describes the performance using the following metrics (figure 8.29):

- *True positive (TP)*—Correctly classified examples as Okay
- *True negative (TN)*—Correctly classified examples as Perfect
- *False positive (FP)*—Perfect examples classified as Okay
- *False negative (FN)*—Okay examples classified as Perfect

	Predicted positive	Predicted negative	
Actual positive	True positive TP	False negative FN	Sensitivity $TP / (TP + FN)$
Actual negative	False positive FP	True negative TN	Specificity $TN / (TN + FP)$
	Precision $TP / (TP + FP)$	Negative precision $TN / (TN + FN)$	Accuracy $\frac{TP + TN}{TP + TN + FP + FN}$

**Figure 8.29 A confusion matrix**

The outcomes of testing the model with unseen examples can be used to deduce several measurements:

- *Precision*—How often Okay examples are classified correctly
- *Negative precision / Negative Predictive Value (NPV)*—How often Perfect examples are classified correctly
- *Sensitivity or recall*—Also known as the *true-positive rate*; the ratio of correctly classified Okay diamonds to all the actual Okay diamonds in the training set
- *Specificity*—Also known as the *true-negative rate*; the ratio of correctly classified Perfect diamonds to all actual Perfect diamonds in the training set
- *Accuracy*—How often the classifier is correct overall between classes

Figure 8.30 shows the resulting confusion matrix, with the results of the diamond example listed as input. Accuracy is important, but the other measurements can unveil additional useful information about the model’s performance.

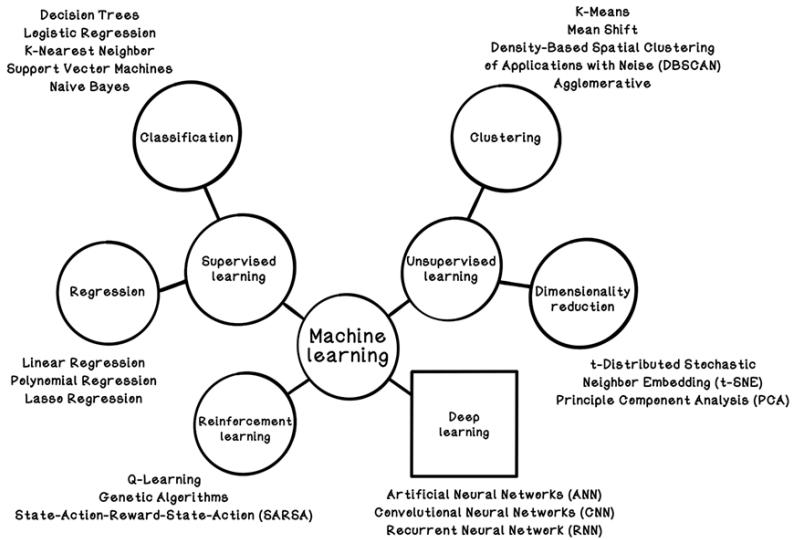
	Predicted positive	Predicted negative	
Actual positive	True positive 4	False negative 1	Sensitivity $4 / (4 + 1) = 0.8$
Actual negative	False positive 2	True negative 3	Specificity $3 / (3 + 2) = 0.6$
	Precision $4 / (4 + 2) = 0.67$	Negative precision $3 / (1 + 3) = 0.75$	Accuracy $\frac{4 + 3}{4 + 3 + 2 + 1} = 0.7$

**Figure 8.30 Confusion matrix for the diamond test example**

By using these measurements, we can make more-informed decisions in a machine learning life cycle to improve the performance of the model. As mentioned throughout this chapter, machine learning is an experimental exercise involving some trial and error. These metrics are guides in this process.

## 8.5 Other popular machine learning algorithms

This chapter explores two popular and fundamental machine learning algorithms. The linear-regression algorithm is used for regression problems in which the relationships between features are discovered. The decision-tree algorithm is used for classification problems in which the relationships between features and categories of examples are discovered. But many other machine learning algorithms are suitable in different contexts and for solving different problems. Figure 8.31 illustrates some popular algorithms and shows how they fit into the machine learning landscape.



**Figure 8.31 A map of popular machine learning algorithms**

The classification and regression algorithms satisfy problems similar to the ones explored in this chapter. Unsupervised learning contains algorithms that can help with some of the data preparation steps, find hidden underlying relationships in data, and inform what questions can be asked in a machine learning experiment.

Notice the introduction of deep learning in figure 8.31. Chapter 9 covers artificial neural networks—a key concept in deep learning. This chapter will give us a better understanding of the types of problems that can be solved with these approaches and how the algorithms are implemented.

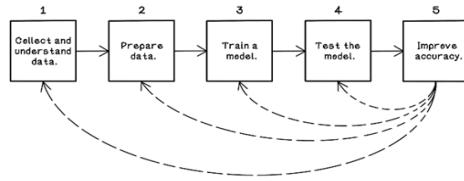
## 8.6 Use cases for machine learning algorithms

Machine learning can be applied in almost every industry to solve a plethora of problems in different domains. Given the right data and the right questions, the possibilities are potentially endless. We have all interacted with a product or service that uses some aspect of machine learning and data modeling in our everyday lives. This section highlights some of the popular ways machine learning can be used to solve real-world problems at scale:

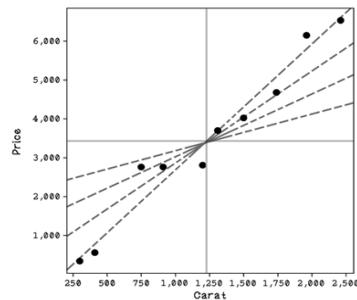
- *Fraud and threat detection*—Machine learning has been used to detect and prevent fraudulent transactions in the finance industry. Financial institutions have gained a wealth of transactional information over the years, including fraudulent transaction reports from their customers. These fraud reports are an input to labeling and characterizing fraudulent transactions. The models might consider the location of the transaction, the amount, the merchant, and so on to classify transactions, saving consumers from potential losses and the financial institution from insurance losses. The same model can be applied to network threat detection to detect and prevent attacks based on known network use and reported unusual behavior.
- *Product and content recommendations*—Many of us use e-commerce sites to purchase goods or media streaming services for audio and video consumption. Products may be recommended to us based on what we’re purchasing, or content may be recommended based on our interests. This functionality is usually enabled by machine learning, in which patterns in purchase or viewing behavior is derived from people’s interactions. Recommender systems are being used in more and more industries and applications to enable more sales or provide a better user experience.
- *Dynamic product and service pricing*—Products and services are often priced based on what someone is willing to pay for them or based on risk. For a ride-sharing system, it might make sense to hike the price if there are fewer available cars than the demand for a ride, sometimes referred to as *surge pricing*. In the insurance industry, a price might be hiked if a person is categorized as high-risk. Machine learning is used to find the attributes and relationships between the attributes that influence pricing based on dynamic conditions and details about a unique individual.
- *Health-condition risk prediction*—The medical industry requires health professionals to acquire an abundance of knowledge so that they can diagnose and treat patients. Over the years, they have gained a vast amount of data about patients: blood types, DNA, family-illness history, geographic location, lifestyle, and more. This data can be used to find potential patterns that can guide the diagnosis of illness. The power of using data to find diagnoses is that we can treat conditions before they mature. Additionally, by feeding the outcomes back into the machine learning system, we can strengthen its reliability in making predictions.

## 8.7 Summary of machine learning

In machine learning, the domain being explored, questions being asked, and data are as important as the algorithms used

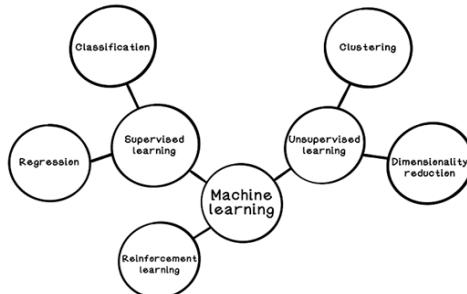
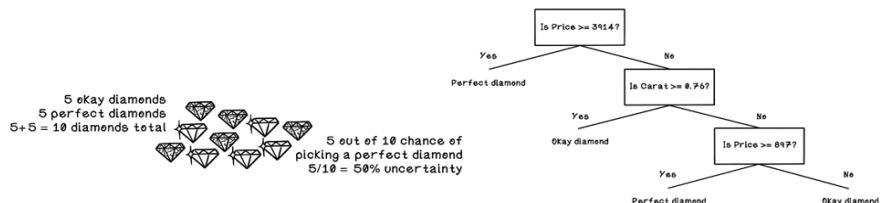


The lifecycle of ML projects is iterative and experimental



Linear regression involves finding the best line to fit the data, which means minimizing the error to each data point

Decision trees split data using questions until the dataset is perfectly split into categories - the key concept is reducing uncertainty in the dataset



Different ML algorithms are used to answer different types of questions and achieve different goals in different scenarios and contexts

# 9 Artificial neural networks

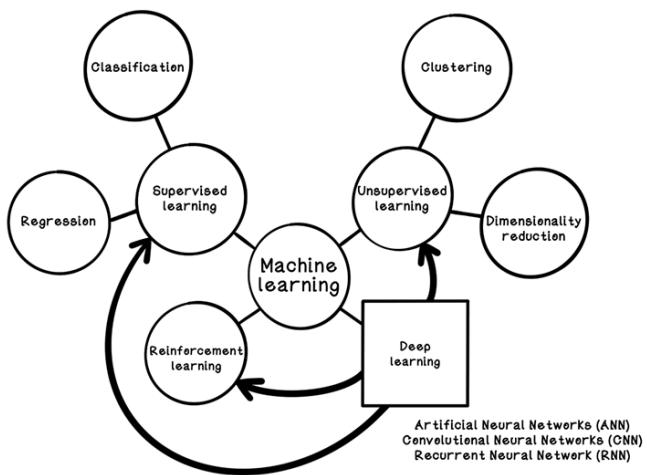
## This chapter covers

- Understanding the inspiration and intuition of artificial neural networks
- Identifying problems that can be solved with artificial neural networks
- Understanding and implementing forward propagation using a trained network
- Understanding and implementing backpropagation to train a network
- Designing artificial neural network architectures to tackle different problems

## 9.1 What are artificial neural networks?

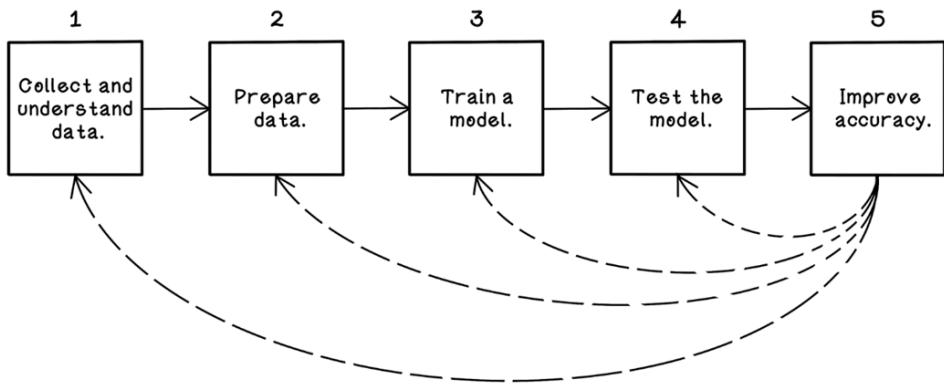
*Artificial neural networks* (ANNs) are powerful tools in the machine learning toolkit, used in a variety of ways to accomplish objectives such as image recognition, natural language processing, and game playing. ANNs learn in a similar way to other machine learning algorithms: by using training data. They are best suited to unstructured data where it's difficult to understand how features relate to one another. This chapter covers the inspiration of ANNs; it also shows how the algorithm works and how ANNs are designed to solve different problems.

To gain a clear understanding of how ANNs fit into the bigger machine learning landscape, we should review the composition and categorization of machine learning - algorithms. *Deep learning* is the name given to algorithms that use ANNs in varying architectures to accomplish an objective. Deep learning, including ANNs, can be used to solve supervised learning, unsupervised learning, and reinforcement learning problems. Figure 9.1 shows how deep learning relates to ANNs and other machine learning concepts.



**Figure 9.1** A map describing the flexibility of deep learning and ANNs

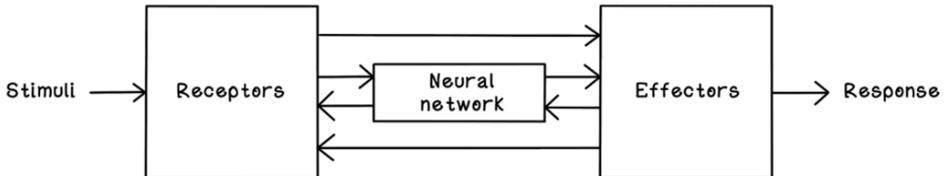
ANNs can be seen as just another model in the machine learning life cycle (chapter 8). Figure 9.2 recaps that life cycle. A problem needs to be identified; that data needs to be collected, understood, and prepared; and the ANN model will be tested and improved if necessary.



**Figure 9.2** A workflow for machine learning experiments and projects

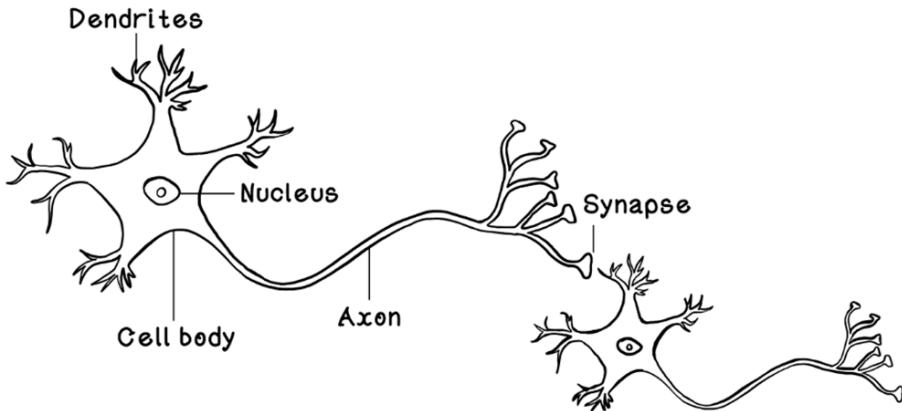
Now that we have an idea of how ANNs fit into the abstract machine learning landscape and know that an ANN is another model that is trained in the life cycle, let's explore the intuition and workings of ANNs. Like genetic algorithms and swarm-intelligence algorithms, ANNs are inspired by natural phenomena—in this case, the brain and nervous system. The nervous system is a biological structure that allows us to feel sensations and is the basis of how our brains think and operate. We have nerves across our entire bodies and neurons that behave similarly in our brains.

Neural networks consist of interconnected neurons that pass information by using electrical and chemical signals. Neurons pass information to other neurons and adjust that information to accomplish a specific function. When you grab a cup and take a sip of water, millions of neurons process the intention of what you want to do, the physical action to accomplish it, and the feedback to determine whether you were successful. Think about little children learning to drink from a cup. They usually start out poorly, dropping the cup a lot. Then they learn to grab it with two hands. Gradually, they learn to grab the cup with a single hand and take a sip without any problems. This process takes months. What's happening is that their brains and nervous systems are learning through practice or training. In our bodies, we have billions of neurons that are harnessed to learn from the signals of what we are doing, towards what goal, while determining our level of success. Figure 9.3 depicts a simplified model of this: receiving inputs (stimuli), processing them in a neural network, and providing outputs (response).



**Figure 9.3 A simplified model of a biological neural system**

Simplified, a *neuron* (figure 9.4) consists of *dendrites* that receive signals from other neurons; a *cell body and a nucleus* that activates and adjusts the signal; an *axon* that passes the signal to other neurons; and *synapses* that carry, and in the process adjust, the signal before it is passed to the next neuron's dendrites. Through approximately 90 billion of these neurons working together, our brains can function at the high level of intelligence that we know.



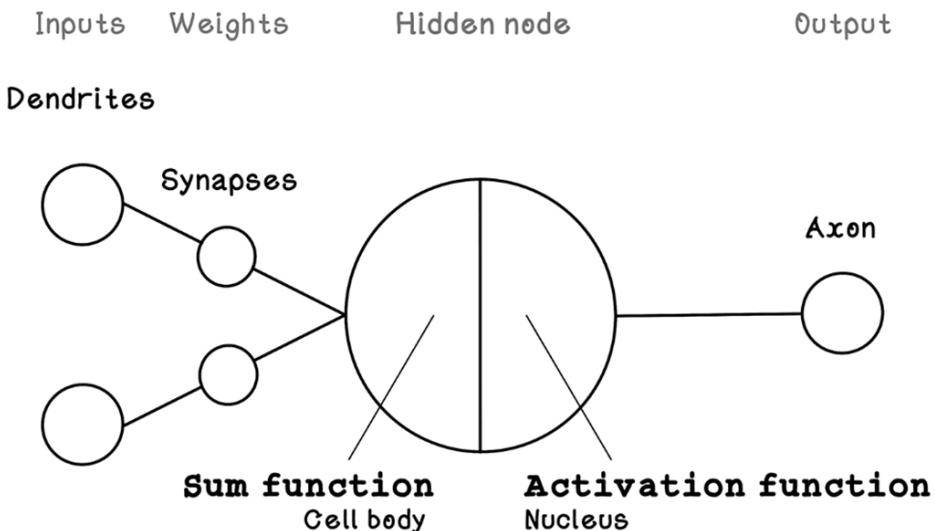
**Figure 9.4 The general composition of neurons**

Although ANNs are inspired by biological neural networks and use many of the concepts that are observed in these systems, they are not an identical representation of biological neural systems. From a biological standpoint, we still have a lot to learn about the human brain and nervous system.

## 9.2 The Perceptron: A representation of a neuron

The neuron is the fundamental concept that makes up the brain. As mentioned earlier, it accepts many inputs from other neurons, processes those inputs, and transfers the result to other connected neurons. ANNs are based on the fundamental concept of the *Perceptron* —a logical representation of a single biological neuron.

Like neurons, the Perceptron receives inputs (like dendrites), alters these inputs by using weights (like synapses), processes the weighted inputs (like the cell body and nucleus), and outputs a result (like axons). The Perceptron is loosely based on a neuron. You will notice that the synapses are depicted after the dendrites, representing the influence of synapses on incoming inputs. Figure 9.5 depicts the logical architecture of the Perceptron.



**Figure 9.5 Logical architecture of the Perceptron**

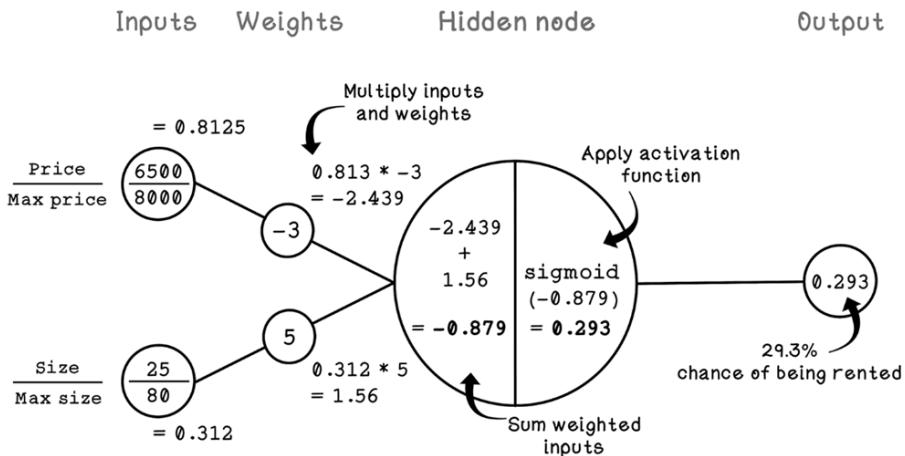
The components of the Perceptron are described by variables that are useful for calculating the output. Weights modify the inputs; that value is processed by a hidden node; and finally, the result is provided as the output.

Here is a brief description of the components of the Perceptron:

- *Inputs*—Describe the input values. In a neuron, these values would be an input signal.
- *Weights*—Describe the weights on each connection between an input and the hidden node. Weights influence the intensity of an input and result in a weighted input. In a neuron, these connections would be the synapses.
- *Hidden node (sum and activation)*—Sums the weighted input values and then applies an activation function to the summed result. An activation function determines the activation/output of the hidden node/neuron.
- *Output*—Describes the final output of the Perceptron.

To understand the workings of the Perceptron, we will examine the use of one by revisiting the apartment-hunting example from chapter 8. Suppose that we are real estate agents trying to determine whether a specific apartment will be rented within a month of listing it, based on the size of the apartment and the price of the apartment. Assume that a Perceptron has already been trained, meaning that the weights for the Perceptron have already been adjusted to make good predictions. We explore the way Perceptrons and ANNs are trained later in this chapter; for now, understand that the weights encode relationships among the inputs by adjusting the strength of inputs to make accurate predictions.

Figure 9.6 shows how we can use a pretrained Perceptron to classify whether an apartment will be rented or not. The inputs represent the price of a specific apartment and the size of that apartment. We're also using the maximum price and size to scale the inputs (\$8,000 for maximum price and 80 square meters for maximum size). For more about scaling data, see the next section.



**Figure 9.6 An example of using a trained Perceptron**

Notice that the price and size are the inputs and that the predicted chance of the apartment being rented is the output. The weights are key to achieving the prediction. Weights are the variables in the network that learn relationships among inputs. The summation and activation functions are used to process the inputs multiplied by the weights to make a prediction.

Think of a neuron as a sound engineer mixing a track:

- *Inputs*: The raw instruments (drums, guitar, vocals).
- *Weights*: The volume sliders. If the guitar is too loud (contributes too much to the error), the engineer lowers that specific weight. If the vocals are important, they increase that weight.
- *Bias*: The master gain. It boosts or cuts the overall signal level before it goes to the speakers.
- *Activation Function*: The speakers. They use the combined signal to produce a sound to the audience.

Notice that we're using an activation function called the *sigmoid function*. Activation functions play a critical role in the Perceptron and ANNs. In this case, the activation function is helping us solve a linear problem. But when we look at ANNs in the next section, we will see how activation functions are useful for receiving inputs to solve nonlinear problems. Figure 9.7 describes the basics of linear problems.

The sigmoid function takes any input value (from  $-\infty$  to  $+\infty$ ) and squashes it into an "S" curve between 0 and 1.

While this is useful for calculating probabilities, it comes with a risk: The *vanishing gradient*. For very large or very small inputs, the curve becomes almost perfectly flat. When the curve is flat, the gradient (slope) is near zero, which can cause the network to stop learning entirely. This is why modern networks often use other activation functions, like ReLU, for hidden layers. When we get to the deeper workings of ANNs later in this chapter, we will see how activation functions help solve nonlinear problems as well.

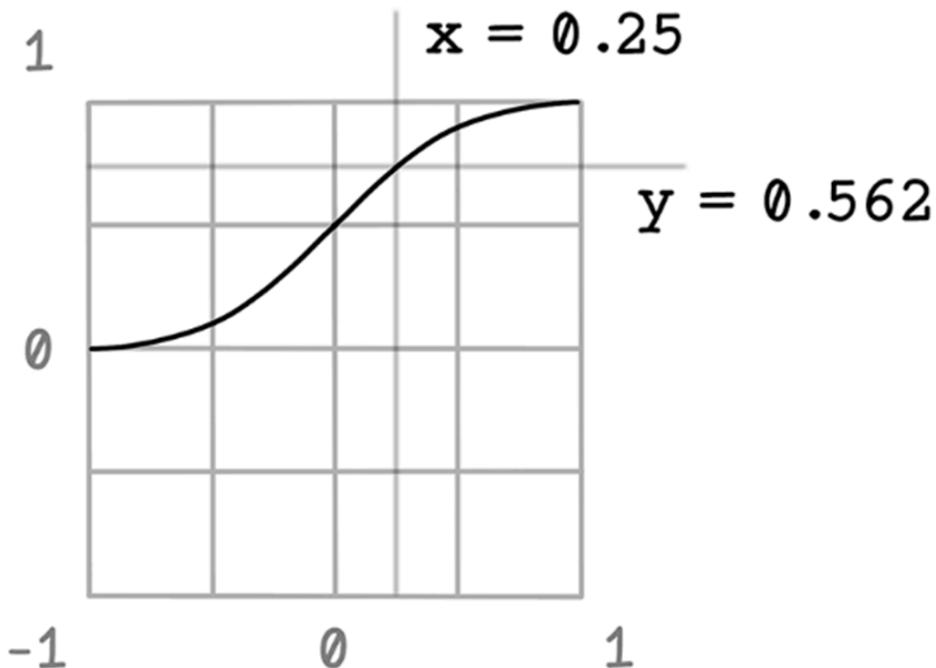
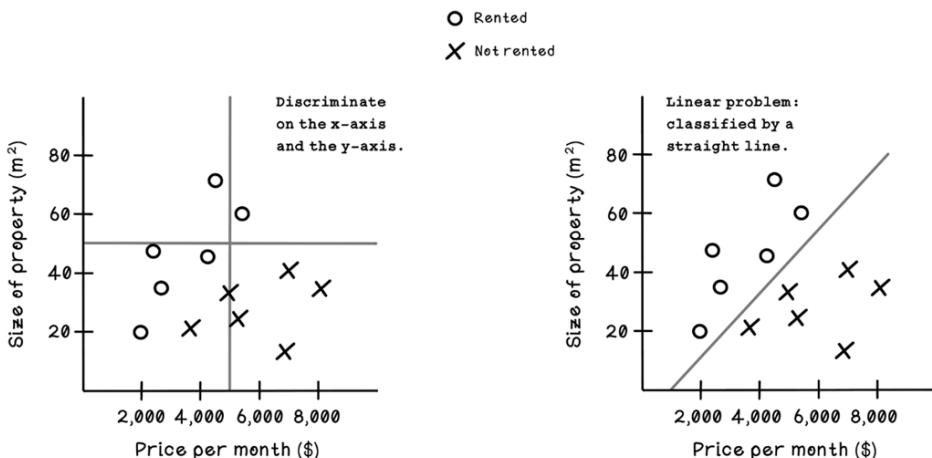


Figure 9.7 The sigmoid function

Let's take a step back and look at the data that we're using for the Perceptron. Understanding the data related to whether an apartment was sold is important for understanding what the Perceptron is doing. Figure 9.8 illustrates the examples in the dataset, including the price and size of each apartment. Each apartment is labeled as one of two classes: rented or not rented. The line separating the two classes is the function described by the Perceptron.



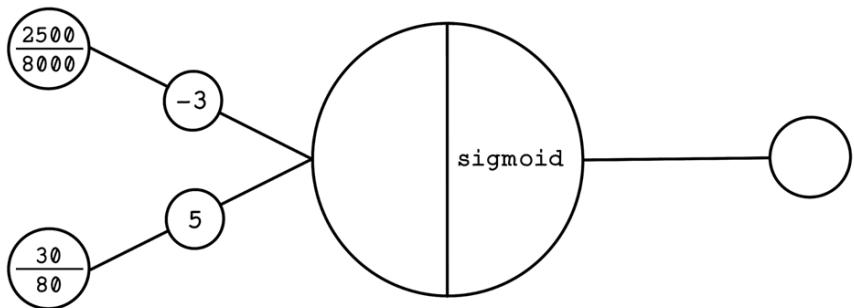
**Figure 9.8 Example of a linear classification problem**

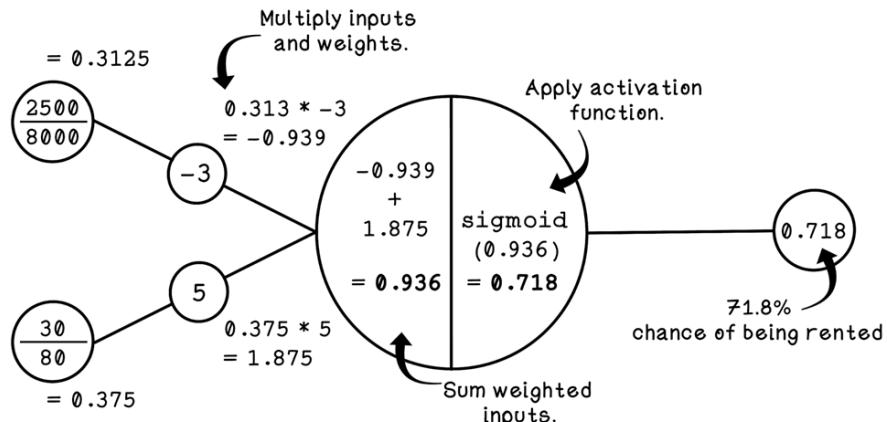
Although the Perceptron is useful for solving linear problems, it cannot solve nonlinear problems. If a dataset cannot be classified by a straight line, the Perceptron will fail.

ANNs use the concept of the Perceptron at scale. Many neurons similar to the Perceptron work together to solve nonlinear problems in many dimensions.

**Exercise: Calculate the output of the following input for the Perceptron**

- Using your knowledge of how the Perceptron works, calculate the output for the following:



**SOLUTION: CALCULATE THE OUTPUT OF THE FOLLOWING INPUT FOR THE PERCEPTRON**


### 9.3 Defining artificial neural networks

The Perceptron is useful for solving simple problems, but as the dimensions of the data increase, it becomes less feasible. ANNs use the principles of the Perceptron and apply them to many hidden nodes as opposed to a single one.

To explore the workings of multi-node ANNs, consider an example dataset related to car collisions. Suppose that we have data from several cars at the moment that an unforeseen object enters the path of their movement. The dataset contains features related to the conditions and whether a collision occurred, including the following:

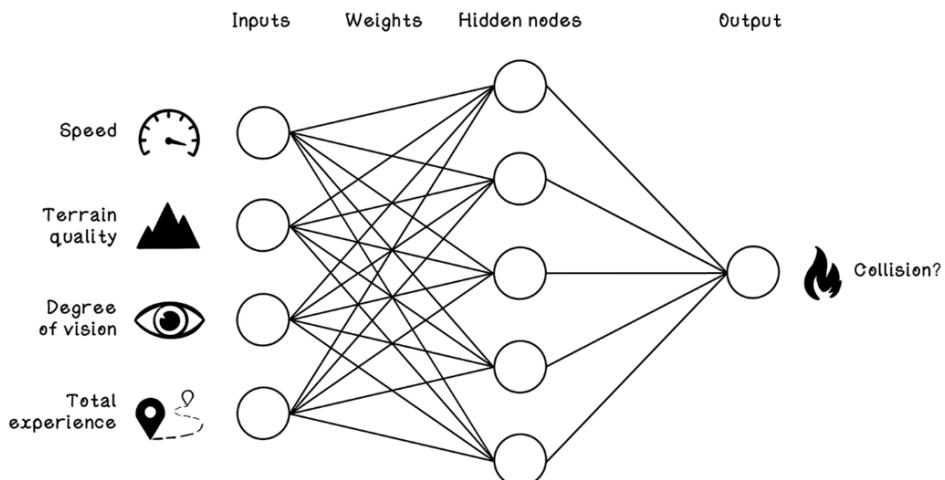
- *Speed*—The speed at which the car was traveling before encountering the object
- *Terrain quality*—The quality of the road on which the car was traveling before encountering the object
- *Degree of vision*—The driver’s degree of vision before the car encountered the object
- *Total experience*—The total driving experience of the driver of the car
- *Collision occurred?*—Whether a collision occurred or not

Given this data, we want to train a machine learning model—namely, an ANN—to learn the relationship between the features that contribute to a collision, as shown in table 9.1.

**Table 9.1 Car collision dataset**

	<b>Speed</b>	<b>Terrain quality</b>	<b>Degree of vision</b>	<b>Total experience</b>	<b>Collision occurred?</b>
1	65 km/h	5/10	180 °	80,000 km	No
2	120 km/h	1/10	72 °	110,000 km	Yes
3	8 km/h	6/10	288 °	50,000 km	No
4	50 km/h	2/10	324 °	1,600 km	Yes
5	25 km/h	9/10	36 °	160,000 km	No
6	80 km/h	3/10	120 °	6,000 km	Yes
7	40 km/h	3/10	360 °	400,000 km	No

An example ANN architecture can be used to classify whether a collision will occur based on the features we have. The features in the dataset must be mapped as inputs to the ANN, and the class that we are trying to predict is mapped as the output of the ANN. In this example, the input nodes are speed, terrain quality, degree of vision, and total experience; the output node is whether a collision happened (figure 9.9).

**Figure 9.9 Example ANN architecture for the car-collision example**

As with the other machine learning algorithms that we've worked through, preparing data is important for making an ANN classify data successfully. The primary concern is representing data in comparable ways. As humans, we understand the concept of speed and degree of vision, but the ANN doesn't have this context. Directly feeding 65 km/h and a 36-degree angle into the ANN causes a problem. Because 65 is numerically larger than 36, the network's math will be biased to think that speed is the "more important" feature.

To fix this, we need to scale our data. By squashing both values into a range between 0 and 1, we ensure the network weighs them based on their actual predictive power, not just their raw size.

A common way to scale data so that it can be compared is to use the *min-max* scaling approach, which aims to scale data to values between 0 and 1. By scaling all the data in a dataset to be consistent in format, we make the different features comparable. Because ANNs do not have any context about the raw features, we also remove bias with large input values. As an example, 1,000 seems to be much larger than 65, but 1,000 in the context of total driving experience is poor, and 65 in the context of driving speed is significant. Min-max scaling represents these pieces of data with the correct context by taking into account the minimum and maximum possible values for each feature.

Here are the minimum and maximum values selected for the features in the car-collision data:

- *Speed*—The minimum speed is 0, which means that the car is not moving. We will use the maximum speed of 120, because 120 km/h is the maximum legal speed limit in most places around the world. We will assume that the driver follows the rules.
- *Terrain quality*—Because the data is already in a rating system, the minimum value is 0, and the maximum value is 10.
- *Degree of vision*—We know that the total field of view in degrees is 360. So the minimum value is 0, and the maximum value is 360.
- *Total experience*—The minimum value is 0 if the driver has no experience. We will subjectively make the maximum value 400,000 for driving experience. The rationale is that if a driver has 400,000 km of driving experience, we consider that driver to be highly competent, and any further experience doesn't matter.

Min-max scaling uses the minimum and maximum values for a feature and finds the percentage of the actual value for the feature. The formula is simple: subtract the minimum from the value, and divide the result by the minimum subtracted from the maximum. Figure 9.10 illustrates the min-max scaling calculation for the first row of data in the car-collision example:

	<b>Speed</b>	<b>Terrain quality</b>	<b>Degree of vision</b>	<b>Total experience</b>	<b>Collision occurred?</b>
1	65 km/h	5/10	180°	80,000 km	No

	Speed	Terrain quality	Degree of vision	Total experience
	65 km/h	5 / 10	180°	80,000
	Min: 0 Max: 120	Min: 0 Max: 10	Min: 0 Max: 360	Min: 0 Max: 400,000
$\frac{\text{value} - \text{min}}{\text{max} - \text{min}}$	$\frac{65 - 0}{120 - 0}$	$\frac{5 - 0}{10 - 0}$	$\frac{180 - 0}{360 - 0}$	$\frac{80000 - 0}{400000 - 0}$
Scaled value	0.542	0.5	0.5	0.2

**Figure 9.10** Min-max scaling example with car collision data

Notice that all the values are between 0 and 1 and can be compared equally. The same formula is applied to all the rows in the dataset to ensure that every value is scaled. Note that for the value for the “Collision occurred?” feature, Yes is replaced with 1, and No is replaced with 0. Table 9.2 depicts the scaled car-collision data.

**Table 9.2** Car collision dataset scaled

	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
1	0.542	0.5	0.5	0.200	0
2	1.000	0.1	0.2	0.275	1
3	0.067	0.6	0.8	0.125	0
4	0.417	0.2	0.9	0.004	1
5	0.208	0.9	0.1	0.400	0
6	0.667	0.3	0.3	0.015	1
7	0.333	0.3	1.0	1.000	0

## PYTHON CODE SAMPLE

The code for scaling the data follows the logic and calculations for min-max scaling identically. We need the minimums and maximums for each feature, as well as the total number of features in our dataset. The `scale_dataset` function uses these parameters to iterate over every example in the dataset and scale the value by using the `scale_data_feature` function:

```
def scale_data_feature(data, feature_min, feature_max):
    return (data - feature_min) / (feature_max - feature_min)

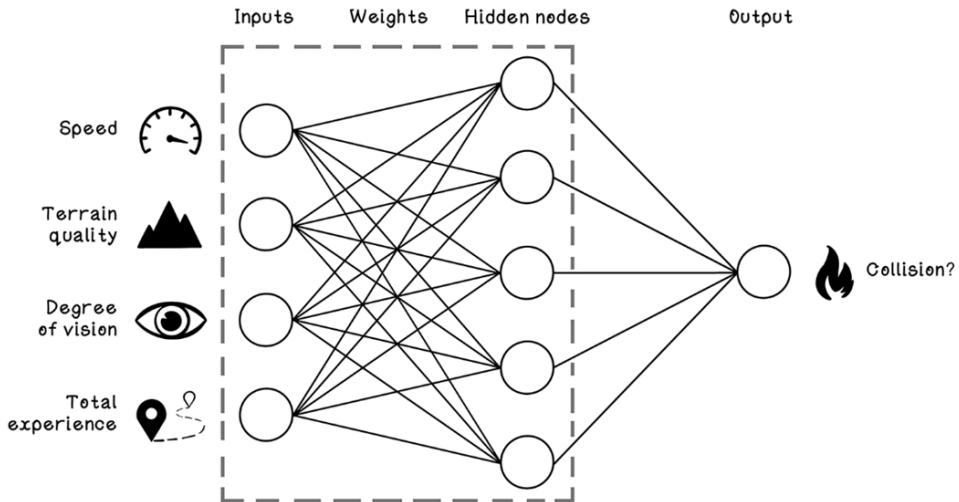
def scale_dataset(dataset, feature_count, feature_min, feature_max):
    scaled_data = []

    for data_row in dataset:
        example = []
        for i in range(feature_count):
            scaled_value = scale_data_feature(
                data_row[i],
                feature_min[i],
                feature_max[i]
            )
            example.append(scaled_value)
        scaled_data.append(example)

    return scaled_data
```

Now that we have prepared the data in a way that is suitable for an ANN to process, let's explore the architecture of a simple ANN. Remember that the features used to predict a class are the input nodes, and the class that is being predicted is the output node.

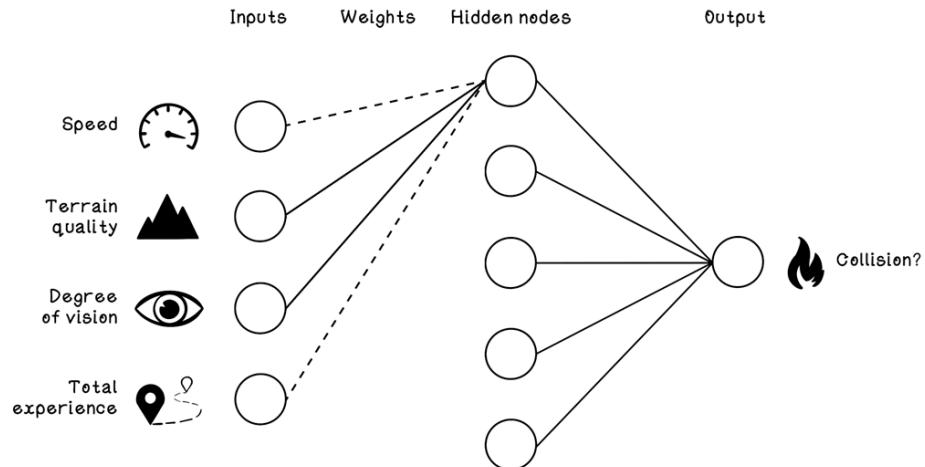
Figure 9.11 shows an ANN with one hidden layer, which is the single vertical layer in the figure, with five hidden nodes. These layers are called *hidden layers* because they are not directly observed from outside the network. Only the inputs and outputs are interacted with, which leads to the perception of ANNs as being black boxes. Each hidden node is similar to the Perceptron. A hidden node takes inputs and weights and then computes the sum and an activation function. Then the results of each hidden node are processed by a single output node.



**Figure 9.11 Example ANN architecture for the car-collision problem**

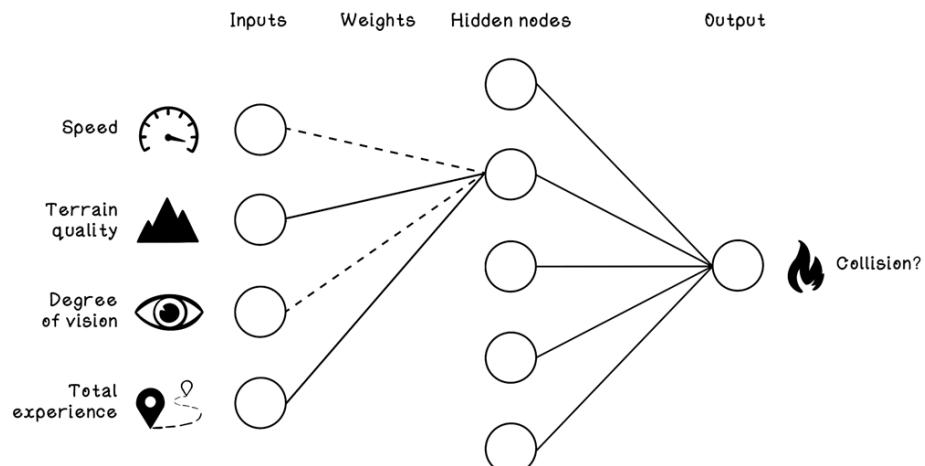
Before we consider the calculations and computation of an ANN, let's try to intuitively dig into what the network weights are doing at a high level. Because a single hidden node is connected to every input node but every connection has a different weight, independent hidden nodes might be concerned with specific relationships among two or more input nodes.

Figure 9.12 depicts a scenario in which the first hidden node has strong weightings on the connections to terrain quality and degree of vision but weak weightings on the connections to speed and total experience. This specific hidden node is concerned with the relationship between terrain quality and degree of vision. It might gain an understanding of the relationship between these two features and how it influences whether collisions happen; poor terrain quality and poor degree of vision, for example, might influence the likelihood of collisions more than good terrain quality and an average degree of vision. These relationships are usually more intricate than this simple example.



**Figure 9.12 Example of a hidden node comparing terrain quality and degree of vision**

In figure 9.13, the second hidden node might have strong weightings on the connections to terrain quality and total experience. Perhaps there is a relationship among different terrain qualities and variance in total driving experience that contributes to collisions.



**Figure 9.13 Example of a hidden node comparing terrain quality and total experience**

The nodes in a hidden layer can be conceptually compared with the analogy of ants discussed in chapter 6. Individual ants fulfill small tasks that are seemingly insignificant, but when the ants act as a colony, intelligent behavior emerges. Similarly, individual hidden nodes contribute to a greater goal in the ANN.

By analyzing the figure of the car-collision ANN and the operations within it, we can describe the data structures required for the algorithm:

- *Input nodes*—The input nodes can be represented by a single array that stores the values for a specific example. The array size is the number of features in the dataset that are being used to predict a class. In the car-collision example, we have four inputs, so the array size is 4.
- *Weights*—The weights can be represented by a matrix (a 2D array), because each input node has a connection to each hidden node and each input node has five connections. Because there are 4 input nodes with 5 connections each, the ANN has 20 weights toward the hidden layer and 5 toward the output layer, because there are 5 hidden nodes and 1 output node.
- *Hidden nodes*—The hidden nodes can be represented by a single array that stores the results of activation of each respective node.
- *Output node*—The output node is a single value representing the predicted class of a specific example or the chance that the example will be in a specific class. The output might be 1 or 0, indicating whether a collision occurred; or it could be something like 0.65, indicating a 65% chance that the example resulted in a collision.

## PYTHON CODE SAMPLE

The next piece of code describes a class that represents a neural network. Notice that the layers are represented as properties of the class and that all the properties are arrays, with the exception of the weights, which are matrices. An `output` property represents the predictions for the given examples, and an `expected_output` property is used during the training process:

```

class NeuralNetwork:
    def __init__(self, features, labels, hidden_node_count):
        num_features = features.shape[1]
        num_samples = features.shape[0]
        num_output = labels.shape[1]

        self.input = features
        self.expected_output = labels

        self.weights_input = np.random.randn(num_features, hidden_node_count) *
0.01

        self.hidden = np.zeros((num_samples, hidden_node_count))

        self.weights_hidden = np.random.randn(hidden_node_count, num_output) *
0.01

        self.output = np.zeros((num_samples, num_output))

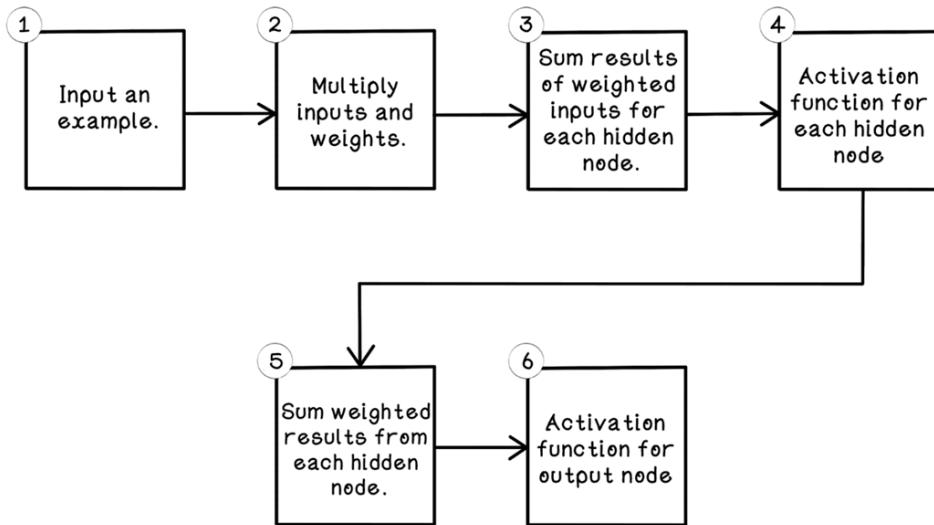
        self.hidden_node_count = hidden_node_count
        self.num_output = num_output
        self.num_samples = num_samples

```

## 9.4 Forward propagation: Using a trained ANN

A trained ANN is a network that has learned from examples and adjusted its weights to best predict the class of new examples. Don't panic about how the training happens and how the weights are adjusted; we will tackle this topic in the next section. Understanding forward propagation will assist us in grasping *backpropagation* (how weights are trained).

Now that we have a grounding in the general architecture of ANNs and the intuition of what nodes in the network might be doing, let's walk through the algorithm for using a trained ANN (figure 9.14).

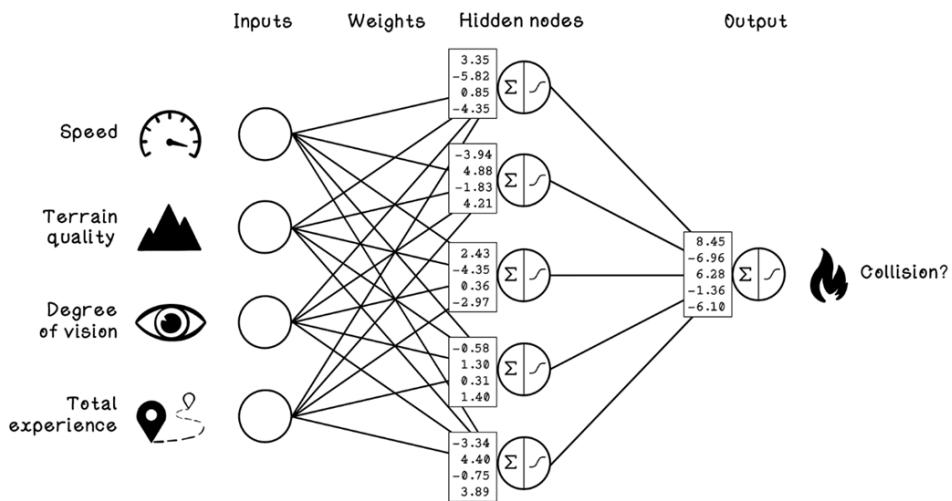


**Figure 9.14 Life cycle of forward propagation in an ANN**

As mentioned previously, the steps involved in calculating the results for the nodes in an ANN are similar to the Perceptron. Similar operations are performed on many nodes that work together; this addresses the Perceptron's flaws and is used to solve problems that have more dimensions. The general flow of forward propagation includes the following steps:

1. *Input an example*—Provide a single example from the dataset for which we want to predict the class.
2. *Multiply inputs and weights*—Multiply every input by each weight of its connection to hidden nodes.
3. *Sum results of weighted inputs for each hidden node*—Sum the results of the weighted inputs.
4. *Activation function for each hidden node*—Apply an activation function to the summed weighted inputs.
5. *Sum weighted results of hidden nodes to the output node*—Sum the weighted results of the activation function from all hidden nodes.
6. *Activation function for output node*—Apply an activation function to the summed weighted hidden nodes.

For the purpose of exploring forward propagation, we will assume that the ANN has been trained and the optimal weights in the network have been found. Figure 9.15 depicts the weights on each connection. The first box next to the first hidden node, for example, has the weight 3.35, which is related to the Speed input node; the weight -5.82 is related to the Terrain Quality input node; and so on.



**Figure 9.15 Example of weights in a pretrained ANN**

Because the neural network has been trained, we can use it to predict the chance of collisions by providing it with a single example. Table 9.3 serves as a reminder of the scaled dataset that we are using.

**Table 9.3 Car collision dataset scaled**

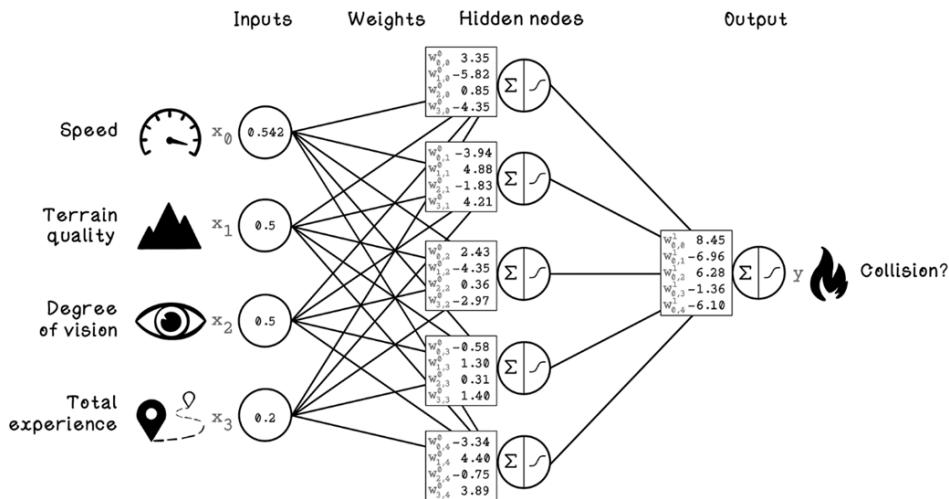
	<b>Speed</b>	<b>Terrain quality</b>	<b>Degree of vision</b>	<b>Total experience</b>	<b>Collision occurred?</b>
1	0.542	0.5	0.5	0.200	0
2	1.000	0.1	0.2	0.275	1
3	0.067	0.6	0.8	0.125	0
4	0.417	0.2	0.9	0.004	1
5	0.208	0.9	0.1	0.400	0
6	0.667	0.3	0.3	0.015	1
7	0.333	0.3	1.0	1.000	0

If you've ever looked into ANNs, you may have noticed some potentially frightening mathematical notations. Let's break down some of the concepts that can be represented mathematically.

The inputs of the ANN are denoted by  $X$ . Every input variable will be  $X$  subscripted by a number. Speed is  $X_0$ , Terrain Quality is  $X_1$ , and so on. The output of the network is denoted by  $y$ , and the weights of the network are denoted by  $W$ . Because we have two layers in the ANN—a hidden layer and an output layer—there are two groups of weights. The first group is superscripted by  $W_0$ , and the second group is  $W_1$ . Then each weight is denoted by the nodes to which it is connected. The weight between the Speed node and the first hidden node is  $W_{0,0}$ , and the weight between the Terrain Quality node and the first hidden node is  $W_{0,1}$ . These denotations aren't necessarily important for this example, but understanding them now will support future learning.

Figure 9.16 shows how the following data is represented in an ANN:

	<b>Speed</b>	<b>Terrain quality</b>	<b>Degree of vision</b>	<b>Total experience</b>	<b>Collision occurred?</b>
1	0.542	0.5	0.5	0.200	0



**Figure 9.16 Mathematical notation of an ANN**

As with the Perceptron, the first step is calculating the weighted sum of the inputs and the weight of each hidden node. In figure 9.17, each input is multiplied by each weight and summed for every hidden node.

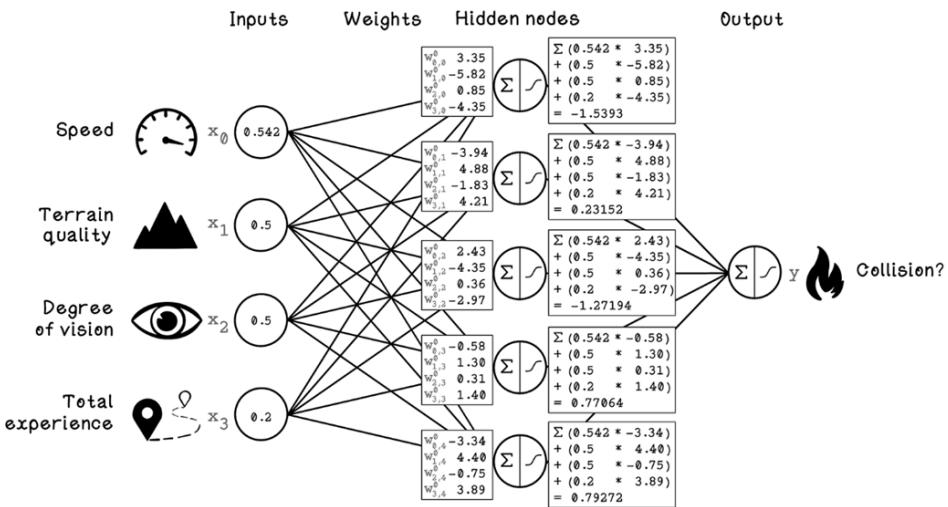


Figure 9.17 Weighted sum calculation for each hidden node

The next step is calculating the activation of each hidden node. We are using the sigmoid function, and the input for the function is the weighted sum of the inputs calculated for each hidden node (figure 9.18).

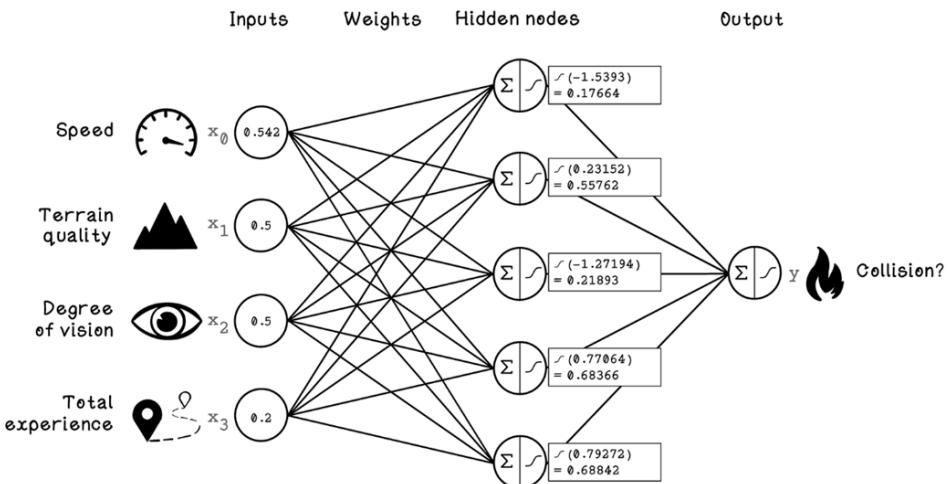
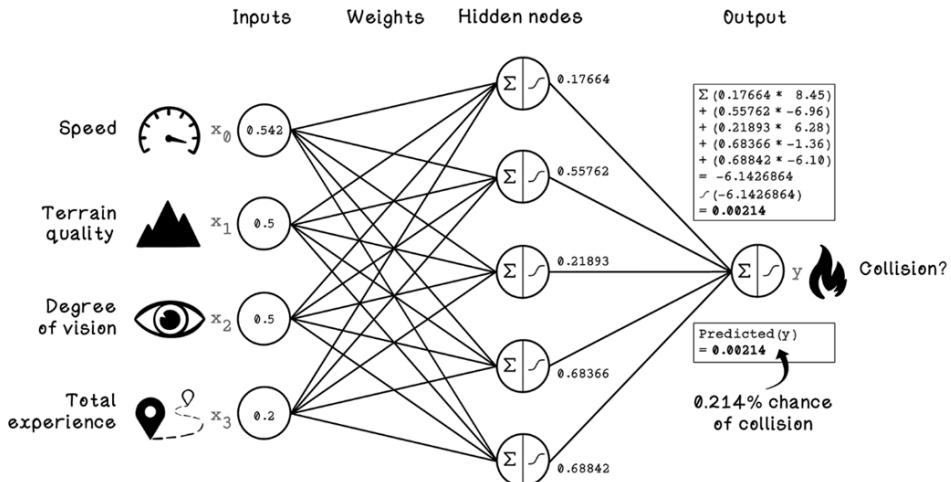


Figure 9.18 Activation function calculation for each hidden node

Now we have the activation results for each hidden node. When we mirror this result back to neurons, the activation results represent the activation intensity of each neuron. Because different hidden nodes may be concerned with different relationships in the data through the weights, the activations can be used in conjunction to determine an overall activation that represents the chance of a collision, given the inputs.

Figure 9.19 depicts the activations for each hidden node and the weights from each hidden node to the output node. To calculate the final output, we repeat the process of calculating the weighted sum of the results from each hidden node and applying the sigmoid activation function to that result.

**NOTE** The sigma symbol ( $\Sigma$ ) in the hidden nodes depicts the sum operation.



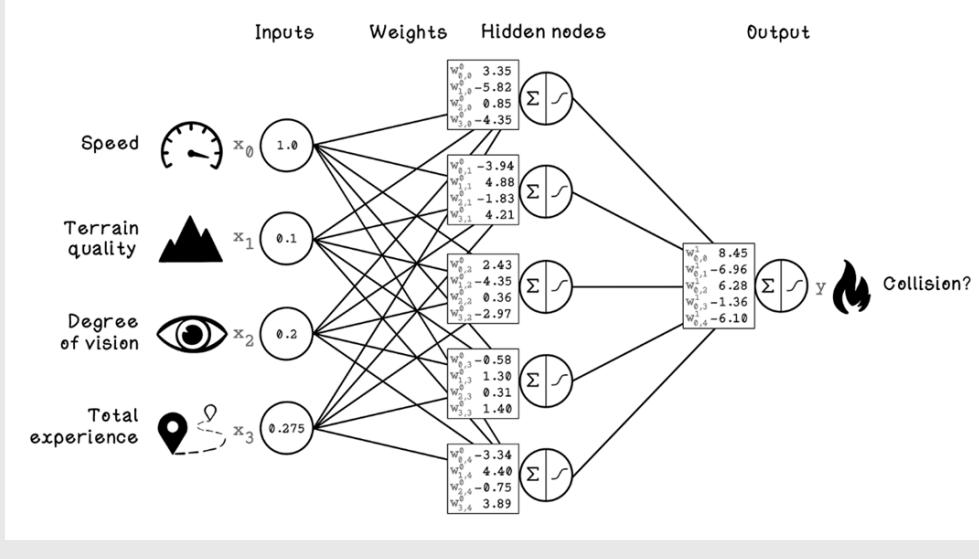
**Figure 9.19** Final activation calculation for the output node

We have calculated the output prediction for our example. The result is 0.00214, but what does this number mean? The output is a value between 0 and 1 that represents the probability that a collision will occur. In this case, the output is 0.214 percent (0.00214 \* 100), indicating that the chance of a collision is almost 0 percent.

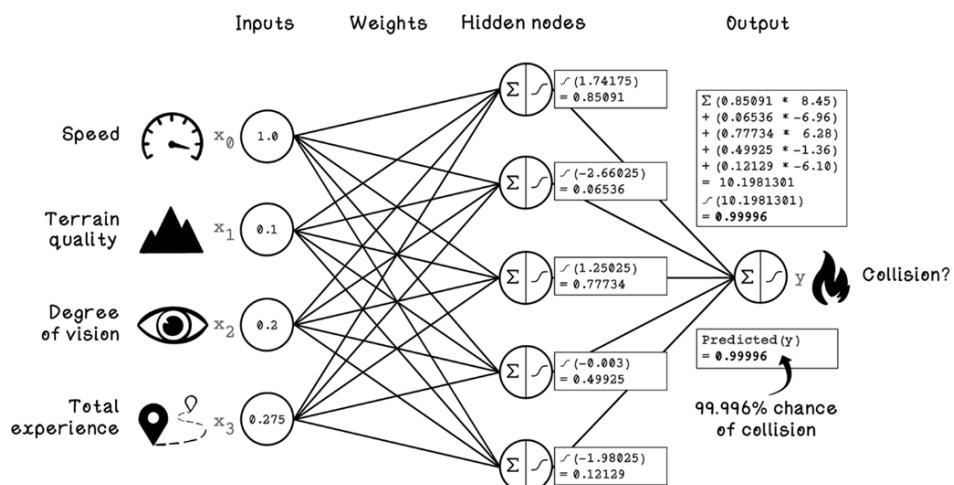
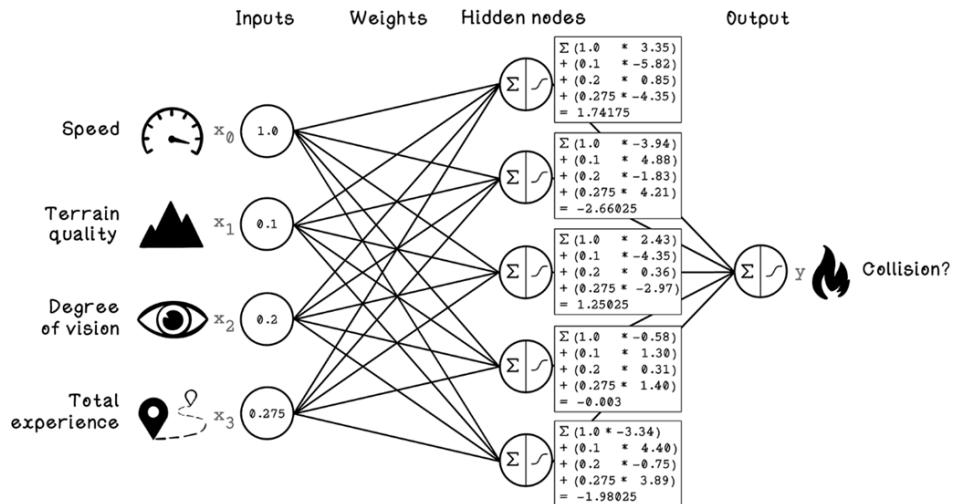
The following exercise uses another example from the dataset.

**EXERCISE: CALCULATE THE PREDICTION FOR THE EXAMPLE BY USING FORWARD PROPAGATION WITH THE FOLLOWING ANN**

	Speed	Terrain quality	Degree of vision	Total experience	Collision occurred?
2	1.000	0.1	0.2	0.275	1



**Solution: Calculate the prediction for the example by using forward propagation with the following ANN**



When we run this example through our pretrained ANN, the output is 0.99996, or 99.996 percent, so there is an extremely high chance that a collision will occur. By applying some human intuition to this single example, we can see why a collision is likely. The driver was traveling at the maximum legal speed, on the poorest-quality terrain, with a poor field of vision.

## PYTHON CODE SAMPLE

One of the important functions for activation in our example is the sigmoid function. This method describes the mathematical function that represents the S curve:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

---

**NOTE** `exp` is a mathematical constant called Euler's number, also denoted by `e`, approximately **2.71828**.

---

Notice that the same neural network class defined earlier in the chapter is described in the following code. This time, a `forward_propagation` function is included. This function sums the input and weights between input and hidden nodes, applies the sigmoid function to each result, and stores the output as the result for the nodes in the hidden layer. This is done for the hidden node output and weights to the output node as well.

```

class NeuralNetwork:
    def __init__(self, features, labels, hidden_node_count):
        num_features = features.shape[1]
        num_samples = features.shape[0]
        num_output = labels.shape[1]

        self.input = features
        self.expected_output = labels
        self.hidden_node_count = hidden_node_count

        self.weights_input = np.random.randn(num_features, hidden_node_count) *
0.01
        self.weights_hidden = np.random.randn(hidden_node_count, num_output) *
0.01

        self.hidden = np.zeros((num_samples, hidden_node_count))
        self.output = np.zeros((num_samples, num_output))

    def forward_propagation(self):
        self.hidden = sigmoid(np.dot(self.input, self.weights_input))
        self.output = sigmoid(np.dot(self.hidden, self.weights_hidden))

```

## 9.5 Backpropagation: Training an ANN

The machine learning life cycle and principles covered in chapter 8 are important for tackling backpropagation in ANNs. An ANN can be seen as another machine learning model. We still need to have a question to ask. We're still collecting and understanding data in the context of the problem, and we need to prepare the data in a way that is suitable for the model to process.

We need a subset of data for training and a subset of data for testing how well the model performs. Also, we will be iterating and improving through collecting more data, preparing it differently, or changing the architecture and configuration of the ANN.

Training an ANN consists of three main phases. Phase A involves setting up the ANN architecture, including configuring the inputs, outputs, and hidden layers. Phase B is forward propagation. And phase C is backpropagation, which is where the training happens (figure 9.20).

Imagine a relay race team that lost the race by 5 seconds (the total error). The coach walks backward from the finish line to fix it.

“Runner 4, you were 2 seconds slow. Fix it.”

“Runner 3, you handed the baton off poorly, which caused Runner 4 to be late. Fix it.”

Backpropagation looks at the final error and works backward through the network, calculating exactly how much each specific weight contributed to that error, and tells it to adjust.

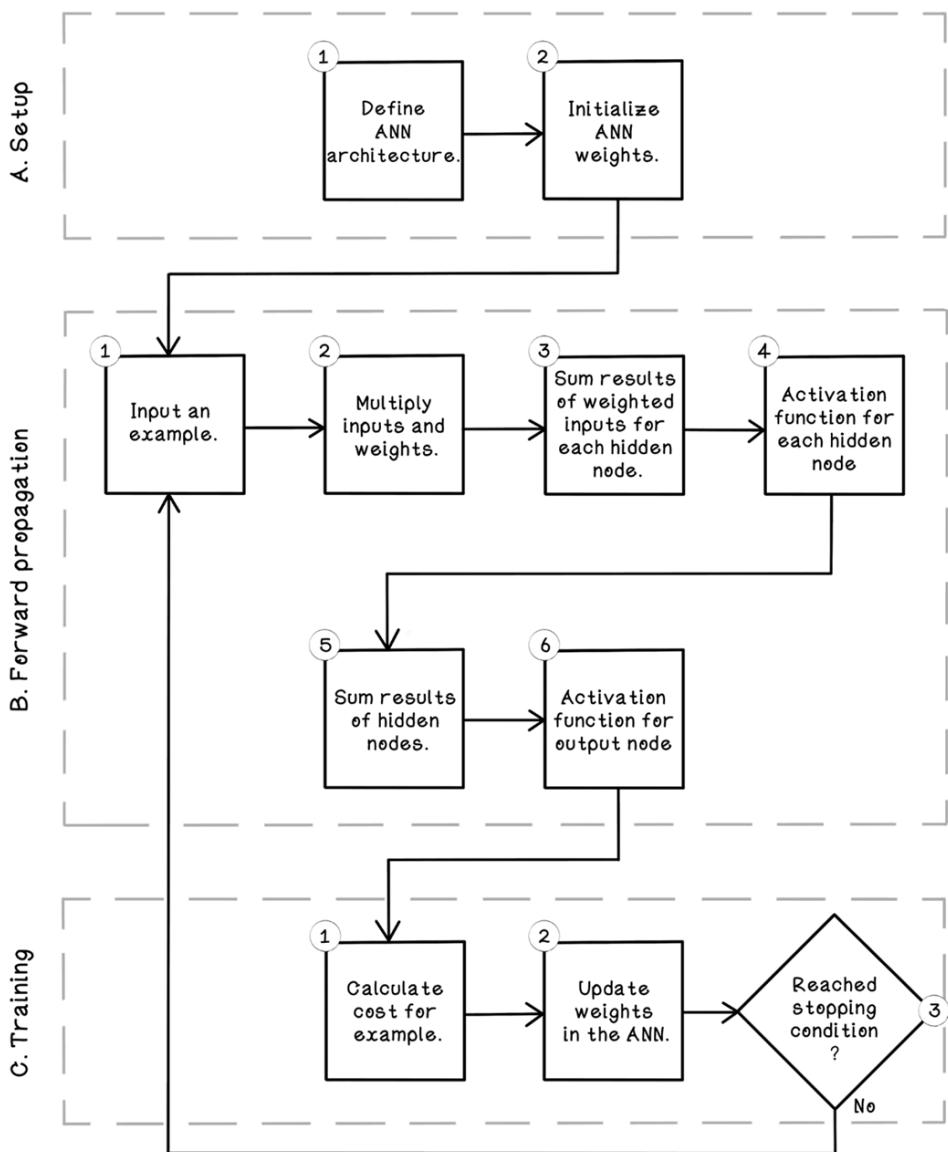


Figure 9.20 Life cycle of training an ANN

Sections of the lifecycle are broken down into Phase A, Phase B, and Phase C for ease of understanding the operations involved in the back-propagation algorithm.

### 9.5.1 Phase A: Setup

1. *Define ANN architecture.* This step involves defining the input nodes, the output nodes, the number of hidden layers, the number of neurons in each hidden layer, the activation functions used, and more.
2. *Initialize ANN weights.* The weights in the ANN must be initialized to some value. We can take various approaches, with random numbers from a standard normal distribution being an effective approach. The key principle is that the weights will be adjusted constantly as the ANN learns from training examples.

### 9.5.2 Phase B: Forward propagation

This process is the same one that we covered earlier in this chapter, and the same calculations are carried out to produce an output for the network. The predicted output, however, will be compared with the actual class of the example from the training data set that is fed into the network. So if the network produced the prediction of 0.99996 (99.996%) after the final hidden layer's activation function, then it will be compared to the actual value which would indicate if a collision did occur (1, or 100%) or did not occur (0). We'll explore how the difference between the predicted output and actual value is critical to learning in the Phase C.

### 9.5.3 Phase C: Training

1. *Calculate cost.* Following from forward propagation, the cost is the difference between the predicted output and the actual class for the examples in the training set. The cost effectively determines how bad the ANN is at predicting the class of examples. If the network predicted 0.7 and the actual value was 1.0, trivially, we have a loss of 0.3.
2. *Update weights in the ANN.* The weights of the ANN are the only things that can be adjusted by the network itself. The architecture and configurations that we defined in phase A don't change during training the network. The weights essentially encode the intelligence of the network. Weights are adjusted to be larger or smaller, affecting the strength of the inputs (adjusts the importance of the relationships between certain inputs to adjust predictions).
3. *Define a stopping condition.* Training cannot happen indefinitely. As with many of the algorithms explored in this book, a sensible stopping condition needs to be determined. If we have a large dataset, we might decide that we will use 500 examples in our training dataset over 1,000 iterations to train the ANN. In this example, the 500 examples will be passed through the network 1,000 times, and the weights will be adjusted in every iteration.

When we worked through forward propagation, the weights were already defined because the network was pretrained. Before we start training the network, we need to initialize the weights to some value, and the weights need to be adjusted based on training examples. One approach to initializing weights is to choose random weights from a normal distribution.

Figure 9.21 illustrates the randomly generated weights for our ANN. It also shows the calculations for forward propagation for the hidden nodes, given a single training example. The first example input used in the forward propagation section is used here to highlight the differences in output, given different weights in the network.

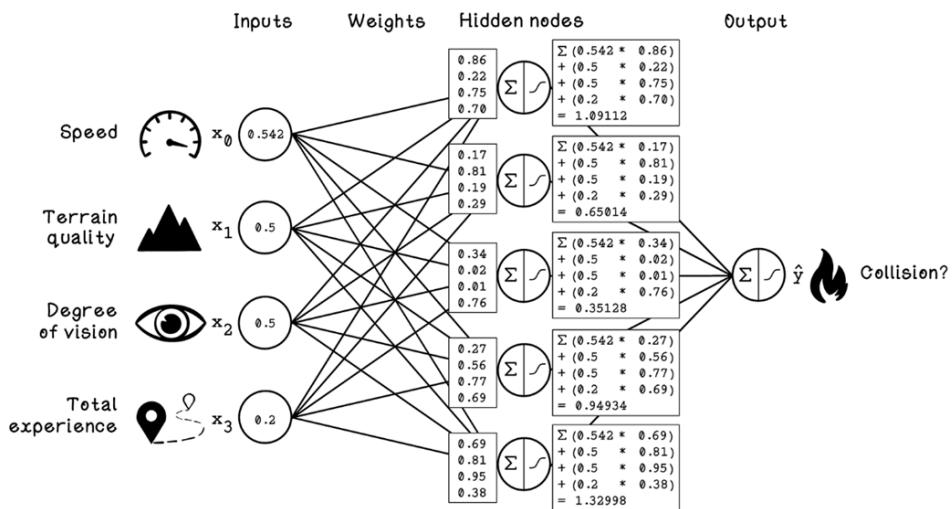


Figure 9.21 Example initial weights for an ANN

The next step is forward propagation (figure 9.22). The key change is checking the difference between the obtained prediction and the actual class.

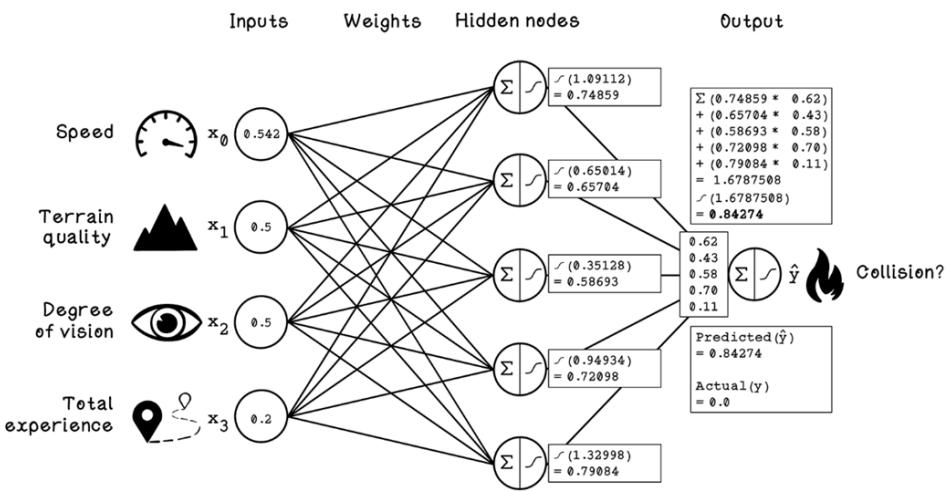
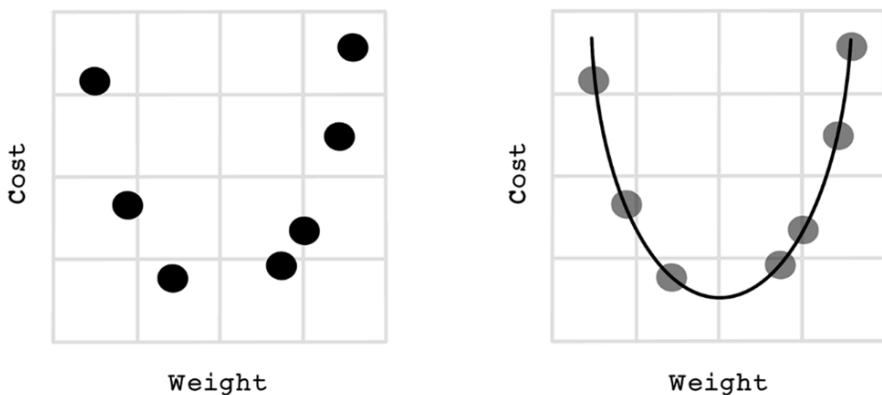


Figure 9.22 Example of forward propagation with randomly initialized weights

By comparing the predicted result with the actual class, we can calculate a cost. The cost function that we will use is simple: subtract the predicted output from the actual output. In this example, 0.84274 is subtracted from 0.0, and the cost is -0.84274. This result indicates how incorrect the prediction was and can be used to adjust the weights in the ANN. Weights in the ANN are adjusted slightly every time a cost is calculated. This happens thousands of times using training data to determine the optimal weights for the ANN to make accurate predictions. Note that training too long on the same set of data can lead to overfitting, as described in chapter 8.

Here is where some potentially unfamiliar math comes into play: the *Chain Rule*. Before we use the Chain Rule, let's gain some intuition about what the weights mean and how adjusting them improves the ANN's performance.

If we plot possible weights against their respective cost on a graph, we find some function that represents the possible weights. Some points on the function yield a lower cost, and other points yield a higher cost. We are seeking points that minimize cost (figure 9.23).



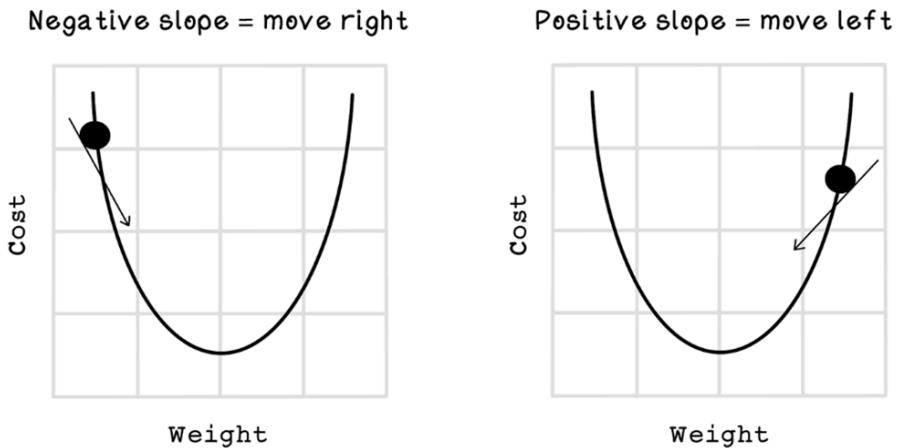
**Figure 9.23 Weight versus cost plotted**

A useful tool from the field of calculus, called *gradient descent*, can help us move the weight closer to the minimum value by finding the derivative. The *derivative* is important because it measures the sensitivity to change for that function.

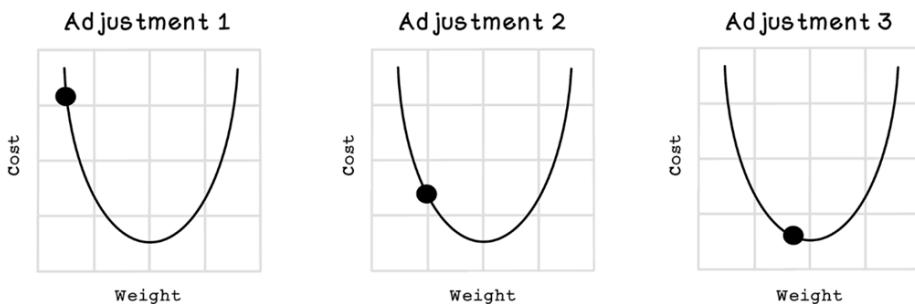
Imagine you are standing on a hilly landscape (the cost function) in the pitch dark. You want to get to the bottom of the valley (minimum cost), but you can't see where the bottom is. All you can feel is the angle of the ground under your feet. The derivative is the slope: It tells you how steep the ground is right where you are standing.

- If the slope tilts up (positive derivative), you know that moving forward will take you higher (more error), so you should move backward.
- If the slope tilts down (negative derivative), you know that moving forward will take you lower (less error), so you should move forward.
- If the slope is perfectly flat (derivative is zero), you have likely reached the bottom.

Other examples are, velocity might be the derivative of an object's **position** with respect to **time**; and acceleration is the derivative of the object's **velocity** with respect to **time**. Derivatives can find the slope at a specific point in the function. Gradient descent uses the knowledge of the slope to determine which way to move and by how much. Figures 9.24 and 9.25 describe how the derivatives and slope indicate the direction of the minimums.



**Figure 9.24 Derivatives' slopes and direction of minimums**



**Figure 9.25 Example of adjusting a weight by using gradient descent**

When we look at one weight in isolation, it may seem trivial to find a value that minimizes the cost, but many weights being balanced affect the cost of the overall network. Some weights may be close to their optimal points in reducing cost, and others may not, even though the ANN performs well.

Because many functions comprise the ANN, we can use the Chain Rule. The Chain Rule is a theorem that calculates the derivative of a composite function. A composite function uses a function  $g$  as the parameter for a function  $f$  to produce a function  $h$ , essentially using a function as a parameter of another function.

Figure 9.26 illustrates the use of the Chain Rule in calculating the update value for weights in the different layers of the ANN.

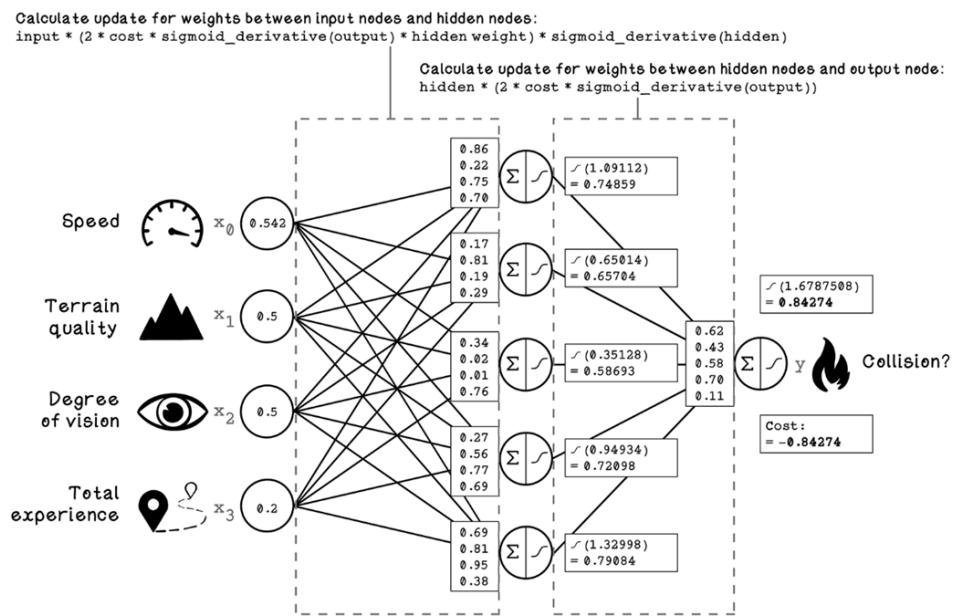


Figure 9.26 Formula for calculating weight updates with the Chain Rule

We can calculate the weight update by plugging the respective values into the formula described. The calculations look complicated, but pay attention to the variables being used and their role in the ANN. Although the formula looks complex, it uses the values that we have already calculated (figure 9.27).

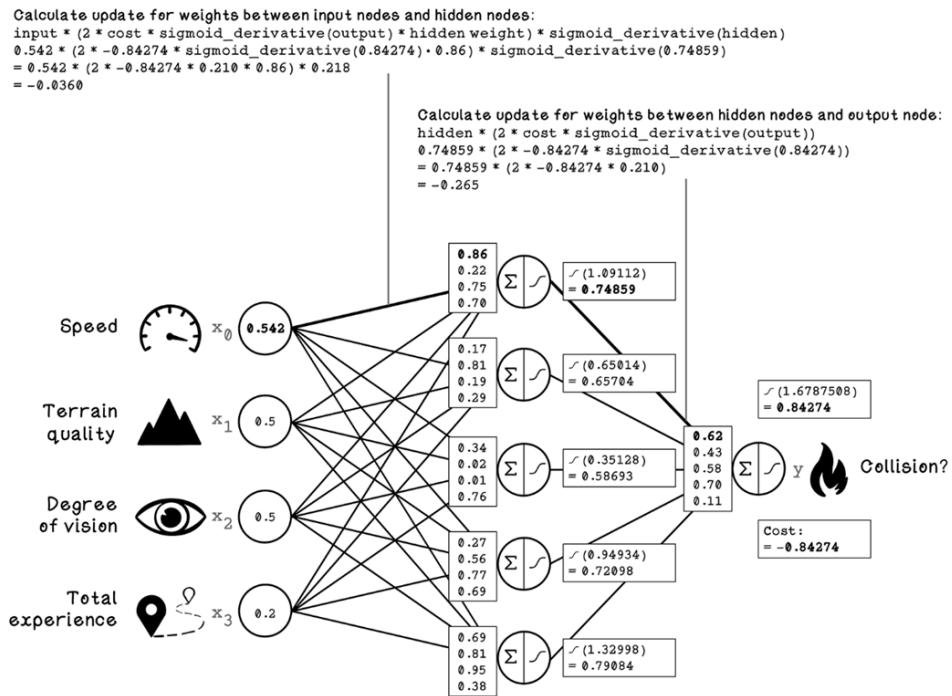


Figure 9.27 Weight-update calculation with the Chain Rule

Here's a closer look at the calculations used in figure 9.27:

**Calculate update for weights between hidden nodes and output node:**  
 $\text{hidden} * (2 * \text{cost} * \text{sigmoid\_derivative}(\text{output}))$

$$0.74859 * (2 * -0.84274 * \text{sigmoid\_derivative}(0.84274))$$

$$= 0.74859 * (2 * -0.84274 * 0.210)$$

$$= -0.265$$

**Calculate update for weights between input nodes and hidden nodes:**  
 $\text{input} * (2 * \text{cost} * \text{sigmoid\_derivative}(\text{output}) * \text{hidden weight}) * \text{sigmoid\_derivative}(\text{hidden})$

$$0.542 * (2 * -0.84274 * \text{sigmoid\_derivative}(0.84274) * 0.86) * \text{sigmoid\_derivative}(0.74859)$$

$$= 0.542 * (2 * -0.84274 * 0.210 * 0.86) * 0.218$$

$$= -0.0360$$

Now that the update values are calculated, we can apply the results to the weights in the ANN by adding the update value to the respective weights. Figure 9.28 depicts the application of the weight-update results to the weights in the different layers.

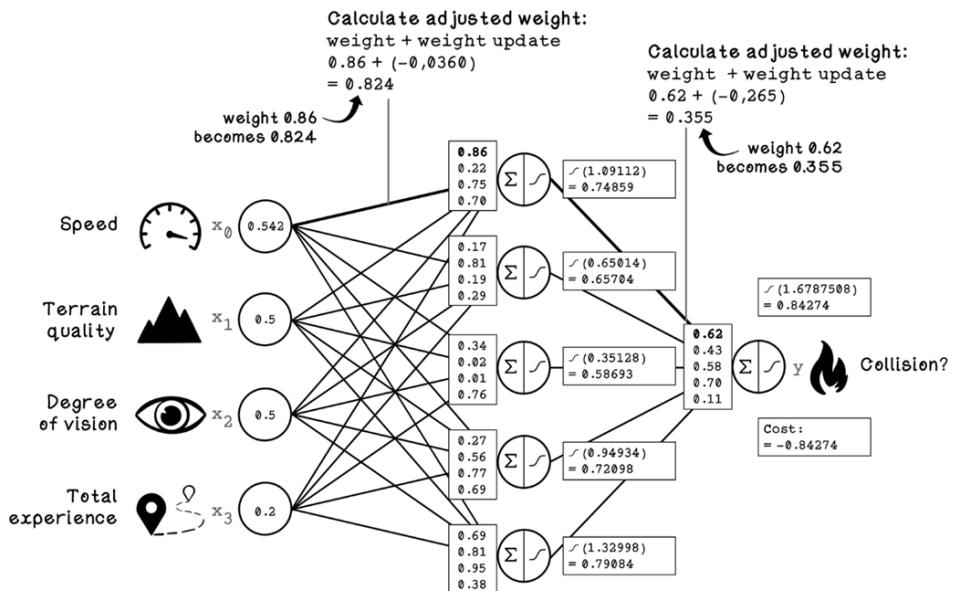
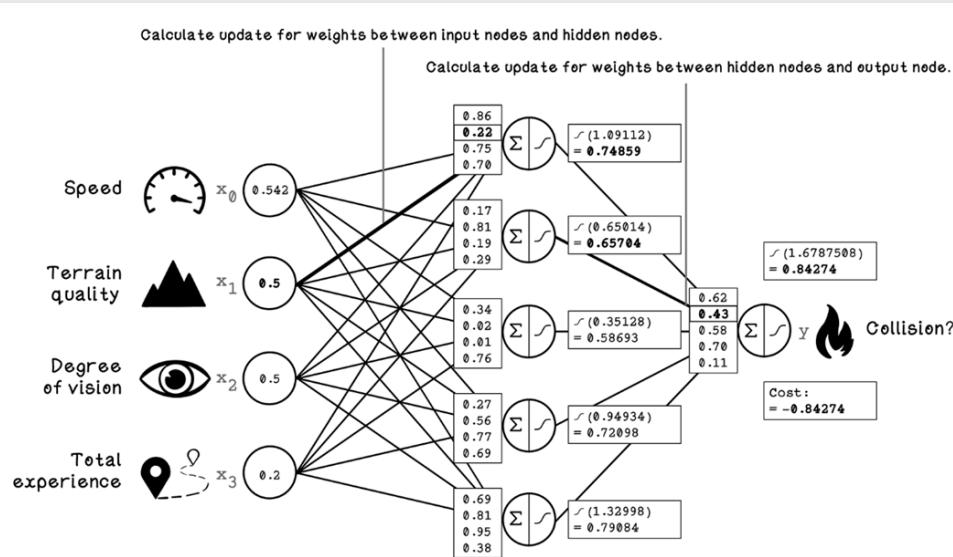


Figure 9.28 Example of the final weight-update for the ANN

#### EXERCISE: CALCULATE THE NEW WEIGHTS FOR THE HIGHLIGHTED WEIGHTS

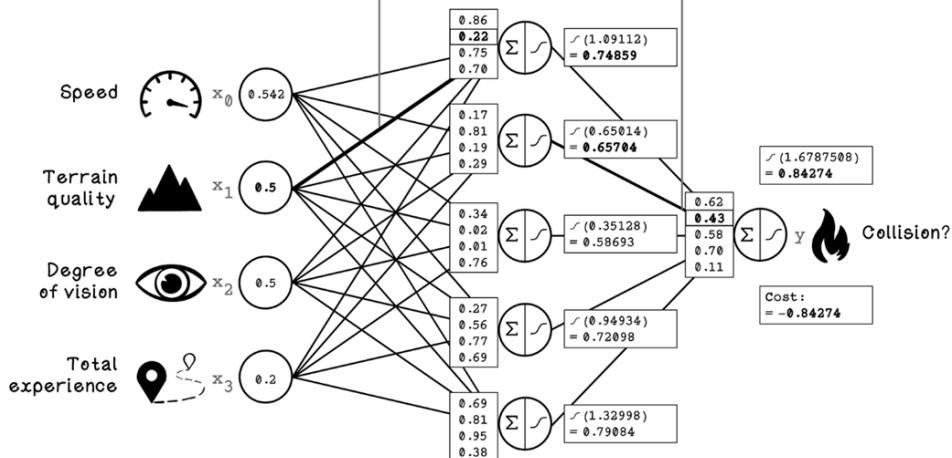


## SOLUTION: CALCULATE THE NEW WEIGHTS FOR THE HIGHLIGHTED WEIGHTS

```

Calculate update for weights between input nodes and hidden nodes:
input * (2 * cost * sigmoid_derivative(output) * hidden weight) * sigmoid_derivative(hidden)
0.5 * (2 * -0.84274 * sigmoid_derivative(0.84274) * 0.22) * sigmoid_derivative(0.74859)
= 0.5 * (2 * -0.84274 * 0.210 * 0.22) * 0.218
= -0.008
weight + weight update
0.22 + (-0.008)
= 0.212

Calculate update for weights between hidden nodes and output node:
hidden * (2 * cost * sigmoid_derivative(output))
0.65704 * (2 * -0.84274 * sigmoid_derivative(0.84274))
= 0.65704 * (2 * -0.84274 * 0.210)
= -0.233
weight + weight update
0.43 + (-0.233)
= 0.197
  
```



The problem that the Chain Rule is solving may remind you of the drone problem example in chapter 7. Particle-swarm optimization is effective for finding optimal values in high-dimensional spaces such as this one, which has 25 weights to optimize. Finding the weights in an ANN is an optimization problem. Gradient descent is not the only way to optimize weights; we can use many approaches, depending on the context and problem being solved.

## PYTHON CODE SAMPLE

The derivative is important in the backpropagation algorithm. The following piece of code revisits the sigmoid function and describes the formula for its derivative, which we need to adjust weights:

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x)) #A

#A Calculates the derivative (slope) of the sigmoid function at point x
```

---

**NOTE** exp is a mathematical constant called Euler's number, also denoted by e, approximately 2.71828.

---

We revisit the neural network class, this time with a backpropagation function that computes the cost, the amount by which weights should be updated by using the Chain Rule, and adds the weight-update results to the existing weights. This process will compute the change for each weight given the cost. Remember that cost is calculated by using the example features, predicted output, and expected output. The difference between the predicted output and expected output is the cost:

```

class NeuralNetwork:
    def __init__(self, features, labels, hidden_node_count):
        num_features = features.shape[1]
        num_samples = features.shape[0]
        num_output = labels.shape[1]

        self.input = features
        self.expected_output = labels
        self.hidden_node_count = hidden_node_count

        self.weights_input = np.random.randn(num_features, hidden_node_count) *
0.01
        self.weights_hidden = np.random.randn(hidden_node_count, num_output) *
0.01

        self.hidden = np.zeros((num_samples, hidden_node_count))
        self.output = np.zeros((num_samples, num_output))

    def forward_propagation(self):
        self.hidden = sigmoid(np.dot(self.input, self.weights_input))
        self.output = sigmoid(np.dot(self.hidden, self.weights_hidden))

    def back_propagation(self):
        # 1. Calculate Cost/Error ( $Y - Y_{\text{hat}}$ )
        error_output = self.expected_output - self.output

        # 2. Calculate Output Layer Gradient (Delta Output)
        d_output = 2 * error_output * (self.output * (1 - self.output)) #A

        # 3. Calculate Update for Weights Hidden -> Output ( $W_{HO}$ )
        d_weights_hidden = self.hidden.T @ d_output #B

        # 4. Propagate Error Back to Hidden Layer
        error_hidden = d_output @ self.weights_hidden.T * (self.hidden * (1 -
self.hidden)) #C

        # 5. Calculate Update for Weights Input -> Hidden ( $W_{IH}$ )
        d_weights_input = self.input.T @ error_hidden #D

        # 6. Apply Updates ( $W = W + dW$ )
        self.weights_hidden += d_weights_hidden
        self.weights_input += d_weights_input

```

---

```
#A 2 * Cost * sigmoid_derivative(Output); Note: We use self.output (the sigmoid output) for the derivative
#B Formula: Hidden_T * Delta_Output
#C Delta_Hidden = (Delta_Output @ W_HO_T) * sigmoid_derivative(Hidden)
#D Formula: Input_T * Delta_Hidden
```

---

**NOTE** The symbol  $\cdot$  implies matrix multiplication.

---

Because we have a class that represents a neural network, functions to scale data, and functions for forward propagation and backpropagation, we can piece this code together to train a neural network.

In the next piece of code, we have a `run_neural_network` function that accepts `epochs` as an input. This function scales the data and creates a new neural network with the scaled data, labels, and number of hidden nodes. Then the function runs `forward_propagation` and `back_propagation` for the specified number of `epochs`:

```
def run_neural_network(epochs, feature_data, label_data, hidden_node_count):

    scaled_feature_data = scale_dataset(
        feature_data, FEATURE_COUNT, FEATURE_MIN, FEATURE_MAX
    )

    nn = NeuralNetwork(
        scaled_feature_data,
        label_data,
        hidden_node_count
    )

    for epoch in range(epochs):
        nn.forward_propagation()
        nn.back_propagation()

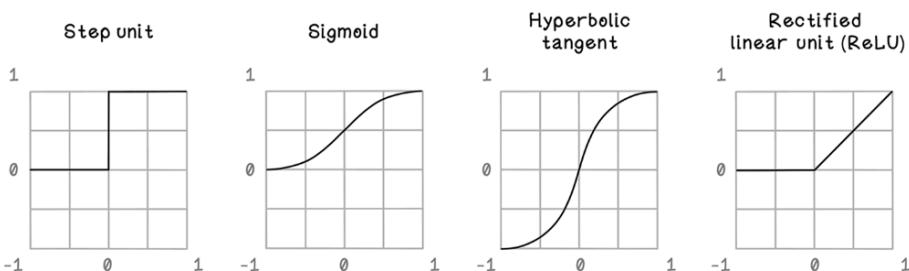
        if epoch % 100 == 0:
            loss = np.mean(np.square(nn.expected_output - nn.output))
            print(f"Epoch {epoch:03d}: Loss = {loss:.6f}")

    return nn
```

You have now learned the complete lifecycle from scaling input data and feeding it through an ANN, to training the ANN using backpropagation by calculating the loss of predictions versus actual outcomes. This is the essence of how large neural networks operate and is the fundamental concept for understanding more sophisticated applications of it like Large Language Models, and Generative Image Models.

## 9.6 Options for activation functions

This section aims to detail the intuition about activation functions and their properties. In the examples of the Perceptron and ANN, we used a sigmoid function as the activation function, which was satisfactory for the examples that we were working with. Activation functions introduce nonlinear properties to the ANN. If we do not use an activation function, the neural network will behave similarly to linear regression as described in chapter 8. Figure 9.29 describes some commonly used activation functions.



**Figure 9.29 Commonly used activation functions**

Different activation functions are useful in different scenarios and have different benefits:

- **Step unit**—The step unit function is used as a binary classifier. Given an input between -1 and 1, it outputs a result of exactly 0 or 1. A binary classifier is not useful for learning from data in a hidden layer, but it can be used in the output layer for binary classification. If we want to know whether something is a cat or a dog, for example, 0 could indicate cat, and 1 could indicate dog.
- **Sigmoid**—The sigmoid function results in an S curve between 0 and 1, given an input between -1 and 1. Because the sigmoid function allows changes in  $x$  to result in small changes in  $y$ , it allows for learning and solving nonlinear problems. The problem sometimes experienced with the sigmoid function is that as values approach the extremes, derivative changes become tiny, resulting in poor learning. As mentioned earlier, this problem is known as the vanishing gradient problem.

- *Hyperbolic tangent*—The hyperbolic tangent function is similar to the sigmoid function, but it results in values between -1 and 1. The benefit is that the hyperbolic tangent has steeper derivatives, which allows for faster learning. The vanishing gradient problem is also a problem at the extremes for this function, as with the sigmoid function.
- *Rectified linear unit (ReLU)*—The ReLU function results in 0 for input values between -1 and 0, and results in linearly increasing values between 0 and 1. In a large ANN with many neurons using the sigmoid or hyperbolic tangent function, all neurons activate all the time (except when they result in 0), resulting in lots of computation and many values being adjusted finely to find solutions. The ReLU function allows some neurons to not activate, which reduces computation and may find solutions faster.

The next section touches on some considerations for designing an ANN.

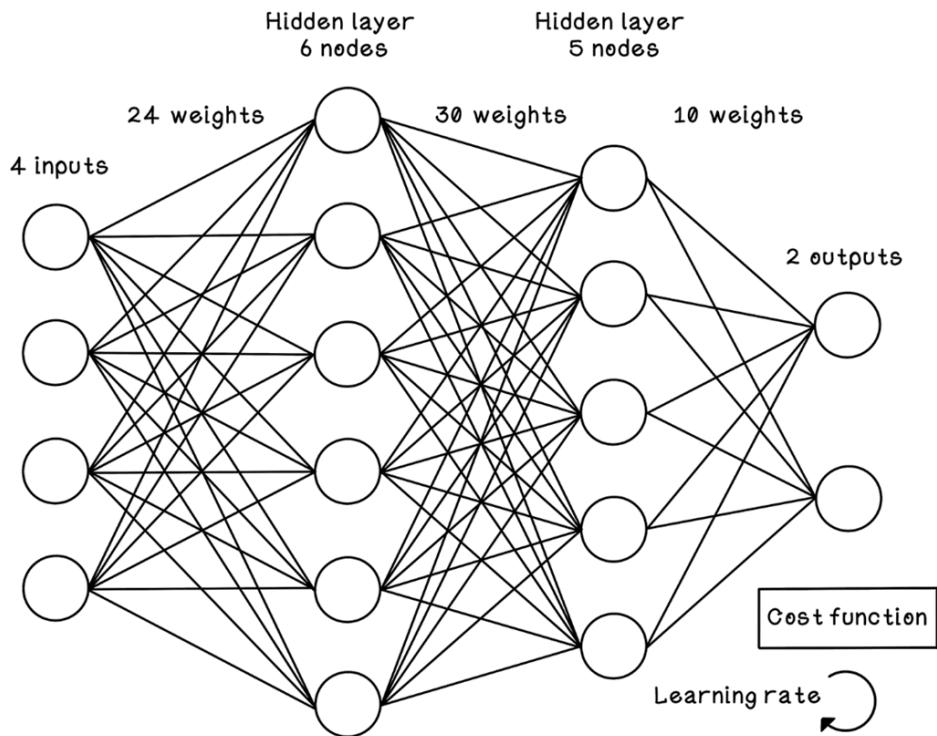
## 9.7 Designing artificial neural networks

Designing ANNs can be experimental and is dependent on the problem that is being solved. The architecture and configuration of an ANN usually change through trial and error as we attempt to improve the performance of the predictions, or based on well-researched architectures that have been proven to work. This section lists the parameters of the architecture that we can change to improve performance or address different problems. Figure 9.30 illustrates an artificial neural network with a different configuration to the one seen throughout this chapter. The most notable difference is the introduction of a new hidden layer and the network now has two outputs.

---

**NOTE** As in most scientific or engineering problems, the answer to “What is the ideal ANN design?” is often “It depends.” Configuring ANNs requires a deep understanding of the data and the problem being solved.

---



**Figure 9.30 An example of a multilayer ANN with more than one output**

### 9.7.1 Inputs and outputs

The inputs and outputs of an ANN are the fundamental parameters for use of the network. After an ANN model has been trained, the trained ANN model will potentially be used in different contexts and systems, and by different people. The inputs and outputs define the interface of the network. Throughout this chapter, we saw an example of an ANN with four inputs describing the features of a driving scenario and one output describing the likelihood of a collision. We may have a problem when the inputs and outputs mean different things, however. If we have a 16- by 16-pixel image that represents a handwritten digit, for example, we could use the pixel values as inputs and the digit they represent as the output. The input would consist of 256 nodes representing the pixel values, and the output would consist of 10 nodes representing 0 to 9, with each result indicating the probability that the image is the respective digit.

### 9.7.2 Hidden layers and nodes

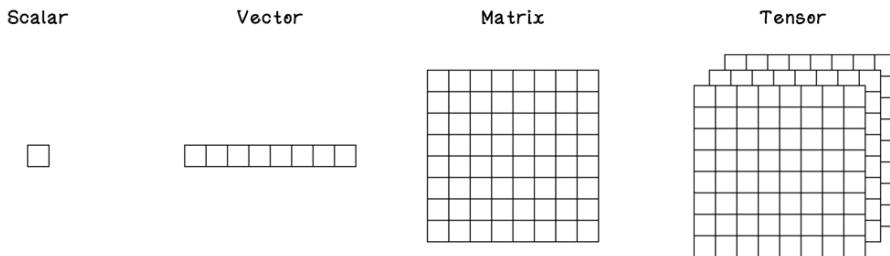
An ANN can consist of multiple hidden layers with varying numbers of nodes in each layer. Adding more hidden layers allows us to solve problems with higher dimensions and more complexity in the classification discrimination line. Think of additional layers as finding more high-level abstractions of the relationships among the outputs of the layer before it. In the example in figure 9.8, a simple straight line classified data accurately. Sometimes, the line is nonlinear but fairly simple. But what happens when the line is a more-complex function with many curves potentially across many dimensions (which we can't even visualize)? Adding more layers allows these complex classification functions to be found. The selection of the number of layers and nodes in an ANN usually comes down to experimentation and iterative improvement. Over time, we may gain intuition about suitable configurations, based on experiencing similar problems and solving them with similar configurations.

## LAYERS ARE TENSORS

While it's common to describe artificial neural networks (ANNs) using nodes (representing neurons) and edges (representing connections), high performing implementations are designed for computational efficiency and rely on tensors instead. A tensor is a flexible, multi-dimensional data structure that generalizes simpler mathematical objects. To understand tensors, consider the following:

- *Scalar*: A single numerical value (0-dimensional tensor), like, 5.
- *Vector*: A 1D array of numbers (1-dimensional tensor), like, [2.1, -0.3, 4.7].
- *Matrix*: A 2D grid of numbers (2-dimensional tensor), like, a table with rows and columns.
- *Tensor*: A generalization of all of the above, a tensor can have 3 or more dimensions. For example:
  - A 3D tensor might represent a stack of matrices (e.g., color image with height × width × RGB channels).
  - A 4D tensor might represent a batch of images (batch size × height × width × channels).

In deep learning frameworks like TensorFlow or PyTorch, entire ANN layers are treated as tensor operations. Inputs, weights, and outputs are all represented as tensors, allowing for highly optimized, parallel computation. Instead of simulating each node and edge individually, the network performs efficient bulk transformations on entire layers using tensor algebra—making large-scale training and inference practical leveraging the power of GPUs that are optimized for operations on tensors. We will learn how this math works shortly.



**Figure 9.31 Comparison of tensor data structures**

### 9.7.3 Weights

Weight initialization is important because it establishes a starting point from which the weight will be adjusted slightly over many iterations. Weights that are initialized to be too small lead to the *vanishing gradient problem* described earlier, and weights that are initialized to be too large lead to another problem, the *exploding gradient problem*—in which weights move erratically around the desired result.

Various weight-initialization schemes exist, each with its own pros and cons. A rule of thumb is to ensure that the mean of the activation results in a layer is 0—the mean of all results of the hidden nodes in a layer. Also, the variance of the activation results should be the same: the variability of the results from each hidden node should be consistent over several iterations.

### THE “GOLDILOCKS” PRINCIPLE OF INITIALIZATION

Why do we care about the mean and variance? Imagine a game of “Telephone” played by 100 layers of neurons.

- If the weights are too small: The whisper gets quieter and quieter until the last person hears nothing. The network stops learning because the signal died. (This is the *vanishing gradient*.)
- If the weights are too large: The whisper turns into shouting. By the end, the signal is a distorted, screaming mess. (This is the *exploding gradient*.)

The solution is smart initialization. To fix this, we don't just pick random numbers. We use smart formulas (for example, Xavier / Glorot Initialization) to pick weights that are “just right”. The goal is simple: to ensure that the variance (the spread of the signal strength) remains the same from the first layer to the last. If the input signal has a strength of “1,” we want the output signal to also roughly have a strength of “1,” preventing the signal from exploding or vanishing as it travels deep into the network.

### 9.7.4 Bias

We can use bias in an ANN by adding a value to the weighted sum of the input nodes or other layers in the network. A bias can shift the activation value of the activation function. A bias provides flexibility in an ANN and shifts the activation function left or right.

A simple way to understand bias is to imagine a line that always passes through 0,0 on a plane; we can influence this line to pass through a different intercept by adding +1 to a variable. This value will be based on the problem to be solved.

### 9.7.5 Activation functions

Earlier we covered the common activation functions used in ANNs. A key rule of thumb is to ensure that all nodes on the same layer use the same activation function. In multilayer ANNs, different layers may use different activation functions based on the problem to be solved. A network that determines whether loans should be granted, for example, might use the sigmoid function in the hidden layers to determine probabilities and a step function in the output to get a clear 0 or 1 decision.

### 9.7.6 Cost function

In our example, we used a simple cost function where the predicted output is subtracted from the actual expected output, but many different cost functions can be used. Cost functions influence the ANN greatly, and using the correct function for the problem and dataset at hand is important because it describes the goal for the ANN. One of the most common cost functions is *mean square error*, which is similar to the function used in the machine learning chapter (chapter 8). But cost functions must be selected based on understanding of the training data, size of the training data, and desired precision and recall measurements. As we experiment more, we should look into the cost function options.

### 9.7.7 Learning rate

Finally, the learning rate of the ANN describes how dramatically weights are adjusted during backpropagation. A slow learning rate may result in a long training process because weights are updated by tiny amounts each time, and a high learning rate might result in dramatic changes in the weights, making for a chaotic training process. One solution is to start with a fixed learning rate and to adjust that rate if the training stagnates and doesn't improve the cost. This process, which would be repeated through the training cycle, requires some experimentation. Stochastic gradient descent is a useful tweak to the optimizer that combats these problems. It works similarly to gradient descent but allows weights to jump out of local minimums to explore better solutions.

Standard ANNs such as the one described in this chapter are useful for solving nonlinear classification problems. If we are trying to categorize examples based on many features, this ANN style is likely to be a good option.

That said, an ANN is not a silver bullet and shouldn't be the go-to algorithm for anything. Simpler, traditional machine learning algorithms described in chapter 8 often perform better in many common use cases. Remember the machine learning life cycle. You may want to try several machine learning models during your iterations while seeking improvement.

## 9.8 Expressing ANNs mathematically

We have looked at ANNs as biological neurons, visualized as graphs, and as code. But there is another language that ANNs speak: *linear algebra*.

When you read AI research papers, you won't usually see code or diagrams of neurons; you will see mathematical equations. Let's translate what we've learned into this universal language.

### 9.8.1 The weighted sum as a dot product

Recall that a single neuron takes inputs, multiplies them by weights, sums them up, and adds a bias. Mathematically, if we have inputs  $x$ , weights  $w$ , and bias  $b$  the weighted sum  $z$  is:

$$z = x_1w_1 + x_2w_2 + x_3w_3 \dots + b$$

Writing this out for every neuron is tedious. In linear algebra, we treat the inputs as a vector (a list of numbers) and the weights as a vector. This allows us to use the dot product. Dot product means multiply pairwise and sum the results. For example, given:

Inputs ( $X$ ): [0.542, 0.5, 0.5, 0.2]

Weights ( $W$ ): [3.35, -5.82, 0.85, -4.35]

The dot product is:  $(0.542 \times 3.35) + (0.5 \times -5.82) + (0.5 \times 0.85) + (0.2 \times -4.35)$

The following single equation replaces the loop we would otherwise have to write in code:

$$z = X \cdot W + b$$

### 9.8.2 The hidden layer as matrix multiplication

In our car collision example, we didn't just have one neuron; we had a layer of 5 hidden nodes. We don't want to calculate five separate dot products. Instead, we bundle the weights of all 5 neurons into a matrix (a 2D grid).

- Inputs ( $X$ ): A vector of size 4 (Speed, Terrain, Vision, Experience).
- Weights ( $W$ ): A matrix of size  $4 \times 5$  (4 inputs connecting to 5 hidden nodes).

Now, the entire layer's calculation happens in one operation:

$$Z_{\text{hidden}} = X \cdot W_{\text{input}}$$

This matrix multiplication calculates the weighted sums for all 5 hidden neurons simultaneously.

### 9.8.3 Adding the activation function

Once we have the weighted sums ( $Z$ ), we apply the activation function (like the Sigmoid).

We denote the activation function with  $\sigma$ —the Greek letter sigma.

The output of the hidden layer ( $H$ ) becomes:

$$H = \sigma(X \cdot W_{\text{input}})$$

### 9.8.4 The output layer

The output of the hidden layer ( $H$ ) becomes the input for the next layer. This is where the concept of composite functions (functions inside functions) appears.

To get the final prediction ( $\hat{y}$ —pronounced “y hat”), we multiply the hidden layer results by the second set of weights and apply the activation function again.

$$\hat{y} = \sigma(H \cdot W_{\text{hidden}})$$

You would pronounce this as “y hat equals sigma of H dot W hidden”.

### 9.8.5 The final neural network equation

If we substitute  $H$  with the equation from the activation function step, we can express the entire neural network in a single mathematical line. This is the “sentence” that describes the architecture we built:

$$\hat{y} = \sigma(\sigma(X \cdot W_{\text{input}}) \cdot W_{\text{hidden}})$$

This elegant equation tells the full story of forward propagation:

1. Take inputs  $X$ .
2. Multiply by first weights  $W_{\text{hidden}}$ .
3. Apply activation  $\sigma$ .
4. Multiply that result by second weights  $W_{\text{hidden}}$ .
5. Apply activation  $\sigma$  again to get the final prediction  $\hat{y}$ .

### 9.8.6 The cost function

Finally, how do we express the “error”? A commonly used method is *Mean Squared Error*: a type of cost function used to measure how well—or how poorly—the neural network is performing. It calculates the average of the squares of the errors. The “error” is simply the difference between what the network predicted and what the actual expected value was (*Actual – Predicted*). Mathematical notation for the cost function ( $J$ ) using *mean squared error* looks like this:

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

$y$ : The actual value (did the collision happen?).

$\hat{y}$ : The predicted value (probability of collision).

### 9.8.7 Expressing backpropagation mathematically

If forward propagation is about composing functions (wrapping inputs inside layers), backpropagation is about decomposing them using the *Chain Rule*. Our goal is simple: We want to know how much to adjust a specific weight ( $W$ ) to decrease the Cost ( $J$ ). In calculus terms, we are looking for the *partial derivative* ( $\partial$ —the Greek letter del) of the Cost with respect to the *Weight*.

$$\frac{\partial J}{\partial W}$$

If this value is positive, increasing the weight increases error (so we should decrease the weight). If it’s negative, increasing the weight decreases error (so we should increase it).

## THE CHAIN RULE – THE DOMINO EFFECT

We cannot calculate  $\frac{\partial J}{\partial W}$  directly because the weight ( $W$ ) doesn’t touch the cost ( $J$ ) directly. It is buried deep inside the network.  $W$  affects the *hidden layer*, which affects the *predicted output*, which affects the *cost*. To find the relationship, we multiply the derivatives of these steps together. This is the *Chain Rule*:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial Output} \cdot \frac{\partial Output}{\partial Hidden} \cdot \frac{\partial Hidden}{\partial W}$$

## CALCULATING THE GRADIENTS – THE BACKWARD PASS

We calculate the gradients starting from the end and moving backward.

**Step 1:** The Output Error ( $\delta_{output}$ —the Greek lowercase letter, delta). First, we see how “wrong” our prediction was and how sensitive the activation function is at that point.

$$\delta_{output} = (\hat{y} - y) \cdot \sigma'(Z_{output})$$

- $(\hat{y} - y)$ : The difference between prediction and reality (the raw error).
- $\sigma'(Z_{output})$ : The derivative (slope) of the activation function (e.g., Sigmoid derivative).

The symbol  $\sigma'$  is pronounced “sigma prime”. In calculus, the little tick mark ( $'$ ) indicates the derivative of a function. So, while  $\sigma$  gives us the activation value (e.g., 0.7),  $\sigma'$  gives us the **slope** of the curve at that point (how steep it is).

**Step 2:** The Output Weights Gradient ( $\nabla W_{hidden}$ —the Greek letter, Nabla). Now we find how much the weights connecting the hidden layer to the output layer contributed to that error.

A note on notation: In the Chain Rule above, we used  $\frac{\partial J}{\partial w}$  to describe the change for a single weight. In the steps below, we use the symbol  $\nabla W$ . This represents the gradient, which is simply a matrix containing the partial derivatives for every weight in the layer at once.

$$\nabla W_{hidden} = H^T \cdot \delta_{output}$$

- We multiply the transpose of the hidden layer values ( $H^T$ ) by the output error.

What does it mean to transpose? The  $T$  stands for *Transpose*. It is a simple instruction to “flip” the matrix over its diagonal. Imagine turning a table on its side. The rows become columns, and the columns become rows. To perform matrix multiplication, the inner dimensions must match (e.g., a  $1 \times 5$  matrix cannot multiply a  $1 \times 1$  matrix directly). Transposing the matrix flips its dimensions (becoming  $5 \times 1$ ), allowing the multiplication operation to work.

**Step 3:** Propagating error to the hidden layer ( $\delta_{hidden}$ ). This is the “relay race” moment. We pass the error backward through the weights to find out how much the hidden nodes were responsible for the error.

$$\delta_{hidden} = (\delta_{output} \cdot W^T_{hidden}) \cdot \sigma'(Z_{hidden})$$

- We take the error from the future ( $\delta_{output}$ ), pull it back through the weights ( $W^T$ ), and multiply by the derivative of the hidden activation.

**Step 4:** The input weight gradient ( $\nabla W_{\text{input}}$ ). Finally, we calculate the gradient for the very first set of weights.

$$\nabla W_{\text{input}} = X^T \cdot \delta_{\text{hidden}}$$

## THE WEIGHT UPDATE – GRADIENT DESCENT

Now that we have the gradients ( $\nabla W$ ), which represent the “slope” of the cost function, we simply move in the opposite direction.

$$\nabla W_{\text{new}} = W_{\text{old}} - \alpha \cdot \nabla W$$

- $\alpha$  (the Greek letter, alpha): Is the learning rate. This controls how big of a step we take.
- $\nabla W$ : Is the gradient we calculated in Steps 2 and 4.

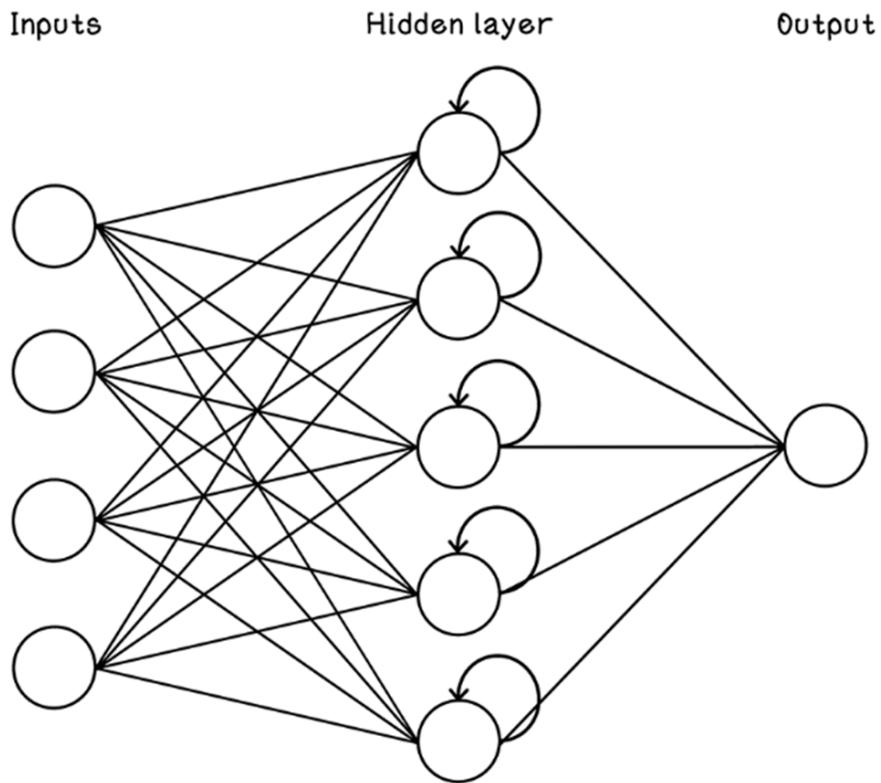
With that, you’ve unlocked the ability to see ANNs not just as code or diagrams, but as mathematical equations. Don’t worry if the notation feels dense at first; the more you look at these types of equations, the more familiar the symbols will become.

## 9.9 Artificial neural network types and use cases

ANNs are versatile and can be designed to address different problems. Specific architectural styles of ANNs are useful for solving certain problems. Think of an ANN architectural style as being the fundamental configuration of the network. The examples in this section highlight different configurations.

### 9.9.1 Recurrent neural network

Whereas standard ANNs accept a fixed number of inputs, *recurrent neural networks* (RNNs) accept a sequence of inputs with no predetermined length. These inputs are like spoken sentences. RNNs have a concept of memory consisting of hidden layers that represent time; this concept allows the network to retain information about the relationships among the sequences of inputs. When we are training a RNN, the weights in the hidden layers throughout time are also influenced by backpropagation; multiple weights represent the same weight at different points in time (figure 9.32).



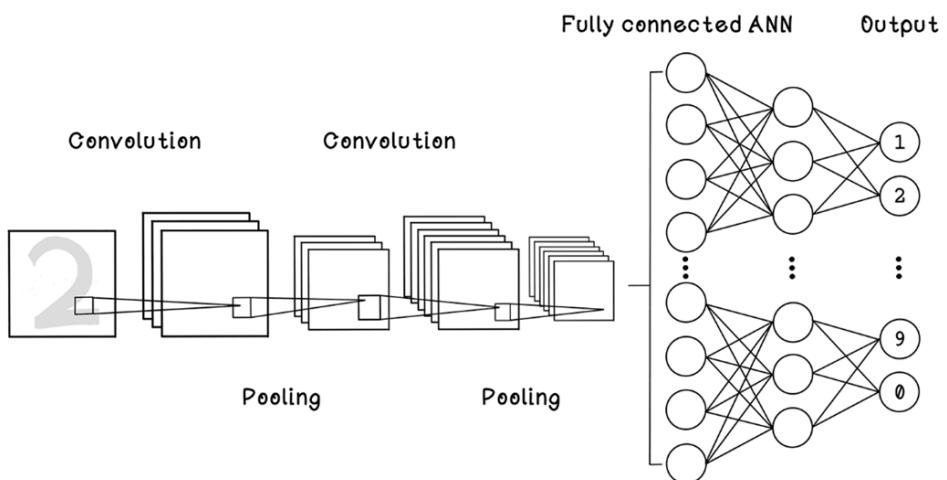
**Figure 9.32 Simple example of a RNN**

RNNs are useful in applications pertaining to speech and text recognition and prediction. Related use cases include autocompletion of sentences in messaging applications, translation of spoken language to text, and translation between spoken languages.

Although RNNs perform decently, the Transformer architecture has become the state-of-the-art option for tasks like text generation and Large Language models. We will explore Transformers further in the LLM chapter.

### 9.9.2 Convolutional neural network

*Convolutional neural networks* (CNNs) were originally designed for image recognition. These networks can be used to find the relationships among different objects and unique areas within images. In *image recognition*, convolution operates on a single pixel and its neighbors within a certain radius. This technique is traditionally used for edge detection, image sharpening, and image blurring. CNNs use convolution and pooling to find relationships among pixels in an image. *Convolution* finds features in images, and *pooling* downsamples the “patterns” by summarizing features, allowing unique signatures in images to be encoded concisely through learning from multiple images (figure 9.31).



**Figure 9.33 Simple example of a CNN**

CNNs are used for image classification. If you’ve ever searched for an image online, you have likely interacted indirectly with a CNN. These networks are also useful for optical character recognition for extracting text data from an image. CNNs have been used in the medical industry for applications that detect anomalies and medical conditions via X-rays and other body scans.

### 9.9.3 Generative adversarial network

A *generative adversarial network* (GAN) consists of a generator network and a discriminator network. For example, the *generator* creates a potential solution such as an image or a landscape, and a *discriminator* uses real images of landscapes to determine the realism or correctness of the generated landscape. The error or cost is fed back into the network to further improve its ability to generate convincing landscapes and determine their correctness. The term *adversarial* is key, as we saw with game trees in chapter 3. These two components are competing to be better at what they do and, through that competition, generate incrementally better solutions (figure 9.33).

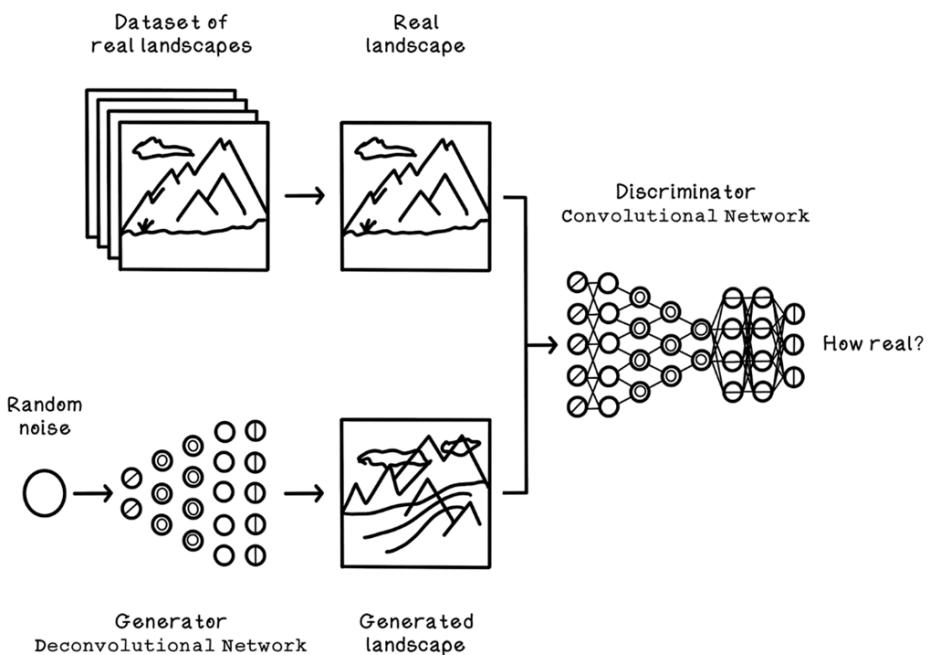


Figure 9.34 Simple example of a GAN

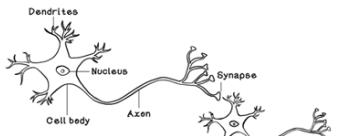
GANs are used to generate convincing fake videos (also known as deepfakes) of famous people, which raises concern about the authenticity of information in the media. GANs also have useful applications such as overlaying hairstyles on people's faces. GANs have been used to generate 3D objects from 2D images, such as generating a 3D chair from a 2D picture. This use case may seem to be unimportant, but the network is accurately estimating and creating information from a source that is incomplete. It is a huge step in the advancement of AI and technology in general.

Although GANs have been useful, we will explore how CNNs and the U-Net architecture have become the “standard” for modern image generation in the Generative Image Models chapter.

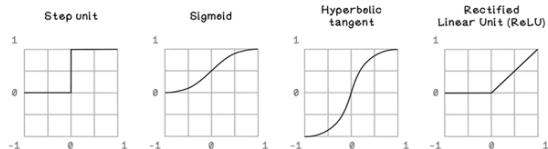
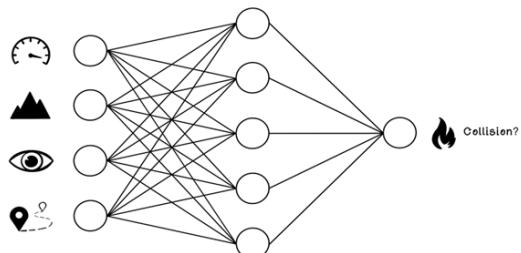
This chapter aimed to tie together the concepts of machine learning with the somewhat-mysterious world of ANNs. For further learning about ANNs and deep learning, try *Grokking Deep Learning* (Manning Publications); and for a practical guide to a framework for building ANNs, see *Deep Learning with Python* (Manning Publications).

## 9.10 Summary of artificial neural networks

Artificial neural networks are inspired by the brain and can be seen as just another ML model

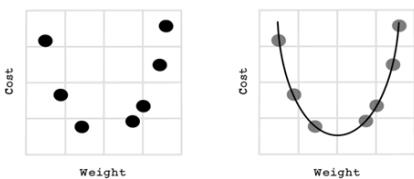
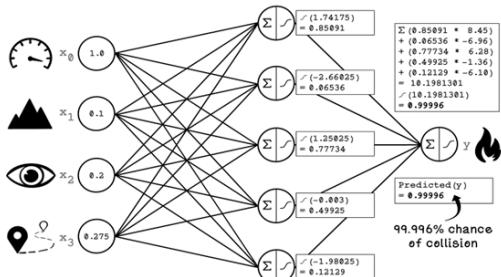


ANNS are based on the idea of a Perceptron (single neuron) at scale



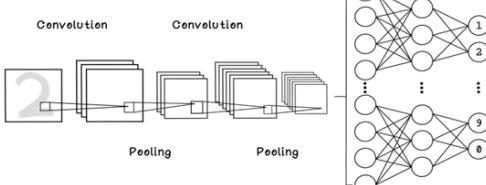
Activation functions help solve nonlinear problems

Forward propagation is used to feed data into an ANN to make predictions



Gradient descent optimization is a popular way to train ANNs

ANNS are flexible and can be adapted to solve many different problems



# 10 Reinforcement learning

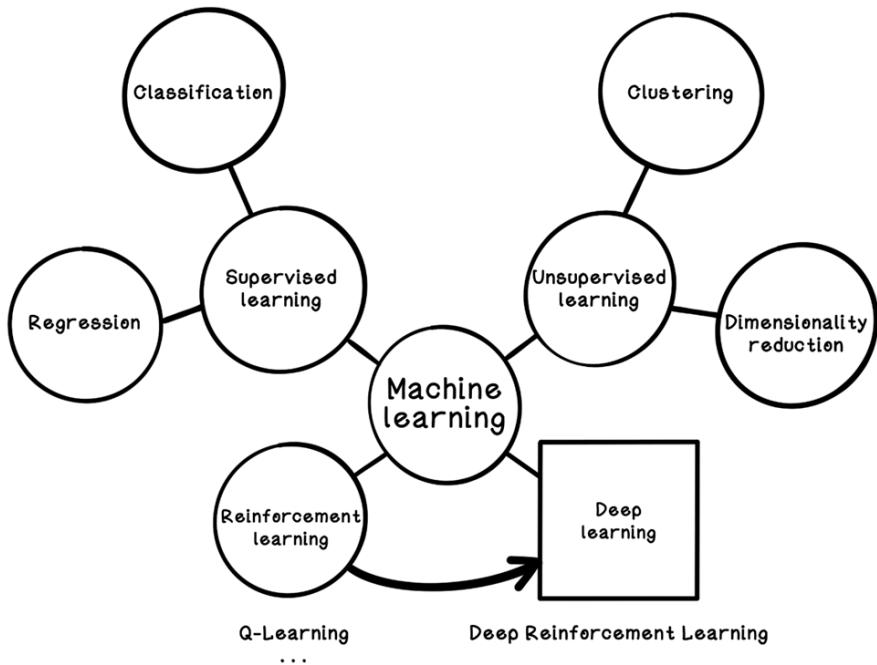
## This chapter covers

- Understanding the inspiration for reinforcement learning
- Identifying problems to solve with reinforcement learning
- Designing and implementing a reinforcement learning algorithm
- Understanding reinforcement learning approaches

### 10.1 What is reinforcement learning?

*Reinforcement learning* (RL) is an area of machine learning inspired by behavioral psychology. The concept of reinforcement learning is based on cumulative rewards or penalties for the actions that are taken by an agent in a dynamic environment. Think about a young dog growing up. The dog is the agent in an environment that is our home. When we want the dog to sit, we might simply say, "Sit." The dog doesn't understand English, so we might nudge it by lightly pushing down on its back. After the dog sits, we pet it or give it a treat - this is a welcomed reward. We need to repeat this many times, but after some time, we have positively reinforced the idea of sitting for the dog. The trigger in the environment is saying "Sit"; the behavior learned is sitting; and the reward is pets or treats.

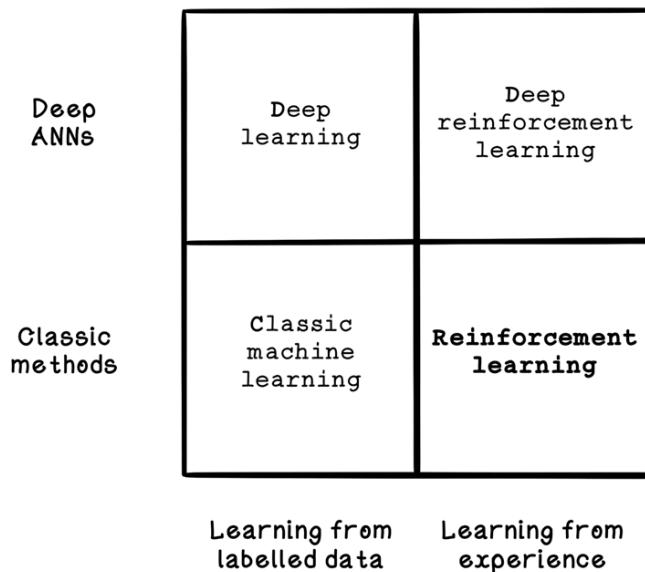
Reinforcement learning is another approach to machine learning alongside *supervised learning* and *unsupervised learning*. Where supervised learning uses labeled data to make predictions and classifications, and unsupervised learning uses unlabeled data to find clusters and trends, reinforcement learning uses feedback from actions performed to learn what actions or sequence of actions are more beneficial in different scenarios toward an ultimate goal. Reinforcement learning is useful when you know what the goal is but don't know what actions are reasonable to achieve it. Figure 10.1 shows the map of machine learning concepts and how reinforcement learning fits in.



**Figure 10.1** How reinforcement learning fits into machine learning

Reinforcement learning can be achieved through classical techniques or deep learning involving artificial neural networks. Depending on the problem being solved, either approach may work better.

Figure 10.2 illustrates when different machine learning approaches may be used. We will be exploring reinforcement learning through classical methods in this chapter.



**Figure 10.2 Categorization of machine learning, deep learning, and reinforcement learning**

### 10.1.1 The inspiration for reinforcement learning

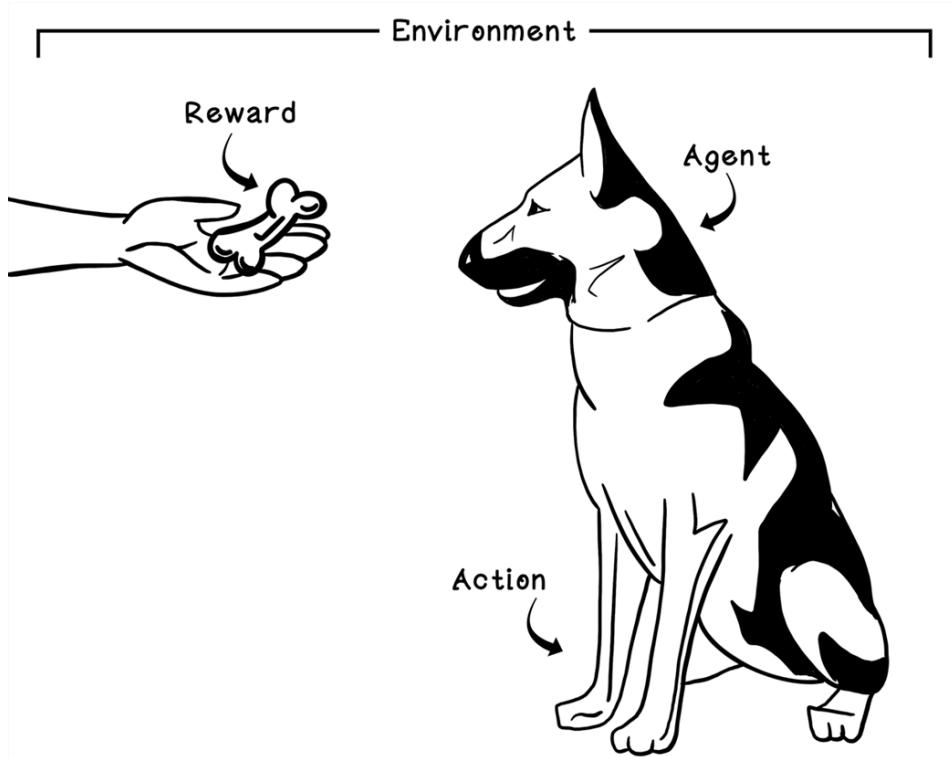
Reinforcement learning in machines is derived from behavioral psychology, a field that is interested in the behavior of humans and other animals. Behavioral psychology usually explains behavior by a reflex action, or something learned in the individual's history. The latter includes exploring reinforcement through rewards or punishments as motivators for behaviors, and aspects of the individual's environment that contribute to the behavior.

Trial and error is one of the most common ways that most evolved animals learn what is beneficial to them and what is not. Trial and error involves trying something, potentially failing at it, and trying something different until you succeed. This process may happen many times before a desired outcome is obtained, and it's largely driven by some reward.

This behavior can be observed throughout nature. Newborn chicks, for example, try to peck any small piece of material that they come across on the ground. Through trial and error, the chicks learn to peck only food. As they grow, they peck less at random things, and learn to identify seeds and grain.

Another example are chimpanzees learning through trial and error that using a stick to dig the soil is more favorable than using their hands. Goals, rewards, and penalties are important in reinforcement learning. A goal for a chimpanzee is to find food; a reward or penalty may be the number of times it has dug a hole or the time taken to dig a hole. The faster it can dig a hole, the faster it will find some food.

Figure 10.3 looks at the terminology used in reinforcement learning with reference to the simple dog-training example.



**Figure 10.3 Example of reinforcement learning: teaching a dog to sit by using food as a reward**

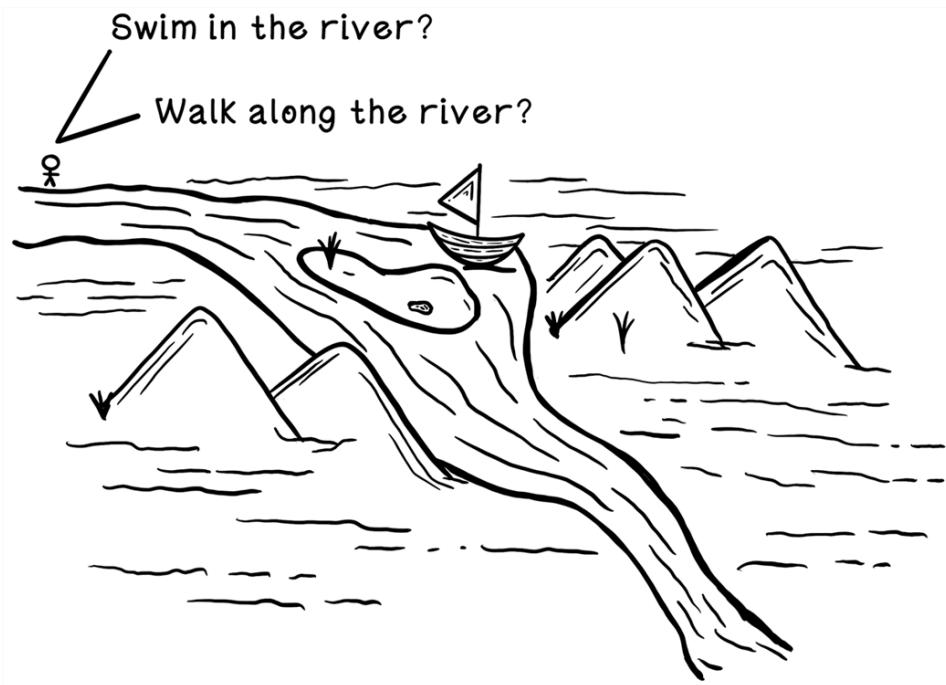
Reinforcement learning relies on a feedback loop of rewards and penalties (sometimes called punishments).

- *Positive Reward:* Receiving value after performing a good action, such as a dog getting a treat after it sits. This encourages the behavior.
- *Negative Reward (Penalty):* Receiving a negative value after performing a bad action, such as a dog getting scolded after tearing up a carpet. This discourages the behavior.

Positive reinforcement is meant to motivate desired behavior, and negative reinforcement is meant to discourage undesired behavior.

Another concept in reinforcement learning is balancing instant gratification with long-term consequences. Eating a chocolate bar is great for getting a boost of sugar and energy; this is *instant gratification*. But eating a chocolate bar every 30 minutes will likely cause serious health problems over time; this is a *long-term consequence*. Reinforcement learning aims to maximize the long-term benefit over short-term benefit, although short-term benefit may contribute to long-term benefit in some scenarios, because we want to exploit knowledge about current situations but also explore new things and discover new strategies.

Reinforcement learning is concerned with the long-term consequence of actions in an environment, so time and the sequence of actions are important. Suppose that we're stranded in the wilderness, and our goal is to survive as long as possible while traveling as far as possible in hopes of finding safety. We're positioned next to a river and have two options: jump into the river to travel downstream faster or walk along the side of the river. Notice the boat on the side of the river in figure 10.4. By swimming, we will travel faster but might miss the boat by being dragged down the wrong fork in the river. By walking, we will be guaranteed to find the boat, which will make the rest of the journey much easier, but we don't know this at the start. This example shows how important the sequence of actions is in reinforcement learning. It also shows how instant gratification may lead to long-term detriment. Furthermore, in a landscape that didn't contain a boat, the consequence of swimming is that we will travel faster but have soaked clothing, which may be very problematic when it gets cold. The consequence of walking is that we will travel slower but avoid wetting our clothing, which highlights the fact that a specific action may work in one scenario but not in others. Learning from many simulation attempts is important to finding more-generalist approaches.



**Figure 10.4 An example of possible actions that have long-term consequences**

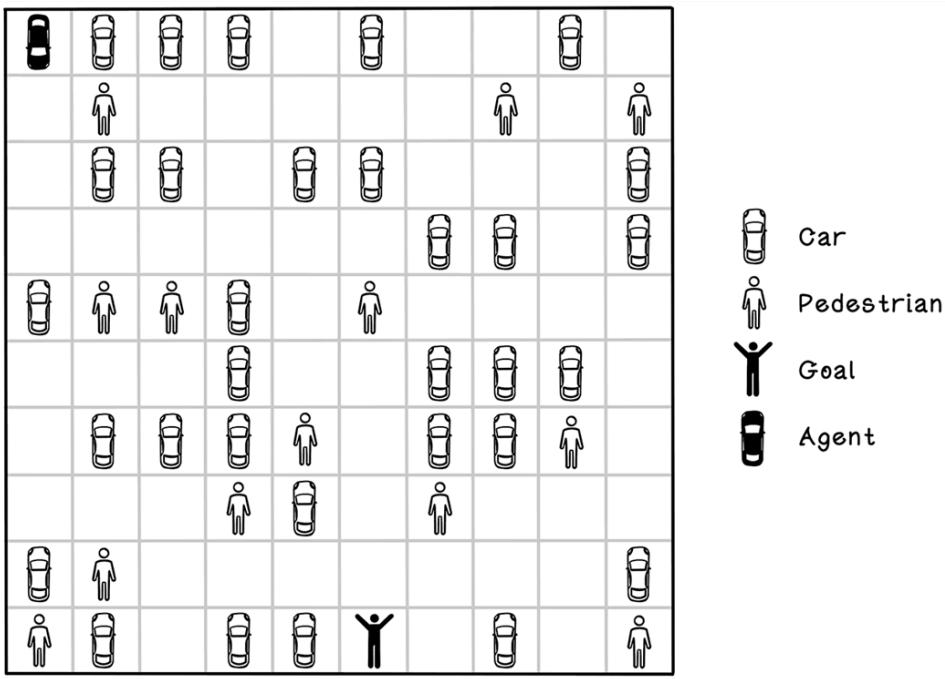
## 10.2 Problems applicable to reinforcement learning

To sum it up, reinforcement learning aims to solve problems in which a goal and allowed actions are known but the best plan to use the available actions is not. These problems involve controlling an agent's actions in an environment. Individual actions may be rewarded more than others, but the main concern is the *cumulative reward* of all actions.

Reinforcement learning is most useful for problems where individual actions build up toward a greater goal. Areas such as strategic planning, industrial-process automation, and robotics are good cases for the use of reinforcement learning. In these areas, individual actions may be suboptimal to gain a favorable outcome. Imagine a strategic game like chess. Some moves may be suboptimal based on the current state of the board, but they help set the board up for a greater strategic win later in the game. Reinforcement learning works well in domains in which chains of events are important for a good solution.

To work through the steps in a reinforcement learning algorithm, we will use the example car-collision problem from chapter 9 as inspiration. This time, however, we will be working with visual data about a self-driving car in a parking lot trying to navigate to its owner. Suppose that we have a map of a parking lot, including a self-driving car, other cars, and pedestrians. Our self-driving car can move north, south, east, and west. The other cars and pedestrians remain stationary in this example.

The goal is for our car to navigate the road to its owner while colliding with as few cars and pedestrians as possible—ideally, not colliding with anything. Colliding with a car is not good because it damages the vehicles, but colliding with a pedestrian is more severe. In this problem, we want to minimize collisions, but if we have a choice between colliding with a car and a pedestrian, we should choose the car. Figure 10.5 depicts this scenario.

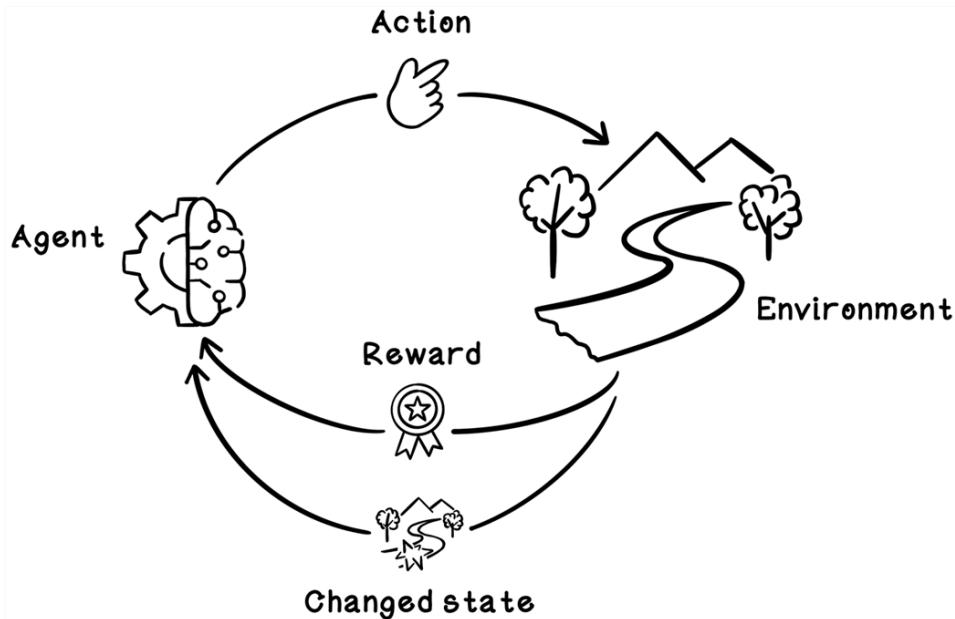


**Figure 10.5 The self-driving car in a parking lot problem**

We will be using this example problem to explore the use of reinforcement learning for learning good actions to take in dynamic environments.

### 10.3 The life cycle of reinforcement learning

Like other machine learning algorithms, a reinforcement learning model needs to be trained before it can be used. The training phase centers on exploring the environment and receiving feedback, given specific actions performed in specific circumstances or states. The life cycle of training a reinforcement learning model is based on the *Markov Decision Process*, which provides a mathematical framework for modeling decisions (figure 10.6). By quantifying decisions made and their outcomes, we can train a model to learn what actions toward a goal are most favorable.



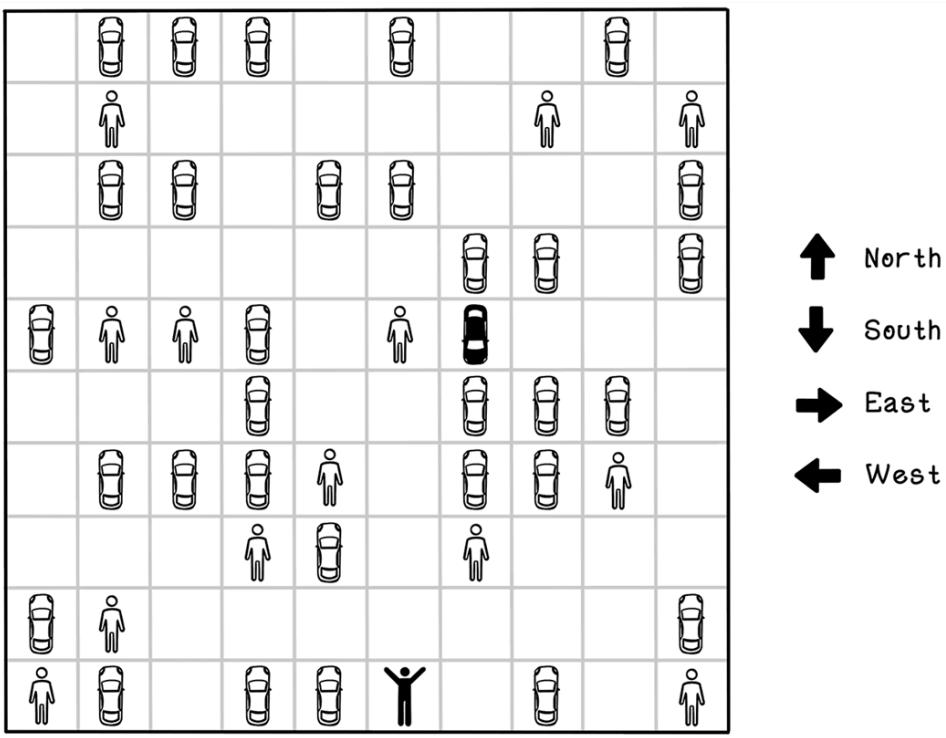
**Figure 10.6 The Markov Decision Process for reinforcement learning**

Before we can start tackling the challenge of training a model by using reinforcement learning, we need an environment that simulates the problem space we are working in. Our example problem entails a self-driving car trying to navigate a parking lot filled with obstacles to find its owner while avoiding collisions. This problem needs to be modeled as a simulation so that actions in the environment can be measured toward the goal. This simulated environment is different from the model that will learn what actions to take. This example is intentionally simplified for clarity. In a real-world application, the environment would be far more dynamic and intricate, presenting the agent with a much broader array of possible actions and a more nuanced spectrum of consequences for each choice.

### 10.3.1 Simulation and data: Make the environment come alive

Figure 10.7 depicts a parking-lot scenario containing several other cars and pedestrians. The starting position of the self-driving car and the location of its owner are represented as black figures. In this example, the self-driving car that applies actions to the environment is known as the *agent*.

The self-driving car, or agent, can take several actions in the environment. In this simple example, the actions are moving north, south, east, and west. Choosing an action results in the agent moving one block in that direction. The agent can't move diagonally.



**Figure 10.7 Agent actions in the parking-lot environment**

When actions are taken in the environment, rewards or penalties occur. Figure 10.8 shows the reward points awarded to the agent based on the outcome in the environment. A collision with another car is bad; a collision with a pedestrian is terrible. A move to an empty space is good; finding the owner of the self-driving car is better. The specified rewards aim to discourage collisions with other cars and pedestrians, and to encourage moving into empty spaces and reaching the owner. Note that there could be a reward for out-of-bounds movements, but we will simply disallow this possibility for the sake of simplicity.

Move into car.			-100
Move into pedestrian.			-1,000
Move into empty space.			+100
Move into goal.			+500

Figure 10.8 Rewards due to specific events in the environment due to actions performed

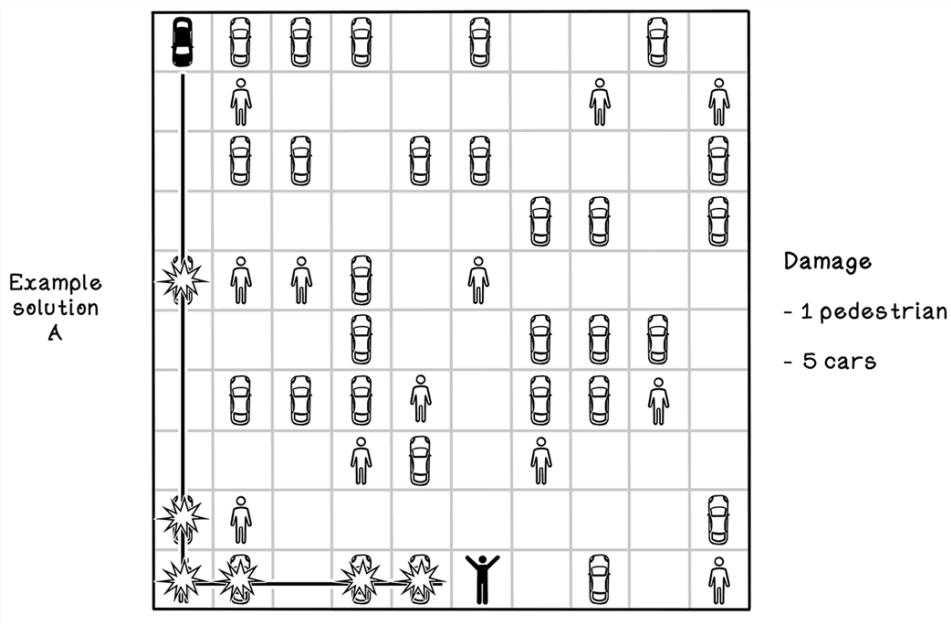
**NOTE** An interesting outcome of the rewards and penalties described is that the car may drive forward and backward on empty spaces indefinitely to accumulate rewards. We will dismiss this as a possibility for this example, but it highlights the importance of crafting good rewards.

The simulator needs to model the environment, the actions of the agent, and the rewards received after each action. A reinforcement learning algorithm will use the simulator to learn through practice by taking actions in the simulated environment and measuring the outcome. The simulator should provide the following functionality and information at minimum:

- *Initialize the environment*—This function involves resetting the environment, including the agent, to the starting state.
- *Get the current state of the environment*—This function should provide the current state of the environment, which will change after each action is performed.
- *Apply an action to the environment*—This function involves having the agent apply an action to the environment. The environment is affected by the action, which may result in a reward.

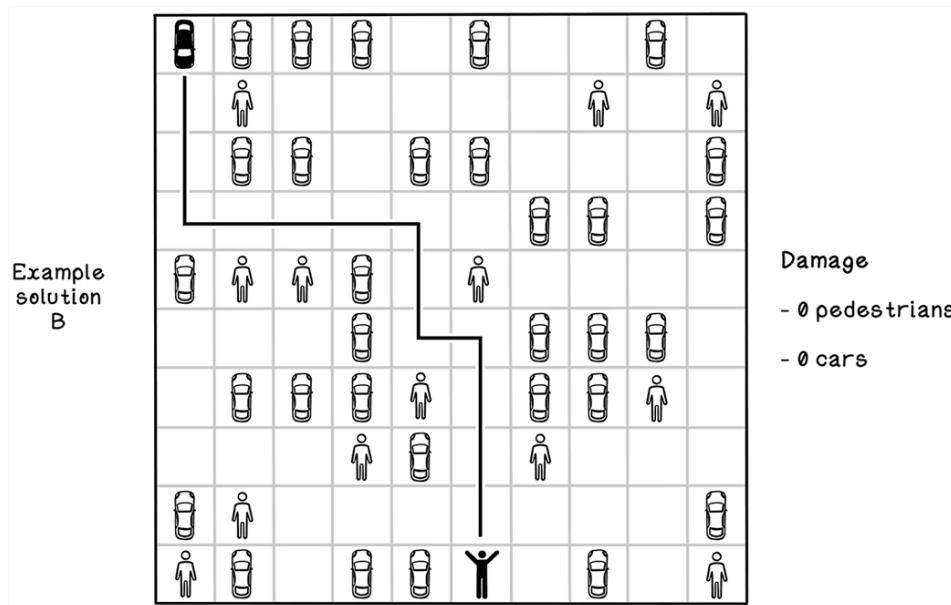
- *Calculate the reward of the action*—This function is related to applying the action to the environment. The reward for the action and effect on the environment need to be calculated.
- *Determine whether the goal is achieved*—This function determines whether the agent has achieved the goal. The goal can also sometimes be represented as completed. In an environment in which the goal cannot be achieved, the simulator needs to signal completion when it deems necessary.

Figures 10.9 and 10.10 depict possible paths in the self-driving-car example. In figure 10.9, the agent travels south until it reaches the boundary; then it travels east until it reaches the goal. Although the goal is achieved, the scenario resulted in five collisions with other cars and one collision with a pedestrian—not an ideal result.



**Figure 10.9** A bad solution to the parking-lot problem

Figure 10.10 depicts the agent traveling along a more specific path toward the goal, resulting in no collisions, which is great. It's important to note that given the rewards that we have specified, the agent is not guaranteed to achieve the shortest path; because we heavily encourage avoiding obstacles, the agent may find any path that is obstacle-free.



**Figure 10.10 A good solution to the parking-lot problem**

At this moment, there is no automation in sending actions to the simulator. It's like a game in which we provide input as a person instead of an AI providing the input. The next section explores how to train an autonomous agent.

### PYTHON CODE SAMPLE

The code for the simulator encompasses the functions discussed in this section. The simulator class would be initialized with the information relevant to the starting state of the environment.

The `move_agent` function is responsible for moving the agent north, south, east, or west, based on the action. It determines whether the movement is within bounds, adjusts the agent's coordinates, determines whether a collision occurred, and returns a reward score based on the outcome:

```

class Simulator:
    def __init__(self, road, road_size_x, road_size_y, agent_start_x,
agent_start_y):
        self.road = road
        self.road_size_x = road_size_x
        self.road_size_y = road_size_y
        self.agent_x = agent_start_x
        self.agent_y = agent_start_y

    def move_agent(self, action):
        next_x, next_y = self.agent_x, self.agent_y

        if action == COMMAND_NORTH:
            next_x -= 1
        elif action == COMMAND_SOUTH:
            next_x += 1
        elif action == COMMAND_EAST:
            next_y += 1
        elif action == COMMAND_WEST:
            next_y -= 1

        reward_update = 0

        if self.is_within_bounds(next_x, next_y):
            reward_update = self.cost_movement(next_x, next_y)
            self.agent_x, self.agent_y = next_x, next_y
        else:
            reward_update = ROAD_OUT_OF_BOUNDS_REWARD

    return reward_update

```

Here are descriptions of the next functions in the code:

- The `cost_movement` function determines the object in the target coordinate that the agent will move to and returns the relevant reward score.
- The `is_within_bounds` function is a utility function that makes sure the target coordinate is within the boundary of the road.
- The `is_goal_achieved` function determines whether the goal has been found, in which case the simulation can end.
- The `get_state` function uses the agent's position to determine a number that enumerates the current state. Each state must be unique. In other problem spaces, the state may be represented by the actual native state itself.

```

class Simulator:
    def __init__(self, road, road_size_x, road_size_y, agent_start_x,
agent_start_y):
        self.road = road
        self.road_size_x = road_size_x
        self.road_size_y = road_size_y
        self.agent_x = agent_start_x
        self.agent_y = agent_start_y

    def cost_movement(self, x, y):
        cell_content = self.road[x][y]

        if cell_content == 'G':
            return 10
        if cell_content == 'C':
            return -5
        return -1

    def is_within_bounds(self, x, y):
        return 0 <= x < self.road_size_x and 0 <= y < self.road_size_y

    def is_goal_achieved(self):
        return self.agent_x == self.goal_x and self.agent_y == self.goal_y

    def get_state(self):
        return (self.road_size_x * self.agent_x) + self.agent_y

```

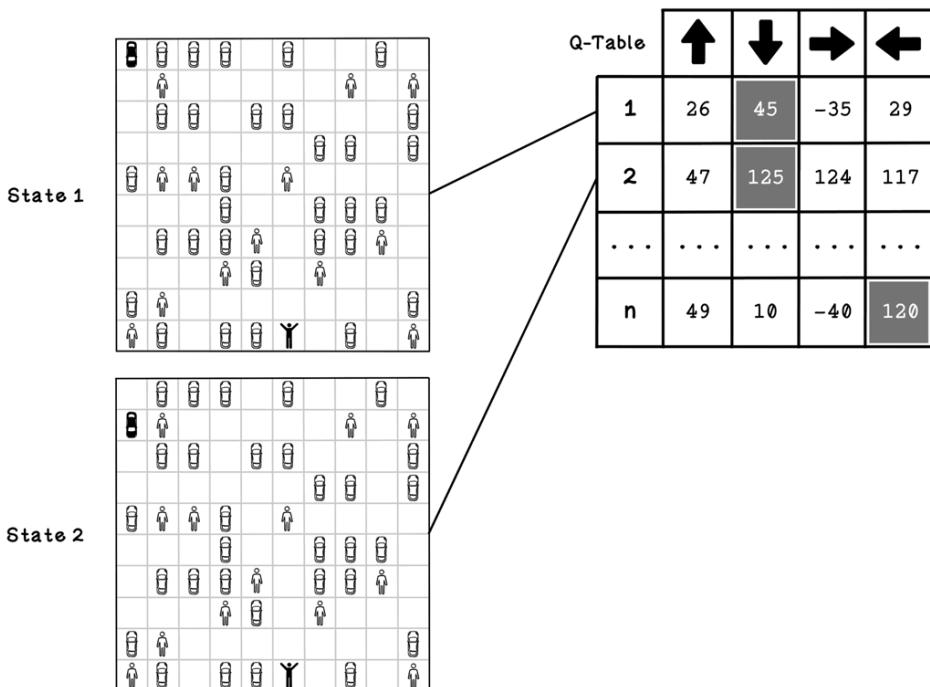
### 10.3.2 Training with the simulation using Q-learning

*Q-learning* is an approach in reinforcement learning that uses the states and actions in an environment to model a table that contains information describing favorable actions based on specific states. Q-learning is a model-free approach (meaning it doesn't need to know the physics of the world beforehand). It learns to estimate the quality of taking actions in specific states.

Reinforcement learning with Q-learning employs a reward table called a *Q-table*. Think of a Q-table as a scorecard. The rows are the possible states, and the columns are the possible actions. Each cell contains a Q-Value (Quality Value). The table doesn't explicitly store "the best action"; rather, it stores a score for every possible action. To find the best action, the agent looks at the row for its current state and picks the action with the highest Q-Value.

The point of a Q-table is to describe which actions are most favorable for the agent as it seeks a goal. The values that represent favorable actions are learned through simulating the possible actions in the environment and learning from the outcome and change in state. It's worth noting that the agent has a chance of choosing a random action or an action from the Q-table, as shown later in figure 10.13. The Q represents the function that provides the reward, or quality, of an action in an environment.

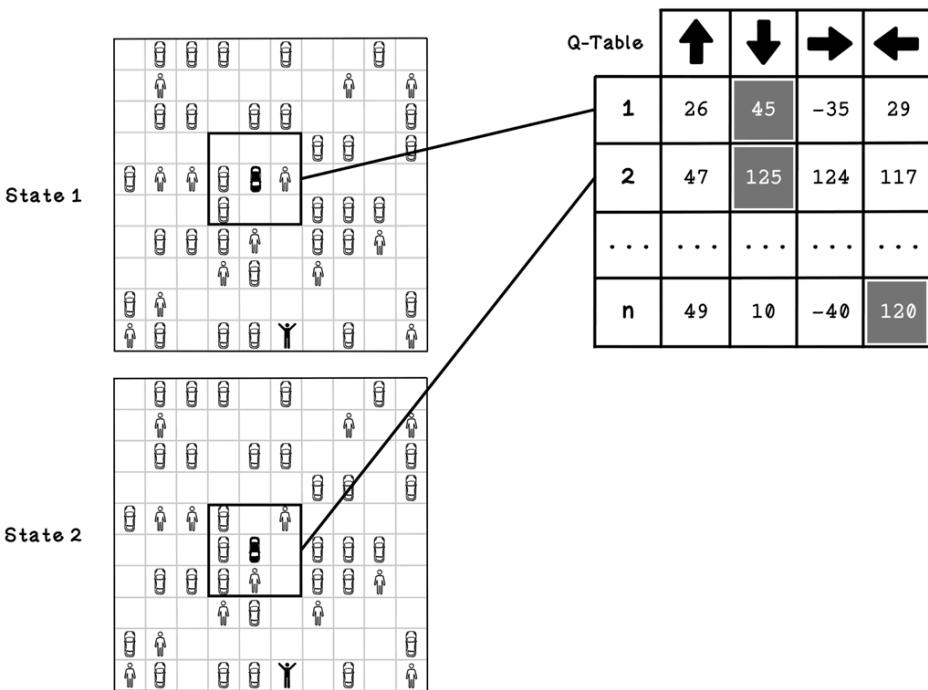
Figure 10.11 depicts a trained Q-table and two possible states that may be represented by the action values for each state. These states are relevant to the problem we're solving; another problem might allow the agent to move diagonally as well. Note that the number of states differs based on the environment and that new states can be added as they are discovered. In state 1, the agent is in the top-left corner, and in state 2, the agent is in the position below its previous state. The Q-table encodes the estimated value of taking an action in a state. Once the table is fully trained (converged), the action with the largest number is considered the most beneficial. Note that at the start of training, these numbers are just random guesses; they only become accurate "best actions" after many trials. Soon, we will see how they're calculated.



**Figure 10.11 An example Q-table and states that it represents**

The big problem with representing the state using the entire map is that the configuration of other cars and people is specific to this problem configuration. The Q-table learns the best choices only for this map layout.

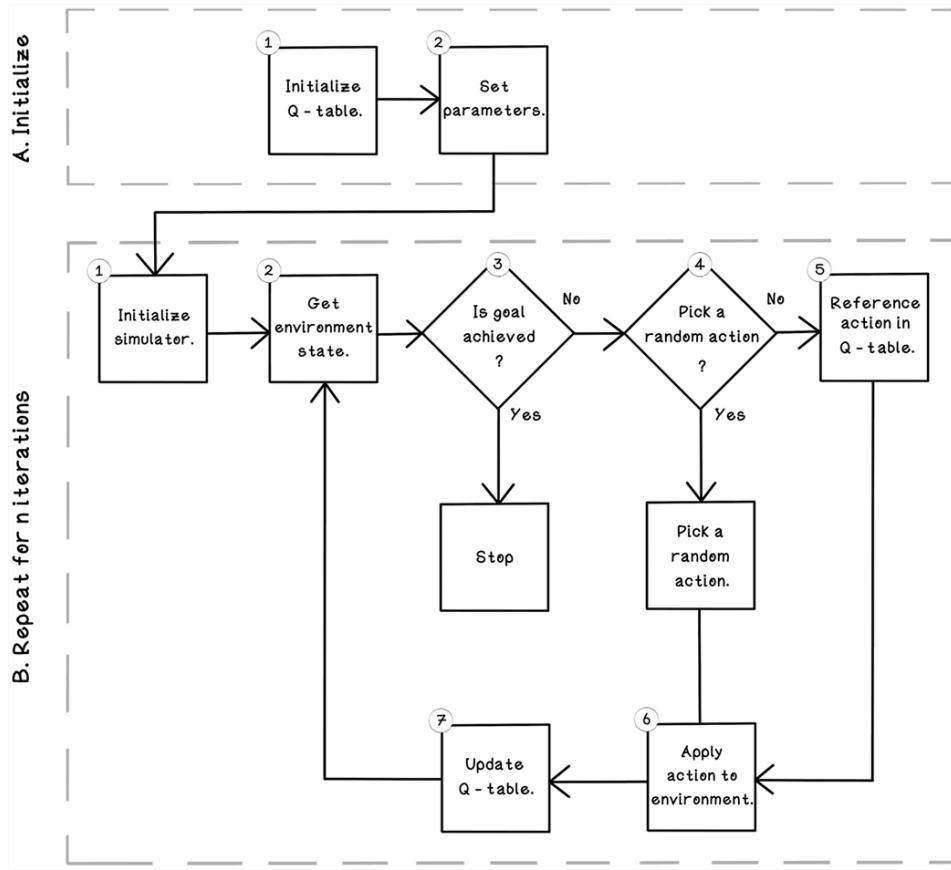
A better way to represent state in our example problem is to look at the objects adjacent to the agent. This approach allows the Q-table to adapt to other parking-lot configurations, because the state is less specific to the example parking lot from which it is learning. This approach may seem to be trivial, but a block could contain another car or a pedestrian, or it could be an empty block or an out-of-bounds block, which works out to four possibilities per block, resulting in a massive 65,536 possible states. With this much variety, we would need to train the agent in many parking-lot configurations many times for it to learn good short-term action choices (figure 10.12).



**Figure 10.12 A better example of a Q-table and states that it represents**

Keep the idea of a reward table in mind as we explore the life cycle of training a model using reinforcement learning with Q-learning. It will represent the model for actions that the agent will take in the environment.

Let's take a look at the life cycle of a Q-learning algorithm, including the steps involved in training. We will look at two phases: initialization, and what happens over several iterations as the algorithm learns (figure 10.13):



**Figure 10.13 Life cycle of a Q-learning reinforcement learning algorithm**

*Initialize.* The initialize step involves setting up the relevant parameters and initial values for the Q-table:

1. *Initialize Q-table.* Initialize a Q-table in which each column is an action and each row represents a possible state after actions based on the current state. Note that states can be added to the table as they are encountered, because it can be difficult to know the number of states in the environment at the beginning. The initial action values for each state are initialized with 0s.
2. *Set parameters.* This step involves setting the parameters for different hyperparameters of the Q-learning algorithm, including:
  - a. *Chance of choosing a random action*—This is the value threshold for choosing a random action over choosing an action from the Q-table.

- b. *Learning rate*—The learning rate is similar to the learning rate in supervised learning. It describes how quickly the algorithm learns from rewards in different states. With a high learning rate, values in the Q-table change erratically, and with a low learning rate, the values change gradually but it will potentially take more iterations to find good values.
- c. *Discount factor*—This determines the present value of future rewards. It works like inflation: a reward received today is worth more than the same reward received 10 steps from now.  
Think of the discount factor as the *Marshmallow Test* (a study on delayed gratification, where a child is offered one treat now or two treats if they wait for a short period).
  - $0 = \text{Impulsive}$ : "I want one marshmallow right now". The agent only cares about the immediate reward.
  - $0.9 = \text{Patient}$ : "I will wait. One marshmallow now is less valuable than five marshmallows in 10 minutes". The agent plans for long-term rewards.
  - $1 = \text{Infinite patience}$ : "I will wait forever if it means getting the highest score eventually".
 Mathematically, the value decays exponentially over time. This decay forces the agent to find the shortest path to the goal, rather than wandering around aimlessly gathering rewards eventually.

*Repeat for n iterations.* To learn the optimal path, the agent needs to learn from experience through thousands of trial-and-error repetitions. Each full trial, from a starting point to a concluding state (like reaching the goal or failing), is called an episode. Within an episode, the agent takes a series of steps (e.g., the car making a single move), receiving feedback and updating the Q-table after each one. Running many episodes is crucial because it allows the estimated value of future rewards to slowly propagate backward from the goal state. This is how the agent learns to make good short-term decisions that lead to the best possible long-term outcomes.

1. *Initialize simulator*. This step involves resetting the environment to the starting state, with the agent in a neutral state.
2. *Get environment state*. This function should provide the current state of the environment. The state of the environment will change after each action is performed.
3. *Is goal achieved?* Determine whether the goal is achieved (or the simulator deems the exploration to be complete). In our example, this goal is picking up the owner of the self-driving car. If the goal is achieved, the algorithm ends.

4. *Pick a random action.* Determine whether a random action should be selected. If so, a random action will be selected (north, south, east, or west). Random actions are useful for exploring the possibilities in the environment instead of learning a narrow subset.
5. *Reference action in Q-table.* If the decision to select a random action is not selected, the current environment state is transposed to the Q-table, and the respective action is selected based on the values in the table. More about the Q-table is coming up.
6. *Apply action to environment.* This step involves applying the selected action to the environment, whether that action is random or one selected from the Q-table. An action will have a consequence in the environment and yield a reward.
7. *Update Q-table.* The following material describes the concepts involved in updating the Q-table and the steps that are carried out.

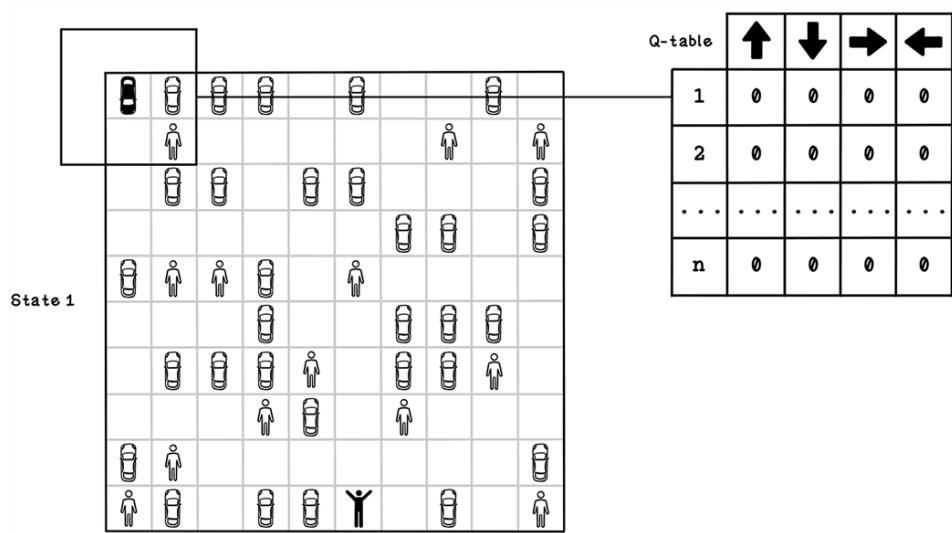
The key aspect of Q-learning is the equation used to update the values of the Q-table. This equation is based on the *Bellman equation*, which determines the value of a decision made at a certain point in time, given the reward or penalty for making that decision. The Q-learning equation is an adaptation of the Bellman equation. In the Q-learning equation, the most important properties for updating Q-table values are the current state, the action, the next state given the action, and the reward outcome. The learning rate is similar to the learning rate in supervised learning, which determines the extent to which a Q-table value is updated. The discount is used to indicate the importance of possible future rewards, which is used to balance favoring immediate rewards versus long-term rewards.

To clarify how the discount factor enables long-term planning, it's important to distinguish between the agent's final policy and its training process. While a trained agent's policy may be to simply choose the action with the highest Q-value in a given state (which seems short-sighted), those Q-values themselves are inherently farsighted. During thousands of training episodes, the discount factor ensures that the value of distant rewards is propagated backward through every state-action pair that leads to them. As a result, a high Q-value doesn't just represent a good immediate reward; it represents the start of a path that the agent has learned will lead to the highest total discounted future reward.

$$\begin{aligned} Q(\text{state}, \text{action}) = & \quad \text{The maximum value of all actions on next state} \\ & (1 - \alpha) * Q(\text{state}, \text{action}) + \alpha * (\text{reward} + \gamma * Q(\text{next state}, \text{all actions})) \end{aligned}$$

Learning rate      Current value      Learning rate      Discount

In this specific example, we initialize the Q-table with 0s (figure 10.14). While this is a common starting point, advanced implementations sometimes use *optimistic initialization*—filling the table with high numbers to trick the agent into exploring every state to “verify” those high rewards.

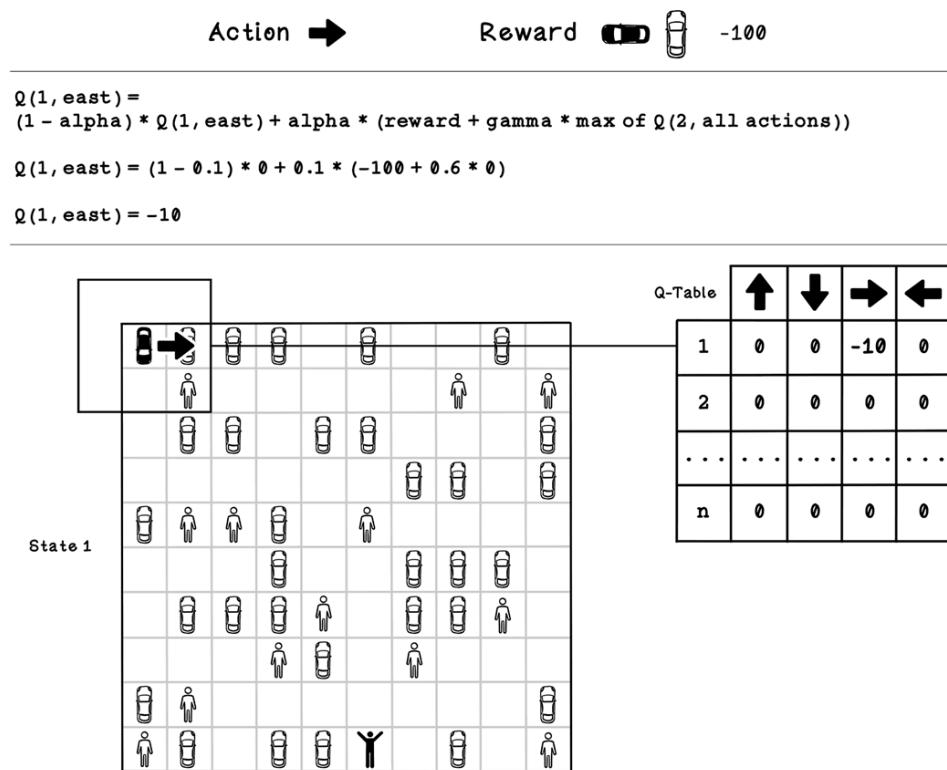


**Figure 10.14 An example initialized Q-table**

Next, we explore how to update the Q-table by using the Q-learning equation based on different actions with different reward values. These values will be used for the learning rate (alpha) and discount (gamma):

- Learning rate (alpha): 0.1
- Discount (gamma): 0.6

Figure 10.15 illustrates how the Q-learning equation is used to update the Q-table, if the agent selects the East action from the initial state in the first iteration. Remember that the initial Q-table consists of 0s. The learning rate (alpha), discount (gamma), current action value, reward, and next best state are plugged into the equation to determine the new value for the action that was taken. The action is East, which results in a collision with another car, which yields -100 as a reward. After the new value is calculated, the value of East on state 1 is -10.



**Figure 10.15 Example Q-table update calculation for state 1**

The next calculation is for the next state in the environment following the action that was taken. The action is South and results in a collision with a pedestrian, which yields -1,000 as the reward. After the new value is calculated, the value for the South action on state 2 is -100 (figure 10.16).

Action Reward -1000

$$\begin{aligned} Q(2, \text{south}) &= \\ (1 - \alpha) * Q(2, \text{south}) + \alpha * (\text{reward} + \gamma * \max \text{ of } Q(3, \text{all actions})) \end{aligned}$$

$$Q(2, \text{south}) = (1 - 0.1) * 0 + 0.1 * (-1000 + 0.6 * 0)$$

$$Q(2, \text{south}) = -100$$

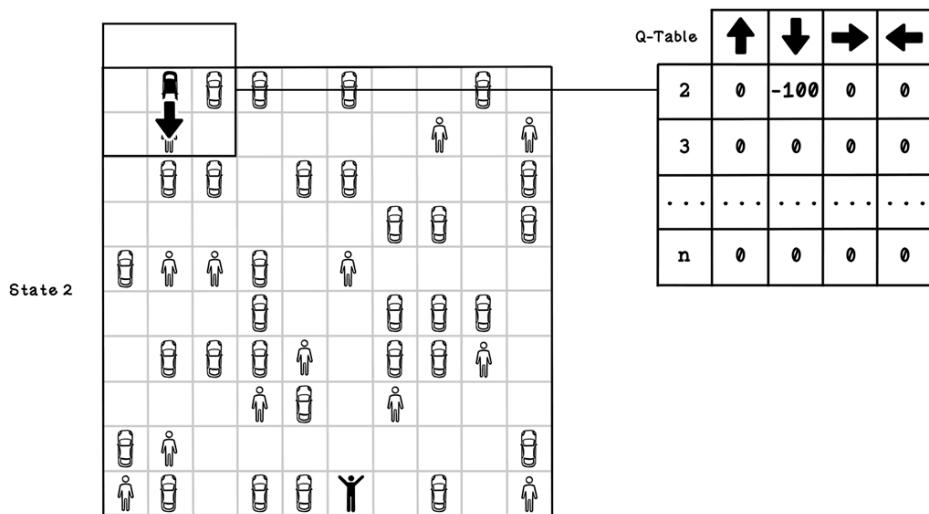


Figure 10.16 Example Q-table update calculation for state 2

Figure 10.17 illustrates how the calculated values differ in a Q-table with populated values because we worked on a Q-table initialized with 0s. The figure is an example of the Q-learning equation updated from the initial state after several iterations. The simulation can be run multiple times to learn from multiple attempts. So, this iteration is succeeding many before it, where the values of the table have been updated. The action for East results in a collision with another car and yields -100 as a reward. After the new value is calculated, the value for East on state 1 changes to -34.

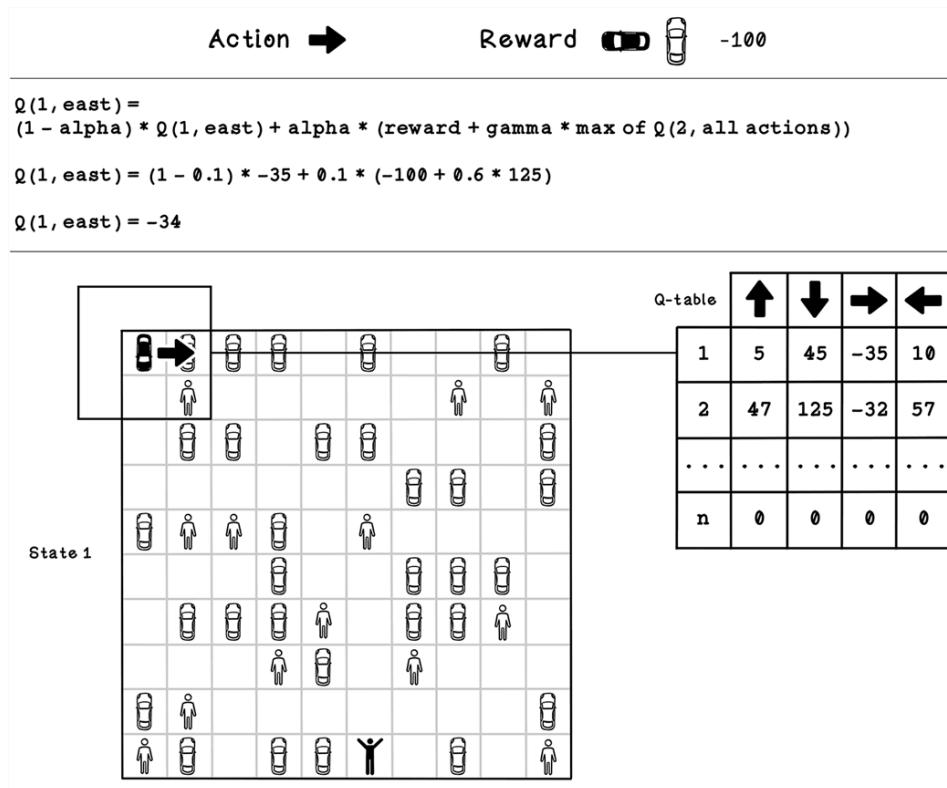


Figure 10.17 Example Q-table update calculation for state 1 after several iterations

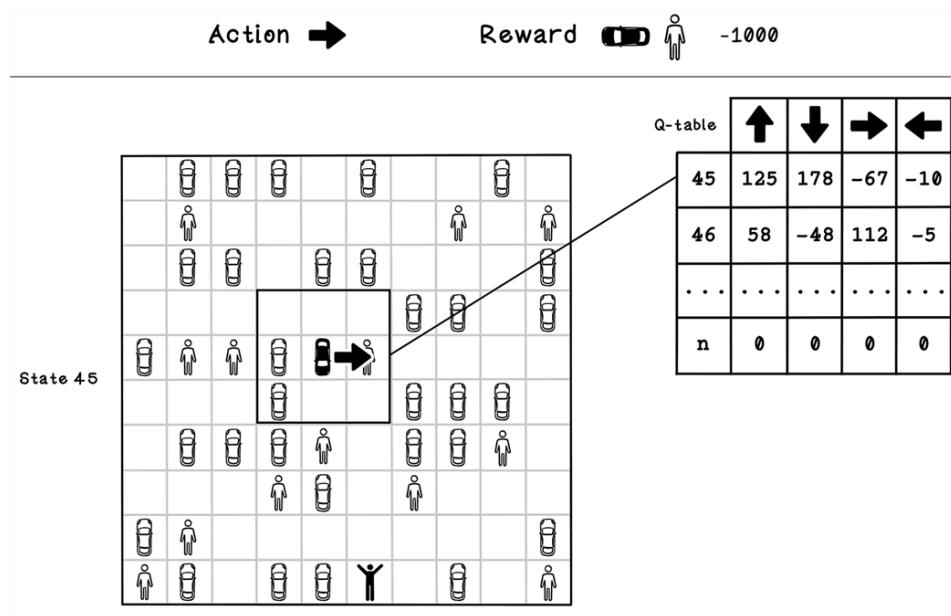
### EXERCISE: CALCULATE THE CHANGE IN VALUES FOR THE Q-TABLE

Using the Q-learning update equation and the following scenario, calculate the new value for the action performed. Assume that the last move was East with a value of -67:

$$Q(\text{state}, \text{action}) = (1 - \alpha) * Q(\text{state}, \text{action}) + \alpha * (\text{reward} + \gamma * \max_{\text{next state}} Q(\text{next state}, \text{all actions}))$$

Learning rate      Current value      Learning rate      Discount

The maximum value of all actions on next state



## SOLUTION: CALCULATE THE CHANGE IN VALUES FOR THE Q-TABLE

The hyperparameter and state values are plugged into the Q-learning equation, resulting in the new value for  $Q(1, \text{east})$ :

- Learning rate (alpha): 0.1
- Discount (gamma): 0.6
- $Q(1, \text{east})$ : -67
- Max of  $Q(2, \text{all actions})$ : 112

```


$$Q(1, \text{east}) =$$


$$(1 - \text{alpha}) * Q(1, \text{east}) + \text{alpha} * (\text{reward} + \text{gamma} * \text{max of } Q(2, \text{all actions}))$$



$$Q(1, \text{east}) = (1 - 0.1) * -67 + 0.1 * (-100 + 0.6 * 112)$$



$$Q(1, \text{east}) = -64$$


```

## PYTHON CODE SAMPLE

This code describes a function that trains a Q-table by using Q-learning. It could be broken into simpler functions but is represented this way for readability. The function follows the steps described in this chapter.

The Q-table is initialized with 0s; then the learning logic is run for several iterations. Remember that an iteration is an attempt to achieve the goal.

The next piece of logic runs while the goal has not been achieved:

1. Decide whether a random action should be taken to explore possibilities in the environment. If not, the highest value action for the current state is selected from the Q-table.
2. Proceed with the selected action, and apply it to the simulator.
3. Gather information from the simulator, including the reward, the next state given the action, and whether the goal is reached.
4. Update the Q-table based on the information gathered and hyperparameters. Note that in this code, the hyperparameters are passed through as arguments of this function.
5. Set the current state to the state outcome of the action just performed.

These steps will continue until a goal is found. After the goal is found and the desired number of iterations is reached, the result is a trained Q-table that can be used to test in other environments. We look at testing the Q-table in the next section:

```

def train_with_q_learning(observation_space, action_space_size,
number_of_iterations, learning_rate, discount, chance_of_random_move):

    q_table = np.zeros((observation_space, action_space_size))

    for i in range(number_of_iterations):

        simulator = Simulator(DEFAULT_ROAD, DEFAULT_ROAD_SIZE_X,
DEFAULT_ROAD_SIZE_Y, DEFAULT_START_X, DEFAULT_START_Y, DEFAULT_GOAL_X,
DEFAULT_GOAL_Y)

        state = simulator.get_state()
        done = False

        while not done:

            if random.uniform(0, 1) > chance_of_random_move:
                action_index = get_action_index_from_max_q(q_table[state])
            else:
                action_index = get_random_action_index(action_space_size)

            action_command = COMMANDS[action_index]

            reward = simulator.move_agent(action_command)
            next_state = simulator.get_state()
            done = simulator.is_goal_achieved()

            current_value = q_table[state, action_index]
            next_state_max_value = np.max(q_table[next_state])

            new_value = (1 - learning_rate) * current_value + learning_rate * \
(reward + discount * next_state_max_value)

            q_table[state, action_index] = new_value
            state = next_state

        if i % 100 == 0:
            print(f"Episode {i:04d} complete.")

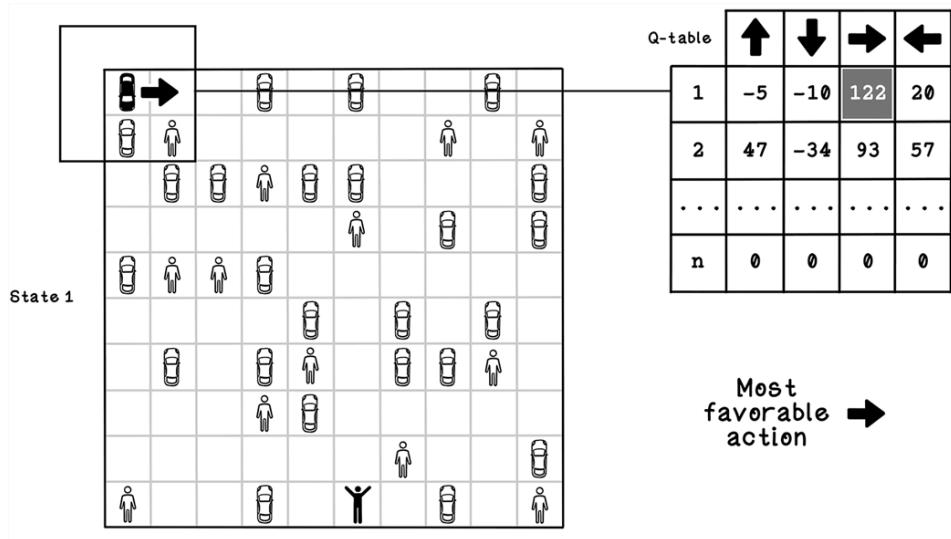
    return q_table

```

### 10.3.3 Testing with the simulation and Q-table

We know that in the case of using Q-learning, the Q-table is the model that encompasses the learnings. When presented with a new environment with different states, the algorithm references the respective state in the Q-table and chooses the highest-valued action. Because the Q-table has already been trained, this process consists of getting the current state of the environment and referencing the respective state in the Q-table to find an action until a goal is achieved (figure 10.18).

This, however, highlights a fundamental limitation of a Q-table. A basic Q-table cannot generalize to new situations. If a state is not in the table, the agent has no learned information and must resort to a default policy, such as picking a random action, which is rarely optimal. This inability to handle novel states is a key reason why, for complex problems, we often replace the rigid Q-table with models like neural networks that can estimate the best action even for situations they haven't seen before.



**Figure 10.18** Referencing a Q-table to determine what action to take

Because the state learned in the Q-table considers the objects directly next to the agent's current position, the Q-table has learned good and bad moves for short-term rewards, so the Q-table could be used in a different parking-lot configuration, such as the one shown in figure 10.18. The disadvantage is that the agent favors short-term rewards over long-term rewards because it doesn't have the context of the rest of the map when taking each action.

One term that will likely come up in the process of learning more about reinforcement learning is *episodes*. An *episode* includes all the states between the initial state and the state when the goal is achieved. If it takes 14 actions to achieve a goal, we have 14 episodes. If the goal is never achieved, the episode is called *infinite*.

#### **10.3.4 Measuring the performance of training**

Reinforcement learning algorithms can be difficult to measure generically. Given a specific environment and goal, we may have different penalties and rewards, some of which have a greater effect on the problem context than others. In the parking-lot example, we heavily penalize collisions with pedestrians. In another example, we may have an agent that resembles a human and tries to learn what muscles to use to walk naturally as far as possible. In this scenario, penalties may be falling or something more specific, such as too-large stride lengths. To measure performance accurately, we need the context of the problem.

One generic way to measure performance is to count the number of penalties in a given number of attempts. Penalties could be events that we want to avoid that happen in the environment due to an action.

Another measurement of reinforcement learning performance is *average reward per action*. By maximizing the reward per action, we aim to avoid poor actions, whether the goal was reached or not. This measurement can be calculated by dividing the cumulative reward by the total number of actions.

#### **10.3.5 Model-free and model-based learning**

To support your future learning in reinforcement learning, be aware of two approaches for reinforcement learning: *model-based* and *model-free*, which are different from the machine learning models discussed in this book. Think of a model as being an agent's abstract representation of the environment in which it is operating.

We may have a model in our heads about locations of landmarks, intuition of direction, and the general layout of the roads within a neighborhood. This model has been formed from exploring some roads, but we're able to simulate scenarios in our heads to make decisions without trying every option. To decide how we will get to work, for example, we can use this model to make a decision; this approach is model-based. Model-free learning is similar to the Q-learning approach described in this chapter; trial and error is used to explore many interactions with the environment to determine favorable actions in different scenarios.

Figure 10.19 depicts the two approaches in road navigation. Different algorithms can be employed to build model-based reinforcement learning implementations.

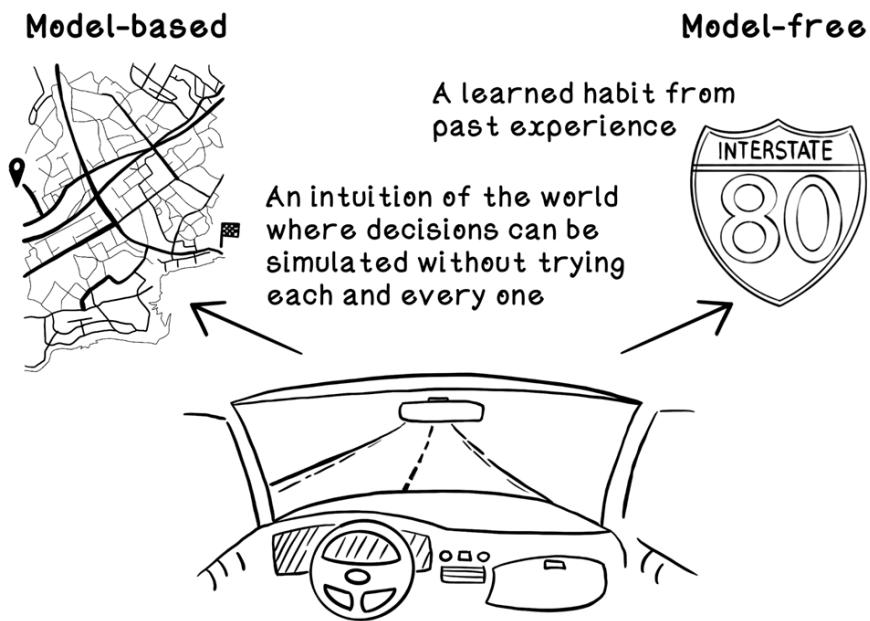


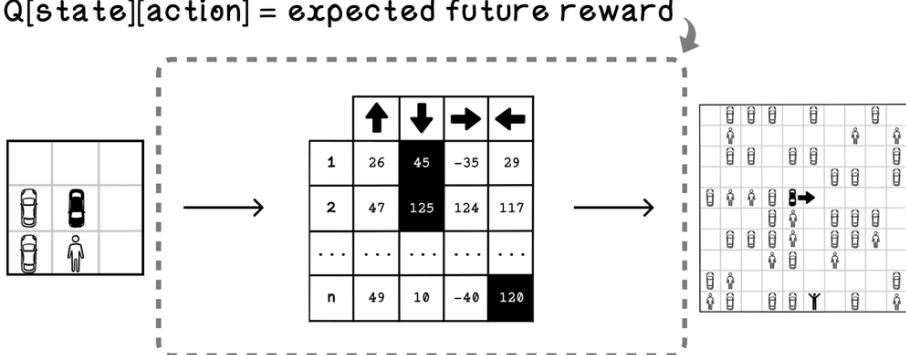
Figure 10.19 Examples of model-based and model-free reinforcement learning

## 10.4 Deep learning approaches to reinforcement learning

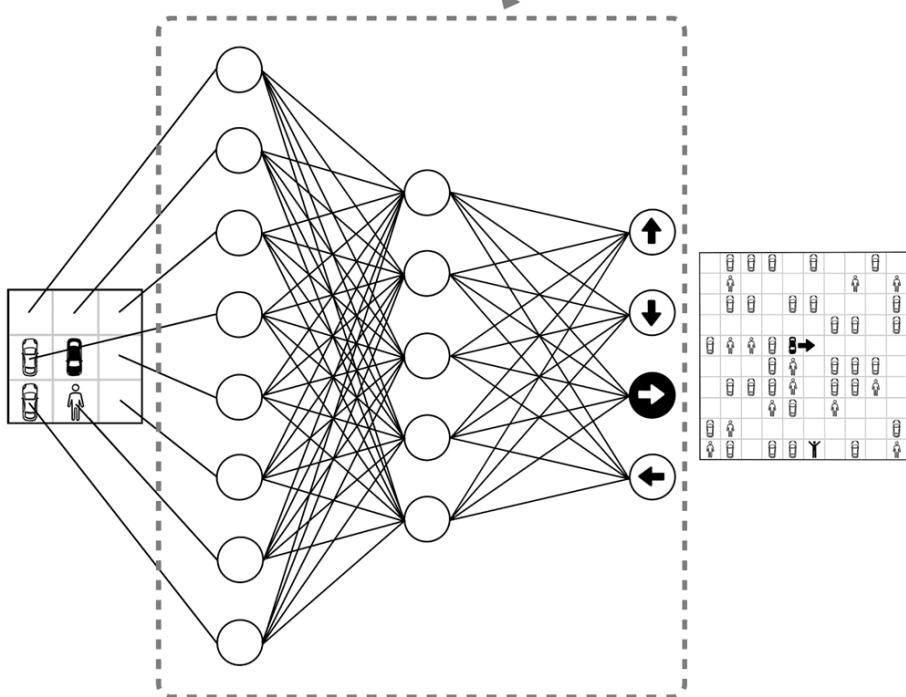
Q-learning is one approach to reinforcement learning. Having a good understanding of how it functions allows you to apply the same reasoning and general approach to other reinforcement learning algorithms. Several alternative approaches depend on the problem being solved. One popular alternative is *deep reinforcement learning*, which is useful for applications in robotics, video-game play, and problems that involve images and video.

Deep reinforcement learning can use artificial neural networks (ANNs) to process the states of an environment and produce an action. The actions are learned by adjusting weights in the ANN, using the reward feedback and changes in the environment. Reinforcement learning can also use the capabilities of convolutional neural networks (CNNs) and other purpose-built ANN architectures to solve specific problems in different domains and use cases. Instead of having the “learnings” encoded into a table of actions as in Q-learning, the learnings are encoded as weights in an ANN. Figure 10.20 visualizes this difference.

$Q[\text{state}][\text{action}] = \text{expected future reward}$



$Q(\text{state}, \text{action}) \approx \text{output of ANN}$



**Figure 10.20 The difference between using a Q-table and ANN for the parking-lot problem**

In deep Q-learning, the neural network serves as a function approximator that learns to predict Q-values: the expected long-term reward of taking a particular action in a particular state.

As you can probably tell, the primary change is how the actions for specific states are learned. As we know from Chapter 9, neural networks are great at finding fine-grained relationships, then finding abstractions of those relationships that are learned from a loss function. In this case the loss function is based on the reward or penalty for a specific action taken given a specific state.

### 10.4.1 Training with an artificial neural network

Let's revisit our parking-lot problem, but instead of using a Q-table, we will use a neural network as the model that will be trained to predict actions with a state of the car's surroundings as the input, and an action as the output.

- *Input nodes*—The input nodes represent a specific state of the map. Remember in Chapter 9's example we had 4 input nodes representing the speed, terrain quality, degree of vision, and total experience as scaled values between 0 and 1. In the case of the parking-lot example, we need to encode the state in a way that's conducive to effective learning by the neural network. The way that the input is expressed also influences the design of the hidden layers and activation functions. We will look at two options for this soon.
- *Hidden layers*—The hidden layers are dependent on the number of input nodes we have. If we have more nodes in the input, we likely need more nodes in the first layer to find patterns well from the inputs. And naturally, this would impact the subsequent hidden layers in the network. The choice of two hidden layers is sensible for this use case. If we have too many hidden layers, we increase training time, and risk overfitting to the training data.
- *Output node*—The output will be 4 nodes representing one score for each respective action. This configuration is not influenced by the input nodes and hidden nodes since we know that we need an output that is an action, and the most clear way is to have 4 discrete values that represent the associated reward for each action. We will use a linear activation function for the nodes in the output since the values can be positive or negative.

Training the network follows the same reinforcement learning loop; we simply replace the Q-table update with a backpropagation step. Let's use this opportunity to learn about the difference that the input encoding makes to the design of the neural network.

## SCALAR ENCODING APPROACH

The scalar encoding approach represents each category in a cell with a single numeric value that reflects its relative importance or impact. For example, a goal might be encoded as 1, an empty space as 0, a vehicle as -0.5, and a pedestrian as -1. This compact representation reduces the number of input nodes and can help the model learn faster by embedding domain knowledge directly into the input. However, it assumes a meaningful ordinal relationship between the categories, which may limit flexibility or mislead the model if the relationships are not truly linear.

In this case, the values hold meaning:

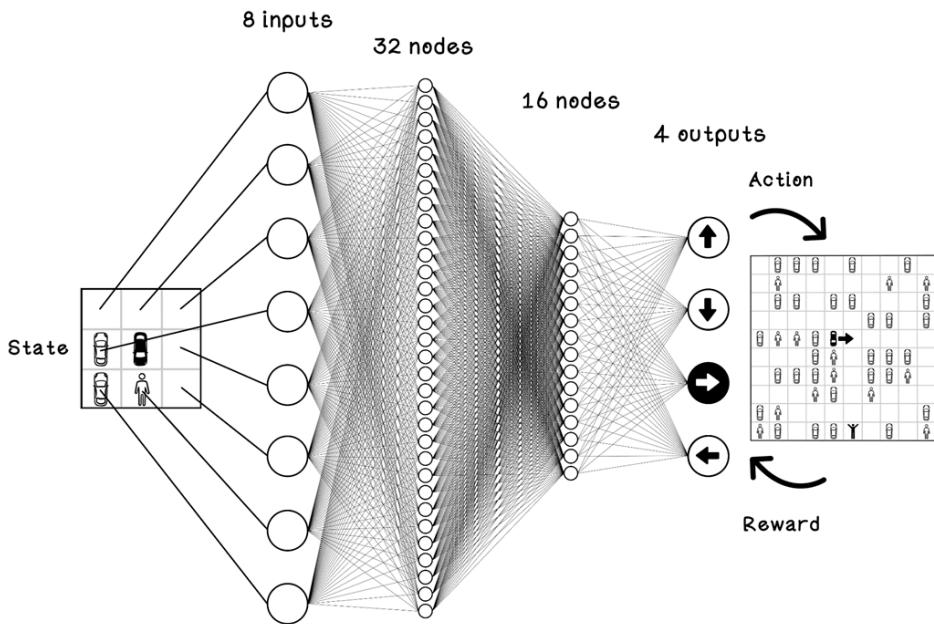
- 1 is as positive as can be and represents reaching the goal.
- 0 is neutral and represents an empty space.
- -0.5 represents another vehicle. It's bad, but not as bad as -1.
- -1 is the worst and represents a pedestrian.

These values represent a “risk / reward” gradient within the network. It’s more compact but less explicit, and may lead the network to learn unrelated relationships. With scalar encoding, we use 8 input nodes to represent all surrounding blocks (including diagonals). Why 8 inputs if we can only move in 4 directions? Even though the agent cannot move diagonally, it needs full situational awareness. An obstacle in the North-East corner becomes an immediate danger if the agent decides to move North. By observing all 8 surrounding cells, the network can anticipate future collisions and recognize complex environment shapes.

Given this input, and given that we have decided to have 2 hidden layers in the network, let’s look at what a possible configuration could look like.

Using 32 nodes in the first hidden layer for the 8 input nodes provides the network with enough capacity to learn complex feature interactions from a compact input. Each of the 8 input nodes (representing the state of nearby cells) may carry nonlinear signals about obstacles, goals, or dangers. A 4x expansion to 32 hidden nodes allows the network to extract diverse patterns, like recognizing dangerous obstacles or promising paths to the goal, without overfitting on such a small input space.

Following this, 16 nodes in the second hidden layer act as a refinement stage. It compresses the features extracted by the first layer into a more abstract representation, suitable for selecting one of the four possible actions. The structure of  $8 \rightarrow 32 \rightarrow 16 \rightarrow 4$  encourages the network to distill the raw input into meaningful, generalizable decisions while avoiding complexity. Figure 10.21 illustrates what this network looks like.



**Figure 10.21 Example 8 input ANN for the parking-lot problem**

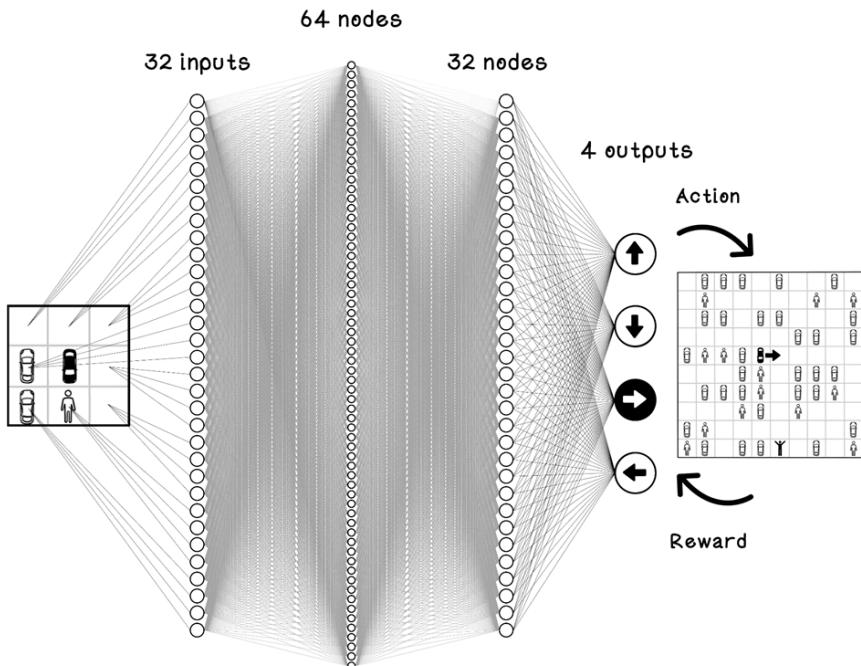
For scalar inputs, like 8 input values ranging from -1 to 1, feeding into the 32 hidden nodes, the recommended activation function is ReLU (Rectified Linear Unit), as seen in Chapter 9. ReLU is computationally efficient and introduces nonlinearity while preserving the magnitude of positive values, making it ideal when inputs already carry semantic meaning, such as danger or reward. It activates only when useful patterns emerge, helping the network focus on relevant signals. If negative input values, like -1 for pedestrians, are critical to the task, *Leaky ReLU* can be used instead to retain some gradient flow for negative signals.

## ONE-HOT ENCODING APPROACH

The one-hot encoding approach represents each possible category in a cell with a separate input node, activating only one node per block. For example, a block could be encoded as [1, 0, 0, 0] for empty, [0, 1, 0, 0] for vehicle, [0, 0, 1, 0] for pedestrian, and [0, 0, 0, 1] for the goal. This creates a larger input vector of size 32 (because we have 8 adjacent blocks with 4 possible states). Although it is large, it avoids implying any ordinal or linear relationship between categories, because values are simply 0 or 1 for specific input nodes. One-hot encoding gives the model full flexibility to learn independent outcomes for each category, making it ideal when the categories (empty / vehicle / person / goal) are distinct and unordered, as in the parking-lot example.

Using 64 nodes in the first hidden layer for 32 one-hot encoded input nodes allows the network to learn a rich set of patterns and interactions across the input features. Since each cell in the input grid is represented by multiple one-hot nodes, the model needs more capacity to combine and interpret these sparse inputs, for example, recognizing dangerous arrangements like a pedestrian to the left and a vehicle ahead. Doubling the size of the input gives the network enough flexibility to capture nonlinear relationships and local spatial patterns without becoming overly complex.

The 32 nodes in the second hidden layer then serve to refine and condense the higher-level features learned in the first layer. This gradual reduction helps the model generalize by forcing it to prioritize the most relevant information for decision-making, ultimately feeding into the output layer of 4 actions. This structure balances learning capacity with training efficiency, making it well-suited for one-hot encoded grid inputs where categorical relationships need to be learned independently.



**Figure 10.22 Example 32 input ANN for the parking-lot problem**

One-hot inputs are sparse by nature, and ReLU efficiently activates only when meaningful signals are present, ignoring inactive inputs. So it is a good choice as the activation function for the nodes in the hidden layers for our one-hot encoding scenario as well. Since one-hot vectors don't carry ordinal meaning, ReLU enables the network to learn useful patterns from scratch. Activation functions like sigmoid or tanh are less suitable here, as they can unnecessarily squash values and slow down learning.

## CALCULATING LOSS AND BACKPROPAGATION

The goal is to train the network to approximate the Q-function, which guides the agent toward actions that maximize cumulative rewards over time. Instead of learning from fixed labeled data, the network learns through interactions with the environment, using feedback from each action it takes.

After each action, the agent receives a reward and observes the next state. These four elements, the current state, action taken, reward received, and resulting next state, form a training sample. The network uses this to compute a target Q-value: the immediate reward plus the discounted maximum predicted Q-value for the next state. It then compares this target to its own predicted Q-value for the action taken, and the squared difference becomes the loss.

Assume that the algorithm has the current state ( $S$ ) and it chooses east as the action. The outcome is a new state ( $S'$ ) and a reward of +1 for performing that action. Let's assume that the rewards for  $S'$  looks like the following:

- North: 1.2
- South: 0.5
- East: 2.0
- West: 0.3

Based on this, the network thinks that the best possible move will be East and give a reward of 2.0 - the maximum reward from the options. This is good, but remember, we want the network to learn "Moving right from the previous state should be worth today's reward plus future rewards".

The discount factor controls how much the agent values future rewards compared to immediate ones, and 0.9 is a typical default value used in many reinforcement learning problems.

- A high factor (e.g. 0.9 or 0.99) means the agent values long-term rewards almost as much as immediate rewards.
- A low factor (e.g. 0.1 or 0.3) means the agent is short-sighted. It cares more about immediate rewards and less about the future.

So given these values, we calculate the target as follows:

$$\begin{aligned}
 \text{target} &= \text{reward} + \text{discount} * \max \text{ future reward} \\
 &= 1.0 + 0.9 * 2.0 \\
 &= 1 + 1.8 \\
 &\Rightarrow = 2.8
 \end{aligned}$$

If I go right, I'll get 1 point immediately and likely 1.8 more later

Now we look at what the network currently thinks the value of going right from state S is:

- North: 0.7
- South: 1.1
- East: 2.2
- West: 0.5

So the network believes that moving east is 2.2, but we know from our target calculation that it should be 2.8. We compute the difference as the loss, and as usual, the network minimizes this loss for future epochs by using backpropagation and gradient descent. During backpropagation, the gradients of the loss are used to adjust the weights in the direction that reduces future error. Over time, this tunes the network to predict more accurate Q-values, enabling better decision-making.

## NUMBER OF EPISODES

Learning occurs across many episodes. Each episode is a simulation run from the starting state to either a goal state or a failure / timeout condition. Contrary to what might be expected, the agent does not need to complete the episode for learning to happen. Even partial progress, such as a few steps before a failure, contributes useful training data, allowing the model to improve gradually.

Before starting the training loop, we must decide how each new episode begins. A key choice is the initial state. In some problems, like chess, the agent always starts from a single, fixed state. However, for many tasks, a more powerful approach is to begin each episode from a randomly chosen valid state. For our self-driving car, this would mean placing it in a different spot in the parking lot for each new trial run. This method forces the agent to learn a more robust and complete strategy, as it must discover valuable paths from all over the environment, not just from a single starting point.

As training progresses, the agent balances exploration and exploitation. Early on, it explores the environment by trying random actions to discover valuable paths. Over time, it exploits the best-known actions more frequently as the network's predictions improve, much like many algorithms that we have learned about in this book.

Reinforcement learning with artificial neural networks offers a powerful way to handle complex decision-making problems where rules are not explicitly known, and the solution must be discovered through “trial and error”. By continually adjusting its predictions based on outcomes, the network learns not only to react, but to strategize, balancing short-term gains against long-term rewards. This approach opens up new possibilities for intelligent agents in dynamic, unpredictable environments, from robotics to game AI and beyond.

## 10.5 Use cases for reinforcement learning

Reinforcement learning excels at solving complex problems where an agent must make a sequence of decisions to achieve a goal. The key to applying RL is not simply the absence of a static dataset, but rather the ability to learn through active trial and error. This approach is most powerful in situations where we can both simulate a realistic environment at scale and define a clear reward function that provides feedback on the agent's actions. By interacting with this environment, taking actions and receiving rewards or penalties, the agent learns the optimal strategy over time. The use cases for this approach are potentially endless, and this section will describe some of the most popular ones.

### 10.5.1 Robotics

Robotics involves creating machines that interact with real-world environments to accomplish goals. Some robots are used to navigate difficult terrain with a variety of surfaces, obstacles, and inclines. Other robots are used as assistants in a laboratory, taking instructions from a scientist, passing the right tools, or operating equipment. When it isn't possible to model every outcome of every action in a large, dynamic environment, reinforcement learning can be useful. By defining a greater goal in an environment and introducing rewards and penalties as heuristics, we can use reinforcement learning to train robots in dynamic environments. A terrain-navigating robot, for example, may learn which wheels to drive power to and how to adjust its suspension to traverse difficult terrain successfully. This goal is achieved after many attempts.

These scenarios can be simulated virtually if the key aspects of the environment can be modeled in a computer program. Computer games have been used in some projects as a baseline for training self-driving cars before they're trained on the road in the real world. The aim in training robots with reinforcement learning is to create more-general models that can adapt to new and different environments while learning more-general interactions, much the way that humans do.

### 10.5.2 Recommendation engines

Recommendation engines are used in many of the digital products we use. Video streaming platforms use recommendation engines to learn an individual's likes and dislikes in video content and try to recommend something most suitable for the viewer. This approach has also been employed in music streaming platforms and e-commerce stores. Reinforcement learning models are trained by using the behavior of the viewer when faced with decisions to watch recommended videos. The premise is that if a recommended video was selected and watched in its entirety, there's a strong reward for the reinforcement learning model, because it has assumed that the video was a good recommendation. Conversely, if a video never gets selected or little of the content is watched, it's reasonable to assume that the video did not appeal to the viewer. This result would result in a weak reward or a penalty.

### 10.5.3 Financial trading

Financial trading is a classic example of a sequential decision-making problem, making it a prime candidate for reinforcement learning. While vast amounts of historical market data are publicly available, this raises an important question: why not simply use supervised learning?

The answer is that trading isn't about making a single, isolated "correct" prediction. A supervised model might predict if a stock will go up tomorrow, but it can't tell you if buying today is part of a long-term winning strategy. The consequence of your action, owning the stock, changes your situation for all future decisions.

Reinforcement learning is designed for exactly this. An RL agent's state includes not just market data, but also its current portfolio (what it owns and its cash balance). It learns a policy by taking actions (buy, sell, hold) and receiving rewards or penalties based on the change in its portfolio's value. To avoid the risk of real financial loss, the agent is trained in a simulated market environment, a "sandbox", that uses historical data to let the agent practice and learn from the past without real-world consequences.

Although a reinforcement learning model could help generate a good return on investment, here's an interesting question: if all investors were automated and completely rational, and the human element was removed from trading, what would the market look like?

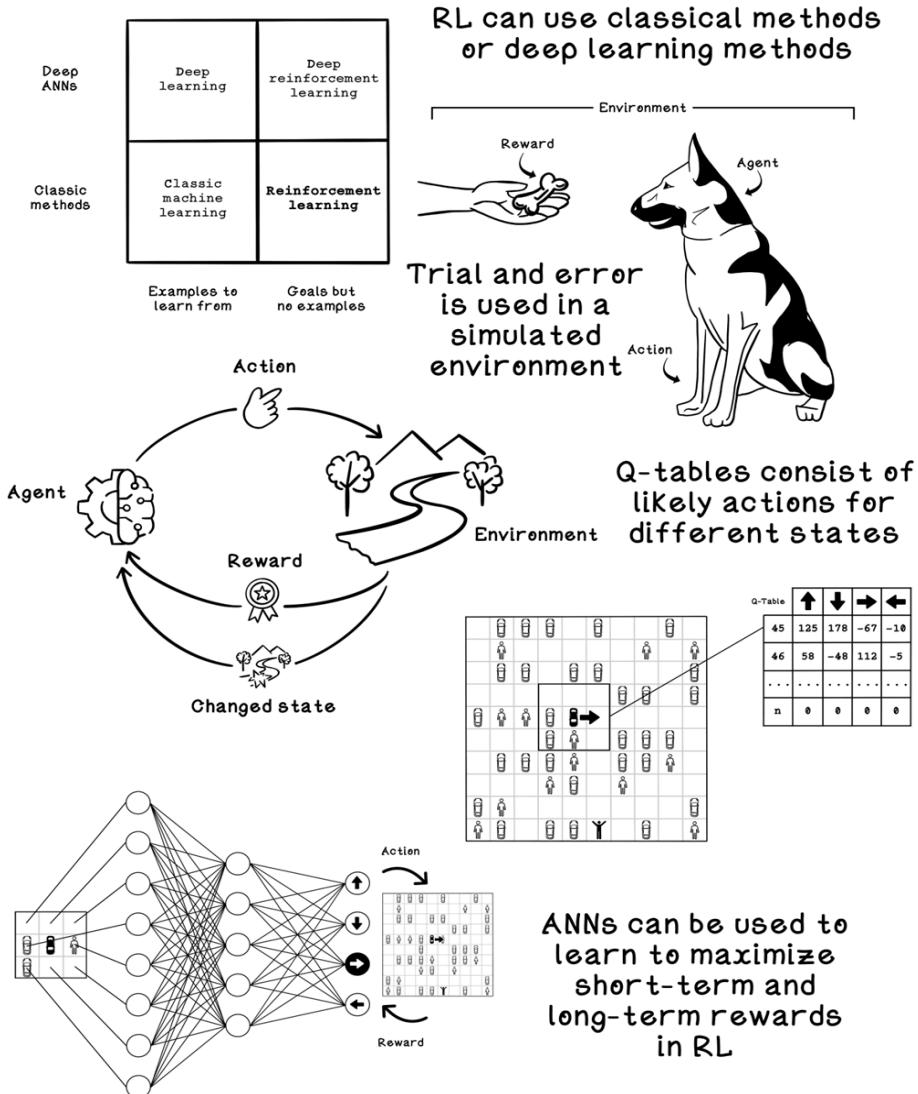
#### 10.5.4 Game playing

Popular strategy computer games have been pushing players' intellectual capabilities for years. These games typically involve managing many types of resources while planning short-term and long-term tactics to overcome an opponent. These games have filled arenas, and the smallest mistakes have cost top-notch players in many matches. Reinforcement learning has been used to play these games at the professional level and beyond. These reinforcement learning implementations usually involve an agent watching the screen the way a human player would, learning patterns, and taking actions. The rewards and penalties are directly associated with the game. After many iterations of playing the game in different scenarios with different opponents, a reinforcement learning agent learns what tactics work best toward the long-term goal of winning the game. The goal of research in this space is related to the search for more-general models that can gain context from abstract states and environments and understand things that cannot be mapped out logically. As children, for example, we never got burned by multiple objects before learning that hot objects are potentially dangerous. We developed an intuition and tested it as we grew older. These tests reinforced our understanding of hot objects and their potential harm or benefit.

AI research and development strives to make computers learn to solve problems in ways that humans are already good at: in a general way, stringing abstract ideas and concepts together with a goal in mind and finding good solutions to problems.

## 10.6 Summary of reinforcement learning

Reinforcement learning is useful when a goal is well defined but robust examples to learn from are not



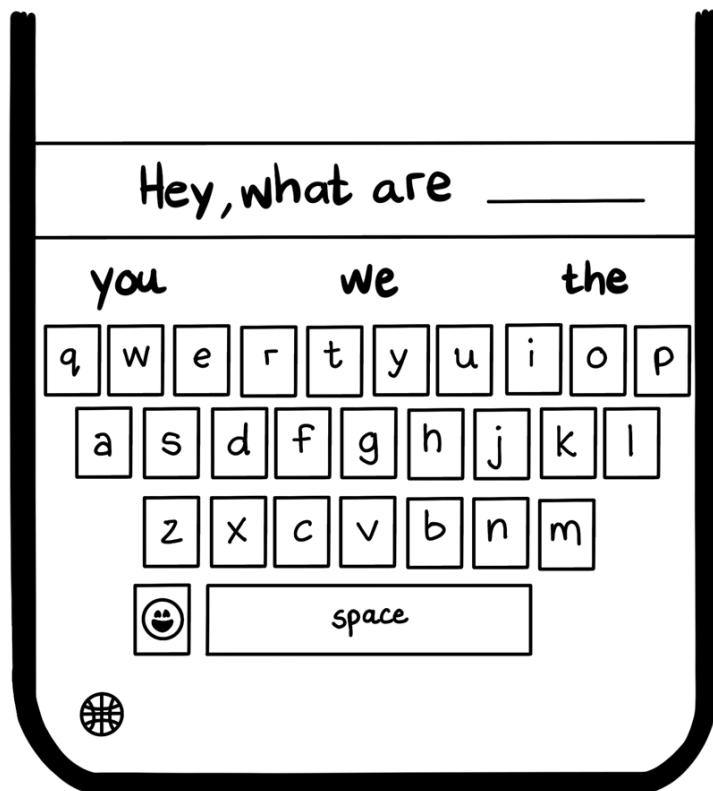
# 11 Large Language Models (LLMs)

## This chapter covers

- Understanding the intuition of large language models
- Identifying and preparing LLM training data
- Deeply understanding the operations in training a large language model
- Implementation details and LLM tuning approaches

### 11.1 What are large language models?

Large language models (LLMs) are machine learning models that are specialized for natural language processing problems, like language generation. Consider the autocomplete feature on your mobile device's keyboard (figure 11.1). When you start typing "Hey, what are...", the keyboard likely predicts that the next word is "you", "we", or "the", because these are the most common next words after the phrase. It makes this choice by scanning a table of probabilities that was trained on commonly available pieces of content - this simple table is a language model.



**Figure 11.1 Example of autocomplete as a language model**

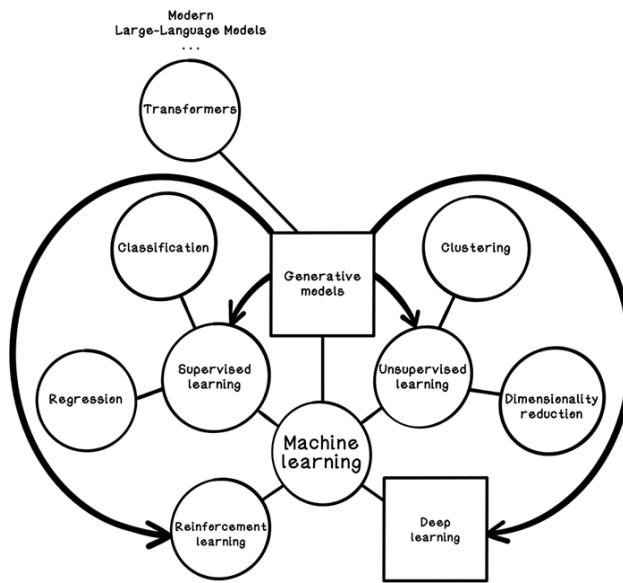
A large language model (LLM) is exactly the same idea, with some fundamental upgrades to enable interesting capabilities that come with predicting more than one word at a time:

- *More capacity:* Instead of a table of thousands of probabilities, an LLM holds billions (or trillions) of parameters. Parameters behave as adjustable “settings” that represent subtle patterns in language like relationships between words, vocabulary, grammar, facts, and even writing styles.
- *More training data:* Instead of the most popular and commonly available texts, a typical LLM is trained on entire libraries of content including web pages, novels, research papers, forum posts, code repositories, and more - often reaching terabytes in size.

- *More complex architecture:* Modern LLMs are built on the Transformer framework. While the original Transformer consisted of both an Encoder (to read) and a Decoder (to write), modern LLMs often use just one part (e.g., GPT models are Decoder-only). Regardless of the configuration, the core magic lies in the Transformer Block, which combines Attention Mechanisms (to focus on context), Feed-Forward Networks (to process information), and Normalization layers. The attention mechanism allows the model to decide which earlier words deserve focus when predicting the next ones - imagine reading a book, and highlighting important words or phrases as you go.

With these upgrades, LLMs are capable of performing far beyond predicting single words. They are able to generate coherent paragraphs, translate languages, answer questions, write code, and more possibilities are being unlocked each and every day.

The most popular architecture and approach for modern large language models are transformers; Figure 11.2 shows how they fit into the mental model of machine learning algorithms. Pay attention to how *Generative models* are a broader concept that leverages approaches from the other categories of machine learning algorithms that we have explored through this book, like *Unsupervised learning* (Finding relationships between words), *Supervised learning* (Determining if predicted words are correct), *Reinforcement learning* (Fine-tuning the model based on human preferences), and the broad category of *Deep learning*.



**Figure 11.2 How generative models and Transformers fit into machine learning**

## 11.2 The intuition behind language prediction

Before we explore the complete LLM training workflow, let's warm up with a practical example that we can use basic calculations and our intuition to solve. Suppose we have the following short sentences from the book *Alice's Adventures in Wonderland*.

1. Alice was beginning to get very tired
2. Alice was getting very sleepy
3. The white rabbit was late
4. The white rabbit was very late
5. Alice and the rabbit were friends

From simply reading these sentences, you might already recognize some patterns, like "Alice" and "The white rabbit" are always the nouns, "was" normally comes after "Alice" and after "The white rabbit", the adjective "very" is used a lot (figure 11.3). It's normal for us because our brains are pattern matching machines.

**Alice was beginning to get very tired**

**Alice was getting very sleepy**

**The white rabbit was late**

**The white rabbit was very late**

**Alice and the rabbit were friends**

**Figure 11.3 How we subconsciously identify patterns in language**

But let's quantify the words into something that a computer can understand and find patterns in using calculations.

A simple concept in language models is Bigrams. Bigrams are 2-word "sentences", for example "Alice was" or "white rabbit". If we consider the first word as the word we're making a prediction about, and the next word as a possible prediction, we can build a table of probabilities based on the sentences that we have available to train on.

Probabilities are calculated using the formula in figure 11.4. The count for a specific word is the total occurrences of a word (Next) after the word (Previous) divided by the sum of all occurrences of the word (Previous). This *Next* and *Previous* pairing is a Bigram.

Think of this like your phone's autocomplete. If you type "Alice", the model looks at its training data to see what word usually comes next. If "was" appears 90 times after "Alice", and "ran" only appears 10 times, the model assigns a 90% probability to "was" and suggests it as the next word.

$$P(\text{Next}|\text{Previous}) = \frac{\text{number of times Next occurs after Previous}}{\text{total occurrences that start with Previous}}$$

**Figure 11.4 Formula for probability of a word occurring after another word**

Table 11.1 details all bigrams in our dataset of five sentences and their respective probabilities.

**Table 11.1 Bigram counts for the five sample sentences**

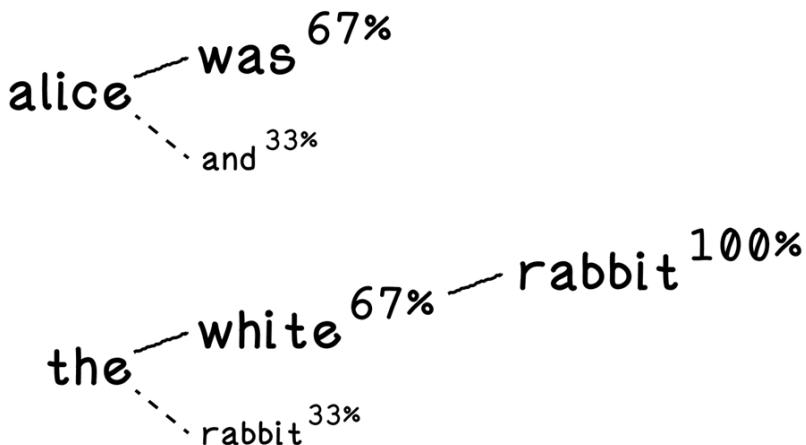
	<b>Previous word</b>	<b>Next word</b>	<b>Occurrence count</b>	<b>Probability calculation</b>	<b>Probability</b>
1	alice	was	2	2/3	0.67
2	alice	and	1	1/3	0.33
"alice" occurrence count			3		
3	was	beginning	1	1/4	0.25
4	was	getting	1	1/4	0.25
5	was	late	1	1/4	0.25
6	was	very	1	1/4	0.25
"was" occurrence count			4		
7	beginning	to	1	1/1	1
8	to	get	1	1/1	1
9	get	very	1	1/1	1
10	very	tired	1	1/3	<b>0.33</b>
11	very	sleepy	1	1/3	<b>0.33</b>
12	very	late	1	1/3	<b>0.33</b>
"very" occurrence count			3		
13	getting	very	1	1/1	<b>1</b>
14	the	white	2	2/3	<b>0.67</b>
15	the	rabbit	1	1/3	<b>0.33</b>
"the" occurrence count			3		
16	white	rabbit	2	2/2	<b>1</b>
17	rabbit	was	2	2/3	<b>0.67</b>
18	rabbit	were	1	1/3	<b>0.33</b>
"rabbit" occurrence count			3		
19	and	the	1	1/1	<b>1</b>

20	were	friends	1	1/1	<b>1</b>
----	------	---------	---	-----	----------

Now, using this table of probabilities, we can make predictions.

Given the word “alice”, the model has two options for the next word: “was” with a probability of 0.67, and “and” with a probability of 0.33. This means that the next word predicted will be “was” with the probability of 67%, so the result is “alice was” (figure 11.5).

Given the word “the”, the options are “white” with a probability of 0.67 and “rabbit” with a probability of 0.33. So, the completion will be “the white” with the probability of 67%, then for the next word, “white” has only one option with a probability of 100%, “rabbit”. Progressing to the sentence “the white rabbit”.



**Figure 11.5 Example of the probability of words occurring**

You can see how this tiny example of just 5 sentences and 29 “tokens”, can start producing a semblance of intelligent behavior that’s expected from a language model.

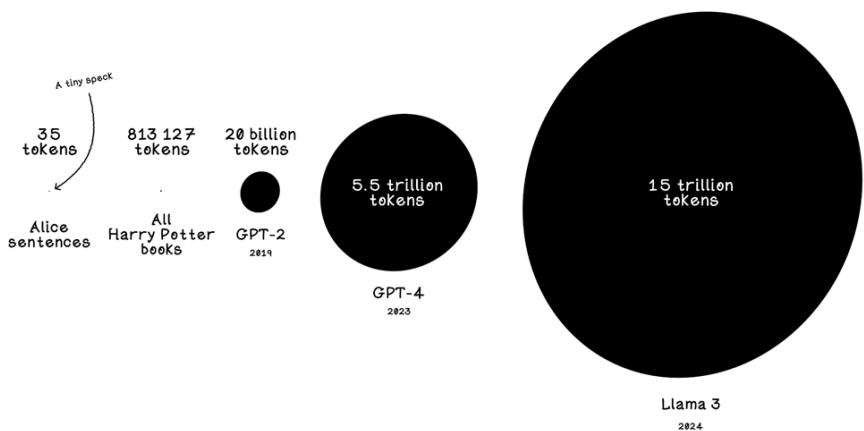
A token in this example is a single word in the 5 sentences. There are 29 tokens (words) total. A token is the smallest unit of text the model processes. While often a whole word, tokens can also be parts of words (like “ing”) or even punctuation, depending on the tokenizer used.

A simplified understanding of a “parameter” is each row in the table. There are 20 parameters. In actual LLMs parameters are weights in the Transformer used to train it.

On a conceptual level, LLMs use a similar counting and normalizing approach but on billions of tokens, allowing them to model much larger contexts. In the next sections we will build on this idea, swap the simple bigram count-table for trainable embeddings and attention layers, and explore how the same underlying principle scales up to modern LLMs.

### 11.2.1 Why the size of tokens and parameters matter

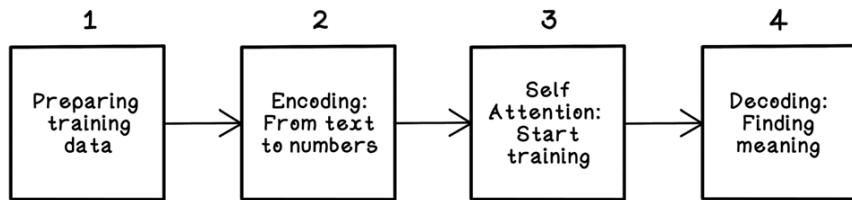
More tokens mean a larger training set to learn from; this means more weights in the artificial neural network and that gives the model room to store fine relationships between words, and other combinations of words, not just pairs. This means that the LLM can learn edge cases, rare words, and different writing styles - this powers the goal of generalization, where a model is able to answer any prompt sensibly. In the example that we will explore soon, we will be working with 35 tokens to train the LLM. However, modern production-grade LLMs have been trained with over 5 trillion tokens (figure 11.6).



**Figure 11.6 Comparison of size of training sets in popular language models**

### 11.2.2 An LLM training workflow

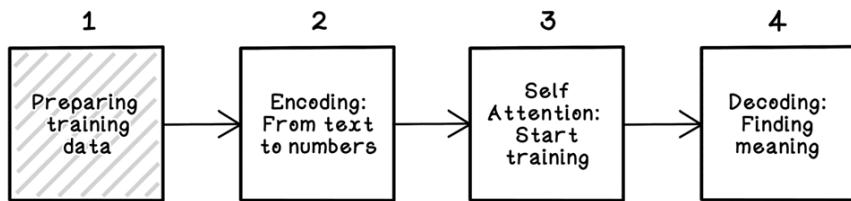
Training an LLM is conceptually similar to any other machine learning endeavor. We need to collect and prepare the right data, architect the model towards the goal we're trying to achieve, and train a model using that data (figure 11.7).



**Figure 11.7 An overview of the LLM training workflow**

1. *Prepare training data:* Before we can teach a model anything, we need to build its library. This first step involves gathering and cleaning a massive and diverse dataset of text (including books, research papers, web forum discussions, code, and more), which will serve as the foundation for all the knowledge the model will eventually learn.
2. *Encoding: From text to numbers:* Computers don't understand words, so this step acts as a translator. We create a vocabulary of unique tokens (words or parts of words) and then convert our entire text dataset into a long sequence of corresponding numerical IDs that the model can process mathematically.
3. *Self-attention: Start training:* This is where the core learning happens. The model repeatedly processes the numerical data using the self-attention mechanism, which allows it to look at all the tokens in a sequence and learn the complex relationships between them, figuring out which words are important to each other in different contexts.
4. *Decoding: Finding meaning:* After the model has learned these relationships, it holds its understanding in a series of complex vectors. The decoding step is the process of taking these internal numerical representations and projecting them back into human-readable language, allowing the model to make predictions and generate new text based on the patterns it has found.
5. *Controlling the LLM:* Like with any machine learning project, we need to understand what hyperparameters we can adjust like, the configuration of the transformer, stacking transformers, adjusting attention heads, epochs, and more.

## 11.3 Preparing training data



**Figure 11.8 Preparing training data step in the LLM training workflow**

Before any training can happen, we need to select and prepare the data that we want the LLM to train on and be knowledgeable in (figure 11.8). After reading Chapter 8, you should understand the importance of preparing data and the various techniques available for preparing discrete and continuous data. Preparing data for LLMs is similar, but requires a massive amount of data to make the model perform anywhere close to satisfactory. The main steps in preparing data to train LLMs are selecting and collecting data, and cleaning and preprocessing that data. From reading Chapter 8 and chapter 9, you probably have the intuition of the importance of selecting the right data given the question that you're trying to answer, however, with LLMs that question is a big one - understanding an entire language and the domain-specific meanings and knowledge that the language expresses. Cleaning the training data is equally as important as with numeric data.

### 11.3.1 Selecting and collecting data

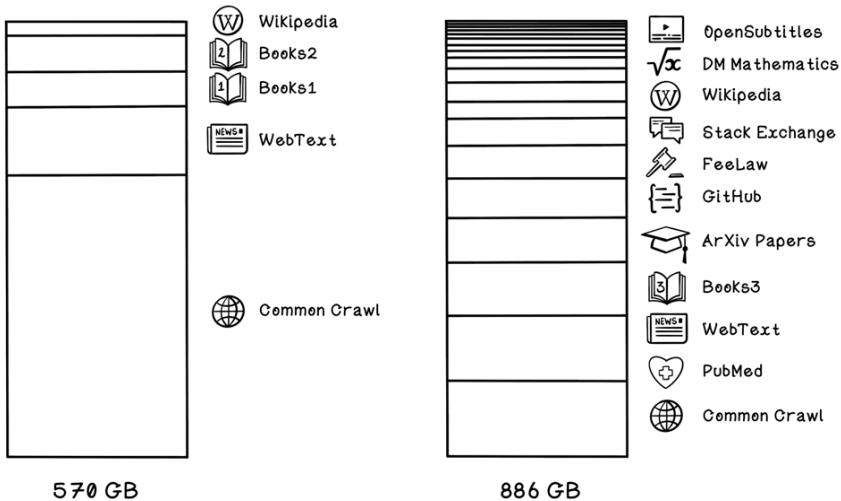
The LLM will become as broad-minded or narrow-minded as the data that it's trained on. The goal when selecting the data to train with is to choose data that is broad enough to teach general language skills, and relevant enough in the domain that you specifically care about. Any well performing LLM requires many general texts to understand a specific language, like English. And if you're interested in the LLM performing well in a specific domain like rocket science, it would need to be trained on a wealth of texts about that discipline. Here are some sources that powerful modern LLMs are trained on:

- *Public domain books*: These are literary works where the copyright has expired, making them free to use. Sources like Project Gutenberg provide a wealth of high-quality, well-edited texts that helps the model learn grammar, vocabulary, and coherent and long-form narrative structures.
- *Web crawling*: This is the process of using automated programs to download vast amounts of text from the public internet. While this is the best way to achieve massive scale and capture a wide range of current topics, the resulting data is very noisy and contains everything from high-quality articles to repetitive spam, requiring significant cleaning.

- *User-generated text:* This includes content from public forums, Q&A sites, and social media platforms. This type of data is invaluable for teaching the model how real people have conversations, how to answer questions, and the specific jargon used in different communities, like technical problem-solving or casual discussions.
- *Domain-specific text:* This refers to large collections of text focused on a single field, such as legal documents or medical textbooks. Including this type of data is crucial for training a model to be an expert in a specific domain's concepts and jargon.
- *Code repositories:* Sources like GitHub provide access to millions of public software projects. This data is essential for teaching a model how to understand, write, and debug code across a wide variety of programming languages, and common design patterns, and likely implementations of common functions.
- *News archives:* These are collections of public articles from news organizations, often spanning many decades. They are a valuable source of high-quality, factual information about historical and current events, which helps ground the model's knowledge of the world.

## VOLUME AND QUALITY OF TOKENS

Different LLMs have different goals. For example, you might want to make an LLM with the intention of helping make a breakthrough in scientific research. A good approach might be to include many academic papers from different specialty fields like medicine, mathematics, computer science, engineering, law, etc. However, the resulting model might come across as very formal when interacting with it - since research papers are intended to be formal and clear. Additionally, although the dataset on the right in figure 11.9 has more sources, we might not know the quality of the data extracted from those sources. Pure size of the data doesn't matter as much as the quality of the data. A smaller curated dataset could be better than a large uncurated dataset.



**Figure 11.9 Sample training datasets and their domain composition**

Although accessing information on the internet is easy, there are complexities in using that information to train your LLM. With current intellectual property laws (IP laws), not everything you see is free to use as training data.

## LICENSING OF CONTENT

Much of the text and media we find online is protected by copyright, which means it can't be freely used to train a model, especially for commercial purposes. To avoid legal issues, it's best to rely on data that is in the public domain (like old books where the copyright has expired), content with permissive licenses that allow for this kind of use, or data that has been explicitly licensed from the owners. Specialty content becomes more valuable as the LLM landscape evolves. If a model can be trained on special knowledge not available to others, it might have an edge.

For working through the LLM training algorithm, we will use the open-source text *Alice's Adventures in Wonderland*.

### 11.3.2 Cleaning and preprocessing data

Similarly to datasets in previous chapters, we need to clean our text data from the *Alice's Adventures in Wonderland* book. For the sake of brevity, we will use the first paragraph of the book only, but these principles will apply to the entire text within the book and all training data in reality.

Figure 11.10 shows an excerpt of the original text from the book that we will modify as we progress in data preparation:

\*\*\* START OF THE PROJECT GUTENBERG EBOOK ALICE'S ADVENTURES IN WONDERLAND \*\*\*  
 [Illustration]

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice "without pictures or conversations?"

**Figure 11.10 Excerpt of training data from Alice's Adventures in Wonderland**

## LANGUAGE FILTERING

If there are multiple languages, the model needs to juggle multiple vocabularies and grammars in the same weights of the Transformer. Less common languages get drowned out, and common languages get trained poorly. Having multiple languages also increases the number of tokens drastically - you'll see why this is important in the next section on tokenizing data.

For our example, the book is already in English, so we don't need to do any language filtering. But in a real-world dataset, you will need to filter out languages that you are not interested in.

## BOILER-PLATE STRIPPING

Some data might contain boiler-plate content like licensing declarations, HTML tags, cookie banner code from websites, or menu code from a website, among other noise. This content provides no semantic meaning in the context of the domain that you might be trying to train for. This content can be stripped out - reducing the token size, and reducing the model learning strange language relationships from repeated content.

Notice that in the original text example, there are two boiler-plate lines that don't actually provide any context related to the meaning of the story. This should be stripped out, resulting in the text in figure 11.11.

### Boiler-plate stripped

Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice "without pictures or conversations?"

**Figure 11.11 Training data with boiler-plate stripped**

## NEAR-DUPLICATE REMOVAL

Sometimes the same content is mirrored in many places that the training data might have been sourced from. For example, a quote may appear hundreds of times on many forums and blogs. The model will assign an artificially high probability to that piece of text, and will start to provide that text verbatim, even when it's not the correct response. By removing duplicates, we encourage the network to learn how to generalize meaning instead of memorizing specific phrases.

In our example, there are no duplicates.

## SAFETY AND COMPLIANCE FILTERS

Another area to look at is removing profanity, slurs, toxicity (toxicity means highly opinionated views that may be harmful), and personally identifiable information (PII). We normally don't want our model to use obscene words - this carries a reputational risk, and could also be a poor experience for the users of the model. Training data could contain phone numbers, credit card numbers, and other personally identifiable information that we would not want our model to reproduce due to data protection laws, so stripping out this data is important.

## NORMALIZATION

Normalization involves standardizing the content of the text in terms of special characters, white space, and dealing with emojis. café and CAFÉ may look the same to us humans, but they are two different tokens, because the "e" is not the same in both words. Here are some common operations for normalization.

- *Lower-case all the words:* This ensures the model treats the same word, like "The" at the start of a sentence and "the" in the middle, as a single token. It also reduces the size of the vocabulary the model needs to learn and helps it generalize better. Lowercasing is common in traditional NLP pipelines to reduce vocabulary size; in modern NLP pipelines (especially for LLMs), casing is preserved because capitalization carries meaning (e.g., apple vs Apple, us vs US). For simplicity in our example, we will lowercase the training data.
- *Strip smart quotes:* Word processors often create curly "smart" quotes, which are different characters from the standard straight " quotes. This step converts them all to the standard version to ensure consistency across the dataset.
- *Collapse white space:* Text from different sources might have extra spaces, tabs, or line breaks for formatting. This process replaces any sequence of these with a single space, creating clean and uniform separation between words.
- *Handle accented characters:* This involves deciding how to handle characters with diacritics, like in "café". You might want to convert them to their basic alphabet equivalents, like "cafe", to simplify the vocabulary, or keep them to retain their original meaning in other languages.

For our example, the normalized text appears in Figure 11.12. Notice that all characters are lowercase, opening and closing quote characters are normalized to normal quote characters, all white space is single white space characters only, and special characters have white space around them to identify them as their own semantic symbol.

alice was beginning to get very tired of sitting by her sister on the bank,  
and of having nothing to do : once or twice she had peeped into the book her  
sister was reading , but it had no pictures or conversations in it , " and what  
is the use of a book," thought alice "without pictures or conversations ? "

↳ Lowercased, single spaced,  
normalized special characters

**Figure 11.12 Training data normalized**

## DATA QUALITY AUDIT

Once our data has been collected and normalized, we can perform a data quality audit to ensure the dataset is clean, balanced, and suitable for our goals. This process involves calculating key metrics to check for common issues like repetitive text, toxic content, or formatting problems. By setting a target for each metric, we can systematically measure and improve the quality of our data before feeding it to the model. Table 11.2 is a checklist of areas that can be evaluated with goals for each.

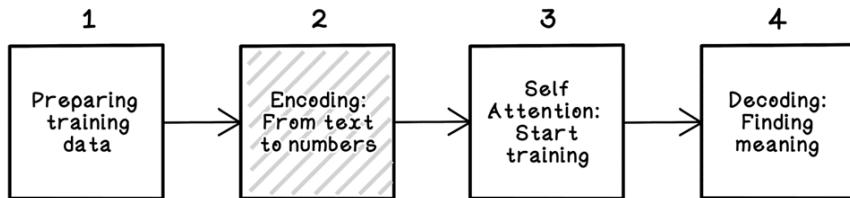
**Table 11.2 A summary of areas important for data quality**

Metric	Why it matters	Goals
Duplicate rate	Prevents memorization and reduces dataset size	< 5% exact-paragraph matches
Average sentence length	Very short sentences lack context, and very long sentences overflow the model window	5-40 tokens per segmented sentence
Toxicity rate	Keeps generation safe and not offensive	< 0.3% words flagged for toxicity
Profanity ratio	A more precise count apart from toxicity rate	Depends on the use case. < 0.2% for general models, but can be higher for "edgy" domains
Language purity	Mixed languages confuse the model and adds unnecessary vocabulary	> 98% of the content in the desired language
PII leaks	Reduces privacy and legal risks	0 emails, 0 phone numbers, 0 ID numbers
Non-UTF8 character rate	Binary junk characters can cause the tokenizer (explored next) to crash	0 control characters, realistically < 0.01% is good enough
Forbidden phrases	Prevents generating phrases that you don't want to see. E.g. "All rights reserved" or any phrase you deem fit	0 occurrences of forbidden phrases
Markup density	Markup like HTML and markdown noise wastes the context window and makes tokenizing less accurate	< 2% lines containing code characters like <, [ , ^ , > , ] , + , > , or links
Readability score	If you want to make the LLM generate text that is easy to read, e.g. write at a certain reading level	Flesch-Kincaid Grade-Level readability score (F-K score) is a formula to determine ease of readability. A score of 6 means that 11 year olds can read it easily. A score of 12 means that 17 year olds can read it easily

Domain balance	If you curate multiple domains (E.g. Medical, Fiction, Geography, etc.), you want to represent as intended	Not more than 10% variance per domain. E.g. if you planned 20% Medical content, your actual data should not be less than 10% or more than 30%
----------------	--	---

Note: There are reliable libraries that can be used for these tasks. You can get to know some of them by having a look at the accompanying Python repository to the book.

## 11.4 Encoding: From text to numbers



**Figure 11.13 Encoding step in the LLM training workflow**

Now that we have our clean training data, we face a fundamental challenge. Computers don't understand words, they understand numbers. For a model to learn from our text, we first need to convert it into a numerical format it can work with (figure 11.13). This entire process of translating text into a list of numbers is known as encoding.

### 11.4.1 Tokenization

As humans who understand the English language, our lowest building blocks for reading, writing, and processing the language are *words*, and punctuation. In "Alice was beginning to get very tired of sitting", we understand each word independently, and based on the rules of language, we understand nouns, like "Alice", verbs, like "sitting", and adjectives, like "very" and "tired". We also know the meaning of a word, like "beginning" is something different to "ending", but they are also related to each other as opposites.

LLMs don't understand language in the same intuitive way humans do. Instead, the input text undergoes a process called tokenization, where the text is broken down into a sequence of these smaller units.

For example, the sentence "Alice was beginning to get very tired of sitting" might be tokenized as: "Alice", "was", "beginning", "to", "get", "very", "tired", "of", "sitting". In this simple case, each token is a word. However, actual tokens end up looking quite different so the model can learn the core semantics of a language. For example, the word "unbelievable", could be broken down into the tokens "un", "believ", "able". This allows the model to handle a vast vocabulary, like "un" having a similar meaning in "unrealistic" or "unforgettable". Understanding that tokens are not necessarily words is an important intuition in LLMs. This allows the model to more flexibly and efficiently handle a large vocabulary.

Let's explore how text training data can be tokenized and used to build a vocabulary. We will work with just a paragraph of training data.

## CHARACTERS AS TOKENS

First, we break up each word in the sentence into individual characters. We also add a special end-of-word marker, </w>, after the last character of each original word (figure 11.14). This is an important step because it allows the algorithm to distinguish between a sequence of characters that occurs inside a word (like "in" in "spin") and one that makes up a word itself, like the word "in".

```
alice</w> w a s </w> b e g i n n i n g </w> t o </w> g e t </w> v e r y </w> t i r e d </w> o f </w>
s i t t i n g </w> b y </w> h e r </w> s i s t e r </w> o n </w> t h e </w> b a n k </w>, </w> a n d </w> o f
</w> h a v i n g </w> n o t h i n g </w> t o </w> d o </w> : </w> o n c e </w> o r </w> t w i c e </w> s h e
</w> h a d </w> p e e p e d </w> i n t o </w> t h e </w> b o o k </w> h e r </w> s i s t e r </w> w a s </w>
r e a d i n g </w>, </w> b u t </w> i t </w> h a d </w> n o </w> p i c t u r e s </w> o r </w>
c o n v e r s a t i o n s </w> i n </w> i t </w>, </w> " </w> a n d </w> w h a t </w> i s </w> t h e </w> u s e
</w> o f </w> a </w> b o o k </w>, </w> " </w> t h o u g h t </w> a l i c e </w> " </w> w i t h o u t </w>
p i c t u r e s </w> o r </w> c o n v e r s a t i o n s </w> ? </w> " </w>
```

Split into individual characters  
with end-of-word markers

Figure 11.14 Training data split into characters with end-of-word markers

## COUNTING ALL BIGRAMS

Remember, bigrams are 2-word sentences, and in this case, it is pairs of tokens that appear next to each other. The partial bigram frequency table for our sentence is Table 11.3. Notice that there are 8 occurrences of *i* and *n*. Some examples include: "beginning" with 2 occurrences, and "sitting" with 1 occurrence. However, the token pairs with the fewest occurrences are "s" and "?", appearing only in "conversations?", and "?" and "", appearing at the end of "...conversations??".

**Table 11.3 The initial bigram count for the training data**

	<b>Previous word</b>	<b>Next word</b>	<b>Occurrence count</b>
0	i	n	8
1	e	r	7
2	t	h	7
3	t	i	6
4	h	e	6
5	o	n	6
6	i	c	5
7	n	g	5
8	c	e	4
...			
24	a	t	3
25	a	l	2
...			
125	s	?	1
126	?	"	1

## MERGING THE MOST FREQUENT PAIR

Counting the bigrams is the first step of byte-pair encoding (BPE) the data. We can now merge the most common pairs into a single token and repeat the process. In the output shown in Figure 11.15, we have merged all occurrences of "i" and "n" since it has the most occurrences. Note: The end-of-word </w> markers have been omitted for readability in our example.

```

alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do:
once or twice she had peeped in to the book her sister was reading,
but it had no pictures or conversations in it,
"and what is the use of a book," thought Alice
without pictures or conversations?"
```

**Figure 11.15 Training data after one iteration of merging with BPE**

After repeating the process of counting bigrams with the new input, we have the bigram frequency table (Table 11.4). Notice that the new most frequent pair is “e” and “r”. And the pair “t” and “i” which had 6 occurrences in the first iteration, now only has 5 occurrences. And, there’s the introduction of our new token with the pair “in” and “g” which has 5 occurrences.

**Table 11.4 The bigram count after one iteration**

	<b>Previous word</b>	<b>Next word</b>	<b>Occurrence count</b>
0	e	r	7
1	t	h	7
2	h	e	6
3	o	n	6
4	i	c	5
5	in	g	5
6	t	i	5
7	c	e	4
8	r	e	4
...			
129	?	"	1

After merging our most frequent bigram which is “e” and “r”, we now have tokens that would look like the training paragraph in Figure 11.16.

```

alice was beginning to get very tired of sitting by her sister on the bank,
and of having nothing to do:
once or twice she had peeped into the book her sister was reading,
but it had no pictures or conversations in it,
"and what is the use of a book," thought Alice "without pictures or conversations?"

```

**Figure 11.16 Training data after two iterations of merging with BPE**

This process is done until we reach the stopping point. The stopping point is usually defined based on a desired vocabulary size - this is when the most frequent bigram count falls below a minimum threshold. If we iterate until there are no bigrams with frequency more than 1, for example, we might end up with tokens that don't make sense. In our example, going that route would result in one token being "picturesorconversations" - this is not a word or phrase commonly found in languages.

When we stop the iterations at a maximum of 3 iterations (figure 11.17), our final bigram count and result looks like Table 11.5.

**Table 11.5 The bigram count after two iterations**

	<b>Previous word</b>	<b>Next word</b>	<b>Occurrence count</b>
0	ic	e	3
1	t	o	3
2	d	o	3
3	o	f	3
4	th	e	3
...			
136	?	"	1

alice was beginning to get very tired of sitting by her sister on the bank,  
and of having nothing to do:  
once or twice she had peeped into the book her sister was reading,  
but it had no pictures or conversations in it,  
"and what is the use of a book," thought alice  
without pictures or conversations?"

**Figure 11.17 Training data after three iterations of merging with BPE**

### PYTHON CODE SAMPLE

This code implements a simplified version of Byte Pair Encoding (BPE) for tokenizing text. It starts by splitting words into characters and marking word boundaries. It then iteratively finds and merges the most frequent adjacent character pairs (bigrams), forming longer tokens until a target vocabulary size is reached. Finally, it builds a vocabulary of unique tokens and assigns each a unique integer ID, which can be used to convert text into model-ready numerical inputs.

```

def tokenize(text, vocabulary_size):
    words = text.split()
    current_tokens = []

    for word in words:
        current_tokens.extend(list(word) + ['</w>'])

    vocabulary = set(current_tokens)

    while len(vocabulary) < vocabulary_size:

        bigram_counts = collections.defaultdict(int)
        for i in range(len(current_tokens) - 1):
            pair = (current_tokens[i], current_tokens[i+1])
            bigram_counts[pair] += 1

        if not bigram_counts:
            break

        most_frequent_bigram = max(bigram_counts, key=bigram_counts.get)
        count = bigram_counts[most_frequent_bigram]

        if count <= 1:
            break

        new_token = "".join(most_frequent_bigram)
        next_tokens = []
        i = 0

        while i < len(current_tokens):
            is_match = (i < len(current_tokens) - 1 and
                        current_tokens[i] == most_frequent_bigram[0] and
                        current_tokens[i+1] == most_frequent_bigram[1])

            if is_match:
                next_tokens.append(new_token)
                i += 2 #A
            else:
                next_tokens.append(current_tokens[i])
                i += 1

        current_tokens = next_tokens
        vocabulary.add(new_token) #B
    
```

```

final_vocabulary = sorted(list(vocabulary))
token_ids = {token: i for i, token in enumerate(final_vocabulary)}

return final_vocabulary, token_ids

#A Skip the next token since it was just merged
#B Add the new merged token to the vocabulary

```

## ASSIGN IDS TO THE TOKENS

With our final set of bigrams, we can now assign integer IDs to each token. Integer IDs are important because algorithms work with numeric representations of data, so our text tokens can now be interpreted correctly. Table 11.6 is a table of all tokens and their respective IDs. There are 34 unique tokens, so our vocabulary size is 34.

**Table 11.6 The vocabulary based on the training data**

ID	Token	ID	Token	ID	Token
0	a	12	o	24	th
1	l	13	v	25	k
2	ic	14	er	26	,
3	e	15	y	27	:
4	w	16	ti	28	c
5	s	17	re	29	r
6	b	18	d	30	p
7	g	19	f	31	u
8	in	20	i	32	"
9	n	21	h	33	?
10	ing	22	ers		
11	t	23	on		

### 11.4.2 Vectorization

After building our vocabulary through tokenization, the next step is to use it to transform our entire training text into the numerical format the model requires for learning. This process is often called vectorization, as we are turning the text into a long vector, or token stream. Each number in this sequence corresponds to a specific token from our learned vocabulary.

## PACK TOKEN-IDS INTO TRAINING BATCHES

With our tokens represented as IDs, we can now represent our input as a token stream of the token IDs. Our original input training data, shown in figure 11.18...

alice was beginning to get very tired of sitting by her sister on the bank,  
 and of having nothing to do :  
 once or twice she had peeped into the book her sister was reading ,  
 but it had no pictures or conversations in it ,  
 " and what is the use of a book," thought alice  
 "without pictures or conversations ? "

**Figure 11.18 Original training data**

...becomes the token stream shown in figure 11.19:

```
a l i c e ...
\ \ / /
0 1 2 3 4 0 5 6 3 7 8 9 10 11 12 7 3 11 13 14 15 16 17 18 12
19 5 20 11 11 10 6 15 21 22 20 5 11 14 23 24 3 6 0 9 25 26 0 9 18
12 19 21 0 13 10 9 12 24 10 11 12 18 12 27 23 28 3 12 29 11 4 2 3 5
21 3 21 0 18 30 3 3 30 3 18 8 11 12 24 3 6 12 12 25 21 22 20 5 11
14 4 0 5 17 0 18 10 26 6 31 16 24 0 18 9 12 30 2 11 31 17 5 12 29
28 23 13 22 0 16 23 5 8 20 11 26 32 0 9 18 4 21 0 16 5 24 3 31 5
3 12 19 0 6 12 12 25 26 32 24 12 31 7 21 11 0 1 2 3 32 4 20 24 12
31 11 30 2 11 31 17 5 12 29 28 23 13 22 0 16 23 5 33 32
```

**Figure 11.19 Tokenized training data as a token stream**

Our token stream is now a complete numerical representation of the text, but it's far too long for us to work through in training our model all at once. LLMs have a fixed input size called a context window, which is the maximum number of tokens they can look at in a single pass. To prepare the data for training, we break this long stream into smaller, manageable chunks called batches. Each batch is a self-contained training example that fits within the model's context window. We can then create training batches based on a context window size. Our context window size is 32, but modern LLMs have context windows of 1 024 to 30 000 tokens. Figure 11.20 illustrates the 6 batches of tokens that we can use for training. We will use only one at a time.

```
[a, l, ic, e, w, a, s, b, e, g, in, n, ing, t, e, g, e, t, v, er, y, ti, re, d, e, f, s, i, t, t, ing, b]
Batch 0
[0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6] ↗
Batch 1
[15, 21, 22, 20, 5, 11, 14, 23, 24, 3, 6, 0, 9, 25, 26, 0, 9, 18, 12, 19, 21, 0, 13, 10, 9, 12, 24, 10, 11, 12, 18, 12] ↗
32 tokens in size
Batch 2
[27, 23, 28, 3, 12, 29, 11, 4, 2, 3, 5, 21, 3, 21, 0, 18, 30, 3, 3, 30, 3, 18, 8, 11, 12, 24, 3, 6, 12, 12, 25, 21]
Batch 3
[22, 20, 5, 11, 14, 4, 0, 5, 17, 0, 18, 10, 26, 6, 31, 16, 24, 0, 18, 9, 12, 30, 2, 11, 31, 17, 5, 12, 29, 28, 23, 13]
Batch 4
[22, 0, 16, 23, 5, 8, 20, 11, 26, 32, 0, 9, 18, 4, 21, 0, 16, 5, 24, 3, 31, 5, 3, 12, 19, 0, 6, 12, 12, 25, 26, 32]
Batch 5
[24, 12, 31, 7, 21, 11, 0, 1, 2, 3, 32, 4, 20, 24, 12, 31, 11, 30, 2, 11, 31, 17, 5, 12, 29, 28, 23, 13, 22, 0, 16, 23]
Batch 6
[5, 33, 32]
```

**Figure 11.20 Tokenized training data as training batches**

#### EXERCISE: WHAT ARE THE TOKENS FOR BATCH 1?

Determine the text tokens that map to the token IDs for batch 1.

#### SOLUTION: WHAT ARE THE TOKENS FOR BATCH 1?

[15, 21, 22, 20, 5, 11, 14, 23, 24, 3, 6, 0, 9, 25, 26, 0, 9, 18, 12, 19, 21, 0, 13, 10, 9, 12, 24, 10, 11, 12, 18, 12]

[y, h, ers, i, s, t, er, on, th, e, b, a, n, k, , a, n, d, o, f, h, a, v, ing, n, o, th, ing, t, o, d, o]

#### PYTHON CODE SAMPLE

This code segments a continuous stream of token IDs into fixed-size chunks called batches, based on a given `context_window_size`. It moves through the token stream in steps equal to the window size, slicing out sequences of tokens that fit within the model's attention span. These batches are returned as a list of token sequences, ready for input into the model during training or inference.

```
def batch_tokens(token_stream, context_window_size):
    batches = []

    start_index = 0
    while True:
        end_index = start_index + context_window_size

        if end_index > len(token_stream):
            break

        batch = token_stream[start_index:end_index]
        batches.append(batch)

        start_index += 1

    return batches
```

## 11.5 Designing the ANN architecture (And why transformers)

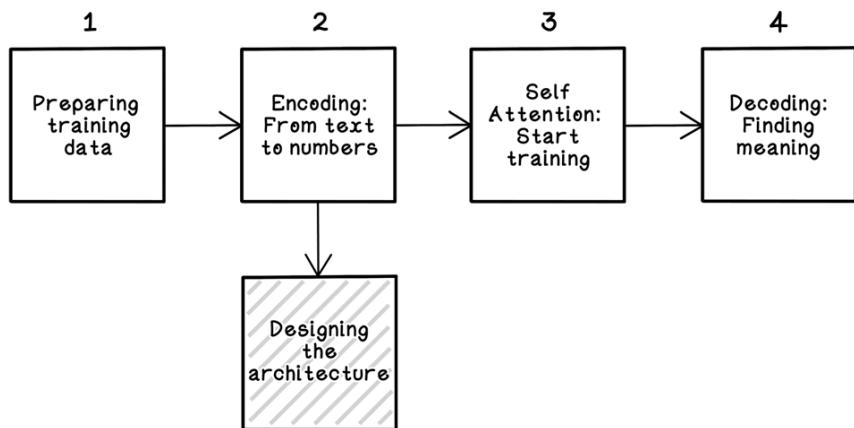


Figure 11.21 Designing the architecture step in the LLM training workflow

Now that we have our training data in a state that is understandable for computing and algorithms, the next step is to choose a training architecture (figure 11.21). In Chapter 9 we explored how artificial neural networks (ANN) take input data, learn the relationships between the inputs, and provide predictions based on those relationships. If we didn't use ANNs to train the LLM, we would need to come up with the rules and relationships between all the tokens in the vocabulary. Even in the simple example that we're using, this would require immense effort.

A major decision in training the LLM is choosing the ANN architecture to use. Different architectures have different trade-offs based on the problem you're solving. Three key concerns when choosing an architecture for training an LLM are the following:

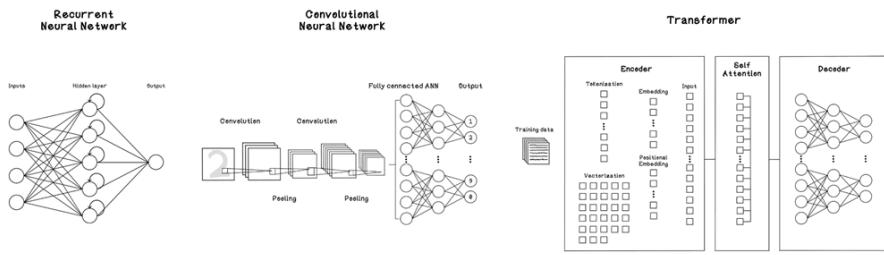
- *Memory capacity*: The amount of RAM or GPU memory required to hold the model's parameters. Larger and more complex models can capture more nuance but demand more memory, which often requires more expensive and specialized hardware.
- *Compute cost*: The sheer amount of processing power and energy usage needed to train the model. Training massive architectures on huge datasets is an enormous and expensive effort, sometimes taking weeks or months on clusters of thousands of GPUs.
- *Latency*: The delay between providing the model with an input and receiving its output. For applications like chatbots, low latency is critical for a good user experience, but larger, more capable models often take longer to generate a response, and the model's architecture dictates this.

In Chapter 9 we explored the different hyperparameters of artificial neural networks, and how they can be adjusted based on different problem spaces. In the context of the car collisions example in Chapter 9, a small artificial neural network was good enough to understand relationships between inputs and use that to make fairly accurate predictions based on a fairly small training dataset. However, you're probably gaining the intuition that the input tokens in an LLM are massive. In our small paragraph, we have a vocabulary of 34 tokens to find relationships between. A usable and "smart" LLM is trained with a vocabulary of 50 000 to more than 250 000 tokens. More tokens usually means more generalization in language understanding and diverse knowledge in different domains.

Three ANN architectures can be used to train an LLM (figure 11.22).

- *Recurrent Neural Network (RNN) / LSTM*: This architecture processes text sequentially, one word at a time, while maintaining a "memory" of what it has seen so far. Basic RNNs have short memories, but a more advanced version is LSTM (Long Short-Term Memory). It uses a sophisticated mechanism to remember context over much longer sequences, making it well-suited for language tasks. However, it suffers from an information bottleneck. It needs to summarize its entire understanding of the preceding text learned into its fixed-size memory vector—losing rich context of early text learned.

- *Convolutional Neural Network (CNN)*: Though famous for image processing, CNNs can be applied to text by using sliding windows to detect local patterns of words. They are effective for tasks like text classification where specific key phrases are important, but they are less suited for capturing the complex, long-range meaning of a sentence.
- *Transformers*: The modern, state-of-the-art architecture that powers virtually all modern large language models. Unlike an RNN, it processes all words in a sequence at once. Its key innovation is the self-attention mechanism, which allows every word to directly look at and weigh the importance of all other words in the sentence, enabling it to build a deep, contextual understanding of language.

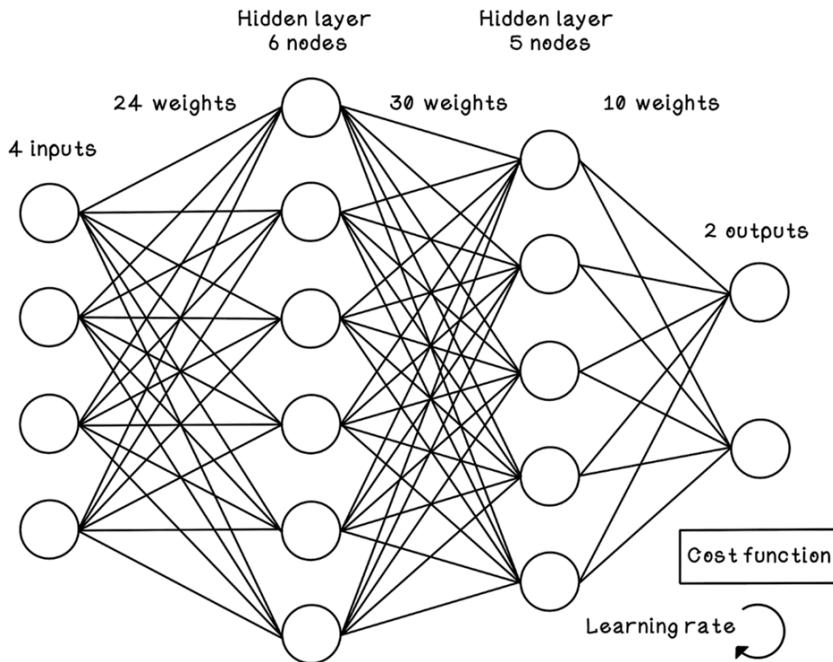


**Figure 11.22 Comparison of different ANN architectures aimed at training LLMs**

We will be training our LLM with the Transformer framework. We need to think about the different parameters of the different components of the Transformer.

- *Vocabulary Size*: The number of unique tokens after BPE merging. We've done this step already and know the number of tokens for our toy example, 34 tokens.
- *Context Window*: The number of tokens that a model can "see" at once. A larger window means more "memory" to work from. A 256 token window won't be able to write a novel, but it could write some sentences, since it would only have context of the last 256 tokens.
- *Width (Embedding)*: How many floating-point numbers describe one token in each layer. Wider embeddings hold more details about the meaning of words in different contexts.
- *Depth (Number of layers)*: Think of layers as steps in a reasoning chain. One layer looks left & right one time, 12 layers can "think a dozen steps ahead".
- *Attention Heads*: Think of it as the number of unique "patterns" or concepts that the model can notice at once.
- *Feed-forward Size*: This is the number of layers and nodes in the feed-forward artificial neural network.

As a refresher, the diagram Figure 11.23 illustrates a typical artificial neural network. These are great for learning from numeric inputs and finding deep relationships between the independent data, and further meaning from the correlations of that data until it can accurately make a prediction.

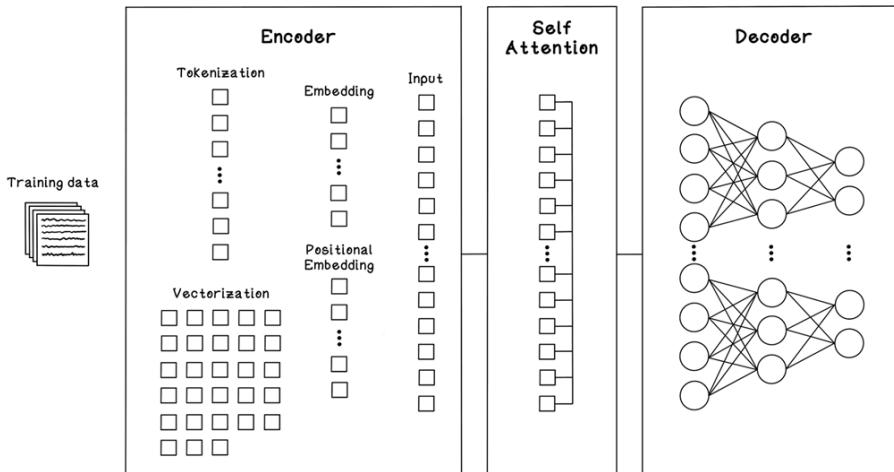


**Figure 11.23 The fundamental composition of an artificial neural network**

The Transformer architecture still leverages artificial neural networks, but has a number of steps before that to make text data efficient to be processed, while maintaining information about the original data, like position of tokens. Furthermore, Transformers have a step called *Self-attention* that is extremely beneficial in helping tokens understand language semantics before further “meaning” can be found. Figure 11.24 illustrates the components of a Transformer.

- *Encoder*: This encompasses the tokenization and vectorization process that we’ve already done. It also creates a data object that is suitable for self-attention and ANN processing by creating a numeric embedding of tokens and their positional data in relation to each other. Note: We’re using “encoder” here to refer to operations in the initial embedding step. Many modern LLMs are “decoder-only” and don’t use the classic, multi-layer Encoder architecture.

- *Self-attention*: This step lets all tokens figure out how they relate to all other tokens with a specific perspective in mind—for example, how tokens relate to other tokens that are nouns.
- *Decoder*: This encompasses the ANN that intends to derive meaning after the Self-attention step, and “decoding” back to human-understandable outcomes, resulting in a probability distribution of predictions for the next token in the sequence.



**Figure 11.24 A simplistic overview of the Transformer framework**

We've come to a point in this chapter where although we can work out the steps in the algorithm with small simple examples, having it perform with some level of intelligence is difficult with only a single paragraph of training data.

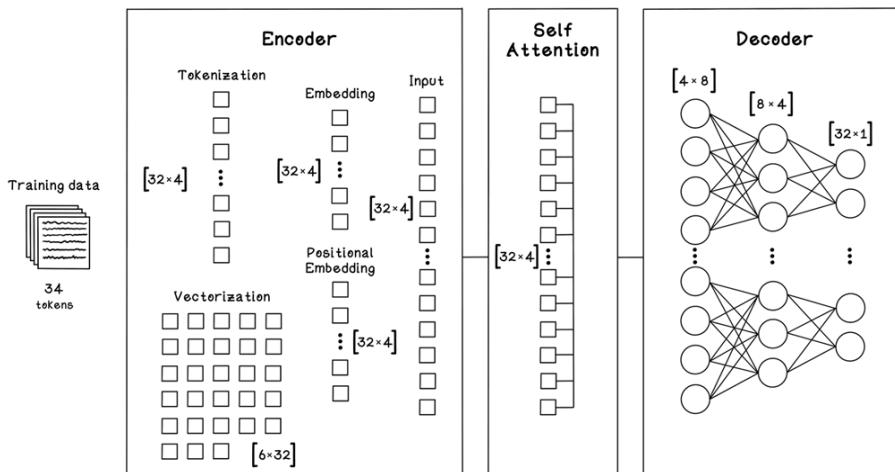
The power of LLMs comes from having massive language text datasets to learn from. This in turn impacts the number of tokens to process, which impacts the vocabulary size, which in turn influences the design of the Transformer, and the complexity of the calculations that need to be done. So from now on, we will still use the single paragraph to explore the steps and logic in the algorithm, but the results of calculations only move us a tiny bit closer to intelligence, rather than explicit progress like we've seen with the operations thus far.

Table 11.7 and Figure 11.25 show the details of the hyper-parameters and how they differ for our examples compared to real-world LLMs.

**Table 11.7 The hyper-parameters for the Transformer framework**

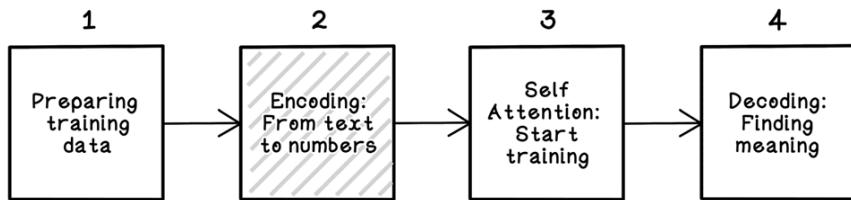
<b>Hyper-parameter</b>	<b>Rule of thumb</b>	<b>Toy LLM example</b>	<b>Real-world LLM</b>
Vocabulary Size (V)	Based on training data	34 sentence pieces	50 000 - Over 250 000 sentence pieces
Context Window (T)	Longest prompt you need, rounded to power of 2, that still fits batch on a GPU	32 tokens	1 024 - Over 32 000 tokens
Width (d)	Start near $4\sqrt{\text{Vocabulary size}}$	4 channels	768 - Over 20 000 channels
Depth (L)	1 - 2 layers per 100 000 characters of training text	2 layers	12 - Over 120 layers
Attention Heads (h)	$h = \text{Width} \div 64$ (round to even)	1 head	12 - Over 128 heads
Feed-forward Size ( $d_n$ )	$d_n = 4 \times \text{Width}$ is a safe default	Layer 1: 8 neurons Layer 2: 4 neurons	3 027 - Over 80 000 neurons

Figure 11.25 is the transformer diagram with the respective hyper-parameter values for the respective components.

**Figure 11.25 The data shapes for the different components in our Transformer framework**

## 11.6 Encoding: Creating trainable embeddings

With a basic understanding of CNNs and our Transformer architecture bedded down, we will now jump back into the encoding step (figure 11.26). Since we designed our Transformer to be a width of 4, it will influence how we further encode our training data. Let's dive into how our numeric token values become embeddings that can be used in the ANN.



**Figure 11.26 Continuation of the encoding step in the LLM training workflow**

We've converted our text into a sequence of token IDs, but these numbers are arbitrary. The ID 10 has no mathematical relationship to the ID 11, so the model can't learn from them directly. To solve this, we map each unique token ID to a list of numbers called an embedding vector. This vector isn't arbitrary; its values represent the token's meaning in a high-dimensional space. Initially, these vectors start as random, but they get adjusted and improved during training.

Think of Embeddings like a grocery store layout:

- “Apple” and “banana” are close together (fruit aisle).
- “Apple” and “apple pie” are somewhat close (the bakery is near the produce area).
- “Apple” and “laundry detergent” are on completely opposite sides of the store.

The vector is simply the “GPS coordinate” of the item in this massive store of meaning. The math calculates distance to see if two concepts are “related flavors” or totally distinct.

### 11.6.1 Sampling a batch of tokens

To walk through the training process, we need to select one of these batches to act as our sample input. Let's take the first one from our list of training data (figure 11.27). This single batch of 32 tokens will be the input that we pass through all the layers of our Transformer model.

```
[a, l, ic, e, w, a, s, b, e, g, in, n, ing, t, e, g, e, t, v, er, y, ti, re, d, e, f, s, i, t, t, ing, b]
Batch 0
[0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6] ↗
32 tokens in size
Batch 1
[15, 21, 22, 20, 5, 11, 14, 23, 24, 3, 6, 0, 9, 25, 26, 0, 9, 18, 12, 19, 21, 0, 13, 10, 9, 12, 24, 10, 11, 12, 18, 12]
Batch 2
[27, 23, 28, 3, 12, 29, 11, 4, 2, 3, 5, 21, 3, 21, 0, 18, 30, 3, 3, 30, 3, 18, 8, 11, 12, 24, 3, 6, 12, 12, 25, 21]
Batch 3
[22, 20, 5, 11, 14, 4, 0, 5, 17, 0, 18, 10, 26, 6, 31, 16, 24, 0, 18, 9, 12, 30, 2, 11, 31, 17, 5, 12, 29, 28, 23, 13]
Batch 4
[22, 0, 16, 23, 5, 8, 20, 11, 26, 32, 0, 9, 18, 4, 21, 0, 16, 5, 24, 3, 31, 5, 3, 12, 19, 0, 6, 12, 12, 25, 26, 32]
Batch 5
[24, 12, 31, 7, 21, 11, 0, 1, 2, 3, 32, 4, 20, 24, 12, 31, 11, 30, 2, 11, 31, 17, 5, 12, 29, 28, 23, 13, 22, 0, 16, 23]
Batch 6
[5, 33, 32]
```

**Figure 11.27 Sampling the first batch of tokens from the training data**

We will sample the first 32 tokens as a batch to work with:

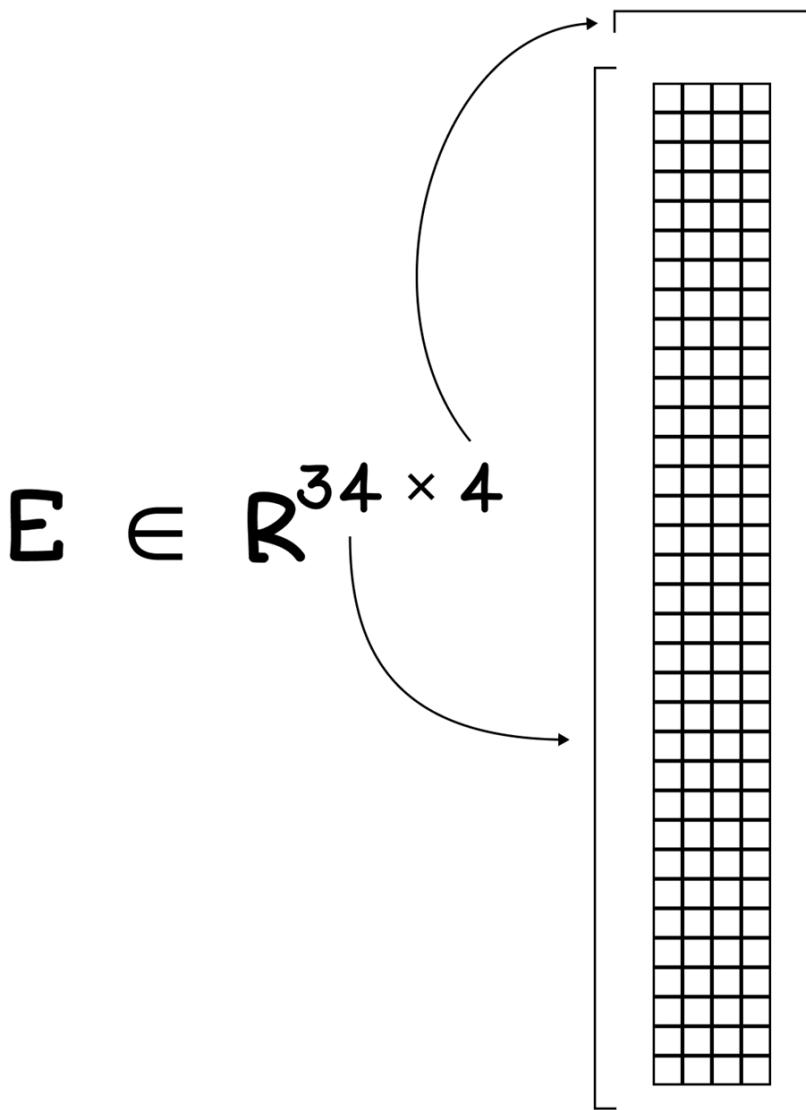
**[0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6]**

What happens to batch 6 since it has only 3 tokens and not 32? Since working with uniform data is more efficient on GPUs and results in consistent operations, batch 6 might be a problem. There are three possible ways to handle it:

- *It gets dropped*: This is the simplest and often the most common approach, especially with very large datasets. The loss of a tiny fraction of the data from the last incomplete batch is considered negligible and won't impact the final trained model.
- *It gets padded*: The batch is filled with a special "padding token" until it reaches the required size of 32. For example, it might become [5, 22, 32, PAD, PAD, ..., PAD]. The model is then specifically taught to ignore these padding tokens during its calculations.
- *It's used as-is*: Some model architectures can handle input sequences of variable lengths. In this case, the model could be fed the smaller batch of 3 tokens directly. However, this can sometimes be less computationally efficient than using fixed-size, padded batches.

### 11.6.2 Creating a trainable embedding matrix

If you hand a transformer the raw ID “17” it has no idea whether that is a verb, a comma, or something else. The network can only add and multiply real numbers, so every discrete ID must first be translated into a small vector of floating-point-numbers to represent its meaning; this is the *embedding*. That trainable lookup table is called the embedding matrix ( $E$ ). We represent the embeddings based on the vocabulary size ( $V$ ) and the width ( $d$ ). The expression in figure 11.28 is used to describe the respective values as a matrix.



**Figure 11.28 A view of how the vocabulary size and width dimensions shape the embedding**

We're choosing a width of 4 ( $d = 4$ ) for ease of understanding. And start with the values in the matrix being random numbers between -0.01 and 0.01, as long as each row starts with a different number.

During training these values will be “nudged”. For example, if “a” often precedes “l”, row 0 will shift closer to row 1 in the massive multidimensional space.

Table 11.8 shows the first two tokens, and the last token, and their respective random starting values.

**Table 11.8 A subset of tokens and their initial embedding values**

Position	Token ID	Token	x0	x1	x2	x3
0	0	a	0.0030	-0.0070	0.0120	0.0010
1	1	l	-0.0080	0.0050	-0.0020	0.0090
	...					
31	6	b	0.7381	-0.2512	-0.6719	0.7938

Note: We’re showing the values at most 4 decimal places for the sake of simplicity in learning. In actual training, the model relies on high-precision to work well, so numbers will have many decimal places.

### 11.6.3 Creating positional encodings

Positional encoding is a technique used to give the model information about the order of words in a sequence. It injects a “positional signal” into each token’s embedding. Self-attention, the next step coming soon (and the core mechanism of the Transformer), processes all words in a sentence simultaneously. By itself, it has no idea of order. To the self-attention layer, the sentences “The dog bit the man” and “The man bit the dog” would look like identical sequences of tokens.

To solve this, we add a unique positional vector to each token’s embedding. This ensures that the model can distinguish between the same token at different positions and understand the overall grammar and meaning of the sentence.

Now, given that the first batch is the following:

[0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6]

We can create the position encoding matrix. We need to calculate position values for the 32 tokens in our first batch. We will find 4 values for each because we have a width of 4.

Imagine a physical odometer on the inside of a car’s dashboard (this is the display that shows how many miles the car has travelled). Each number unit rotates on a dial to tick over to the next number. That means that there is a gear for each dial. If we had four dials, then we can represent four individual numbers that are all read as one number. We can store the rotation of each gear that lets the machine represent a chosen number. For example, in Figure 11.29, the number 2145 is represented as 42, 19, 22, 7, because those numbers represent the rotation of each gear to get the odometer to show the number 2145.

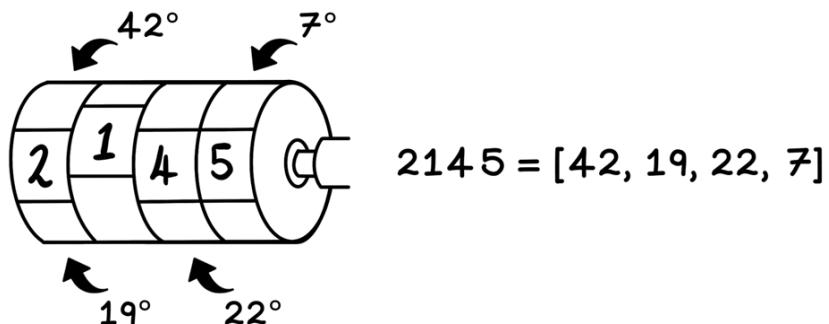


Figure 11.29 An odometer analogy for how sinusoidal positional encoding works

This is the intuition for representing the token position within the 4-width embedding that we're working with using the concept of *sinusoidal positional encoding*.

To find the values, we use the formula in figure 11.30:

$$\text{angle} = \left( \frac{\rho}{1000^{\frac{2\lfloor i/2 \rfloor}{\text{model dimensions}}}} \right)$$

↖

$2\lfloor i/2 \rfloor$  means  $2 \times$  floor of  $i/2$   
and rounds down to the  
nearest integer

So, because  $d = 4$

$$i = 0 \text{ angle: } \rho / 1000^{(2\lfloor 0/2 \rfloor / 4)} = \rho / 1$$

$$i = 1 \text{ angle: } \rho / 1000^{(2\lfloor 1/2 \rfloor / 4)} = \rho / 1$$

$$i = 2 \text{ angle: } \rho / 1000^{(2\lfloor 2/2 \rfloor / 4)} = \rho / 100$$

$$i = 3 \text{ angle: } \rho / 1000^{(2\lfloor 3/2 \rfloor / 4)} = \rho / 100$$

$$PE(\rho, i) = \begin{cases} \sin(\text{angle}) & \text{if } i \text{ is odd} \\ \cos(\text{angle}) & \text{if } i \text{ is even} \end{cases}$$

Figure 11.30 A breakdown of the calculations involved in sinusoidal encoding

Table 11.9 shows the tokens, and their respective calculations for finding their positional encoding.

**Table 11.9 A subset of tokens and their positional encoding calculations**

<b>Position (p)</b>	<b>Token ID</b>	<b>Token</b>	<b>Index (i)</b>	<b>Angle Formula</b>	<b>Formula</b>	<b>Positional encoding</b>
0	0	a	0	0/1	$\sin(0/1)$	0
			1	0/1	$\cos(0/1)$	1
			2	0/100	$\sin(0/100)$	0
			3	0/100	$\cos(0/100)$	1
1	1		0	1/1	$\sin(1/1)$	0.8415
			1	1/1	$\cos(1/1)$	0.5403
			2	1/100	$\sin(1/100)$	0.0100
			3	1/100	$\cos(1/100)$	1.0000
...						
31	6	b	0	31/1	$\sin(31/1)$	-0.4040
			1	31/1	$\cos(31/1)$	0.9147
			2	31/100	$\sin(31/100)$	0.3051
			3	31/100	$\cos(31/100)$	0.9523

This way of positional encoding was introduced in 2017 for the very first Transformer. The results provide uniqueness so that every position has distinct values, nearby positions have similar values, and can be adapted to different widths of embeddings. And it's nifty because any position can be reconstructed by knowing the width (d), position (p), and dimension inside the d-width embedding (i).

#### EXERCISE: WHAT ARE THE POSITIONAL ENCODINGS FOR TOKEN 2?

Calculate the positional encodings for token 2.

#### SOLUTION: WHAT ARE THE POSITIONAL ENCODINGS FOR TOKEN 2?

2	2	ic	0	2/1	$\sin(2/1)$	0.9093
			1	2/1	$\cos(2/1)$	-0.4161
			2	2/100	$\sin(2/100)$	0.0200
			3	2/100	$\cos(2/100)$	0.9998

### 11.6.4 Combining the embedding matrix and positional encodings

For each token in our batch, we now have two pieces of information; 1) its embedding vector (what the token means) and its positional encoding vector (where the token is in the sequence). We need to combine these into a single, information-rich vector that can be fed into the main Transformer architecture. This is done by simply adding the two vectors together, element by element—the two “meanings” are added to become one (figure 11.31).

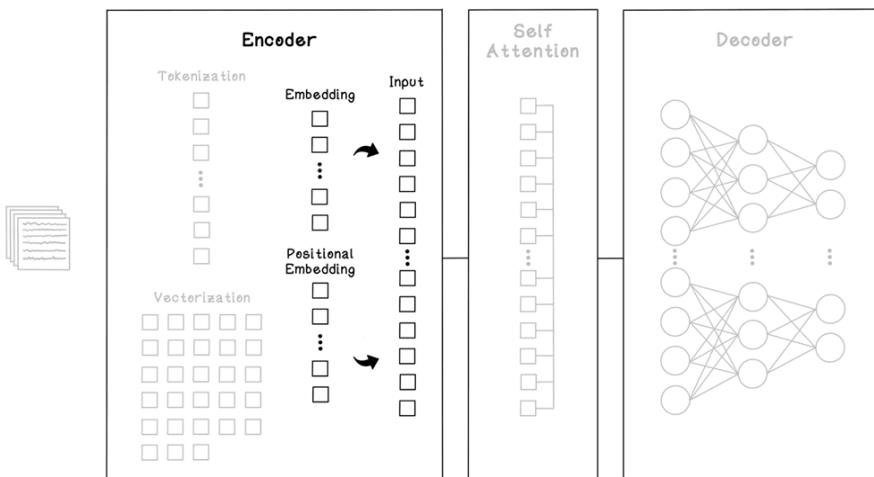


Figure 11.31 A view of the embedding combination step in the encoding stage

With the training batch being:

[**0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6**]

We do addition with embedding and positional encoding. Table 11.10 shows the embedding values for each token, and the positional values for each token, and the operation used to get the final result.

**Table 11.10 A subset of tokens and their embedding added to the positional encodings**

<b>Position</b>	<b>Token ID</b>	<b>Token</b>	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>
E for 0	0	a	0.003	-0.007	0.012	0.001
P for 0			0	1	0	1
X for 0			0.0030	0.9930	0.0120	1.0010
E for 1	1	I	-0.008	0.005	-0.002	0.009
P for 1			0.8415	0.5403	0.0100	1.0000
X for 1			0.8335	0.5453	0.0080	1.0090
...						
E for 31	6	b	0.7381	-0.2512	-0.6719	0.7938
P for 31			-0.4040	0.9147	0.3051	0.9523
X for 31			0.3341	0.6635	-0.3668	1.7461

The complete input data will result in a  $32 \times 4$  matrix, like the following:

**Table 11.11 A subset of tokens and their embedding results**

<b>ID</b>	<b>Token</b>	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>
X for 0	a	0.0030	0.9930	0.0120	1.0010
X for 1	I	0.8335	0.5453	0.008	1.009
...					
X for 31	b	0.3341	0.6635	-0.3668	1.7461

Our token embedding vector now represents the "what" (the concept of "alice"), and the positional encoding vector represents the "where" (position 0). When we added them, we created a new vector that holds both pieces of information simultaneously. In the high-dimensional space that the model operates in, there's plenty of space for both signals to coexist without corrupting each other. The model's subsequent layers (specifically, the linear weight matrices for Query, Key, and Value - these will be explored in the next section) are trained from scratch to understand this combined format. They learn that certain "directions" in this new vector space correspond to meaning, while other "directions" correspond to position.

#### EXERCISE: WHAT IS THE FINAL INPUT VECTOR FOR TOKEN 2?

Determine the final input vector for token 2.

**SOLUTION: WHAT IS THE FINAL INPUT VECTOR FOR TOKEN 2?**

<b>ID</b>	<b>Token</b>	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>
X for 2	ic	0.9093	-0.4161	0.0200	0.9998

**PYTHON CODE SAMPLE**

This code generates embedding vectors for a batch of tokens by combining two components: *token embeddings* and *positional encodings*. Each token ID is first mapped to a vector using an *embedding\_matrix*. To help the model understand the order of tokens, a sinusoidal *positional\_encoding* is calculated for each position in the sequence. These two vectors are then added together to form the *final\_embedding*, a matrix representing both the identity and position of each token, ready to be passed into the transformer layers.

```

def create_embedding(batch, embedding_width):
    vocab_size = VOCABULARY_SIZE
    context_window_size = CONTEXT_WINDOW_SIZE

    embedding_matrix = np.random.randn(vocab_size, embedding_width)

    positional_encoding = np.zeros((context_window_size, embedding_width))

    for position in range(context_window_size):
        for dimension in range(embedding_width):
            if dimension % 2 == 0:
                exp_term = dimension / embedding_width
                positional_encoding[position, dimension] = np.sin(
                    position / np.power(10000, exp_term))
            else:
                exp_term_cos = (dimension - 1) / embedding_width
                positional_encoding[position, dimension] = np.cos(
                    position / np.power(10000, exp_term_cos))

    final_embedding = np.zeros((len(batch), embedding_width))

    for token_position, token_id in enumerate(batch):
        token_embedding = embedding_matrix[token_id]
        position_embedding = positional_encoding[token_position]

        final_embedding[token_position] = token_embedding + position_embedding

    return final_embedding

```

## 11.7 Self-attention: Start training the LLM

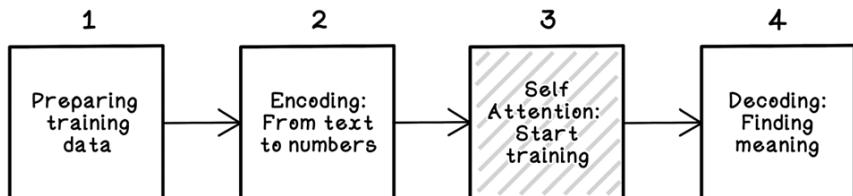
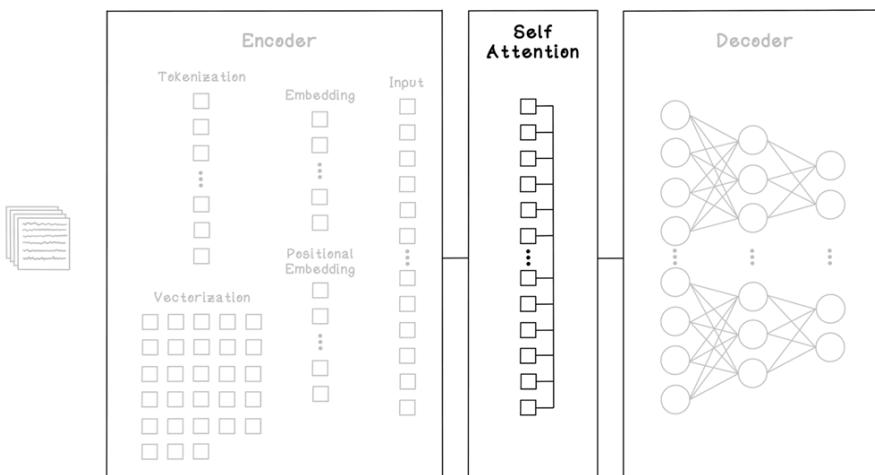


Figure 11.32 The self-attention step in the LLM training workflow

You're reading "Alice was beginning to get very tired..." with a highlighter pen in your brain. When you get to the word "tired", internally, your brain instantly goes back to "Alice" to know *who* is tired and back to "very" to know *how* tired. Self-attention gives the Transformer the same ability (figure 11.32). Every token peeks at other tokens in the current batch to decide how much each one should influence its new meaning.

The math behind self-attention all has a purpose. We follow the steps shown in Figure 11.33 for a single attention head. Think of heads as unique perspectives while figuring out meaning between tokens. Trivially, one attention head might be focusing on the *adjectives* and another might be focusing on the *nouns*, for example. For simplicity, we're working through the operations for a **single** attention head.



**Figure 11.33 Where self-attention fits into the Transformer framework**

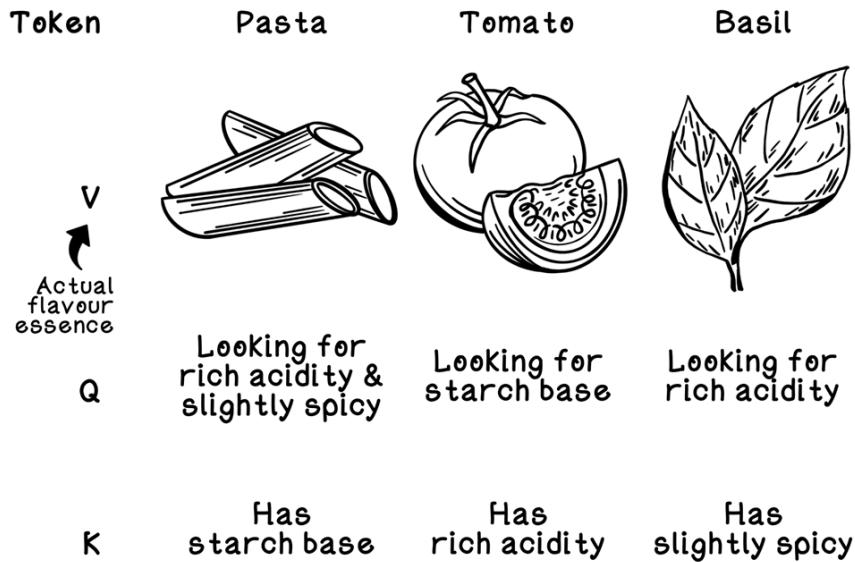
### 11.7.1 Linear weight matrix projections

The first step in self-attention is multiplying the embedding input by three matrices, namely Query (Q), Key (K), and Value (V). These are the self-attention weight matrices (starting off random, then resulting in learned values determined during training). These matrices transform each token into three specialized roles, making the process of comparing them much more effective.

Imagine each token as core ingredients, like pasta, tomato, basil, etc. (figure 11.34) The self-attention mechanism has three roles related to the ingredients:

- **Query (Q):** Creates a "shopping list". It's the matrix that looks at the ingredient, and decides what it needs from other ingredients to maximize its potential. Perhaps the tomato is "looking" for something with "starch base".

- **Key (K)**: Creates a “flavor label”. It’s the matrix that labels ingredients as “starch base”, or “rich acidity”, or “slightly spicy”.
- **Value (V)**: Creates a “flavor essence”. It’s the matrix that looks at the outcome of what ingredients would produce together. This is the actual contribution, not just the label, like K.



**Figure 11.34** An analogy of the purposes of Query, Key, and Value

During self-attention, each ingredient (Query) checks other ingredients’ flavor labels (Keys) to find complementary flavors. Once compatible ingredients are identified through matching the Query and Keys, their actual flavor essence (Values) are combined in a balanced way to achieve the best overall taste.

Replace this analogy with the concept of language, and you can see how flavor labels are the individual meaning of tokens (Key). The flavor essence is the actual meaning being applied together with other tokens (Value). And looking for matching flavor labels for one ingredient is akin to tokens looking for other related tokens (Query).

Q, K, and V start as tiny random numbers that allow for gradient descent to learn. For this example, we will pick random integers to illustrate how the calculations work (figure 11.35), but in an actual self-attention operation, these are random numbers between -1 and 1.

$$\begin{array}{l}
 Q = [2, 0, 1, 0 \\
 \quad 0, 2, 0, 1 \\
 \quad 1, 0, 2, 0 \\
 \quad 0, 1, 0, 2] \\
 \\
 K = [1, 0, 1, 0 \\
 \quad 0, 2, 0, 1 \\
 \quad 1, 0, 2, 0 \\
 \quad 0, 1, 0, 2] \\
 \\
 V = [1, 0, 0, 1 \\
 \quad 0, 1, 1, 0 \\
 \quad 1, 0, 1, 0 \\
 \quad 0, 1, 0, 1]
 \end{array}$$

**Figure 11.35 Initialization of the values for Queries, Keys, and Values**

Given token 0's input value as:

**Table 11.12 The final embedding for token 0**

ID	Token	x0	x1	x2	x3
X for 0	a	0.0030	0.9930	0.0120	1.0010

We add the random numbers that Q, K, and V were initialized with to token 0's values, as seen in Table 11.13, Table 11.14, and Table 11.15:

**Table 11.13 The Query values for token 0**

Q0=	$2 \times 0.0030 + 0 \times 0.9930 + 1 \times 0.0120 + 0 \times 1.0010$	0.0180
Q1=	$0 \times 0.0030 + 2 \times 0.9930 + 0 \times 0.0120 + 1 \times 1.0010$	2.9870
Q2=	$1 \times 0.0030 + 0 \times 0.9930 + 2 \times 0.0120 + 0 \times 1.0010$	0.0270
Q3=	$0 \times 0.0030 + 1 \times 0.9930 + 0 \times 0.0120 + 2 \times 1.0010$	2.9950

**Table 11.14 The Key values for token 0**

K0=	$1 \times 0.0030 + 0 \times 0.9930 + 1 \times 0.0120 + 0 \times 1.0010$	0.0150
K1=	$0 \times 0.0030 + 2 \times 0.9930 + 0 \times 0.0120 + 1 \times 1.0010$	2.9870
K2=	$1 \times 0.0030 + 0 \times 0.9930 + 2 \times 0.0120 + 0 \times 1.0010$	0.0270
K3=	$0 \times 0.0030 + 1 \times 0.9930 + 0 \times 0.0120 + 2 \times 1.0010$	2.9950

**Table 11.15 The Value values for token 0**

V0=	$1 \times 0.0030 + 0 \times 1.0070 + 0 \times 0.9930 + 1 \times 1.0010$	0.0150
V1=	$0 \times 0.0030 + 1 \times 1.0070 + 1 \times 0.9930 + 0 \times 1.0010$	1.9940
V2=	$1 \times 0.0030 + 0 \times 1.0070 + 1 \times 0.9930 + 0 \times 1.0010$	1.0050
V3=	$0 \times 0.0030 + 1 \times 1.0070 + 0 \times 0.9930 + 1 \times 1.0010$	1.0040

This will result in three new matrices where every token has a reference to its input value, its Q values, K values, and V values. Table 11.16 represents how these values relate in one simple view for ease of reference.

**Table 11.16A subset of tokens and their Query, Key, and Value values**

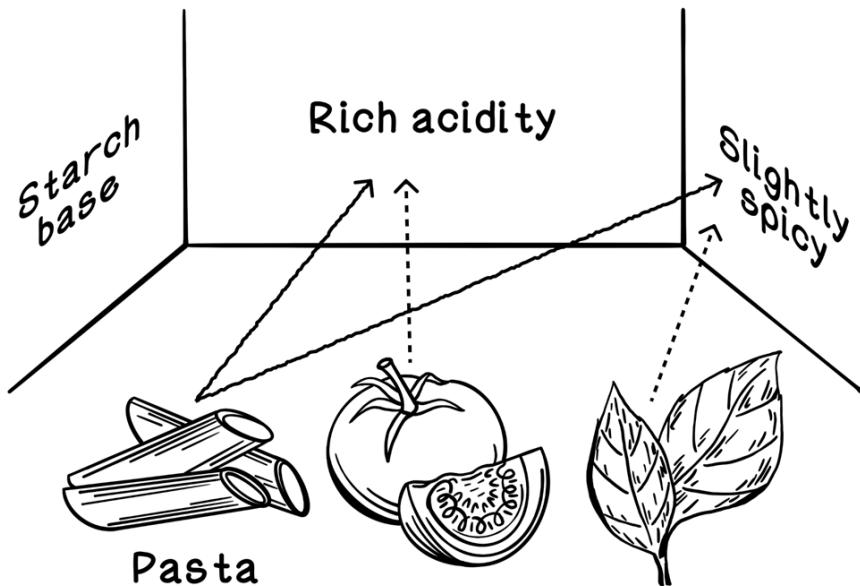
Position	ID	Token	x0	x1	x2	x3	q0	q1	q2	q3	k0	k1	k2	k3	v0	v1	v2	v3
0	0	a	0.0030	0.9930	0.0120	1.0010	0.018	2.9870	0.0270	2.9950	0.0150	2.9870	0.0270	2.9950	0.0150	1.9940	1.0050	1.0040
1	1	i	0.8335	0.5453	0.008	1.009	1.6749	2.0996	0.8495	2.5632	0.8415	2.0996	0.8495	2.5632	0.8415	1.5543	0.5533	1.8424
...																		
31	6	b	0.3341	0.6635	-0.3668	1.7461	0.3013	3.0732	-0.3996	4.1558	-0.0328	3.0732	-0.3996	4.1558	-0.0328	2.4097	0.2967	2.0802

## 11.7.2 Ask every other token

Now that we have initial values for Q, K, and V. We dot-product each Query with every other token's Key. This results in a raw "score" value. This determines "how relevant are you to me" for this attention head. We take the Q values for each row and dot-product it with each set of Keys.

The dot-product is useful to find angles between vectors in multidimensional spaces. Think about the dot product as a mathematical operation that calculates an "alignment score". If you're a chef for a *Query* vector, like Pasta, you are checking each and every other token's *Key*, to see how well they match. Imagine that the ingredients' flavor labels were written on all the walls of the kitchen in the order of relatedness. You point towards the Pasta's flavor label. There's a chef for each token's *Key* pointing to their respective flavor labels. You want to find how close each of them are pointing to where you are pointing. If you're both pointing to the same spot, there is high alignment, meaning that the *Key* is highly relevant to your *Query* - high score. If the other chef is pointing to the wall next to your wall, there is no alignment, with a score of 0. And if the other chef points to the wall behind yours, it's maximally misaligned, resulting in a negative score.

In Figure 11.36, we're concerned with Pasta and we have good matches for both Tomato and Basil.



**Figure 11.36 An analogy of how self-attention works**

Since we want to find the relevance between vectors, this operation is ideal. The angle suggests something like “Do my needs match your content”, and the length indicates “How confident am I?”. And as a bonus, GPUs compute dot-products extremely efficiently, making training faster than on traditional CPUs.

So with  $Q = [0.018, 2.9870, 0.0270, 2.9950]$  for token 0, we dot-product it with each set of K keys. We will have 32 results for token 0 as seen in Table 11.17.

**Table 11.17 A subset of tokens and their Q dot-product K results**

<b>0 (a)</b>	<b>Token ID</b>	<b>Token</b>	<b>Key Vector (Kq)</b>	<b>Score</b>
0	0	a	[0.0150, 2.9870, 0.0270, 2.9950]	17.8932
1	1		[0.8415, 2.0996, 0.8495, 2.5632]	13.9862
...				
31	6	b	[-0.0328, 3.0732, -0.3996, 4.1558]	21.6150

This operation must be done for each token in the embedding matrix. This will result in a  $32 \times 32$  sized matrix overall.

**EXERCISE: WHAT IS THE SCORE FOR TOKEN 2?**

Determine the score for token 2.

**SOLUTION: WHAT IS THE SCORE FOR TOKEN 2?**

<b>0 (a)</b>	<b>Token ID</b>	<b>Token</b>	<b>Key Vector (Kq)</b>	<b>Score</b>
2	2	ic	[0.8415, 2.0996, 0.8495, 2.5632]	-8.2245

**11.7.3 Calculating attention weights**

The next step is to scale the scores and run the scaled scores through softmax - a way of scaling values to emphasise their relative difference. They will then become attention weights where each value is between 0 and 1, and they sum to 1 in total—think of this as a pie chart of relevance for all the tokens. When the dimensions are small, like 4 in our example, the dot-products stay modest. However, in larger dimensions (64, 128, etc.), these numbers explode. If we don't scale the numbers, softmax doesn't behave well.

To scale the values, we use the formula in figure 11.37; remember our dimensions = 4.

$$\text{Scaled score} = \frac{\text{score}}{\sqrt{\text{dimensions}}}$$

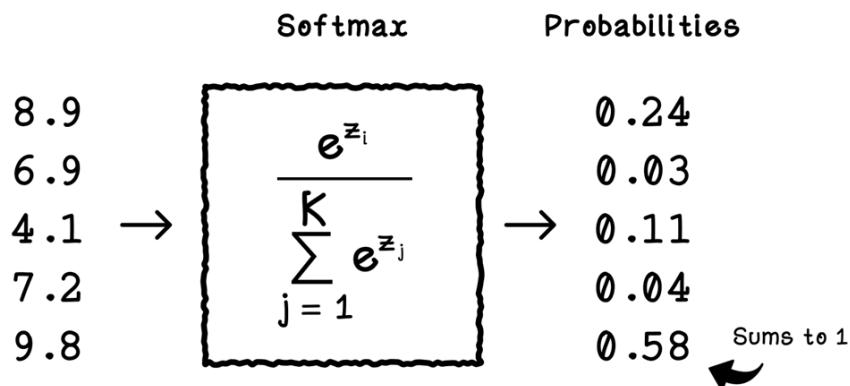
```
Scaled score = score / sqrt(dimensions) = score / sqrt(4)
Scaled score = score / 2
```

**Figure 11.37 The formula for scaling scores given the score and dimensions**

These calculations will result in the following for token 0:

**Table 11.18 A subset of tokens and their initial scaled scores**

<b>0 (a)</b>	<b>Token ID</b>	<b>Token</b>	<b>Score</b>	<b>Scaled score</b>
0	0	a	17.8932	8.9466
1	1	l	13.9862	6.9931
...				
31	6	b	21.6150	10.8075

**Figure 11.38 How softmax scales scores to probabilities**

Next is softmax. We do this by exponentiating the scaled score with  $e^{\text{scaled score}}$ . The reason exponentials work is to make every score positive, and it “magnifies” differences. A score two times larger becomes much larger, highlighting clear differences between scores.

Max value = **10.8075**

**Table 11.19 A subset of tokens and their scaled scores**

<b>0 (a)</b>	<b>Token ID</b>	<b>Token</b>	<b><math>e^{(\text{Scaled score} - \text{Max for token 0})}</math></b>	<b>Exponential</b>
0	0	a	$e^{(8.9466 - 10.81)}$	0.1555
1	1	l	$e^{(6.9931 - 10.81)}$	0.0221
...				
31	6	b	$e^{(10.8075 - 10.81)}$	1.0000

Then to softmax so all weight values summed equal to one, we take the exponential value divided by the sum of all values for all tokens (figure 11.38).

For token 0's Q: In our example, the total is 2.46. (This was calculated by the accompanying Python example.)

**Table 11.20 A subset of tokens and their softmax scaled scores**

<b>0 (a)</b>	<b>Token ID</b>	<b>Token</b>	<b>Exponential / Sum of all</b>	<b>Weight (After softmax)</b>
0	0	a	0.1555 / 2.46	0.0632
1	1	l	0.0221 / 2.46	0.0090
...				
31	6	b	1.0000 / 2.46	0.4064

We now have **attention weights** that have been scaled and normalized with softmax.

#### 11.7.4 Weighted sum

We have our attention weights, which act as a precise recipe for our token in question. The recipe tells us exactly how much influence every other token in the token sequence should have. We also have the Value vector (V) for each token, which represents its actual meaning or substance.

The next step is to use this recipe to create a new, blended representation. To do this, we first multiply each token's Value vector by its corresponding attention weight. This scales each token's contribution, making the influential ones stronger and the irrelevant ones weaker. Finally, we add up all of the new weighted vectors to get our final, context-rich output vector.

Remember that we created linear projections for Q, K, and V. And we used Q and K to create the dot-products for the original score. We now use the values for V and multiply each by the attention weights that we just calculated.

This table shows the tokens, their attention weights, original vectors, and results:

**Table 11.21 A subset of tokens and their result values**

<b>q</b>	<b>Weight</b>	<b>V0</b>	<b>V1</b>	<b>V2</b>	<b>V3</b>	<b>V0 result</b>	<b>V1 result</b>	<b>V2 result</b>	<b>V3 result</b>
0	0.0632	0.0150	1.9940	1.0050	1.0040	0.0009	0.1261	0.0635	0.0635
1	0.0090	0.8415	1.5543	0.5533	1.8424	0.0075	0.0139	0.0050	0.0165
...									
31	0.4064	-0.0328	2.4097	0.2967	2.0802	-0.0133	0.9794	0.1206	0.8455

Finally, add all results for each V as the context vectors. We add all values for V0, V1, V2, and V3 respectively to get the context vector for Q.

So given the following results for V for token 0:

**Table 11.22 A subset of tokens and their final values**

<b>q</b>	<b>Weight</b>	<b>V0 result</b>	<b>V1 result</b>	<b>V2 result</b>	<b>V3 result</b>
0	0.0632	0.0009	0.1261	0.0635	0.0635
1	0.0090	0.0075	0.0139	0.0050	0.0165
...					
31	0.4064	-0.0133	0.9794	0.1206	0.8455

We calculate the attention context vectors as shown in figure 11.39:

$$C0 = 0.0009 + 0.0075 + 0.0000 + \dots + -0.0133 = 0.2283$$

$$C1 = 0.1261 + 0.0139 + 0.0000 + \dots + 0.9794 = 2.1990$$

$$C2 = 0.0635 + 0.0050 + -0.0000 + \dots + 0.1206 = 0.4526$$

$$C3 = 0.0635 + 0.0165 + 0.0001 + \dots + 0.8455 = 1.9747$$

**Figure 11.39 A subset of the context vector calculations**

Token 0's final context is now:

**Table 11.23 Token 0 and their attention context vectors**

<b>q</b>	<b>token</b>	<b>C0</b>	<b>C1</b>	<b>C2</b>	<b>C3</b>
0	a	0.2283	2.1990	0.4526	1.9747

And we will do the same for the remaining 31 tokens. This will result in a 32 x 4 matrix.

**Table 11.24 A subset of tokens and their attention context vectors**

<b>q</b>	<b>token</b>	<b>C0</b>	<b>C1</b>	<b>C2</b>	<b>C3</b>
0	a	0.2283	2.1990	0.4526	1.9747
1	I	0.4641	2.0770	0.4670	2.0741
...					
31	b	0.2383	2.2812	0.3188	2.2007

You have now learned how to do the self-attention step in a LLM Transformer. There is loads of math, but these small steps combined create fascinating emergent intelligence where tokens are able to negotiate the importance of their relationships among each other.

## PYTHON CODE SAMPLE

This code implements scaled dot-product self-attention, a core operation in transformers. It starts by projecting the input embeddings into three matrices, Query (Q), Key (K), and Value (V), using learned weights. It then computes a matrix of attention scores by taking the dot product of each query with every key and scaling by the square root of the dimensionality. After applying the softmax function to get attention weights, these are used to blend the value vectors, producing the final attention output. This mechanism allows the model to selectively focus on different parts of the input sequence.

```
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))
    return e_x / np.sum(e_x, axis=-1, keepdimss=True)

def calculate_attention(embeddings, dimensions):
    query_weights = np.random.randn(dimensions, dimensions)
    key_weights = np.random.randn(dimensions, dimensions)
    value_weights = np.random.randn(dimensions, dimensions)

    Q = embeddings @ query_weights
    K = embeddings @ key_weights
    V = embeddings @ value_weights

    attention_scores = (Q @ K.T) / np.sqrt(dimensions)

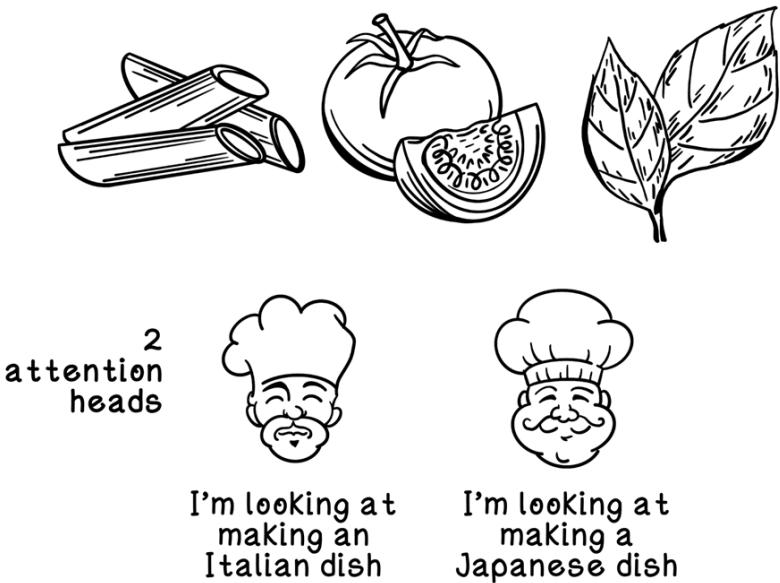
    attention_weights = softmax(attention_scores)

    attention_output = attention_weights @ V

    return attention_output, attention_weights
```

### 11.7.5 Multiple attention heads

In our simple example we have only 1 attention head. Remember, attention heads look at the relationships between tokens from a specific “perspective” - more heads learn more distinct relations. To continue with our cooking analogy, you can see attention heads as different chefs that look at the ingredients from unique perspectives. Similar ingredients with varied preparation can be used to make different tasting dishes, so a chef (attention head) might be looking at the ingredients from an angle of making a Japanese dish, while another chef (attention head) is looking at it from an Italian dish perspective (figure 11.40).



**Figure 11.40 An analogy of the usefulness of multiple attention heads**

Multiple attention heads are a powerful concept for LLMs. We might have an emergent *Grammar expert*, *Pronoun expert*, *Positional expert*, etc. Each expert (attention head) has its own independent set of Q, K, and V matrices, allowing it to specialize in finding one specific type of connection. The model then combines the findings from all these experts to build a single, incredibly rich, and layered understanding of each word.

This approach gives the model the ability to analyze a sentence from many different perspectives at once, capturing a much deeper and more robust understanding of the language than a single head ever could.

So if we had 2 heads, we would repeat this entire self-attention process for each. However, the initial Q, K, V step would have 2 vectors and not 4, this means that each of the two heads will be a  $32 \times 2$  matrix, which will be concatenated into a  $32 \times 4$  matrix, which matches our width.

If we had a width of 8, for example, we could have:

- 1 head that is  $32 \times 8$ , or
- 2 heads that is  $32 \times 4$  each, or
- 4 heads that is  $32 \times 2$  each

We have now completed the self-attention step. In a large set of tokens, this entire process is massive in terms of compute and data calculated. Although we've been doing each operation independently, in practice, we leverage the power of specialized chips like GPUs that have built-in matrix manipulation operations to make these computationally heavy operations happen in an instant compared to running them on traditional CPUs.

### 11.7.6 Layer normalization

We now need to add the calculated attention to our original token embeddings. Layer normalization stabilizes training by keeping every token's vector roughly zero-mean. So later layers in training aren't working with exploding or vanishing magnitudes. The model can always "fall back" on the original embedding representation if the attention heads were unhelpful. It can even learn to ignore the attention heads, if needed.

Given the original embedding for token 0, as seen in table 11.25:

**Table 11.25 Token 0's original weights and attention context**

<b>Original weights</b>					
<b>ID</b>	<b>Token</b>	<b>x0</b>	<b>x1</b>	<b>x2</b>	<b>x3</b>
0	a	0.0030	0.9930	0.0120	1.0010
<b>Attention context</b>					
<b>ID</b>	<b>Token</b>	<b>c0</b>	<b>c1</b>	<b>c2</b>	<b>c3</b>
0	a	0.2283	2.1990	0.4526	1.9747

We add the residual connection by calculating (figure 11.41):

```

y = x + c
x = [0.0030, 0.9930, 0.0120, 1.0010]
c = [0.2283, 2.1990, 0.4526, 1.9747]
y = [0.0030, 0.9930, 0.0120, 1.0010] + [0.2283, 2.1990, 0.4526, 1.9747]
y = [0.2313, 3.1920, 0.4646, 2.9757]

```

**Figure 11.41 Calculations for the residual connection**

Layer normalization is now done with the equation in figure 11.42:

$$\text{LayerNorm}(x_i) = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta$$

Input vector element      Mean of all elements in input vector  
 Variance                    Learnable scale  
 Learnable bias

**Figure 11.42** The formula for layer normalization

To understand this complex looking expression, let's take it step-by-step, given the input:

$x = [0.2313, 3.1920, 0.4646, 2.9757]$

- Calculate Mean ( $\mu$ )  
 $(0.2313 + 3.1920 + 0.4646 + 2.9757) / 4$   
 $= 1.7159$
- Calculate Deviations ( $x - \mu$ )  
 $[0.2313 - 1.7159, 3.1920 - 1.7159, 0.4646 - 1.7159, 2.9757 - 1.7159]$   
 $= [-1.4846, 1.4761, -1.2513, 1.2598]$
- Squared Deviations ( $(x - \mu)^2$ )  
 $[(-1.4846)^2, (1.4761)^2, (-1.2513)^2, (1.2598)^2]$   
 $= [2.040, 2.1789, 1.5657, 1.5871]$
- Calculate Variance ( $\sigma^2$ )  
 $(2.040 + 2.1789 + 1.5657 + 1.5871) / 4$   
 $= 1.8839$
- Normalize  
 $\text{Deviations} / \sqrt{\text{Variance}}$   
 $= [-1.0816, 1.0754, -0.9116, 0.9178]$

For the sake of simplicity, we've omitted the learnable bias and learnable scale variables from this example.

**$z_0 = [-1.0816, 1.0754, -0.9116, 0.9178]$**

We will need to do this operation for all 32 tokens in our example which will result in a 32 x 4 matrix, ready for the decoding components.

#### EXERCISE: WHAT IS THE NORMALIZED VECTOR FOR TOKEN 2?

Determine the normalized vector for token 2.

**SOLUTION: WHAT IS THE NORMALIZED VECTOR FOR TOKEN 2?**

Final Output for z0 : [-1.0816, 1.0754, -0.9116, 0.9178]

**PYTHON CODE SAMPLE**

This code performs layer normalization on the attention output. For each row (each token's vector), it computes the mean and variance, then standardizes each value by subtracting the mean and dividing by the standard deviation (with a small epsilon added for numerical stability). This helps stabilize and accelerate training by ensuring that each token's vector has a consistent distribution, regardless of its raw values.

```
def normalize_layer(attention_output, epsilon=1e-5):

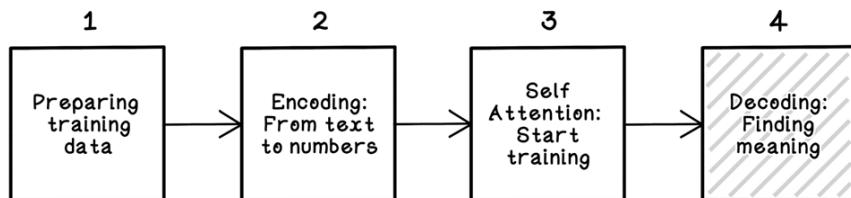
    mean = np.mean(attention_output, axis=1, keepdims=True)

    variance = np.var(attention_output, axis=1, keepdims=True)

    standard_deviation = np.sqrt(variance + epsilon)

    normalized_output = (attention_output - mean) / standard_deviation

    return normalized_output
```

**11.8 Decoding: Meaning through neural networks**

**Figure 11.43** The decoding step in the LLM training workflow

Now that we have found the relationships between tokens from different perspectives (attention heads), the feed-forward network works with that blended data and finds relationships between the features within it (figure 11.43). The self-attention step has delivered a bundle of clues for a word, and now the feed-forward network decides which clues to amplify, suppress, or even merge into higher-level features. If the results of self-attention are ingredients, the results of the feed-forward network is the “full course of dishes” created from the ingredients.

Table 11.26 is our normalized layer for all tokens:

**Table 11.26 A subset of tokens and their normalized values**

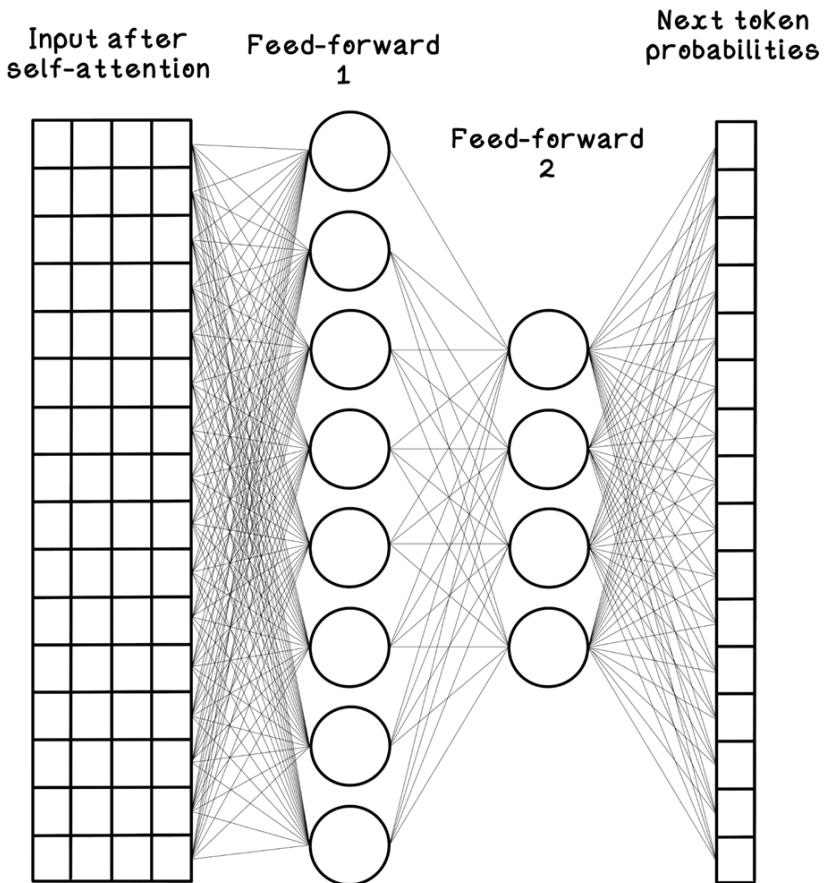
ID	Token	y0	y1	y2	y3
0	a	-1.0816	1.0754	-0.9116	0.9178
1	l	-0.5509	0.7252	-1.3432	1.1689
<hr/>					
31	b	-0.7789	0.6629	-1.1559	1.2719

Thinking back on Chapter 9’s example, we had 4 inputs, for *speed*, *terrain quality*, *degree of vision*, and *total experience*. For the LLM, we have a much larger set of inputs describing the relationships between tokens, the positional relationship between tokens, and outcomes of self-attention.

We need to define the parameters for the feed-forward network.

- Project up layer: This is an 8 node layer with a width of 4 based on our initial width of 4. This layer is called W1. The bias will be 4 values that match our width called b1.
- Project down layer: This is a 4 node layer that takes 8 vectors as inputs (8 x 4) called W2. The bias will be 4 values that match our width called b2.

Figure 11.44 illustrates the architecture of our feed-forward network that will produce probabilities of what the next token will be given our current training batch.



**Figure 11.44 The architecture of the feed-forward network to output token probabilities**

### 11.8.1 Project up layer

This is the first weight matrix,  $W_1$ , in our feed-forward network. Its job is to project the 4-dimensional input vector from the previous layer into a larger, 8-dimensional space. In a real scenario, these weights start as random numbers and are slowly adjusted during training. For our example, we will just define them with the following fixed values.

We choose random numbers between -1 and 1 for the weights in our layer, as seen in Table 11.27:

**Table 11.27 Random weights for the project-up layer**

0.1	-0.2	0.3	0.4	-0.5	0.6	0.7	-0.8
0.9	0.8	-0.7	0.6	0.5	-0.4	0.3	0.2
-0.3	0.2	0.1	-0.9	0.8	0.7	-0.6	0.5
0.5	-0.6	0.7	0.8	-0.9	0.1	0.2	-0.3

We can now take the input for token 0 and multiply it with the weights matrix.

The process of multiplying the input vector with the weight matrix involves calculating each of the 8 new output values one by one. To get the first output value, we multiply each element of our input vector with the corresponding element from the first column of the weights matrix and sum the results. We repeat this process for each of the 8 columns in the weights matrix to get all 8 final values.

So given the input **[-1.0816, 1.0754, -0.9116, 0.9178]** for token 0, we can calculate the weights (figure 11.45).

y0 input      Weight 0, 0	y3 input      Weight 3, 0
$\downarrow$ $\downarrow$	$\downarrow$ $\downarrow$
Output 0: $(-1.0816 \times 0.1) + (1.0754 \times 0.9) + (-0.9116 \times -0.3) + (0.9178 \times 0.5) = 1.5921$ Output 1: $(-1.0816 \times -0.2) + (1.0754 \times 0.8) + (-0.9116 \times 0.2) + (0.9178 \times -0.6) = 0.3436$ Output 2: $(-1.0816 \times 0.3) + (1.0754 \times -0.7) + (-0.9116 \times 0.1) + (0.9178 \times 0.7) = -0.5260$ Output 3: $(-1.0816 \times 0.4) + (1.0754 \times 0.6) + (-0.9116 \times -0.9) + (0.9178 \times 0.8) = 1.7674$ Output 4: $(-1.0816 \times -0.5) + (1.0754 \times 0.5) + (-0.9116 \times 0.8) + (0.9178 \times -0.9) = -0.4769$ Output 5: $(-1.0816 \times 0.6) + (1.0754 \times -0.4) + (-0.9116 \times 0.7) + (0.9178 \times 0.1) = -1.6255$ Output 6: $(-1.0816 \times 0.7) + (1.0754 \times 0.3) + (-0.9116 \times -0.6) + (0.9178 \times 0.2) = 0.2961$ Output 7: $(-1.0816 \times -0.8) + (1.0754 \times 0.2) + (-0.9116 \times 0.5) + (0.9178 \times -0.3) = 0.3492$	$\uparrow$ $\uparrow$
y0 input      Weight 0, 3	

**Figure 11.45 Calculations for feed-forward outputs**

Giving us the result of weight multiplication:

**[1.5921, 0.3436, -0.5260, 1.7674, -0.4769, -1.6255, 0.2961, 0.3492]**

## ADD BIAS

After the weight multiplication, the next step is to add a bias vector. This is simply another list of learnable numbers, one for each of our 8 dimensions. Adding a bias gives the network more flexibility, allowing it to shift the output of each node up or down to better fit the data.

Let's assume that our bias is: **[0.1, -0.05, 0.2, -0.15, 0.0, 0.1, -0.2, 0.05]**

So, the result is as shown in figure 11.46:

```
[1.5921, 0.3436, -0.5260, 1.7674, -0.4769, -1.6255, 0.2961, 0.3492]
+ [0.1, -0.05, 0.2, -0.15, 0.0, 0.1, -0.2, 0.05]
= [1.6921, 0.2936, -0.3260, 1.6174, -0.4769, -1.5255, 0.0961, 0.3992]
```

**Figure 11.46 Calculations for adding bias to the output**

## ACTIVATION FUNCTION (RELU)

So far, all our operations have been linear. To give our network the ability to learn more complex, non-linear patterns, we must apply a non-linear activation function. One of the most common and effective activation functions is ReLU (Rectified Linear Unit). The rule is very simple: it goes through each number in our vector and replaces any negative value with 0, leaving all positive values unchanged (figure 11.47).

Given the last result

```
[1.6921, 0.2936, -0.3260, 1.6174, -0.4769, -1.5255, 0.0961, 0.3992]
```

After ReLU, we have:

```
[1.6921, 0.2936, 0, 1.6174, 0, 0, 0.0961, 0.3992]
```

**Figure 11.47 Calculations for applying the ReLU activation function**

## PYTHON CODE SAMPLE

This code applies a feedforward projection layer that expands or transforms the normalized token representations. Each token vector is multiplied by a learned `projection_weights` matrix to produce a higher-dimensional representation (`projected_output`). This step allows the model to learn richer, more abstract features before continuing through the transformer block. It's commonly referred to as the "project up" step in the feedforward sublayer.

```
def project_up(normalized_output, output_dimensions):
    input_embedding_width = normalized_output.shape[1]
    projection_weights = np.random.randn(input_embedding_width, output_dimensions)
    projected_output = normalized_output @ projection_weights
    return projected_output
```

### 11.8.2 Project down layer

The second and final layer of our feed-forward network takes the 8-dimensional vector from the ReLU step and projects it back down to our original embedding size of 4. This is done using the W2 weight matrix. This step ensures the output of the feed-forward block has the same dimensions as its input, which is necessary for the final residual connection.

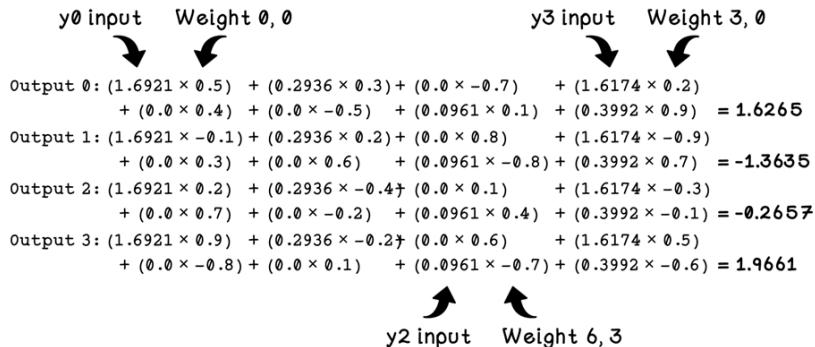
Again, we choose random numbers between -1 and 1 for the weights in our layer, as seen in Table 11.28:

**Table 11.28 Random weights for the project-down layer**

0.5	-0.1	0.2	0.9
0.3	0.2	-0.4	-0.2
-0.7	0.8	0.1	0.6
0.2	-0.9	-0.3	0.5
0.4	0.3	0.7	-0.8
-0.5	0.6	-0.2	0.1
0.1	-0.8	0.4	-0.7
0.9	0.7	-0.1	-0.6

The calculation is the same as before. To get the first of our four new output values, we multiply each of the 8 elements from our input vector with the corresponding element from the first column of the W2 matrix and then sum the results. We repeat this for all 4 columns to produce our final 4-dimensional vector (figure 11.48).

So given the input: [1.6921, 0.2936, 0, 1.6174, 0, 0, 0.0961, 0.3992]



**Figure 11.48 Calculations for the project-down layer output**

Giving us the result of weight multiplication:

**[1.6265, -1.3635, -0.2657, 1.9661]**

### ADD BIAS

As with the first layer, a final bias vector,  $b_2$ , is added to the output of the 'project down' multiplication. Because this layer returns a 4-dimensional vector, the bias also contains 4 values, providing a final adjustment to the output.

Let's assume that our bias is: **[0.05, -0.1, 0.15, -0.02]**

So the result is as shown in figure 11.49:

$$\begin{aligned}
 & [1.6265, -1.3635, -0.2657, 1.9661] \\
 & + [0.05, -0.1, 0.15, -0.02] \\
 & = [1.6765, -1.4635, -0.1157, 1.9461]
 \end{aligned}$$

**Figure 11.49 Calculations for adding bias to the output of the project-down layer**

### ADD THE RESIDUAL

We have now reached the second residual connection in our Transformer block. Just like after the self-attention step, we add the input of the feed-forward network to its output. This shortcut helps stabilize training and allows the model to bypass the feed-forward layer if it determines that its calculations are not needed for a particular token.

Now we take the initial input after self-attention, and add the result after bias was added (figure 11.50):

$$\begin{aligned}
 & [-1.0816, 1.0754, -0.9116, 0.9178] \\
 & + [1.6765, -1.4635, -0.1157, 1.9461] \\
 & = [0.5949, -0.3881, -1.0274, 2.8640]
 \end{aligned}$$
**Figure 11.50 Calculations for adding the residual**

### 11.8.3 Layer normalization

The final step in our Transformer block is to apply layer normalization one last time. We take the output from the second residual connection and normalize each vector independently. This ensures that the data we pass to the next Transformer block is clean and has a consistent scale, which helps to stabilize the training of very deep networks.

We follow the same layer normalization operations as described earlier, to give use the final result for token 0:

**[0.0569, -0.6089, -1.0419, 1.5939]**

The same process happens for all 32 tokens in the batch.

**Table 11.29 A subset of tokens and their results after layer normalization**

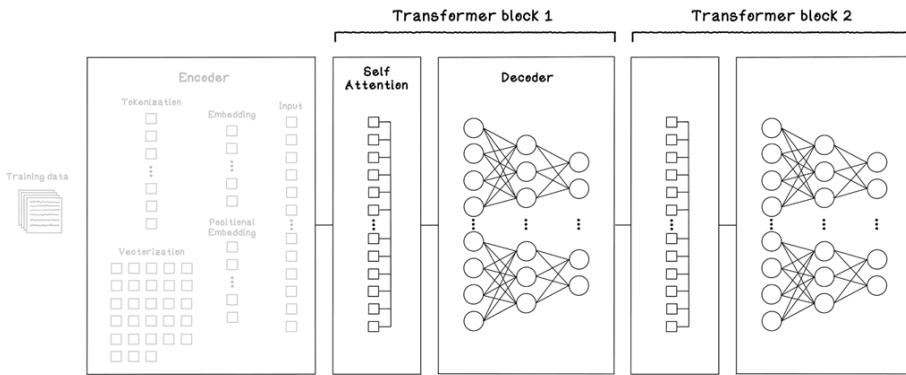
ID	Token	y0	y1	y2	y3
0	a	0.0569	-0.6089	-1.0419	1.5939
1	l	0.2189	-1.0396	-0.7153	1.5361
31	b	-0.0047	-0.9123	-0.7135	1.6305

### 11.8.4 Stacking Transformer blocks

Now that we've completed one full pass of a Transformer block, the process continues in one of two ways, depending on where we are in the model's architecture. The most common next step is to repeat the entire process by feeding the output into another identical Transformer block. However, if this were the final block, the model would move to predict the next token—in practice when utilizing the LLM after training, this can be done repeatedly to generate a complete sensible output.

The power of LLMs comes from stacking many Transformer blocks. The output from the block we just finished ([0.0569, -0.6089, -1.0419, 1.5939]) would become the input for the next block.

This new block would perform its own self-attention and feed-forward calculations using its own unique set of learned weights. This allows the model to refine the token's representation at progressively deeper and more abstract levels. A common production-scale LLM has dozens of these blocks stacked on top of each other (figure 11.51).



**Figure 11.51 How multiple transformer blocks are stacked**

- *Early Layers (Bottom of the Stack)*: Tend to learn simpler, more local patterns. They might focus on basic syntax, word-level relationships, and resolving ambiguity in specific sentences. For example, in the sentence “Alice was beginning to get very tired of sitting” an early layer might learn that “sitting” is a verb associated with “Alice”.
- *Middle Layers (Middle of the Stack)*: These layers take the language “syntactically-aware” representations from the early layers and start building a more semantic understanding. They connect concepts across sentences, recognizing pronoun relationships and understanding the overall structure of the text.
- *Late Layers (Top of the Stack)*: The final layers receive a rich, contextually-aware representation. They use this information to perform high-level inference, understand complex ideas like sarcasm or metaphors, and synthesize all the previous data to make the best possible prediction.

Let’s assume that the current output of this transformer block is the final output for the neural network, for simplicity in our example.

### 11.8.5 Making a prediction

The entire purpose of the Transformer block was to create a final, information-rich vector for each token in our input—making a matrix. Now, we will use this final matrix to perform the model’s main purpose: predicting the next token in the sequence. This is done by taking a specific output vector and transforming it into a list of scores, one for every single token in our vocabulary.

## CREATING LOGITS

This is done by creating logits. Logits are used as an intermediate step because they provide a numerically stable foundation for the final probability calculation. Logits are the raw, unnormalized scores that a model produces right before the final prediction step. They are not probabilities, they can be any real number.

Logits are calculated by taking the final output matrix from the last transformer block and passing it through one final linear layer (sometimes called the language model head).

We need another weighted matrix that is the width x vocabulary in size ( $4 \times 35$  in size).

**Note!** We're no longer using only the first 32-token batch context window. We're now working with the entire vocabulary.

So given the results in Table 11.30:

**Table 11.30 A subset of tokens and their vectors**

ID	Token	y0	y1	y2	y3
0	a	0.0569	-0.6089	-1.0419	1.5939
1	l	0.2189	-1.0396	-0.7153	1.5361
31	b	-0.0047	-0.9123	-0.7135	1.6305

We will use the last token to make a prediction. We do this because the model will attempt to predict what comes after the sequence of tokens in the context window that was trained. That will be token "b". Think of it like reading a sentence. To predict the word that comes after "The cat sat on the...", you use the context of the whole phrase.

**Table 11.31 Token b's vector**

ID	Token	y0	y1	y2	y3
31	b	-0.0047	-0.9123	-0.7135	1.6305

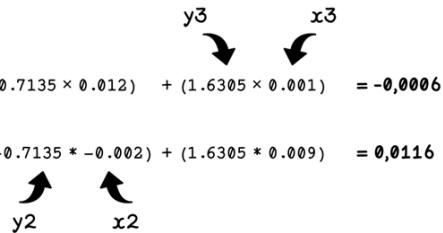
With the original weight matrix that looks like this:

**Table 11.32 A subset of tokens and their vectors**

Position	Token ID	Token	x0	x1	x2	x3
0	0	a	0.0030	-0.0070	0.0120	0.0010
1	1	l	-0.0080	0.0050	-0.0020	0.0090
	...					
31	6	b	0.7381	-0.2512	-0.6719	0.7938

The results are as shown in figure 11.52:

$$\begin{array}{l}
 \text{Logit for "a"} \\
 (-0.0047 \times 0.003) + (-0.9123 \times -0.007) + (-0.7135 \times 0.012) + (1.6305 \times 0.001) = \mathbf{-0,0006} \\
 \\
 \text{Logit for "l"} \\
 (-0.0047 \times -0.008) + (-0.9123 \times 0.005) + (-0.7135 \times -0.002) + (1.6305 \times 0.009) = \mathbf{0,0116}
 \end{array}$$



**Figure 11.52 Calculations for logits for tokens**

#### EXERCISE: WHAT IS THE LOGIT FOR TOKEN 31?

Determine the logit for token 31.

#### SOLUTION: WHAT IS THE LOGIT FOR TOKEN 31?

$$\begin{aligned}
 & (-0.0047 \times 0.7381) + (-0.9123 \times -0.2512) + (-0.7135 \times -0.6719) + (1.6305 \times 0.7938) \\
 & = 1.2
 \end{aligned}$$

## SOFTMAX

Finally, we can use softmax to convert the raw logits to probabilities. If you're still unsure about the softmax calculation, refer to the third step in the self-attention section.

This will result in a vector of 35 probabilities where we have a probability of each token in our vocabulary to appear after the batch in question (figure 11.53).

**Table 11.33 A subset of tokens and their probabilities of appearing after token 0**

<b>ID</b>	<b>Token</b>	<b>Probability</b>
0	a	0.0243
1	l	0.0246
...		
6	b	0.1799
...		
34	?	0.0251

**Final Prediction: “b” with a probability of 17.99%**

**Figure 11.53 Final prediction for the next token after “a”**

Let's see if that makes any sense. Our training batch was the one shown in figure 11.54:

[a, l, ic, e, w, a, s, b, e, g, in, n, ing, t, e, g, e, t, v, er, y, ti, re, d, e, f, s, i, t, t, ing, b]  
 Batch 0  
 [0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6]

**Figure 11.54 Revisiting the first training batch of tokens**

By using our intuition, we can tell that a “b” is probably not the correct token to appear next. It might be suspicious that “b” is predicted and it is also the last token in the training batch that we used, but that is pure coincidence.

We did all that work, only to get an incorrect prediction. But remember, we used only a single 32 token batch to train with, a single attention head, a tiny feed-forward, and only ran one iteration of training. LLMs require masses of data, many attention heads, massive feed-forward networks, stacked transformer blocks, and many iterations of training before they start showing signs of intelligence. This intensity in training is the key reason that GPUs are preferred from a performance perspective, due to their blazing fast matrix manipulation, and even so, LLMs are notorious for requiring immense compute.

## PYTHON CODE SAMPLE

This code projects the model's internal representations back down to the vocabulary space to generate predictions. It multiplies each token's high-dimensional vector by a *projection\_down\_weights* matrix, producing a score for every token in the vocabulary. The index of the highest score in each row is selected as the model's predicted next token.

```
def project_down_and_predict(projected_output, vocabulary_size):

    input_width = projected_output.shape[1]

    projection_down_weights = np.random.randn(input_width, vocabulary_size)

    final_output_logits = projected_output @ projection_down_weights

    predictions = np.argmax(final_output_logits, axis=1)

    return predictions.tolist()
```

### 11.8.6 Backpropagation and calculating loss

The next step is to calculate the loss, which measures how wrong the model's prediction was. This is a critical part of training. You compare the probability the model assigned to the actual correct next token against the ideal probability (1.0 or 100%). This comparison gives you a single number, the loss or "how wrong it was".

We covered computing loss / cost in Chapter 9, but the essence of it is correcting weights given the predicted values. With the car example in Chapter 9 we calculated the cost / loss by comparing it with the training data reference. With LLMs, the "actual" next token is already known because it's the very next token in the training data sequence that we feed the model. The network "understands" the loss because of this.

## CALCULATING CROSS-ENTROPY LOSS

To measure the model's error, we calculate the cross-entropy loss. For each token in the input sequence, we look at the probability the model assigned to the **actual** next token and then take its negative log. A perfect prediction would have a probability of 1.

The negative logarithm function,  $-\log(p)$ , has two properties that make it perfect for measuring error:

When the model is correct, the probability it assigns to the correct token is high (close to 1).  $-\log(1)$  is 0, so the loss is very low, which is what we want.

When the model is wrong, the probability  $p$  for the correct token is low (close to 0). As  $p$  approaches 0,  $-\log(p)$  gets huge, approaching infinity. This heavily penalizes the model for being confident in the wrong answer.

- *Confident & Correct ( $p = 0.99$ ):* The model assigned a 99% probability to the right answer. The loss is tiny, about 0.01.

- *Uncertain ( $p = 0.50$ ):* The model was unsure. Loss is moderate, around 0.69.
- *Confident & Wrong ( $p = 0.01$ ):* The model assigned only a 1% probability to the right answer. The loss is huge, about 4.60.

Given this, we can calculate the loss for each token using the correct next ID and its probability.

**Table 11.34 A subset of tokens and their loss values**

ID	Token	Correct next ID	Correct next token	Probability	Loss
0	a	1	I	0.0243	3.7221
1	I	2	ic	0.0395	3.2304
...					
30	ing	6	b	0.1123	2.1869

This is the expected result because our model is untrained. It has learned no patterns from the data. Its weights are fixed and random. For a vocabulary of 35 tokens, a random guess would have a loss of approximately  $-\log(1/35)$ , about 3.55. Our model's performance is in that exact ballpark.

Therefore, a probability of 11% for the token 'b' does not mean the model is "quite confident" and wrong, it means its performance is only slightly better than a random guess. The high loss value correctly reflects this deep uncertainty.

## BACKPROPAGATION

The loss score that we've just calculated is the starting signal for backpropagation. The algorithm works backward from the loss value through every layer of the network - the feed-forward layers, the attention mechanism, and the embedding matrices. It calculates how much each individual weight and bias contributed to the final error and then "nudges" every parameter in a direction that will make the loss smaller on the next attempt. This is how the network learns the relationships between words and refines its understanding of language based on the training data.

See Chapter 9's discussion of backpropagation if you need a refresher, but in short, backpropagation starts with the loss value and works backward through the network using the chain rule from calculus.

- It first calculates the gradient of the loss with respect to the output of the last layer (the logits).
- It then uses that result to find the gradient with respect to the parameters of that layer (the final weights and biases).
- It continues this process backward, layer by layer, until it has a gradient for every single parameter all the way back to the initial embedding matrix.

Once backpropagation has calculated the gradient for a specific weight, that weight is updated. Let's take a single weight from our final  $4 \times 35$  weight matrix and assume that after the complex backpropagation process, its calculated gradient is -0.5.

- *Old Weight*: Let's say the original value of this weight was 0.5000.
- *Learning Rate*: We need a small number to ensure we don't update the weights too aggressively. A common value is 0.01.
- *Gradient*: The calculated gradient for this weight is -0.5.

The backpropagation update calculation is shown in figure 11.55:

$$\text{New weight} = \text{Old weight} - (\text{Learning rate} \times \text{gradient})$$

$$\text{New weight} = 0.5000 - (0.01 * -0.5)$$

$$\text{New weight} = 0.5000 - (-0.005)$$

$$\text{New weight} = 0.5050$$

**Figure 11.55 Calculations for new weights given the learning rate and gradient**

### The weight is “nudged” from 0.5000 to 0.5050

This tiny adjustment, when applied to all parameters, makes the entire network slightly better at its task. Again, this would be done for all predictions, so that all weights are updated respectively.

### PYTHON CODE SAMPLE

This code computes the training loss using cross-entropy between the predicted tokens and the actual ground-truth tokens. It then performs backpropagation, calculating gradients for all trainable weights in the model, including embedding and projection matrices, and updates them using the specified *learning\_rate*. This step is essential for learning: it gradually adjusts the model's internal parameters to reduce future prediction errors.

```

def calculate_cross_entropy(predictions, actual_tokens):
    return np.mean(np.square(actual_tokens - predictions))

def compute_gradients_and_update_all_weights(weights_list, loss, learning_rate):
    """
    Conceptual function representing the backpropagation step.
    In a real model, this computes dLoss/dW for all weights and updates them.
    """
    print(f" [BP Applied] Loss: {loss:.4f}. Updated {len(weights_list)} weight
matrices.")
    pass #A

def calculate_loss_and_backprop(predictions, actual_tokens, learning_rate):

    loss = calculate_cross_entropy(predictions, actual_tokens)

    trainable_weights = [
        'embedding_matrix', 'query_weights', 'key_weights', 'value_weights',
        'projection_weights', 'projection_down_weights'
    ]

    compute_gradients_and_update_all_weights(trainable_weights, loss,
learning_rate)

    return loss

```

We have now completed one full training cycle on a single batch of 32 tokens. The model is now slightly better than it was before. The training flow continues by sampling a new batch of 32 tokens from the token stream. This cycle is repeated thousands or millions of times, iterating through the entire dataset, until the model's performance is satisfactory.

## 11.9 Controlling the LLM

Well done. We've made it to the end of the core training cycle for an LLM. After digesting the steps, it might seem unbelievable that these operations can create amazing emergent intelligence that we see and use in powerful LLMs, but it's really difficult for humans to visualize and perceive the complexity with the huge data sets and high dimensional spaces that these Transformers play in. With that said, we can train, evaluate, tweak, and improve. Let's explore some concepts for controlling LLMs.

### 11.9.1 Training epochs

A key to transformers is many cycles of learning on different data. To accomplish this, we need to process many batches of the token stream, in different orders, and run this entire process many times.

- An Epoch is one complete cycle through the entire training dataset.
- A Batch is a small chunk of that dataset that you process at one time before updating the model's weights.

One training cycle (figure 11.56) consists of the following:

1. *Pick a batch*: Get the next 32 tokens from the dataset.
2. *Forward pass*: The input goes through the entire model including embeddings, self-attention, and the feed-forward network to produce the final predictions.
3. *Calculate loss*: Compare the model's predictions to the correct next tokens to get a single loss value for the batch.
4. *Backpropagation*: The loss is used to calculate the gradients for all of the model's weights.
5. *Update weights*: Use gradients to "nudge" all the weights, making the model slightly better.
6. *Repeat*: Go back to step 1 and process the next batch with the newly updated weights.

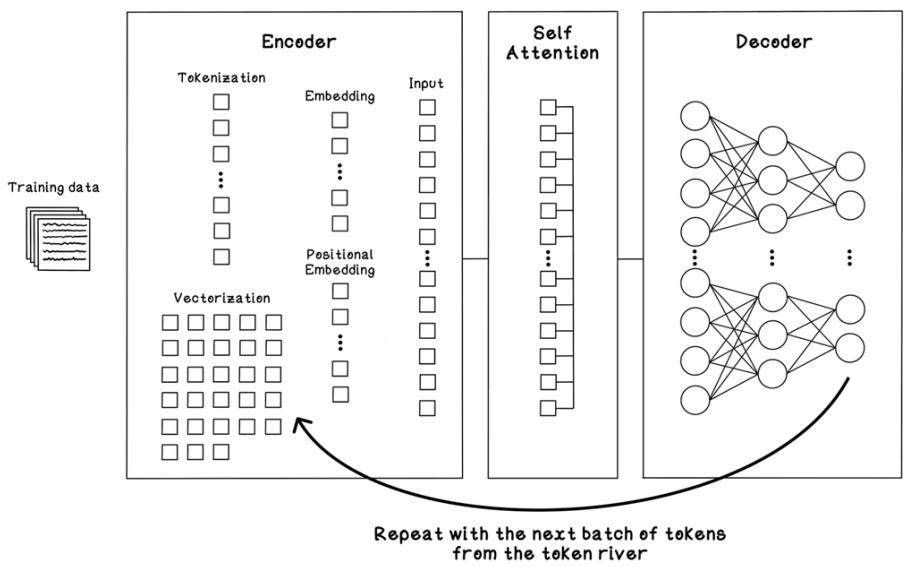


Figure 11.56 How epochs repeat in the Transformer framework

Once all batches in the dataset have been processed, Epoch 1 is complete. The training process then continues by starting Epoch 2, often after shuffling the data, and this cycle repeats for many epochs until the model's performance is satisfactory. Surprisingly some powerful modern LLMs were trained on less than one epoch - this means that the amount of data in their token stream was so massive, that they were able to learn extremely well purely from sampling batches of tokens within a single epoch.

### 11.9.2 Saving checkpoints

During a long training process, you should periodically save the model's state, including all of its learned weights and parameters, to a file. This is called saving a checkpoint. Training can take days, weeks, or months. If the process crashes for any reason, checkpoints allow you to resume from where you left off instead of starting from scratch. They also let you preserve different versions of your model to evaluate later.

- *Periodic Saving:* The most common strategy. The model's state (all weights and parameters) is saved at a regular interval. We can save every N training steps or every M minutes. This should always be used for fault tolerance. The frequency is a trade-off. More frequent saves mean less lost work but use of more disk space and can slightly slow down training.
- *Best-Model Saving:* With this strategy, we only save a checkpoint when the model's performance improves on a validation set of data (i.e., when the loss reaches a new minimum for data that it hasn't seen before). We can overwrite the same file every time a new best is found. A common practice is to combine this with periodic saving.

### 11.9.3 Stopping mechanisms

We need a way to measure if the model is truly learning to generalize or just memorizing the training data (the problem of overfitting). We set aside a portion of the data as a validation set, which the model never trains on. Periodically, we pause training and calculate the loss on this validation set. If the training loss continues to go down but the validation loss starts to go up, it's a clear sign that the model is no longer learning useful patterns and is beginning to memorize. This is the practice of stopping the training process once validation loss stops improving. This saves significant time, computational resources, and costs, and often results in a better-performing final model.

- *Validation set evaluation:* The standard strategy is to run your model on a separate validation dataset that it never trains on, as mentioned.
- *Early Stopping:* This strategy automatically stops the training process to save time and prevent overfitting. The main parameter is "patience". This is the number of evaluation cycles you wait without seeing any improvement in validation loss before you stop training. A patience of 3, for example, would stop training if the validation loss fails to improve for three consecutive evaluations. This saves significant computational cost and prevents the model from getting worse by continuing to train past a known optimal state.

#### 11.9.4 Hyperparameter tuning

Before training begins, you have to set several key parameters that define the model's architecture and how it learns. Finding the best values for these hyperparameters is a critical part of the process. Here's more details about hyperparameters for Transformer LLMs.

- *Learning Rate*: How big of a step the model takes when updating its weights during backpropagation.
- *Batch Size / Context Window*: The number of tokens the model processes at once. In our example, this is 32 tokens.
- *Model Architecture*: The dimensions of the model, such as its width (embedding size), depth (number of layers), and number of attention heads.

Adjusting parameters can be tricky. It might be based on trial and error, or based on another algorithm. Trial and error in training LLMs can be extremely time, and resource consuming. Using known well-performing hyperparameters is preferred.

- *Manual Tuning*: This involves using your intuition and experience to choose parameters, running training cycles, observing the results, and then manually tweaking the parameters for the next run. This is practical when you have a deep understanding of the model or limited computational resources. It's often the starting point for any project.
- *Grid Search*: You define a specific list of values to try for each hyperparameter (e.g. learning rate = [0.01, 0.001, 0.0001], batch size = [32, 64], etc.). The system then exhaustively trains a model for every possible combination. This is useful for a small number of hyperparameters but becomes extremely slow and expensive as the number of options grows.
- *Random Search*: Instead of trying every combination, you define a range for each hyperparameter and then test a fixed number of random combinations from within those ranges. This is often more efficient than grid search. It can explore a wider range of values and frequently finds better models in less time.
- *Bayesian Optimization*: This is an advanced strategy where an algorithm builds a probabilistic model to predict which hyperparameter combinations are likely to perform best. How meta! Machine learning for machine learning. This uses the results from previous runs to intelligently decide which combination to try next.

### 11.9.5 Few-shot and zero-shot learning

In-context learning is the process of adapting a pre-trained model to perform well on a specific task, such as summarization, question-answering, or classification. This is often more efficient than training a model from scratch. This does not mean training or adjusting the weights in the architecture, but rather using prompting techniques in utilizing an already trained model.

One of the most powerful aspects of LLMs is their ability to perform tasks they weren't explicitly trained for, a key goal of generalization.

- *Zero-Shot Learning:* You ask the model to perform a task without giving it any prior examples in the prompt. For example: "Translate 'hello world' to French." The model understands the instruction and provides the answer based on the patterns it learned during its broad training.
- *Few-Shot Learning:* You provide a few examples of the task within the prompt itself to give the model context (figure 11.57).

**Q: What is the capital of Japan?**

**A: Tokyo.**

**Q: What is the capital of USA?**

**A: Washington, D.C.**

**Q: What is the capital of Egypt?**

**A: The model follows the pattern and responds with "Cairo".**

**Figure 11.57 Example of use cases for few-shot learning**

### 11.10 Refining LLMs with Reinforcement Learning

While pre-training teaches an LLM to understand and generate human-like text, it doesn't inherently teach it to be helpful, harmless, or to follow instructions accurately. Reinforcement learning (RL) is a fine-tuning technique used to align the model's behavior with human-centric goals. It "boosts" the base model's capabilities by teaching it what makes a "good" response beyond the basic next-word prediction. The most common method for this is Reinforcement Learning with Human Feedback (RLHF). RLHF can even be used to teach LLMs how to correctly solve problems and can influence the "intelligence" of the model.

RLHF is a multi-step process that uses human preferences to guide the model's learning.

## COLLECT HUMAN FEEDBACK DATA

First, a dataset of human preferences is created. For a given prompt, the pre-trained LLM is used to generate several different responses.

Instead of always picking the single most probable next word, the model creates this variety by sampling from its list of likely options. This process is often controlled by a parameter called temperature. A higher temperature encourages more diverse and diverse responses. Think about temperature as a creativity lever.

- *Low temperature:* The model acts like a strict accountant. It always picks the most likely, safe answer. It is accurate but can be boring or repetitive.
- *High temperature:* The model acts like a jazz musician. It takes risks, picking less likely words to be surprising, creative, or diverse.

Human labelers review the responses and rank them from best to worst. This process is repeated for thousands of prompts, creating a dataset that captures what humans consider to be high-quality and helpful answers.

## TRAIN A REWARD MODEL

Next, a separate, smaller model, called the reward model, is trained on the human feedback data. The goal of this model is to learn to predict which responses a human would prefer. It takes a prompt and a response as input and outputs a single score / reward that represents the response's quality. A higher score means better response.

## FINE-TUNE THE LLM WITH REINFORCEMENT LEARNING

Finally, the original LLM is fine-tuned using the reward model as a guide. The process works as follows:

- The LLM receives a prompt from the dataset and generates a response.
- The response is shown to the reward model, which gives it a reward score.
- This reward is used in a reinforcement learning algorithm to update the LLMs weights. Rewards and penalties are determined with a cost function as seen earlier.

The LLM learns to adjust its responses to maximize the reward score it receives from the reward model. By doing this, it learns to generate answers that are more aligned with the human preferences with the goal of effectively boosting its helpfulness and safety.

## 11.11 LLMs and Mixture of Experts (MoE)

A Mixture of Experts (MoE) model is a type of Transformer architecture designed to make models much larger without a proportional increase in computational cost. Instead of having one massive feed-forward network that every token must pass through, an MoE model has many smaller, specialized feed-forward networks called experts. A small routine network examines each token and decides which one or two experts are best suited to process it. The token is then only sent to those selected experts, while the others remain inactive. This approach means that for any given token, only a fraction of the model's total parameters are used. This allows us to build models with trillions of parameters, giving them a vast capacity for knowledge while keeping the computational cost for training and inference manageable.

## 11.12 LLMs and Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is a technique that enhances an LLM by connecting it to an external knowledge source. This addresses a key weakness of LLMs in that their knowledge is frozen at the time of training and they can sometimes "hallucinate" facts. RAG combines a pre-trained LLM with a retrieval system that can pull information from a database, like a collection of company documents or a recent snapshot of Wikipedia, for example.

When a user asks a question, the system first searches the external database for relevant documents. The original prompt is then combined with the retrieved information. This combined text is fed to the LLM, which uses both its internal knowledge and the provided external context to generate a more accurate and up-to-date answer.

RAG reduces hallucinations, allows the model to cite its sources from the ones provided, and enables it to answer questions about information it was never trained on, making it highly valuable for enterprise and domain-specific applications. For example, analyze libraries of proprietary documents without needing to build a dedicated LLM for that purpose. We see RAG being incorporated into mainstream LLM platforms more and more.

## 11.13 Use cases for large language models

Large language models (LLMs) are a significant leap in AI's ability to understand, generate, and manipulate human language. Because they are trained on vast and diverse datasets from the internet, books, code, and even user-generated content, they learn the intricate patterns, context, and nuances of communication. This deep understanding of patterns allows them to be applied beyond just language and gives them the ability to find relationships between domain-specific contexts. It's like querying a person that has almost the entire human knowledge pool in their head. This unlocks the potential of a wide variety of tasks that were previously difficult to automate.

### 11.13.1 Content generation

Creating original written content, whether it's an email or a fictional novel, can be a slow and difficult process. The initial step of getting ideas with a blank page is often the hardest. An LLM can act as a powerful assistant in this process. By providing the model with a simple prompt or an outline, a user can generate a complete first draft of an article, a marketing slogan, or a business report. The process becomes one of augmentation, where the human guides the creative direction and refines the AI-generated output, rather than spending hours on the initial composition. This makes the creation of high-volume content, like product descriptions for an e-commerce store with thousands of items, a much more manageable task. LLMs also have immensely useful domain-specific information that can enhance the writing in ways that one might not have thought about alone.

- *Marketing and Advertising:* Creative teams can use LLMs to rapidly produce multiple versions of ad copy for testing or generate a wide range of social media posts tailored to different platforms, then test which perform better. This allows them to focus more on strategy and campaign performance instead of the time-consuming task of writing every piece of content from scratch.
- *E-commerce and Retail:* For online stores with thousands of products, writing unique and engaging descriptions for each item is a major challenge. An LLM can be tasked with generating these descriptions based on a list of product specifications, ensuring that every item has a compelling standardized summary for potential buyers.
- *Education:* Teachers and instructional designers can generate customized learning materials, such as practice questions, lesson summaries, or case studies. This allows them to create a richer variety of content to suit different learning styles without being burdened by the creation of every single document.
- *Legal Services:* While not a replacement for a qualified lawyer, an LLM can create a first draft of standard legal documents, like a lease agreement or a formal letter, based on a set of parameters. This allows legal professionals to speed up their workflow by focusing their time on review, customization, and providing expert legal advice unique to the situation at hand.

LLMs can also be used to augment and enhance existing offerings in digital products. For example, if you have a fitness tracker, you can use LLMs to give users clear instructions on how to perform exercises. If you have a recreational flight computer app, you could supplement and standardize take-off site information for a better user experience. The possibilities are endless.

### 11.13.2 Information synthesis

Many professions involve reading and understanding vast amounts of text. A lawyer might have to review hundreds of pages of legal precedent, or a developer might need to stay current with dozens of newly published code frameworks and small packages, all having their own documentation. Manually reading through all this material to find the key information is incredibly time-consuming. LLMs can process these large documents in seconds and generate concise summaries that highlight the most important information based on a unique context. This doesn't replace the need for expert analysis, but it dramatically speeds up the process by allowing the user to quickly determine which documents or areas of information require a closer look.

- *Legal Services:* As mentioned, a paralegal or lawyer can use an LLM to get a high-level summary of hundreds of pages of discovery documents or case law. This helps them quickly identify which documents are most critical to their case, saving countless hours of manual review.
- *Financial Services:* Financial analysts must digest quarterly earnings reports, market news, and complex filings to make timely investment decisions. An LLM can be used to extract the key financial metrics and commentary from these documents, allowing analysts to react faster and make more informed decisions.
- *Market Research:* Companies often collect thousands of customer reviews, survey responses, or support tickets. Instead of having a team read through them one by one, an LLM can analyze the entire dataset and produce a summary of the most common themes, complaints, and feature requests.

### 11.13.3 Coding assistants

Writing code requires precise syntax and logic, and even experienced programmers spend time looking up functions or debugging errors. Because LLMs have been trained on massive amounts of publicly available code from sources like GitHub, they have learned the patterns and structure of many programming languages. LLMs are actually very well suited for programming languages since the languages are very well defined, and many established code design patterns exist with examples available publicly in thousands of code bases. A developer can describe a desired function in plain English, and the model can generate the code to accomplish that task. This is also useful for learning and debugging. If you encounter a complex block of code written by someone else, you can ask the LLM to explain what it does.

- *Translating Between Programming Languages:* If a team needs to convert a utility script from Python to JavaScript to run in a different environment, an LLM can perform the translation. While it requires human review, it handles the bulk of the syntactic and structural conversion automatically.

- *Code Refactoring and Optimization:* A developer can provide a working piece of code and ask the LLM to improve it. This could involve making the code run more efficiently, improving its readability, or rewriting it to conform to a specific coding style guide or best practice.
- *Debugging and Error Analysis:* Instead of spending hours trying to figure out a cryptic error message, a developer can paste both the problematic code and the error message into the LLM. The model can then analyze the context and often suggest the most likely cause of the error and how to fix it.

#### **11.13.4 Enhancing digital products**

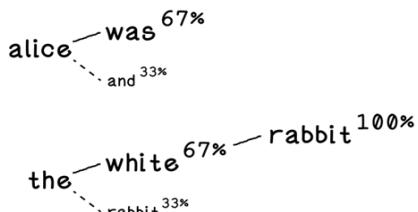
Many digital products and services are built around a core function but involve secondary tasks that require users to process, organize, or create information. These tasks are often a source of friction, slowing the user down from achieving their main goal. A well-implemented LLM can remove this friction by acting as an intelligent assistant embedded directly in the product's workflow. This approach adds a layer of intelligence that makes the product more responsive, personalized, and powerful.

- *Collaborative and Productivity Tools:* An online brainstorming or whiteboarding application is designed to help teams capture ideas. While it's great at collecting raw input, the next step of making sense of a massive board filled with virtual sticky notes is often chaotic and time-consuming. An LLM can be integrated to solve this specific problem. With a single click, it can scan all the text contributed by the group, identify the main themes, group similar ideas together, and generate a structured summary with action items. This transforms a messy, creative brainstorm into a clear, organized output, bridging the gap between ideation and execution.
- *Travel and Hospitality Apps:* Booking a flight and hotel is only the first step in planning a trip. The next, more time-consuming part is figuring out what to do. A travel app can use an LLM to act as a personal concierge. After a user books a 5-day trip to Cape Town, the app could automatically generate a sample itinerary. Based on the user's interests (e.g., "hiking," "history," "wine tasting") and the location of their hotel, the LLM can suggest a logical sequence of activities, grouping them by location to minimize travel time and dramatically simplifying the user's planning process.
- *Personal Finance and Budgeting Tools:* Most budgeting apps are good at categorizing spending, but they often leave the user to figure out the story behind the numbers. An LLM can analyze a user's transaction history and provide narrative insights. For example, instead of just showing a chart, the app could generate a weekly summary like, "You did well on your dining out budget this week, but your grocery spending was 15% higher than average, mostly from a large trip to Whole Foods on Tuesday." This turns raw financial data into a more understandable and actionable story.

## 11.14 Summary of Large Language Models

At their core, LLMs predict the probability of the next word - chained together, this becomes powerful

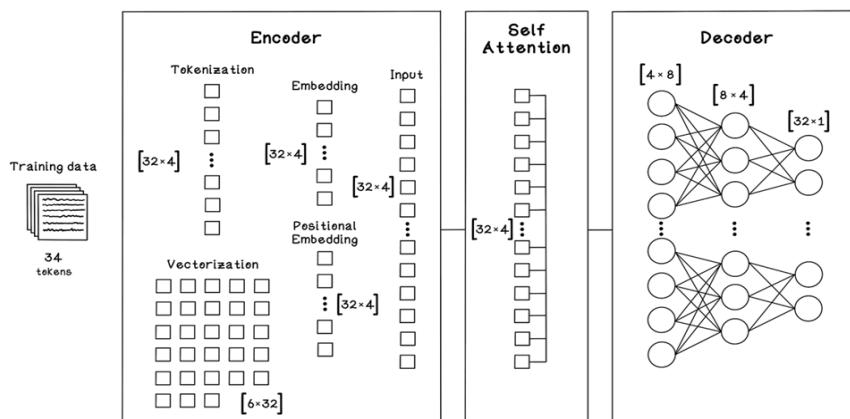
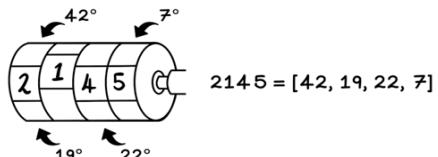
It's key to encode words/tokens as numbers, for LLMs to learn effectively



[a, l, ic, e, w, a, s, b, e, g, in, n, ing, t, o, g, e, t, v, er, y, ti, re, d, o, f, s, i, t, t, ing, b]

[0, 1, 2, 3, 4, 0, 5, 6, 3, 7, 8, 9, 10, 11, 12, 7, 3, 11, 13, 14, 15, 16, 17, 18, 12, 19, 5, 20, 11, 11, 10, 6]

Token positions are a huge component for LLMs to learn meaning of words and how they relate



Attention is the superpower of the Transformer framework that lets words/tokens find meaning among the words around them

# 12 Generative Image Models

## This chapter covers

- Understanding the intuition of generative image models
- Cleaning and preparing image data for training
- Grasping and implementing the steps to train a diffusion model
- Implementing major components for a diffusion model and U-Net

## 12.1 What are generative image models?

Generative image models are a family of algorithms and artificial neural network structures that are specialized towards generating accurate images based on human language input.

Imagine a sculptor who has spent his life observing people who are in deep thought. For years, he walks around the town and thoroughly studies every aspect of every person that he sees in thinking deeply - carefully studying their posture, expressions, and subtle details. Over time, he internalizes what it means to look like someone thinking. He might have seen hundreds of thousands of people pass through the town over the years.

We blindfold the sculptor and give him a random block of marble, and ask "Make me a sculpture of a person thinking". The sculptor can't add new material to marble; instead, he feels the block of marble and chips away a small piece that he's confident does not look like a person thinking. He repeats this thousands of times, each time feeling the edges of the marble, and chipping away a little more "noise". Slowly and methodically, a coherent image of a person thinking emerges from within the random block of stone (figure 12.1).



**Figure 12.1 An analogy showing that Diffusion is about removing to create rather than adding**

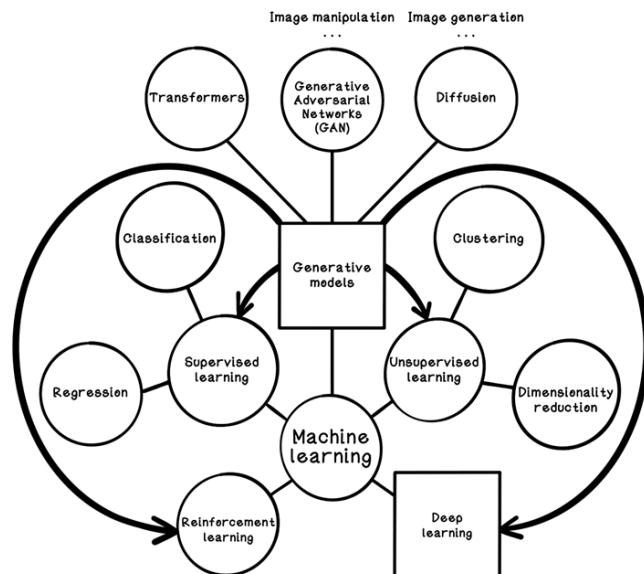
In the same way, a generative image model, trained on millions of images, begins with random noise and iteratively transforms it into a clear image. Not by adding to a blank canvas, but by learning how to remove what doesn't belong.

Modern models that generate images with breathtaking complexity and realism are possible because of three core principles unlocked by the vast amounts of data available, leveraging language models, and harnessing powerful innovations in deep learning research.

- *A visual vocabulary:* Instead of learning the rules of language, an image model learns the “visual grammar” of images. It holds billions of adjustable parameters that store not just facts like “a cat has pointy ears”, but abstract visual concepts. It learns the interplay of light and shadow that makes a scene look dramatic, the specific textures that define rusted metal versus polished chrome, and the compositional rules that distinguish a portrait from a landscape. This allows the model to combine different concepts in novel ways, creating an image of “a snail surfing on a wave of lava in the style of Van Gogh” because it understands the core components of all those ideas.

- *Language models assist image generation:* While a language model is trained on vast amounts of raw text, an image model is trained on two different kinds of data: billions of images, and their related captions. It sees a picture of a futuristic city at night, and reads the caption “a neon-lit cyberpunk cityscape”. By processing millions of these pairs, it builds a powerful understanding between words and pixels. This is how it learns the specific aesthetic of “cyberpunk”, or the subtle difference between a smirk and a smile.
- *The power of noise (Diffusion):* This is the most important innovation, and it’s a counter-intuitive one. Instead of starting with a blank canvas and drawing shapes, the model starts with a canvas of pure random pixels - like a scrambled TV screen with no signal. It then carefully refines every pixel in the static step-by-step, until the desired image emerges from the random noise. This process is called *Diffusion*.

The most popular architectures and approaches for modern image generation are Diffusion models and Generative Adversarial Networks (GANs). Figure 12.2 shows how they fit into the mental model of machine learning algorithms that we have explored throughout this book. You'll notice that *Generative models* is a broad concept that leverages techniques from *Supervised learning*, *Unsupervised learning*, and *Reinforcement learning*.



**Figure 12.2 How Diffusion models fits into machine learning**

## 12.2 The intuition behind image generation

Before we explore the details of a generative image model, let's build some fundamental intuition using a simple practical example that illustrates what a diffusion model actually does. Let's assume that we want to generate an image of a **tree**, and we start with the random unclear image as seen in Figure 12.3.



**Figure 12.3 An example of initial noise**

What do you see? This is the starting point for the model: a canvas of pure, random noise. To our eyes, it's a meaningless, blurry blob. To the model, it is a field of potential, containing every possible image in a faint, ghostly form. At this stage, the model's job is to take its first look and decide which parts of this noise are the least likely to belong in making the image look like a tree.



**Figure 12.4 An example of denoising**

After several steps of “denoising”, a faint structure begins to emerge (figure 12.4). What do you see now? Perhaps a head of broccoli? An explosion? A tree? It’s still very ambiguous, but a general shape is taking form. The model has peeled away the most obvious layers of noise, revealing a low-resolution silhouette. It’s beginning to commit to a general structure, but the details are still fluid. Can you identify which parts of the image need to be denoised to see the tree better?



**Figure 12.5 An example of denoising after more timesteps**

After many more refinement steps, the image becomes much clearer (figure 12.5). It's almost certainly a tree! The main trunk and the leafy canopy are well-defined. The model has now locked-in the high-level concept. Its task is no longer about figuring out the overall structure, but rather, about refining the details, carving out the smaller branches, and adding texture to the leaves. Can you identify the areas that need to be focused on to reveal more details about the tree?

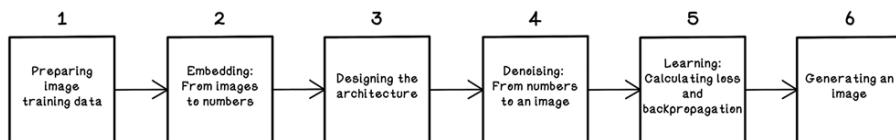


**Figure 12.6 An example of denoising after all timesteps**

Finally, after hundreds of steps, we arrive at the final, crisp image (figure 12.6). We can clearly see the tree, complete with detailed bark, individual leaf clusters, and a coherent structure. The model has successfully removed all the noise that was inconsistent with its goal. This reveals that the tree was hidden within the initial random noise. The journey from a noisy blob to a sharp final image is the essence of the diffusion process. This concept likely seems counterintuitive. When we think of image generation, we might think that it's about drawing and painting because this is how humans create images, but it's the inverse: starting with noise and iteratively removing the noise that doesn't belong there.

### 12.2.1 A generative image model training workflow

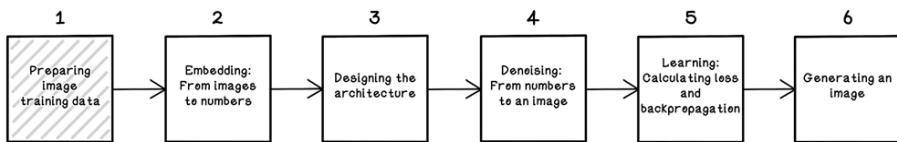
The workflow for training a generative image model involves several key stages, from preparing a vast visual library to teaching the model how to reverse a noising process to create new images (figure 12.7).



**Figure 12.7 A workflow for training a diffusion model**

1. *Preparing image training data:* Before the model can learn to create, it must study existing images. This step involves collecting a massive dataset of images and captions associated with each image, then cleaning and standardizing them so the model has high-quality examples to learn from.
2. *Encoding: From images to numbers:* Computers understand numbers, not pixels or text. This step converts images into numerical data and text captions into embeddings. Crucially, it also includes forward diffusion, where clean training images have noise added to them in stages to create the core training examples.
3. *Designing the architecture:* This step involves choosing the model's structure. For modern image generation, the U-Net architecture is the standard choice because its design is perfect for image-to-image tasks like denoising.
4. *Denoising: From numbers to an image:* This is the U-Net's main task during training. The model is given a noisy image and learns to predict the exact noise that was added to it, using the text caption as a guide.
5. *Learning: Calculating loss and backpropagation:* To learn, the model needs to know how wrong its predictions are. In this step, the model's predicted noise is compared to the actual noise that was added. The difference (loss) is used to update all the model's weights through backpropagation, making it a better noise predictor over time.
6. *Generating an image:* Once trained, the model can generate new images. This is the reverse of the training process. It starts with a canvas of pure random noise and, guided by a user's text prompt, iteratively subtracts the predicted noise until a clean, final image is revealed.

## 12.3 Preparing image training data



**Figure 12.8 Preparing image data in the diffusion model training workflow**

Before a model can learn to create art, it must first go to a “virtual art school”. This first step is curating a massive library of images that the model will learn from (figure 12.8). The resolution quality, diversity, and most importantly, the description of each image will define everything that the model is capable of. Unlike an LLM that learns from text alone, a generative image model learns from the relationship between an image and its caption. If we want to build a generalized model, the dataset would usually be billions of image and text pairs of a huge variety of images like paintings, portrait photography, digital art, satellite photographs, and more.

### 12.3.1 Selecting and collecting image data

As with most machine learning models, a diffusion model’s ultimate ability is a direct reflection of the data that it’s trained on. A model trained only on paintings will not know how to create a photorealistic image. If we train using only photos of landscapes, the model can’t create cartoons. The goal is to collect a dataset that is massive, diverse, and accurately captioned. Some common sources for image data include:

- *Web-crawled data*: The largest datasets are created by crawling the public internet, gathering billions of images and their associated “alt-text” or summarizing the surrounding text. This approach provides immense scale and diversity but requires the most amount of cleaning and filtering to remove low-quality images and irrelevant captions. These images are often poorly captioned with varying aspect ratios and resolutions.
- *Public art & museum archives*: Collections from art museums provide access to high-resolution scans of classical and modern art. This data is good for teaching the model about different artistic styles, from Impressionism to Surrealism. This is how a model is able to understand the “style” of Vincent Van Gogh.
- *Scientific archives*: Specialized datasets of biological diagrams, astronomical images from telescopes, or medical scans teach the model highly specific and technical visual information that is usually not available in general web data.

- *Stock photography databases*: Stock photo sites contain millions of high-quality, professionally shot photographs. Crucially, these images come with detailed, descriptive captions written by the photographers, making them ideal for teaching the model about realism, object composition, and specific terminology.
- *Satellite and geospatial data*: To teach a model about geography, building layouts, or environmental patterns, we can use satellite imagery from sources like NASA and the European Space Agency. This allows the model to generate realistic top-down views of cities, coastlines, and landscapes.
- *Synthetic data & 3D renders*: An increasingly popular source is purely computer-generated imagery. Training on synthetic data from 3D environments created with 3D modeling software gives the model access to perfectly labeled images with precise control over lighting, angle, and composition of the scene. This is also a powerful way to avoid many of the copyright and privacy issues inherent in web-scraped data, although it can be time consuming and requires experts in design to create.

## THE CRUCIAL ROLE OF CAPTIONS

An image without a good caption is useless for training a text-to-image model. The caption is the key that connects the visual data (pixels) to the linguistic concepts (words). The quality of the captions directly determine the model's ability to follow detailed prompts, and generate an accurate image. Let's explore what good and bad captions look like:

- *A good caption*: "A photorealistic close-up of a red-tailed hawk perched on a pine branch, its feathers ruffled by the wind, with a soft-focus lush green forest background". This teaches the model about specific subjects like "red-tailed hawk", actions like "perched", and styles like "photorealistic" and "soft-focus".
- *A bad caption*: "image1.jpg" or "bird". A model trained on millions of pairs like this would learn to associate the word "bird" with many different species in many different poses and situations, and it would struggle to generate a specific type of bird when prompted for one.

## LICENCING OF CONTENT

The data collection process for image models has even more complex legal and ethical challenges than pure text data that we use to train LLMs. Some concerns include: Images are more often copyrighted than not, artists hone their craft and develop their unique style which can then be "stolen", and the philosophical and ethical question about if image generation is doing more good than bad for us.

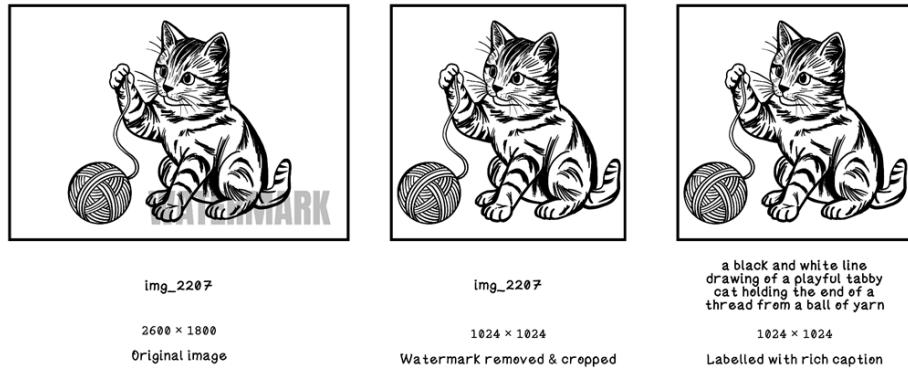
- *Copyright*: Most images found online are copyrighted. Using an artist's work to train a commercial model without a license is a major legal and ethical issue. This is why many foundational models should be trained on datasets with permissive licenses or on large, licensed stock photo archives.
- *Style Mimicry*: Although image generation models create *unique* images each time, they can also create recognizable styles of living artists. This raises profound ethical questions about originality, consent, and compensation. Without that artist's work, the model could not produce that style of image. Furthermore, mass generation of a specific art style could devalue it, making it a commodity.
- *Representation and bias*: If the training data disproportionately shows people of a certain demographic or people in certain stereotypical roles, like "doctors are male" and "nurses are female", the model will learn and amplify these biases. Curating a diverse and representative dataset is one of the most important and difficult challenges in responsible AI development.
- *Misuse of likeness*: Training a model on photographs scraped from the internet without consent can be a major privacy violation. When a model learns a person's face, it has effectively captured that person's visual likeness. This can be used to generate new synthetic images of that person in situations that they were never in, potentially for harassment, misinformation, or creating fake endorsements - going beyond privacy issues and entering the realm of identity appropriation.

### **12.3.2 Cleaning and preprocessing image data**

Just as raw text from the internet is messy when collecting LLM training data, image datasets are filled with "visual noise" that can confuse the model and degrade its performance. The goal of preprocessing is to create a clean, standardized, and high-quality dataset before training begins. Imagine we have a typical image scraped from a pet blog. It might be a high-resolution landscape aspect ratio JPG, have a large watermark from the website, and a non-descriptive filename like "img\_2207" as its only caption (figure 12.9). This single image-text pair must go through some cleaning steps.

- *Resolution and aspect ratio standardization*: Image models are most efficient when they train on images of the same size. This means resizing all images to a standard resolution, like  $512 \times 512$ . Images that are too small are often discarded because they lack detail, while non-square images are usually cropped or padded to fit the target aspect ratio.

- *Watermark and overlay removal:* Watermarks, logos, timestamps, and social media borders are visual noise. If left in the image, the model will learn to replicate them, leading to generated images that are littered with nonsensical text and artifacts. If these artefacts are consistent in the training set, they will be emphasised in the generated images because the model will learn that strong pattern from seeing it many times. Automated tools are used to detect and remove or paint over these overlays, ensuring that the model learns from the content of the image itself.

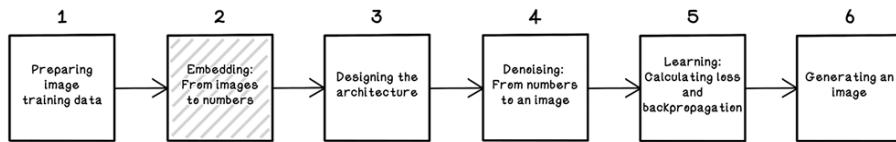


**Figure 12.9 Cleaning image data for training**

- *Duplicate removal:* The same image often appears many times across the web. Removing identical or very similar images prevents the model from becoming over-trained on specific examples and encourages it to learn more general concepts.
- *Corrupted file removal:* Large datasets often contain corrupted files that cannot be opened or are incomplete. These must be filtered to prevent errors during training.
- *Safety and content filtering:* This is a critical step for responsible AI development. The dataset must be scanned to identify and remove harmful or inappropriate content, such as graphic violence or explicit material. This filtering helps ensure that the final model does not generate harmful or offensive images.

- *Caption cleaning and enhancement:* In our example, the initial caption "img\_2207" is not helpful. This is where a significant amount of work can be done to generate a rich caption. Often, a separate, powerful "captioning model" is used to analyze the cleaned image and generate a new, descriptive caption. For our cat example, it might generate: "a black and white line drawing of a playful tabby cat holding the end of a thread from a ball of yarn". This transforms a low-quality data point into a high-quality one.

## 12.4 Embedding: From images to numbers



**Figure 12.10 Embedding in the diffusion model training lifecycle**

We now understand the critical role of collecting and cleaning data and what techniques can be applied. The next fundamental challenge is that computers don't see images or read text like people do - computers only understand numbers. The process of preparing data for the model involves converting the training images into numerical representations, as well as preparing the text label for the image as a numerical representation (figure 12.10).

For our working example, we will train the model to learn how to generate "a happy face", "a sad face", and a neutral face. For the sake of simplicity in following all the steps, we will use an  $8 \times 8$  grayscale image as our training data (figure 12.11).



**Figure 12.11 Example basic image training data**

## IMAGE NORMALIZATION

A computer can see an image as a grid of pixels, where each pixel has a value typically from 0 (black) to 255 (white) in a grayscale image like our examples. While computers can work with these numbers and use them to render colors, deep learning networks train effectively when the input values are small and centered around zero (to leverage gradient descent as explored in Chapter 9). So, we need to normalize the pixel values. A common practice is to scale the 0 to 255 range to a -1 to 1 range. This simple conversion makes the training process stable and efficient.

In Figure 12.12, the image data is an  $8 \times 8$  matrix where 1 indicates complete black, and -1 indicates complete white, and all numbers between -1 and 1 are different shades of gray.

1	1	1	1	1	1	1	1	1
1	-1	-1	-1	-1	-1	-1	-1	1
1	-1	1	-1	-1	1	-1	1	
1	-1	-1	-1	-1	1	-1	1	
1	-1	1	-1	-1	1	-1	1	
1	-1	1	1	1	1	-1	1	
1	-1	-1	-1	-1	-1	-1	-1	1
1	1	1	1	1	1	1	1	1

= [[1, 1, 1, 1, 1, 1, 1, 1, 1],  
 [1,-1,-1,-1,-1,-1,-1,1],  
 [1,-1, 1,-1,-1, 1,-1,1],  
 [1,-1,-1,-1,-1,-1,-1,1],  
 [1,-1, 1,-1,-1, 1,-1,1],  
 [1,-1, 1, 1, 1, 1,-1,1],  
 [1,-1,-1,-1,-1,-1,-1,1],  
 [1, 1, 1, 1, 1, 1, 1, 1]]

## a happy face

**Figure 12.12 How images are represented as normalized numbers**

Most actual training images are in full color and use the RGB (Red, Green, Blue) color scheme. Instead of a single grid of pixels, a color image is composed of three separate grids (also called channels) stacked on top of each other. Each channel represents the intensity of Red, Green, or Blue, with values for each pixel also ranging from 0 to 255.

To normalize a color image, we perform the exact same scaling operation on each of the three channels separately.

If we have the color pixel to normalize, we can scale it using the formula:

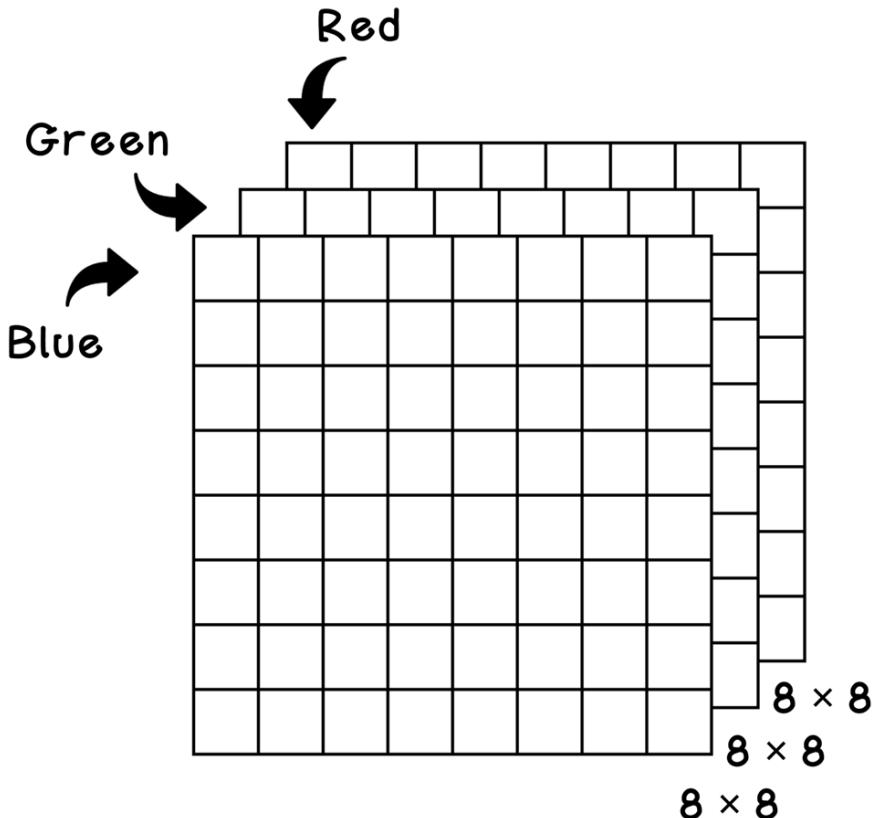
**(original value / 127.5) - 1**

Note that we divide by 127.5 to scale the result between 1 and -1. If we use 255, we will scale to between 1 and 0 which is not what we want.

Red: 100, Green: 150, Blue: 250 will result in:

- Scaled Red: -0.22
- Scaled Green: 0.18
- Scaled Blue: 0.96

After doing this for all pixels in the image, the overall result will be represented as 3 separate matrices, one for each color, all dimensions of the original image (figure 12.13).



**Figure 12.13 How RGB images are represented as matrices**

## FORWARD DIFFUSION

The core of diffusion training is teaching the model to denoise an image. This means iteratively removing noise in a way that makes sense for the goal of revealing the correct final image. To do this, we first need a lot of noisy images to practice with. This is done through the *forward diffusion* process, where we take a clean image from our dataset and deliberately destroy it by adding noise step-by-step.

Imagine taking a perfectly clear photograph and gradually adding layers of static. After one step, it's slightly fuzzy. After a hundred steps, the original image is barely visible. After a thousand steps, it's nothing but pure noise.

Each time we add noise to the image, we save a snapshot that includes:

- The noisy image at that step number
- The exact noise pattern we added to get to that noisy image

Keeping track of the noise pattern added at a specific timestep is critical for the model to understand how noise was added to an image, and what that resulted in. This is called a *noise schedule*.

The algorithm creates different levels of noise at each timestep, derived from the original image each time. This is important to note: a new timestep does not add noise to the previous noisy image, always to the original image.

The values that control the noising schedule are the following:

- *Beta*: We start by defining a noise schedule that gradually increases from a small number at timestep 1 to a larger number at the final timestep, T (the total number of timesteps to create). This controls how much noise is added at each individual step. For example, we might start at 0.0001 and end at 0.02 over 100 steps. This means that Beta will be 0.0001 at timestep 1 and 0.02 at timestep 1000.
- *Alpha*: Alpha is simply defined as 1 - Beta at a specific timestep. It represents the *signal rate*. The signal rate is how much of the original image information is preserved at each step.
- *Alpha-Bar (Cumulative product)*: This value represents the total signal remaining after  $t$  steps. Even though we are “jumping” directly to a specific timestep, the math relies on the idea that the image decays sequentially. At step 1, we keep 99% of the signal. At step 2, we keep 99% of that remaining signal. To jump straight to step 100, we must calculate the cumulative product of all those previous decay rates to determine exactly how much original signal should remain and how much noise should replace it.

So given any timestep,  $t$ , we can calculate out Beta, Alpha, and Alpha-Bar, as shown in Table 12.1.

**Table 12.1 A subset of values for the noise schedule**

Timestep (t)	Beta	Alpha calculation (1 - beta)	Alpha	Alpha-Bar calculation	Alpha-Bar
1	0.00010	1 - 0.00010	0.99990	0.99990	0.99990
2	0.00012	1 - 0.00012	0.99988	0.99990 × 0.99988	0.99978
3	0.00014	1 - 0.00014	0.99986	0.99990 × 0.99988 × 0.99986	0.99964
...					
1000	0.02000	1 - 0.02000	0.98000		0.00004

Using a formula we can calculate the new value for each pixel given the original pixel value, the Alpha-Bar value, and a random number sampled from a standard distribution. In Figure 12.14, we are calculating the random noise for the top-left pixel of the image, i.e. the pixel at 0,0 of the matrix.

```

noisy image      = original image *  $\sqrt{\text{alpha-bar}}$  +  $\sqrt{1 - \text{alpha-bar}} * \text{random}$ 
noisy image[0][0] = 1           *  $\sqrt{0.99990}$  +  $\sqrt{1 - 0.99990} * 0.65$ 
noisy image[0][0] = 1.00645
So, the top-left pixel is 1.00645

```

**Figure 12.14 Calculations for adding noise to an image**

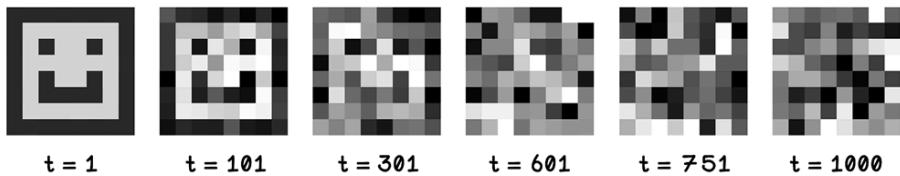
**EXERCISE: WHAT IS THE NOISE FOR PIXEL [1, 1] IF THE RANDOM NUMBER IS 0.25?**

Calculate the noise for pixel [1,1].

**SOLUTION: WHAT IS THE NOISE FOR PIXEL [1, 1] IF THE RANDOM NUMBER IS 0.25?**

0,9974499987

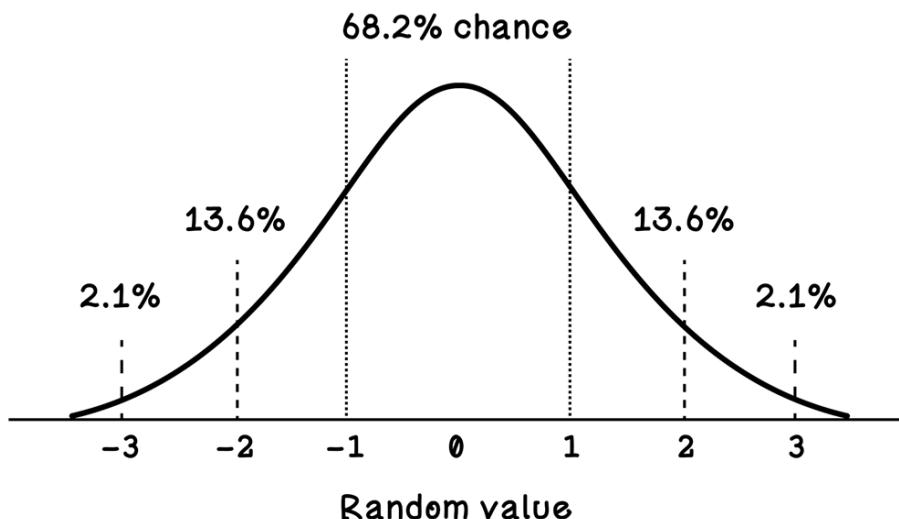
In Figure 12.15, we see the forward diffusion results for “a happy face” image. Note that  $t = 1000$  is not the result of adding noise to  $t = 999$ , but rather the result of adding a lot of noise to the *original* training image.



**Figure 12.15 A progression of adding noise to an image over 1000 timesteps**

This process creates our training pairs of noise and text labels. The model will be given the noisy image and its job is to predict the exact noise pattern that was added. By doing this millions of times with millions of different images and noise levels, it becomes an expert at seeing a noisy image and knowing precisely what noise needs to be removed to make it cleaner.

You might be wondering why we sample a random number from a standard normal distribution. It's because the random number and resulting noise is statistically predictable because most numbers will be centred around 0, with a lower likelihood of the number being between -1 and 1, and even less likelihood of it being between -2 and 2 (figure 12.16). This process is called Gaussian noise. Its unique mathematical properties make both the forward (noising) and reverse (denoising) processes analytically tractable and learnable.



**Figure 12.16 A view of probability of selection in a normal distribution**

## PYTHON CODE SAMPLE

This code describes the forward diffusion process, which gradually adds noise to a clean image over a series of timesteps to generate training data for a diffusion model. It begins by defining a noise schedule (*beta\_schedule*) and computing the corresponding alpha and cumulative *alpha\_bar* values. For each timestep, it samples Gaussian noise and mixes it with the original image using a weighted combination determined by *alpha\_bar*. This produces a progressively noisier version of the image (*noisy\_image*) and stores the exact noise used (*noise\_patterns*) for each step, which are later used to train the model to reverse the process.

```

def define_schedules(timesteps):
    beta_min = 0.0001
    beta_max = 0.02

    beta_schedule = np.linspace(beta_min, beta_max, timesteps)

    alpha_schedule = 1.0 - beta_schedule

    alpha_bar_schedule = np.cumprod(alpha_schedule)

    return beta_schedule, alpha_bar_schedule

def forward_diffusion(image, timesteps):

    _, alpha_bar_schedule = define_schedules(timesteps)

    noisy_images = []
    noise_patterns = []

    for t in range(timesteps):
        alpha_bar = alpha_bar_schedule[t]

        random_noise = np.random.randn(*image.shape)

        sqrt_alpha_bar = np.sqrt(alpha_bar)
        sqrt_one_minus_alpha_bar = np.sqrt(1.0 - alpha_bar)

        noisy_image = (sqrt_alpha_bar * image) + (sqrt_one_minus_alpha_bar * random_noise)

        noisy_images.append(noisy_image)
        noise_patterns.append(random_noise)

    return noisy_images, noise_patterns, alpha_bar_schedule

```

## TIMESTEP EMBEDDING

With an understanding of how timesteps and the noise schedule can indicate the amount of noise added, we will need a way to provide the timestep number to the model for each noisy image being trained with. This is important because an image that is only slightly noisy, like timestep 10, needs a very different, more subtle denoising operation than an image that is almost pure static, like timestep 900.

We give the model this information by creating a timestep embedding. Much like the positional encoding we saw in the LLM chapter, we use a *sinusoidal formula* to convert the integer timestep, for example  $t = 22$ , into a unique vector. This vector gives the model a precise, mathematical representation of the noise level, allowing it to adjust the strength of its denoising prediction accordingly.

To find the values, we use the formula in figure 12.17:

$$\text{angle} = \left( \frac{t}{1000^{\frac{i}{\text{half dimension}}}} \right)$$

↗

embedding dimensions = 16,  
so half dimensions = 8

So, given half dimensions = 8, and  $t = 22$ ,

$$\begin{aligned}
 d = 0 \text{ angle: } 22 / 1000^{(0 / 8)} &= 22 / 1 \\
 d = 1 \text{ angle: } 22 / 1000^{(1 / 8)} &= 22 / 3.16 \\
 d = 2 \text{ angle: } 22 / 1000^{(2 / 8)} &= 22 / 10 \\
 d = 3 \text{ angle: } 22 / 1000^{(3 / 8)} &= 22 / 31.62 \\
 d = 4 \text{ angle: } 22 / 1000^{(4 / 8)} &= 22 / 100 \\
 d = 5 \text{ angle: } 22 / 1000^{(5 / 8)} &= 22 / 316.23 \\
 d = 6 \text{ angle: } 22 / 1000^{(6 / 8)} &= 22 / 1000 \\
 d = 7 \text{ angle: } 22 / 1000^{(7 / 8)} &= 22 / 3162.23
 \end{aligned}$$

**Figure 12.17 Calculations for timestep embeddings**

After calculating this for all timesteps, we will have something like Table 12.2.

**Table 12.2 Calculations for creating the timestep embedding**

<b>Index (i)</b>	<b>angle calculation</b>	<b>angle</b>	<b>sin(angle)</b>	<b>cos(angle)</b>
0	22 / 1	22	-0.0089	-0.9999
1	22 / 3.16	6.957	-0.6277	0.7784
2	22 / 10	2.2	0.8085	-0.5885
3	22 / 31.62	0.696	0.6411	0.7674
4	22 / 100	0.22	0.2182	0.9759
5	22 / 316.23	0.070	0.0699	0.9975
6	22 / 1000	0.022	0.0220	0.9997
7	22 / 3162.23	0.007	0.0070	0.9999

Finally our timestep embedding for  $t = 22$  will be all the  $\sin(\text{angle})$  values concatenated with all the  $\cos(\text{angle})$  values. This will result in a vector of size 16 (figure 12.18):

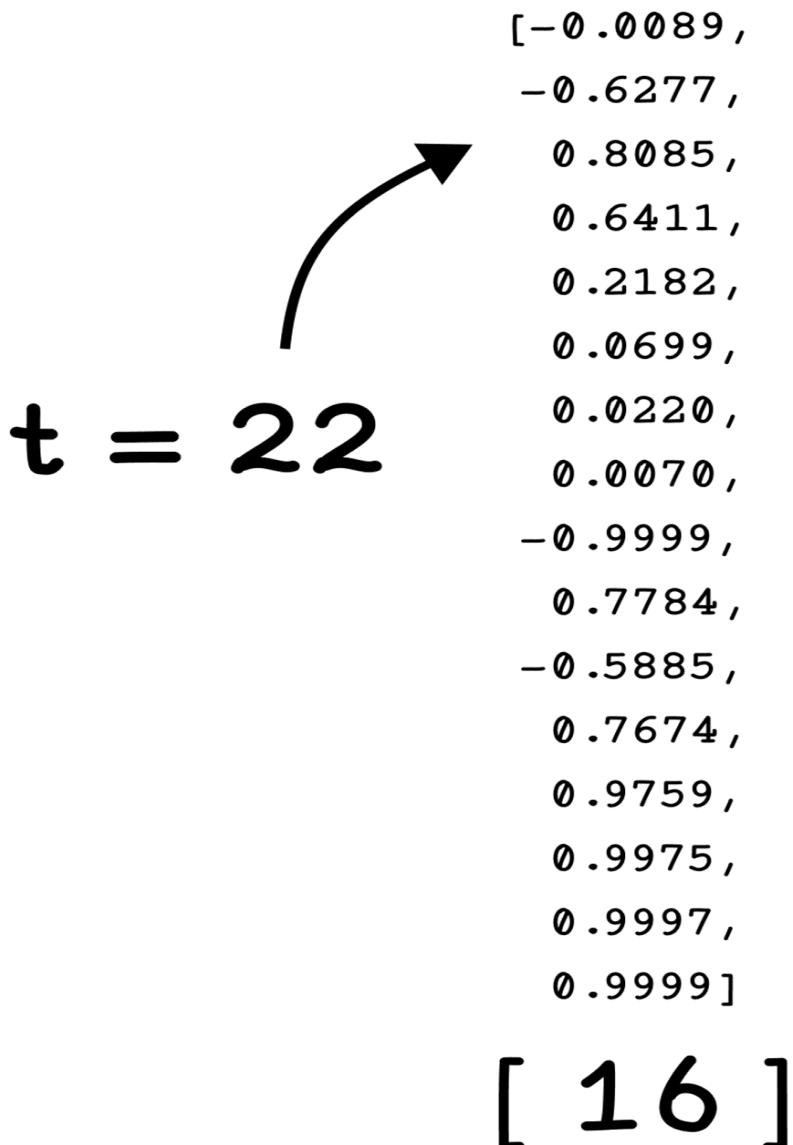


Figure 12.18 The embeddings for timestep 22

## PYTHON CODE SAMPLE

This code generates a timestep embedding vector using a sinusoidal encoding scheme. For a given timestep and embedding size, it computes each element of the vector using alternating sine and cosine functions, scaled by exponential factors of the timestep. This creates a unique, continuous representation of the timestep that captures both absolute and relative position information. The resulting embedding helps the model understand how much noise has been added and how aggressively it should denoise at each stage.

```
def create_timestep_embedding(timestep, embedding_size):
    embedding_vector = np.zeros(embedding_size)

    for i in range(embedding_size):
        exponent = i / embedding_size

        div_term = np.power(10000, exponent)

        argument = timestep / div_term

        if i % 2 == 0:
            value = np.sin(argument)
        else:
            value = np.cos(argument)

        embedding_vector[i] = value

    return embedding_vector
```

## TEXT LABEL EMBEDDING

The model understands images as numbers, but we still have the problem of the text labels. To guide the denoising process, we need to translate the text label like “a happy face” into a numerical representation. We don’t need to train a text model (like a LLM) from scratch; instead, we can leverage a powerful pre-trained text encoder. These models read the text label and output a single, information-rich vector, the embedding. This vector represents the semantic meaning of the label.

After being processed by a text encoder, “a happy face” can be expressed as the following, resulting in an array of size 16 (figure 12.19):

a  
happy  
face



```
[1.0,  
 0.5,  
 0.25,  
 0.125,  
 1.0,  
 0.5,  
 0.25,  
 0.125,  
 1.0,  
 0.5,  
 0.25,  
 0.125,  
 1.0,  
 0.5,  
 0.25,  
 0.125]
```

[ 16 ]

Figure 12.19 The embeddings for “a happy face”

Once we have the initial text embedding, there's one more crucial step: we process it through a normal feed-forward network, like a Multi-Layer Perceptron (MLP). This is a small, linear feed-forward neural network that acts as a specialized translator. The text encoder provides a generic, all-purpose embedding - think of it as a direct dictionary translation. The feed-forward network's task is to take the generic embedding and transform it into a specialized "guidance" vector that is tailored to the unique "language" of our model. It converts the embedding, making it more expressive and useful for our specific model to work with. This network's weights will also be adjusted based on the loss during backpropagation.

More sophisticated architectures use actual multi-layer ANNs instead of a simple MLP.

## PYTHON CODE SAMPLE

This code describes how to create a text embedding vector for a given label to guide image generation. It first uses a pre-trained tokenizer and text encoder to convert the label into a series of token embeddings, which are then averaged into a single vector. This embedding is passed through a two-layer feedforward network with SiLU activation functions, using learned projection weights to transform the generic embedding into a specialized vector (figure 12.20). The result is a refined representation of the text prompt that can be injected into the model to condition and guide the denoising process.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def silu(x):
    return x * sigmoid(x)

def create_text_embedding(text_label, embedding_size):

    token_count = 6
    mock_token_embeddings = np.random.randn(token_count, embedding_size)

    text_embedding = np.mean(mock_token_embeddings, axis=0)

    text_embedding = text_embedding.reshape(1, -1)

    projection_weights_1 = np.random.randn(embedding_size, embedding_size)

    sum_hidden = text_embedding @ projection_weights_1
    hidden_layer = silu(sum_hidden)

    projection_weights_2 = np.random.randn(embedding_size, embedding_size)

    sum_final = hidden_layer @ projection_weights_2
    final_embedding = silu(sum_final)

    return final_embedding.flatten()
```

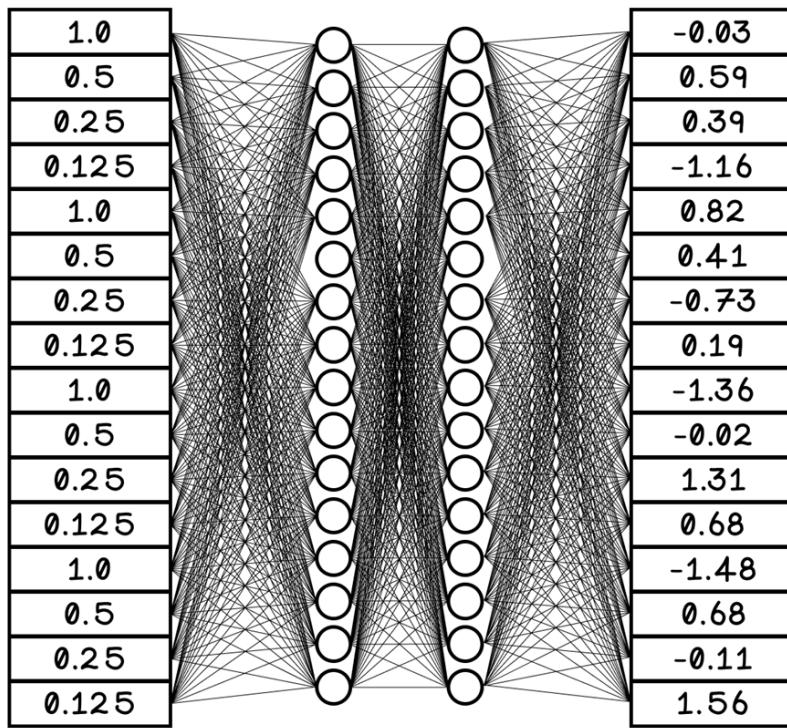


Figure 12.20 A view of a feed-forward ANN processing the text embedding

Similarly, the timestep embedding will also be passed through a feed-forward network that will result in a vector of size 16. It's done for the same reason: to transform the "generic" timestep embedding vector into a vector that is tailored to our generative image model.

Here's an overview of the input data that we have available to feed into the model for training. We will have  $T$  many of the following (figure 12.21):

- *Noise image*: Each being an  $8 \times 8$  matrix.
- *Timestep embedding*: Each being a vector of size 16 after sinusoidal embedding and its feed-forward network.
- *Text label embedding*: Each being a vector of size 16 after being passed through a text encoder and its feed-forward network.

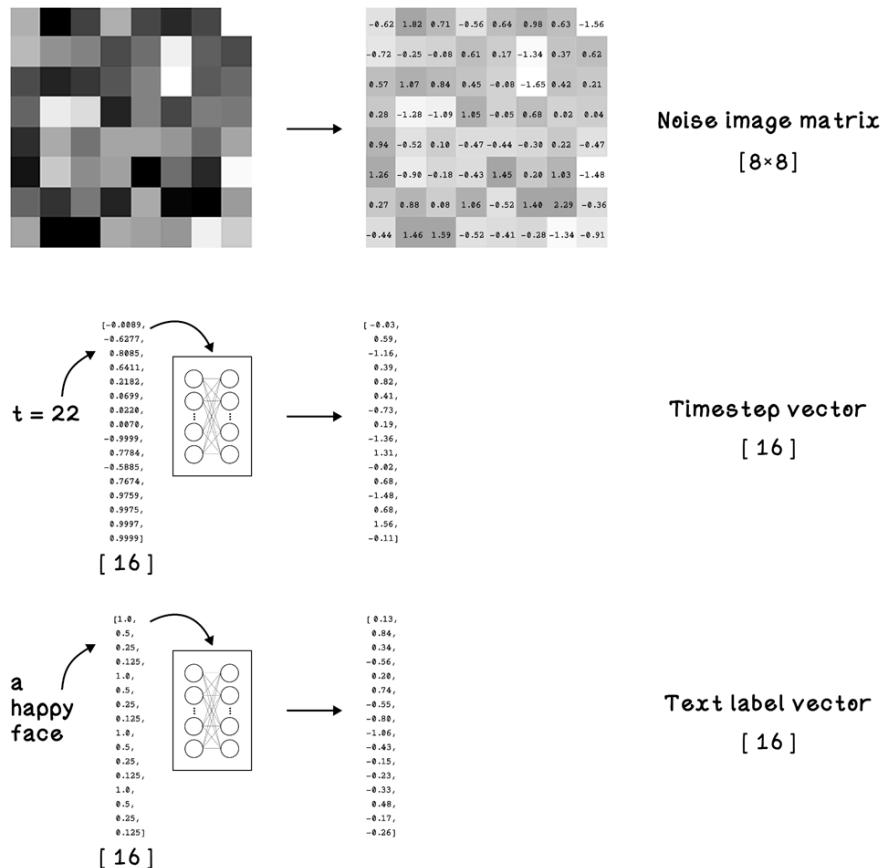


Figure 12.21 A complete representation of one training data input

## 12.5 Designing the architecture (and why U-Nets)

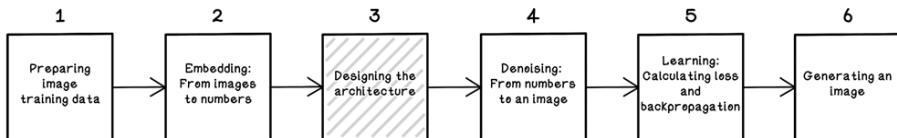


Figure 12.22 Designing the architecture in the Diffusion model training workflow

Now that we have our clean and normalized dataset of image-text pairs as numeric representations, the next step is to choose the training architecture (figure 12.22). This is the “brain” of our operation and the specific type of deep learning architecture that will learn to perform the denoising task.

As seen with LLMs in Chapter 11, a major decision in training specialized generative models is choosing the right architecture for the problem at hand. Different architectures have different trade-offs, and for image generation, the key concerns are:

- *Spatial awareness*: Can the model understand the 2D layout of an image? It needs to know that a pixel's neighbors are far more important than pixels on the other side of the image.
- *Multi-Scale processing*: Can the model see both the “big picture” (the overall composition) and the fine-grained details (textures and edges) at the same time?
- *Parameter efficiency*: Can the model learn complex visual patterns without requiring an unmanageable number of weights, which would make it too slow and expensive to train?

### 12.5.1 Convolutional Neural Networks (CNNs)

Famous for their power in image classification, CNNs are the foundation of modern computer vision. They use sliding filters to detect hierarchies of features. They find anything from simple edges in early layers of the network to complex objects like faces and cars in later layers. While they are masters of analyzing images, they are not inherently designed for generating images from scratch. However, CNNs are used extensively in the U-Net architecture, which is an innovation that makes generating images with diffusion possible.

Let’s explore how CNNs differ from linear feed-forward networks that we’ve been using thus far in the book.

Because we are working with image data, the positions of the “pixels” are of huge importance. The positions of the pixels are just as important as their color when learning and generating coherent images - this is where CNNs are powerful.

Figure 12.23 illustrates the difference between an ordinary linear feed-forward layer and a convolutional layer. Notice that the node in the first one processes all of the inputs provided linearly to learn relationships between them. However, the *convolutional layer* analyzes different regions of the matrix independently to find relationships in a region. CNNs let learning happen with the spatial data intact.

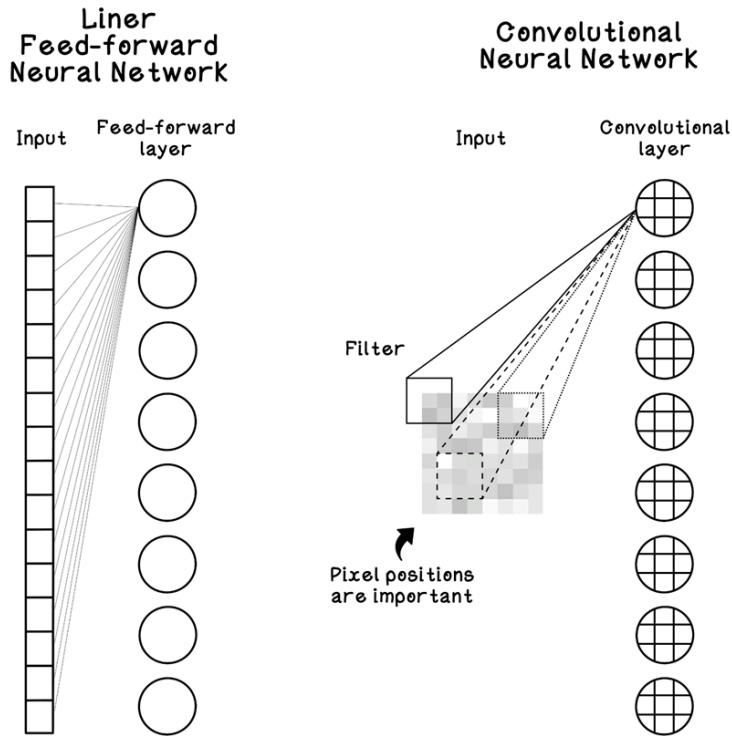


Figure 12.23 The difference between an ANN and a CNN

## CNN INPUT SHAPE

Each node in a convolutional layer takes a matrix (2D grid shape) as input. In the case of our grayscale example, we only have 1 channel so the input will be an  $8 \times 8$  matrix. In a full-color image, we would have 3 channels for *Red*, *Green*, and *Blue* resulting in an  $8 \times 8 \times 3$  tensor. If we're training with PNGs that can have transparency, we would have 4 channels to include the *Alpha* value for the pixels.

## THE FILTER (AKA KERNEL)

The core of a convolutional layer is a small filter, usually a grid that's  $3 \times 3$  in size. These are weights that act as a specialized feature detector. See one filter as a tiny magnifying glass that is trained to find one specific pattern. In the early layers of a network, these patterns are very simple, like:

- One filter might learn to detect vertical edges. Its grid of weights might have positive numbers on the left, negative numbers on the right, and zeros in the middle.
- Another filter might learn to find horizontal edges.
- Others might learn to recognize simple textures or color gradients.

Crucially, the model isn't told what features to look for. These filter weights start as random numbers and are adjusted during the training process. If detecting corners is useful for the model's final goal, then through backpropagation, some filters will naturally evolve into corner detectors because doing so helps reduce the overall loss.

## THE CONVOLUTION OPERATION

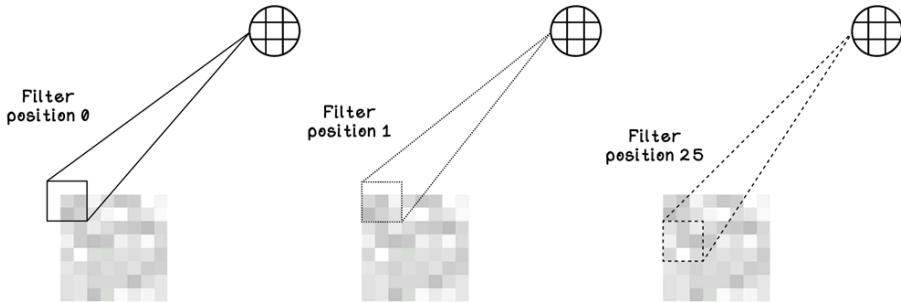
Each node in a convolutional layer “slides” a  $3 \times 3$  filter over every possible position of the input image. At each position, it calculates the dot product between the filter's weights and the image pixels underneath it.

The process starts at the top-left corner of the image. The filter lays over that patch, and the calculation is performed. Then, the filter slides one pixel to the right and repeats the calculation. This continues until it reaches the end of the row. It then moves down one pixel and starts again from the left edge, continuing this scan until every part of the image has been “viewed” by the filter (figure 12.24).

This approach preserves the spatial relationships of the input image. A feature detected in the top-left of the image will result in an activation in the top-left of the output, maintaining the structure and features.

But what happens when the filter reaches the edge of the image? If a  $3 \times 3$  filter tries to center on a border pixel, part of the filter will hang off the edge. To solve this and control the output size, we use a technique called padding. Padding involves adding a border of extra pixels, usually zeros, around the entire input image before convolution begins. This has two major benefits:

- *It preserves the image size:* With the right amount of padding, the output feature map can have the exact same height and width as the input image. This is essential for building very deep networks without the image shrinking.
- *It processes edges properly:* Padding allows the filter to be centered over every pixel of the original image, including the corners and edges, ensuring that the information at the borders is not lost or under-represented during training.



**Figure 12.24 How a CNN filter scans an image**

## THE FEATURE MAP

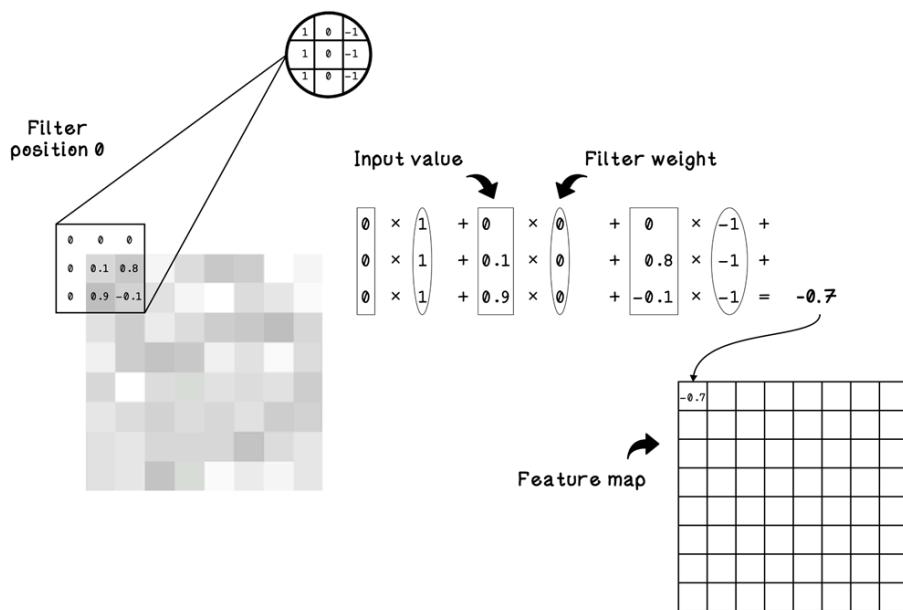
The output of sliding one filter over the entire input is a 2D grid called a *feature map*, this is a new “image” that shows where the filter’s specific feature was detected.

The value of each pixel in this feature map is the result of a dot product. At each position, the numbers in the filter are multiplied by the corresponding pixel values in the patch of the image directly underneath it, and the results are all summed up to produce a single number.

- If a filter is designed to find a vertical line, and it’s currently over a vertical line in the image, the dot product will result in a large positive number.
- If it’s over a blank area, the result will be near zero.
- If it’s over a feature that is the opposite of what it’s looking for, like a horizontal line, it might result in a large negative number.

The final feature map is therefore a new, filtered image that highlights the presence and strength of one specific feature across the entire input image. A convolutional layer has many nodes with filters, each producing its own feature map. These feature maps are passed to the next layer. More on this in the *Denoising* section.

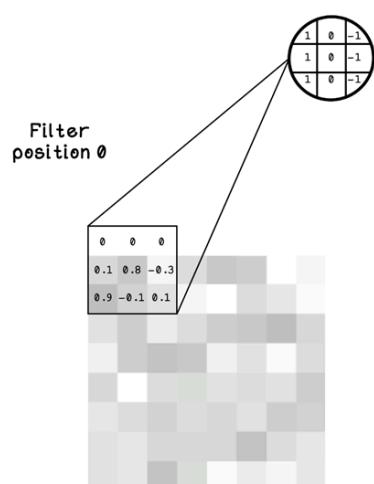
Let’s explore how the filter weights are initialized, and how it produces a value from evaluating the pixel values. In Figure 12.25, our  $3 \times 3$  filter is initialized with a column of 1’s, a column of 0’s, and a column of -1’s. This specific filter is a vertical line detector. It’s designed to produce a high value when it detects a sharp change from dark to light. We’re using this for convenience. Remember that the network would learn this through training.

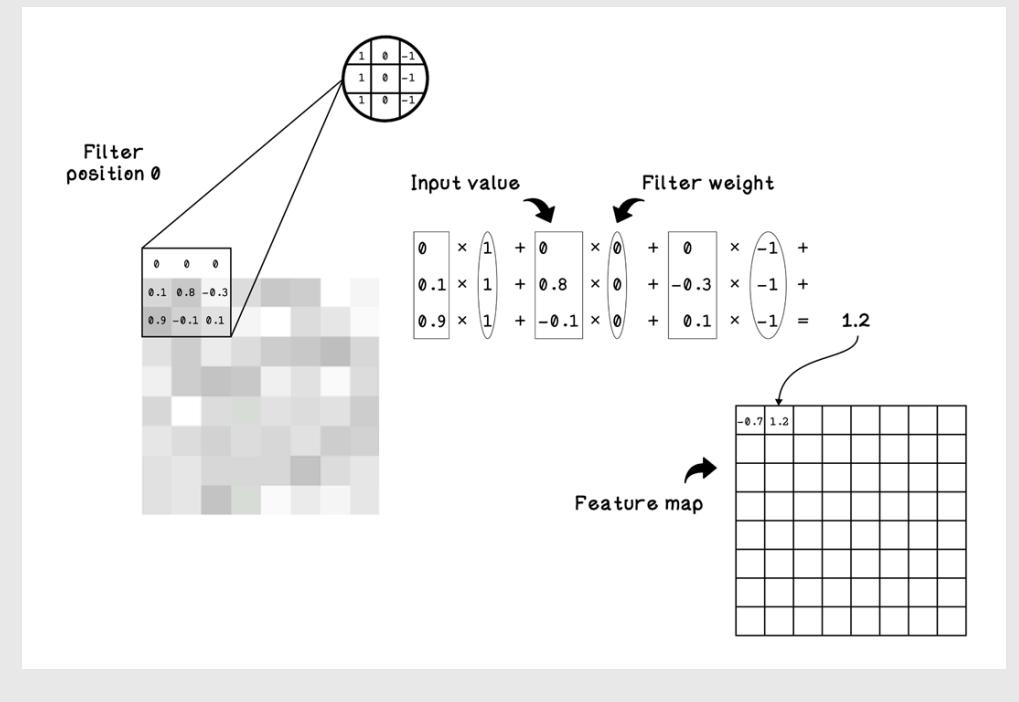


**Figure 12.25 Calculations of a filter on an image for the first pixel**

The result of the filter on the pixels is a dot product between the filter's weights, and the pixel values underneath it. So each filter value is multiplied by the respective pixel value, and all resulting products are summed together. In the above example, it produces the value  $-0.7$  which becomes the first value in the feature map for the top-right value of a  $8 \times 8$  matrix.

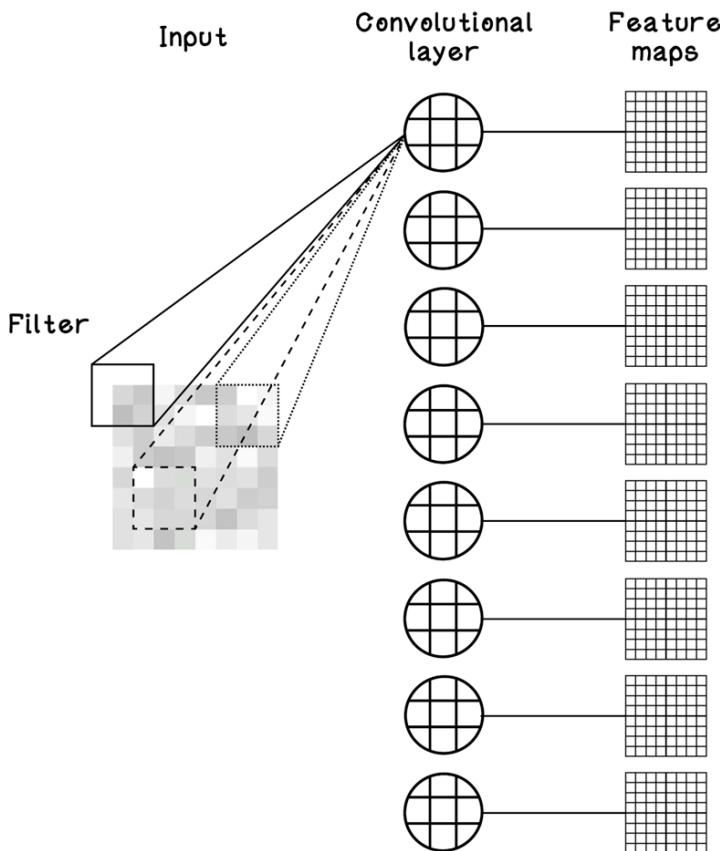
Naturally, this entire process is repeated for every other position that the filter "scans", completing the feature map one pixel at a time.

**EXERCISE: WHAT IS THE OUTCOME OF THE FILTER ON THE NEXT PIXEL POSITION?**

**SOLUTION: WHAT IS THE OUTCOME OF THE FILTER ON THE NEXT PIXEL POSITION?**


With the understanding of how a single node in a convolutional layer works, this exact process is applied to all the other nodes in the layer. We may have emergent nodes to detect vertical lines, horizontal lines, a certain rough texture, and much more. Each node has its own filter weights, and will result in a separate  $8 \times 8$  feature map as depicted in Figure 12.26.

## Convolutional Neural Network



**Figure 12.26 Feature maps produced by a convolutional layer**

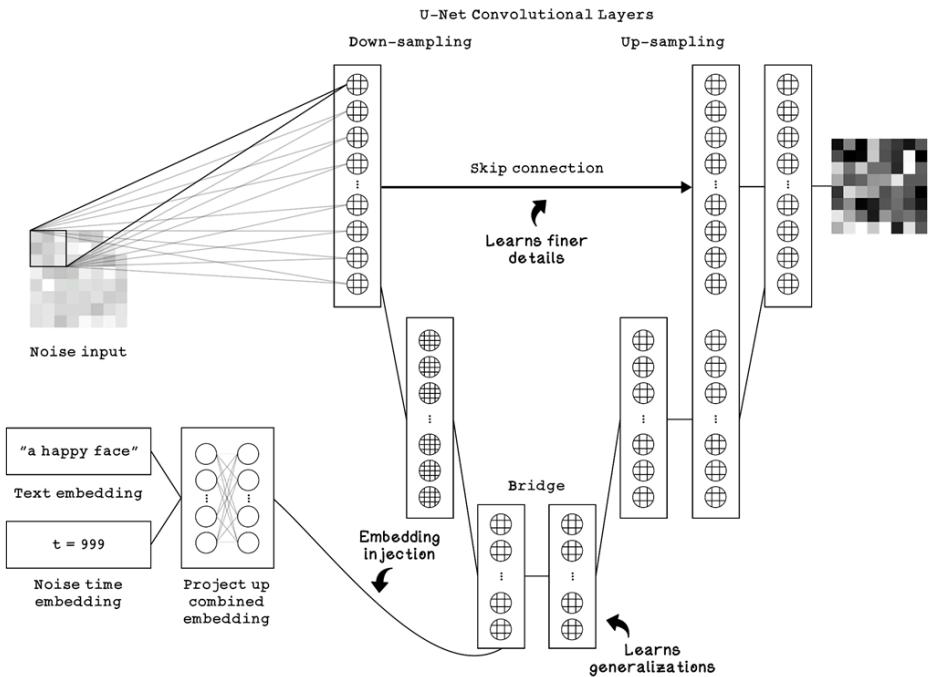
Now that we have a fundamental understanding of CNNs, we're ready to explore the innovation that makes modern image generation possible, the U-Net.

### 12.5.2 The U-Net (A specialized CNN)

The U-Net is the state-of-the-art architecture that powers virtually all modern diffusion models, and it's the one that we will build in this chapter. The U-Net is a special type of CNN designed specifically for image-to-image tasks. Its key innovation is a "U-shaped" design with three core components:

- *Encoder (Down-sample path)*: Think of this path as a summarizer. It progressively shrinks the image using convolutional layers, forcing the network to move beyond individual pixels and capture the high-level context and generalize better. As the image gets smaller, the network gets better at understanding what is in the image, like "this is a face", but loses the precise information about where the fine details are located. This process creates a rich but low-resolution summary of the image's abstract meaning.
- *Decoder (Up-sample path)*: See this path as an artist tasked with reconstructing the full-resolution image from an abstract summary created by the down-sample path. It progressively up-samples the feature maps using transposed convolutions, taking the high-level concepts, like "a face", and attempting to add the fine details back in. By itself, it would struggle to perfectly position every edge and texture because much of the precise spatial information was lost during down-sampling.
- *Skip Connections*: This is the U-Net's super power and the solution to the up-sampling path's problem. A skip connection is a shortcut. It takes the high-resolution feature map from an early stage of the down-sample path, which is rich in fine, spatial details, and feeds it directly to the corresponding layer in the up-sample path. This allows the decoder to combine the abstract, "what" information from the deep layers with the precise, "where" information from the early layers, making it exceptionally good at reconstructing images with high fidelity.
- *The bridge (Also known as the Bottleneck)*: This is the lowest point in the U-shaped path, connecting the end of the down-sample path to the start of the up-sample path. It processes the most compressed, abstract representation of the image. In a diffusion model, this is the critical stage where the timestep and text embeddings are injected, combining the model's understanding of the image with the guidance of the text label before reconstruction begins.

Figure 12.27 illustrates the U-Net. Notice that the initial stage of preparing the training has already been covered in previous sections, and we now need to feed that data into the U-Net itself.



**Figure 12.27 The U-Net architecture**

The unique structure of the U-Net makes it the perfect tool for our diffusion model. It can look at a noisy image, understand the high-level context and use the fine-grained skip connections to precisely predict the noise in every pixel.

Table 12.3 shows the different concerns and hyperparameters for the diffusion model and U-Net. Keep it as a reference as we work through each step in the denoising and learning process.

**Table 12.3 The hyper-parameters for the Diffusion model**

<b>Hyper-parameter</b>	<b>Rule of thumb</b>	<b>Toy Diffusion example</b>	<b>Real-world Diffusion model</b>
Image resolution	Based on training data and available GPU memory	$8 \times 8$	$512 \times 512$ or $1024 \times 1024$
Diffusion Timesteps (T)	Large enough for a smooth transition from data to noise. More steps can improve quality but are slower to train and sample	1000 steps for training 1000 steps for inference	1000 steps for training 20 to 50 steps for inference
Batch Size	As large as can fit on GPU memory for stable gradients	1	256 - 2048+ (Distributed over many GPUs)
Channels	Based on the training image format.	1 (Greyscale pixels only)	3 (For RGB), 4 (For RGB and alpha for transparency)
Attention Mechanism	Essential for conditioning on text and capturing long-range spatial relationships	Not used (text embedding is added directly)	Cross-attention layers at multiple U-Net depths
Text Embedding Dimensions	Determined by the pre-trained text encoder	16 dimensions	768 or 1024 dimensions
Text Embedding Projection	Projects the generic text embedding into a space tailored for the U-Net	Multinode-Linear-Projection (MPL) 16 dimensions to 64 dimensions	Projects text embedding e.g., 768 dimensions to match U-Net's internal width, e.g., 4096 dimensions
Timestep Embedding Dimensions	A hyperparameter balancing expressiveness and efficiency	16 dimensions	Typically larger, e.g., 320 dimensions

Timestep Embedding Projection	Projects the time signal into a space that can be added to image features	MLP 16 dimensions to 64 dimensions	Projects time embedding to match the U-Net
U-Net Base Width	A core capacity parameter. More channels allow for more complex features but increase model size	32 nodes down-sampled to 64 nodes	320 to over 1280 nodes
U-Net Depth	Deeper models capture more hierarchical features. Usually 3-5 down/up-sampling blocks.	1 downsample / upsample block	4-5 downsample / upsample blocks
Bridge Size	The channel count at the deepest point of the U-Net	64 channels	1280 channels

Figure 12.28 is a visualization of the entire diffusion architecture that we will be using in our example. You will see:

- The original “a happy face” training image having noise added to it with forward diffusion to create our noisy training data,
- A noisy image being fed into a convolutional layer to find initial specific features,
- The results of that convolutional layer being fed into a down-sample convolution layer: making the image dimensions go from  $8 \times 8$  to  $4 \times 4$  - compressing the information to find more general features,
- The results of the down-sample layer being fed into the bridge to find the most general features,
- The text embedding and timestep embedding being fed through a feed-forward network,
- The text embedding and timestep embedding being combined with the compressed image data in the bridge - essentially mixing the generalized understanding with the text label,
- The results of those feature maps being up-scaled from  $4 \times 4$ , back to  $8 \times 8$  - to smartly interpolate what the image could be from the generalized information,
- The skip connection combining the results of the initial convolutional layer with the results of the up-scale layer - resulting in both generalized information as well as information on finer details,

- And finally, the convolutional layers that compress those inputs in a single  $8 \times 8$  matrix output to represent the predicted image at a timestep.

Don't worry if this looks daunting. We will dive into each step in the next section.

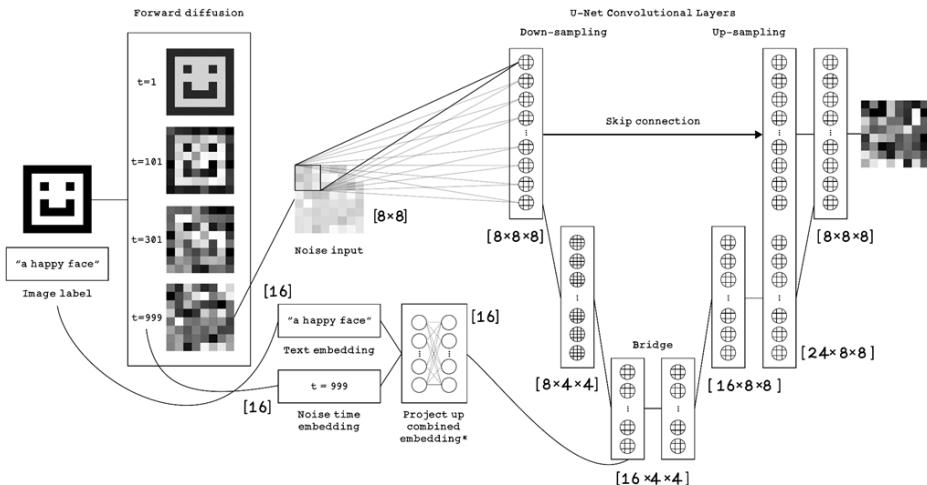


Figure 12.28 The U-Net architecture for our example

## 12.6 Denoising: From numbers to an image

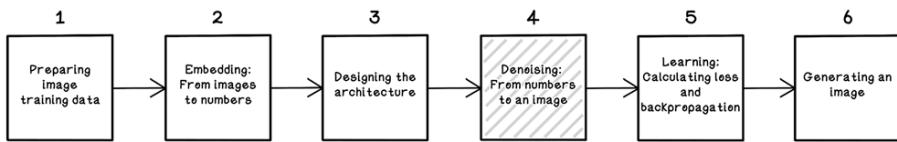


Figure 12.29 Denoising in the Diffusion model training workflow

Now that we have prepared the inputs and designed the deep learning architecture, we can explore the denoising process of the U-Net (figure 12.29). The U-Net's goal is to look at the  $8 \times 8$  input matrix and predict the exact noise pattern that was added to the original clean image. It does this by summarizing the image into more abstracted representations to understand its "essence" and then skillfully reconstructing it by removing the noise.

### 12.6.1 Encoder: Down-sampling layers

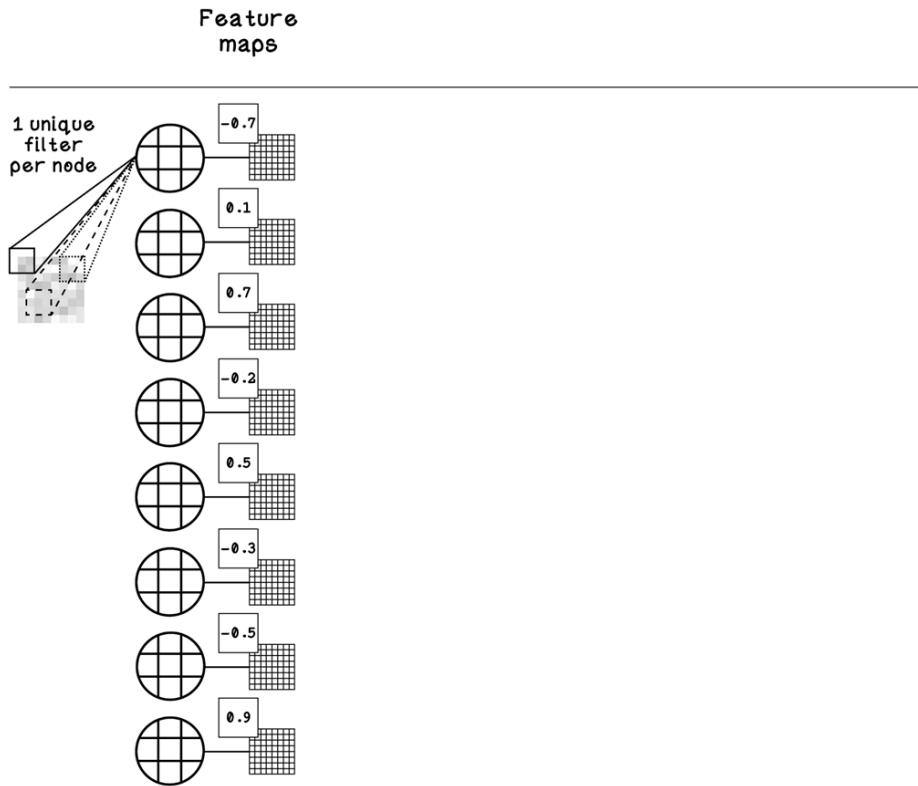
The first half of the U-Net is the encoder, or down-sampling path. Its job is to act like a summarizer. It takes the high-resolution noisy image and progressively shrinks it, forcing the network to move beyond individual pixels and capture the abstract, high-level context of the image. It answers the question, "What am I looking at?".

#### THE FIRST CONVOLUTIONAL LAYER

The  $8 \times 8$  noisy image enters a convolutional layer with 8 unique  $3 \times 3$  filters. Each filter slides across the image, performing a dot product at every position. This operation, along with an added bias for each filter, produces an  $8 \times 8 \times 8$  tensor of feature maps, because we have 8 nodes in this convolutional layer. This tensor then passes through two more steps:

- *Calculating the feature maps:* This is the primary feature extraction step. Each of the 8 unique  $3 \times 3$  filters slides across the entire  $8 \times 8$  noisy input image. As we explored in the previous section, at every one of the 64 positions, each filter performs a dot product calculation with the image patch underneath it. After the dot product is calculated, a single learnable number is added to the result. Because there are 8 independent filters, this process creates 8 unique feature maps, resulting in the  $8 \times 8 \times 8$  tensor.

Figure 12.30 shows only the top-left value for each feature map.

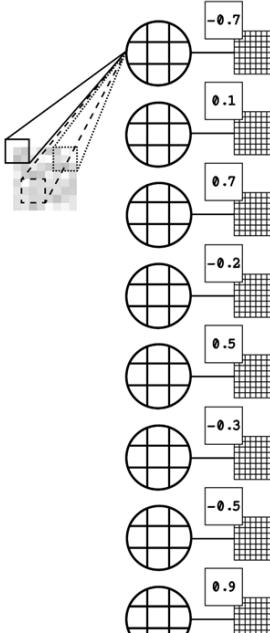


**Figure 12.30** The first pixel of all feature maps produced by the first convolutional layer

- *Group normalization:* Before the feature maps are passed to the activation function, they go through a normalization step. As data flows through a deep network, the distribution of values in each layer can change wildly, a problem known as *internal covariate shift*. This makes it difficult for the network to learn effectively; it's like trying to hit a constantly moving target.  
Group Normalization solves this by re-centering and re-scaling the data. It takes the 8 output feature maps and divides them into smaller, predefined groups. For each group, it calculates a single mean and standard deviation and uses them to normalize the values only within that group.

You can think of it like this: if the 8 feature maps are sketches from a team of artists, Group Normalization is like having 4 art directors, each responsible for a small team of 2 artists. Each director adjusts the overall brightness and contrast of their team's set of sketches to ensure they are consistent before being passed on. This stabilization makes the training process much more robust and efficient.

For simplicity, we are using 1 group (art director) for all 8 feature maps in our layer. In the example shown in Figure 12.31, we are considering just the first pixel in each feature map for simplicity's sake. We first calculate the group mean by adding all 8 values then dividing by 8. Then we find the standard deviation for each by subtracting the original value from the feature map from the mean, and squaring it. And finally we find the normalized values using the original value, mean, and standard deviation.

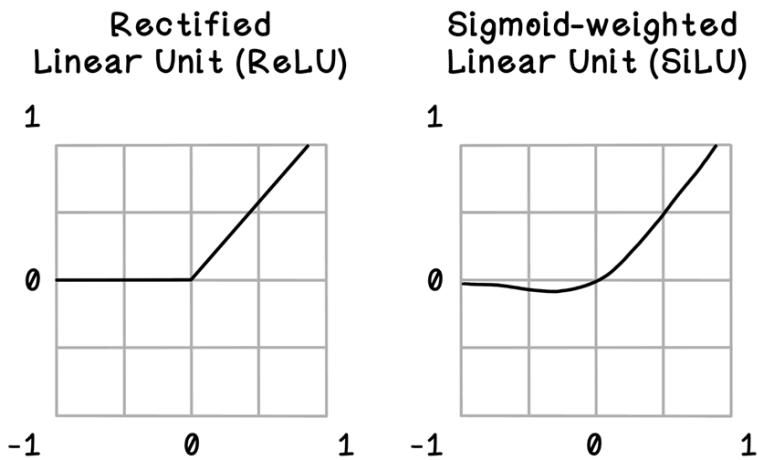
Feature maps	Group mean	Standard deviation $(\text{original} - \text{mean})^2$	Normalized value $(\text{original} - \text{mean}) / \text{std\_dev}$
	$(-0.7 + 0.1 + 0.7 + -0.2 + 0.5 + -0.3 + -0.5 + 0.9) / 8 = 0.063$	$((-0.7 - 0.063)^2 + (0.1 - 0.063)^2 + (0.7 - 0.063)^2 + (-0.2 - 0.063)^2 + (0.5 - 0.063)^2 + (-0.3 - 0.063)^2 + (-0.5 - 0.063)^2 + (0.9 - 0.063)^2) / 8 = \sqrt{0.2995} = 0.55$	$\begin{array}{l} (-0.7 - 0.063) / 0.55 = -1.39 \\ (-0.1 - 0.063) / 0.55 = -0.30 \\ (0.7 - 0.063) / 0.55 = 1.16 \\ (-0.2 - 0.063) / 0.55 = -0.48 \\ (0.5 - 0.063) / 0.55 = 0.79 \\ (-0.3 - 0.063) / 0.55 = -0.66 \\ (-0.5 - 0.063) / 0.55 = -1.02 \\ (0.9 - 0.063) / 0.55 = 1.52 \end{array}$

**Figure 12.31** Normalized values for the first pixel in all feature maps produced from the first convolutional layer

- *Sigmoid-weighted Linear Unit (SiLU) activation function:* A non-linear activation function is applied to every number in the  $8 \times 8 \times 8$  tensor. This is a crucial step that allows the network to learn complex patterns beyond simple linear relationships.

The SiLU function (also known as *Swish*) is calculated by multiplying the input value ( $x$ ) by the output of a sigmoid function applied to that same input ( $x * \text{sigmoid}(x)$ ). For positive input values, it behaves very similarly to the ReLU function, resulting in a nearly linear output. The key difference is its handling of negative values. Instead of outputting a hard zero like ReLU, SiLU produces a smooth curve that dips slightly below zero before approaching zero (figure 12.32). This smoothness can help prevent the “dying ReLU” problem and often leads to better performance by allowing for a more complex response and better gradient flow during training.

See the section *Options for activation functions* in Chapter 9 for more about activation functions.



**Figure 12.32 ReLU vs SiLU activation functions**

The resulting  $8 \times 8 \times 8$  tensor, which we'll call  $h1$ , is now rich with information about low-level features and is saved in memory for the skip connection step that we will work through later (figure 12.33).

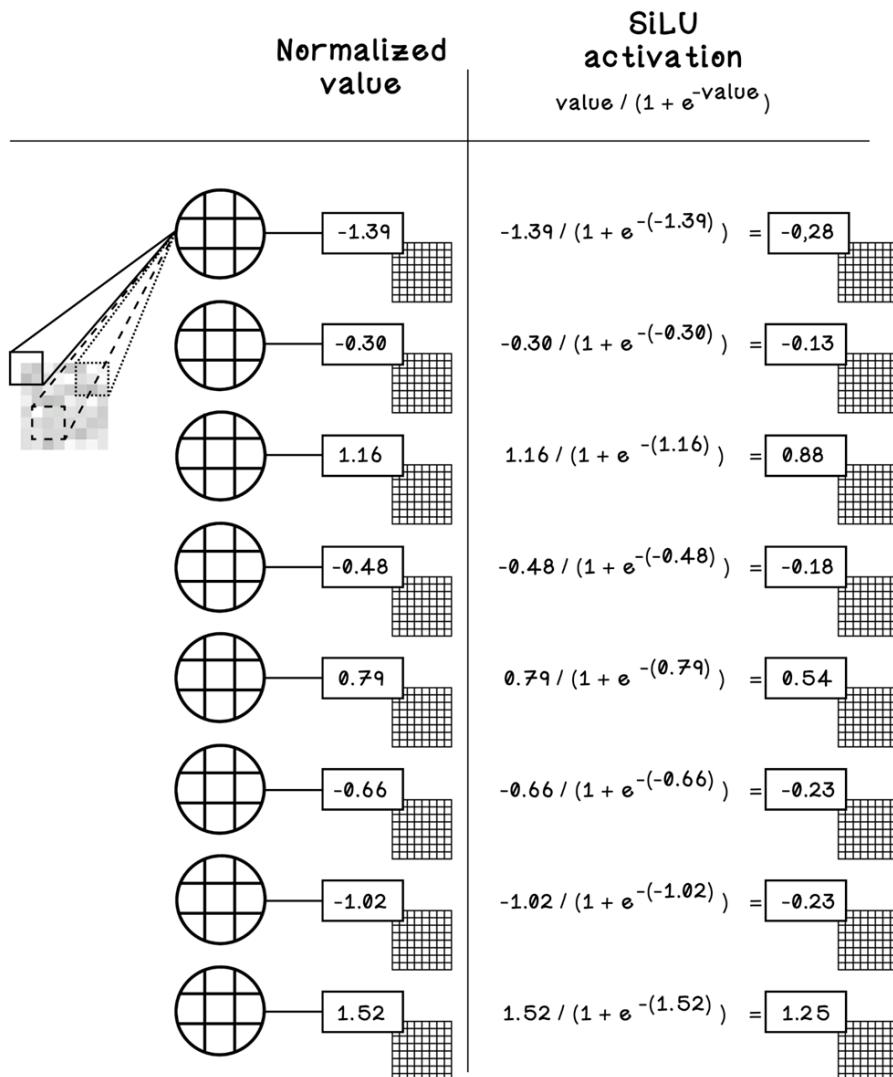


Figure 12.33 Calculations for the SiLU activation function

## PYTHON CODE SAMPLE

This code defines the first convolutional layer in the U-Net encoder, which extracts low-level features from a noisy input image. It begins by applying padding to preserve spatial dimensions, then applies multiple learnable  $3 \times 3$  filters across the image using a sliding window. At each position, it computes the dot product between the filter and the local region of the image, adds a bias term, and stores the result in the corresponding output map. The output of all filters is stacked into a 3D tensor ( $h1$ ), representing the first set of learned feature maps for further processing in the network.

```
def apply_padding(image, padding):
    return np.pad(image, ((padding, padding), (padding, padding)),
mode='constant', constant_values=0)

def first_convolution(noisy_image, num_filters):
    filter_size = 3
    stride = 1
    padding = 1

    padded_image = apply_padding(noisy_image, padding)

    H, W = noisy_image.shape

    filters = np.random.randn(num_filters, filter_size, filter_size) * 0.01
    bias = np.random.randn(num_filters) * 0.01

    feature_maps = []

    for filter_index in range(num_filters):
        current_filter = filters[filter_index]
        current_bias = bias[filter_index]
        output_map = np.zeros((H, W))

        for i in range(H):
            for j in range(W):
                region = padded_image[i : i + filter_size, j : j + filter_size]

                value = np.sum(region * current_filter) + current_bias

                output_map[i][j] = value

        feature_maps.append(output_map)

    h1 = np.stack(feature_maps, axis=0)

    return h1
```

## DOWN-SAMPLING

The entire purpose of down-sampling is to increase the network's high-level understanding of the areas in the original image that each respective node considers. Think of it like squinting your eyes when looking at a detailed painting. You lose the ability to see the fine brushstrokes, but you get a much better sense of the overall composition, shapes, and colors. This process forces the network to learn high-level, abstract concepts (like "eye" or "mouth") instead of just simple features (like "vertical edge" or "corner"). It's a conglomeration of more specific features.

The tensor from the first layer, which we called  $h_1$ , is now passed to a second convolutional layer. This layer's primary job is to down-sample, or shrink, the feature maps. This is essentially lowering the resolution of the image.

In Figure 12.34, notice how each node in the down-sample convolutional layer accepts every feature map from the first convolutional layer. This means each node will have 8 unique filters corresponding to each input. These filters are also  $4 \times 4$  in size (not  $3 \times 3$  like the first convolutional layer). And furthermore, the output of these nodes result in a  $4 \times 4$  feature map instead of an  $8 \times 8$  feature map.

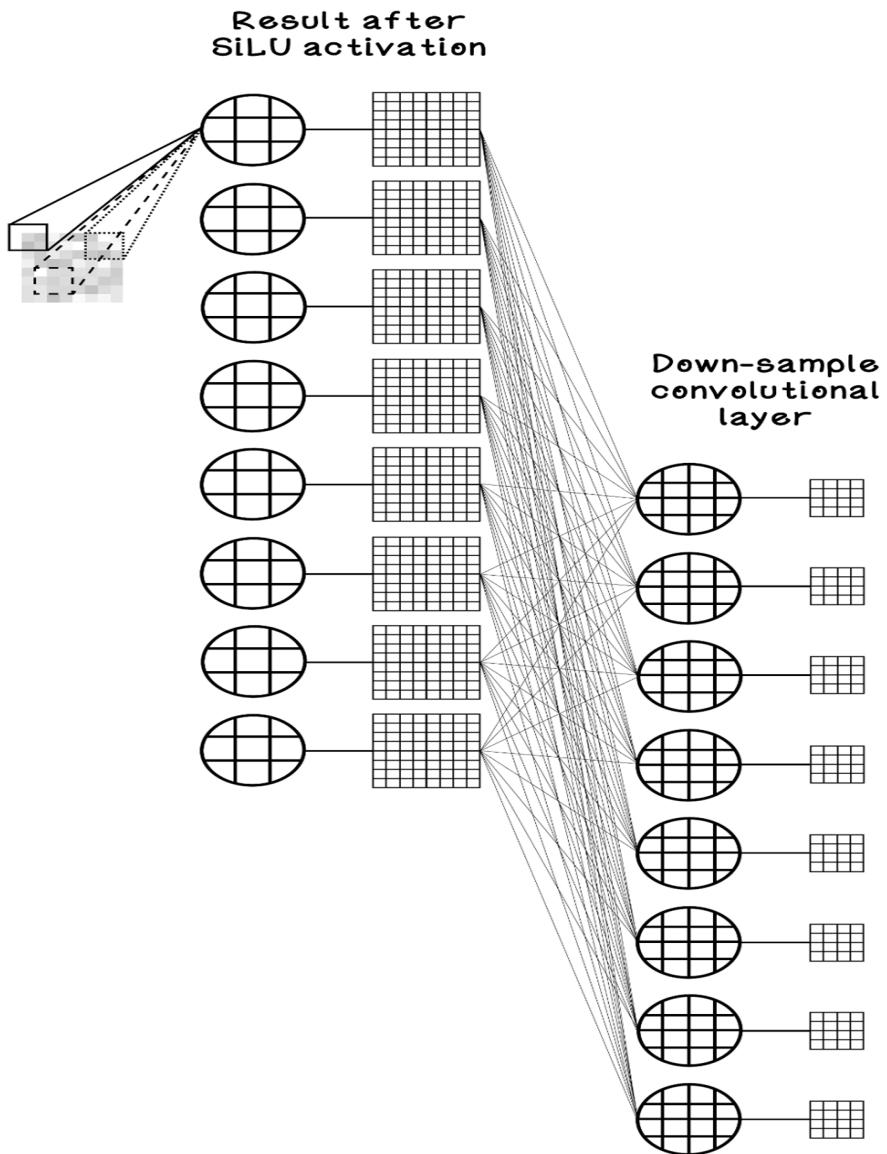
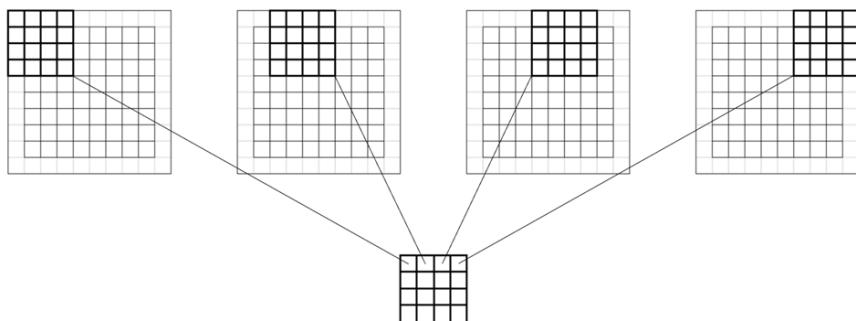


Figure 12.34 The down-sampling convolutional layer in our U-Net

The model achieves producing a  $4 \times 4$  feature map by using filters with a stride of 2, meaning the filters jump 2 pixels at a time as they slide across the input, instead of 1 pixel at a time. This is a good way to halve the height and width of the feature maps, producing a more compressed summary of the image's content. In our example, the  $8 \times 8$  feature maps are shrunk to  $4 \times 4$ . Figure 12.35 illustrates how the different positions of the  $4 \times 4$  filter with a stride of 2 corresponds to the new pixel values.

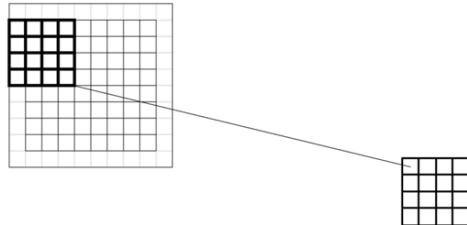
Note the additional “border” of blank values that have been added to the input feature map. This is the padding that was mentioned in the CNN section. It helps preserve the image size and edge data during down-sampling and up-sampling.



**Figure 12.35 How filters scan an image with padding**

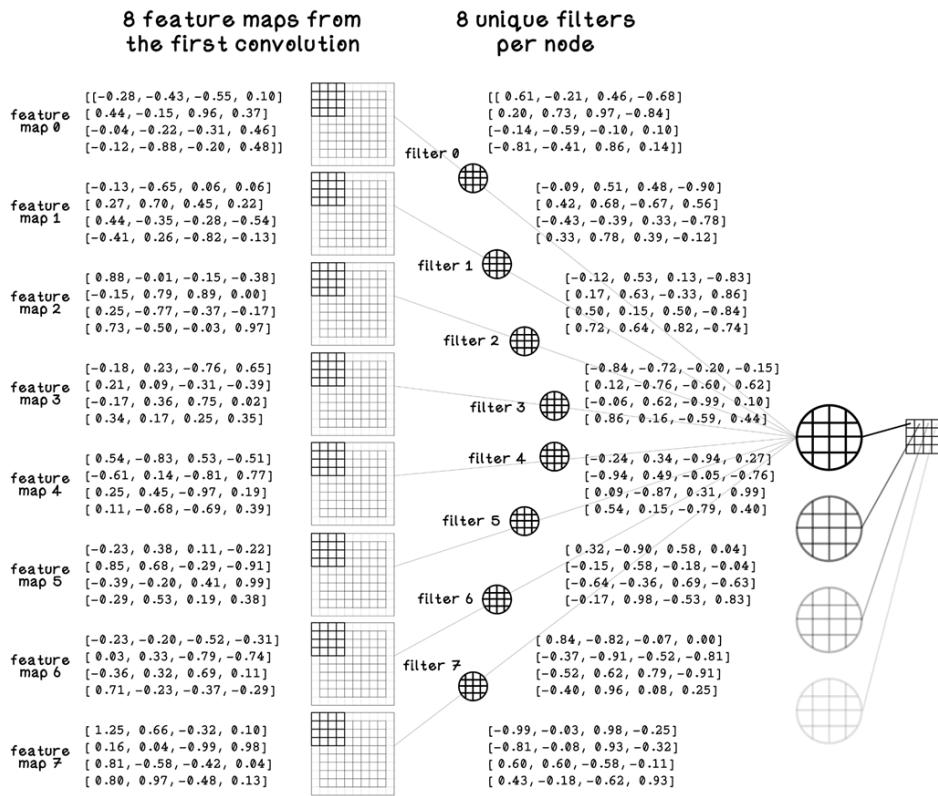
**EXERCISE: WHAT IS THE NEXT POSITION OF THE FILTER?**

Determine the filter position as it “scans”.

**SOLUTION: WHAT IS THE NEXT POSITION OF THE FILTER?**

Since the input  $h_1$  has multiple channels (in this case, 8 feature maps from the first layer), the convolution process at this stage is more complex than the first convolution layer. For our example, each filter in the down-sampling layer is 8 layers deep, designed to process all 8 input channels simultaneously. To calculate a single pixel value in the new output feature map, the network performs 8 separate convolutions (one for each input feature map) and then sums all the results together. This combines the simple features detected in the first layer into more complex and abstract representations.

In Figure 12.36, notice how all feature maps are being processed by the first node of the down-sampling layer. It's important to note that each feature map will be associated with a unique filter of its own unique weights (these weights are adjusted separately during backpropagation), and that all filter "slides" will happen for each node.



**Figure 12.36 Calculations for the down-sampling convolutional layer in our U-Net**

Now that we have the foundational idea of how initial feature maps are processed by the down-sampling layer, let's look at how we find the single value for a single pixel. Each feature map ( $8 \times 8$ ) is multiplied by its respective filter weights ( $4 \times 4$ ). This will result in a single scalar number, for example, **0.75**. This is done for every feature map and filter pair, then all results are summed to obtain our value for the pixel at that position. In Figure 12.37, the top left pixel in the  $4 \times 4$  feature map is **-0.646**.

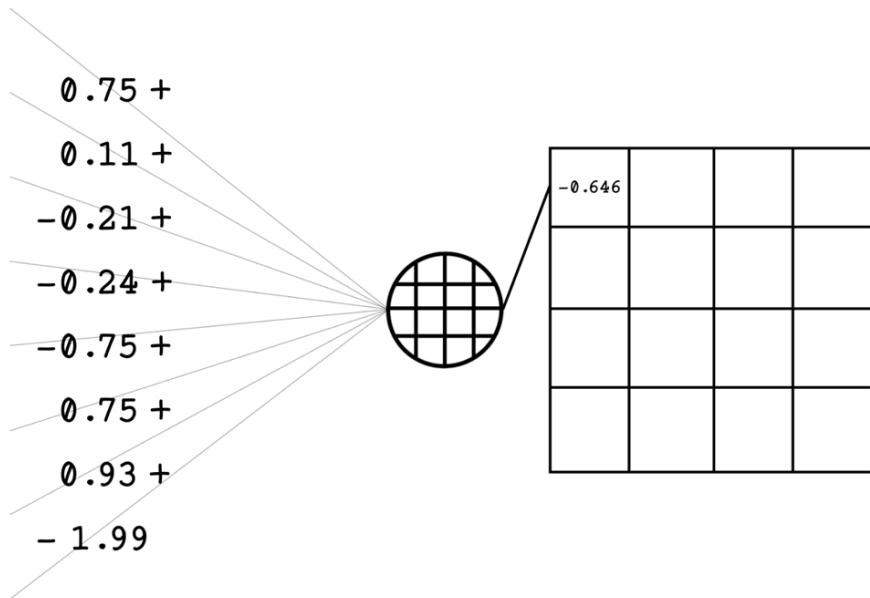
**convolution = sum(feature map \* filter)**

```

convolution 0 = 0.75
convolution 1 = 0.11
convolution 2 = -0.21
convolution 3 = -0.24
convolution 4 = -0.75
convolution 5 = 0.75
convolution 6 = 0.93
convolution 7 = -1.99

```

**final pixel = sum(all convolutions)**



**Figure 12.37 Calculations for producing the final value for the first pixel in the feature map**

As expected, this is done for all nodes in the layer, and in our case, because we have 8 nodes in the down-sampling layer, we will have 8 feature maps that are all 4 x 4 in dimensions.

## PYTHON CODE SAMPLE

This code performs the downsampling convolutional step in the U-Net encoder. It takes a multi-channel feature map and applies a set of learnable  $4 \times 4$  filters with stride 2 and padding 1 to reduce the spatial resolution by half. Each output channel is computed by combining contributions from all input channels through weighted dot products over sliding patches, summing them, and adding a bias. The resulting output maps are stacked into a 3D tensor ( $h2$ ), which captures increasingly abstract features while compressing the spatial size — preparing the data for deeper layers in the network.

```

def apply_padding_3d(tensor, padding):
    return np.pad(tensor, ((0, 0), (padding, padding), (padding, padding)),
mode='constant', constant_values=0)

def downsample_convolution(feature_maps_in, num_output_channels):
    input_channels = feature_maps_in.shape[0]
    input_height = feature_maps_in.shape[1]

    filter_size = 4
    stride = 2
    padding = 1

    padded_maps = apply_padding_3d(feature_maps_in, padding)

    output_height = (input_height + 2 * padding - filter_size) // stride + 1
    output_width = output_height

    filters = np.random.randn(num_output_channels, input_channels, filter_size,
filter_size) * 0.01
    biases = np.random.randn(num_output_channels) * 0.01

    feature_maps_out = []

    for k in range(num_output_channels):
        current_bias = biases[k]
        output_map = np.zeros((output_height, output_width))

        for i in range(output_height):
            for j in range(output_width):

                accumulation_sum = 0

                for c in range(input_channels):

                    current_filter = filters[k, c]

                    r_start, c_start = i * stride, j * stride
                    region = padded_maps[c, r_start : r_start + filter_size,
c_start : c_start + filter_size]

                    accumulation_sum += np.sum(region * current_filter)

                output_map[i, j] = accumulation_sum + current_bias

```

```
feature_maps_out.append(output_map)

h2 = np.stack(feature_maps_out, axis=0)

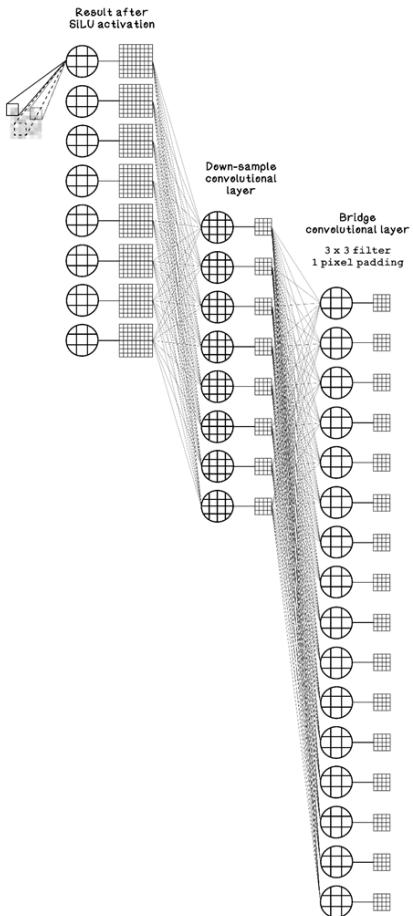
return h2
```

### 12.6.2 Bridge (also known as the bottleneck)

The bridge is the lowest point of the “U”, connecting the encoder and the decoder. It receives the most compressed and abstract summary of the image. This is the critical point where the image information meets the external guidance from our text prompt and timestep. At this point in the network, the image information is as abstract and generalized as possible. In Figure 12.38, we can see that the initial convolutional layer produced 8 feature maps of the original size, then the down-sampling layer produced 8 feature maps that are  $4 \times 4$  in dimensions - these are the inputs for the bridge convolutional layer.

You will notice that the convolution layer in the bridge has 16 nodes instead of 8. By increasing the number of channels at this stage, the network has a greater capacity to create a rich and expressive summary of the image's core content. Think of it as giving the model a larger vocabulary to describe the “what” of the image before it begins to reconstruct the “where” in the up-sampling path.

Increasing the number of nodes in the bridge layer enhances its ability to capture complex, abstract information and prepares it to integrate guidance from the text and time embeddings.



**Figure 12.38** The bridge convolutional layer in our U-Net

## BRIDGE CONVOLUTION

The convolution layer at this level works similarly to the previous layer. The  $8 \times 4 \times 4$  tensor from the previous layer enters the bridge and passes through another convolutional layer. This layer increases the node depth from 8 to 16, resulting in a  $16 \times 4 \times 4$  tensor. This tensor, called  $h2$ , represents the most abstract understanding of the image content and is ready for the text label embedding to be injected.

## EMBEDDING INJECTION

Now the external text label guidance is injected. The text and time embeddings are added together to form a single 16 dimensional vector. This vector is reshaped into a  $16 \times 1 \times 1$  tensor and added to the  $16 \times 4 \times 4$  h2 tensor. The first number from the embedding vector is added to every pixel in the first channel of h2, the second number is added to every pixel in the second channel, and so on. This step “infuses” the prompt and timestep information into the network’s generalized understanding of the image (figure 12.39).

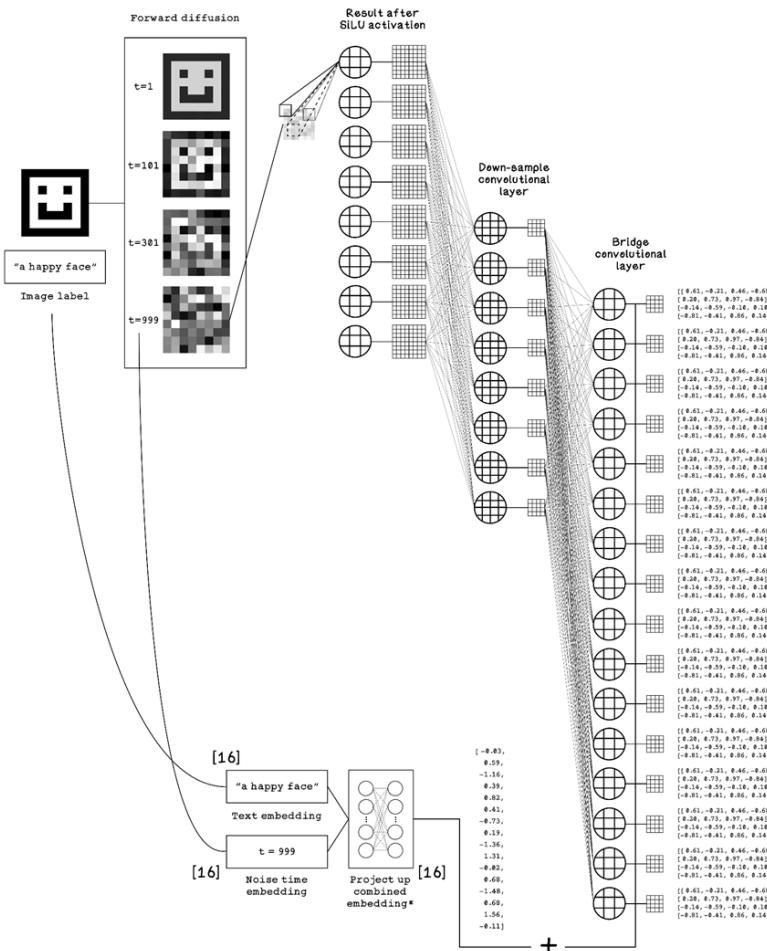


Figure 12.39 How the text embedding and timestep embedding are injected

## PYTHON CODE SAMPLE

This code injects text and timestep embeddings into the bridge tensor at the bottleneck of the U-Net. It first combines the two embeddings elementwise, then reshapes the result to match the spatial dimensions of each feature map channel. The reshaped embedding is broadcast and added to every pixel in its corresponding channel of the bridge tensor. This operation allows the model to condition its internal representation on the prompt and the current diffusion timestep, effectively guiding the denoising process in the decoding path.

```
def inject_embeddings(bridge_tensor, text_embedding, timestep_embedding):

    combined_embedding = text_embedding + timestep_embedding

    channels = combined_embedding.shape[0]
    reshaped_embedding = combined_embedding.reshape(channels, 1, 1)

    output_tensor = bridge_tensor + reshaped_embedding

    return output_tensor
```

### 12.6.3 Decoder: Up-sampling layers

At this point, we now have 16 feature maps that are  $4 \times 4$  in dimensions and include the generalized image data, the text label embedding data, and the timestep embedding data.

The first step of the decoder (up-sampling process) is to take the compressed  $4 \times 4$  feature maps from the bridge and enlarge them.

## UP-SAMPLING

The simplest way to visualize this is through an operation like nearest-neighbor interpolation, as shown in Figure 12.40. With this method, each pixel from the smaller feature map is simply duplicated to fill a  $2 \times 2$  area in the new, larger feature map. While this is a fast and straightforward way to double the resolution to  $8 \times 8$ , it's a basic approach.

Data after embedding combination				Nearest neighbor Interpolation							
<b>0.58</b> 1.02 -1.50 -2.29				<b>0.58</b> 0.58 1.02 1.02 -1.50 -1.50 -2.29 -2.29							
0.38	1.14	0.72	0.27	0.38	0.38	1.14	1.14	0.72	0.72	0.27	0.27
-0.70	0.24	-0.12	2.42	0.38	0.38	1.14	1.14	0.72	0.72	0.27	0.27
-0.29	-0.65	0.78	0.03	-0.70	-0.70	0.24	0.24	-0.12	-0.12	2.42	2.42
				-0.70	-0.70	0.24	0.24	-0.12	-0.12	2.42	2.42
				-0.29	-0.29	-0.65	-0.65	0.78	0.78	0.03	0.03
				-0.29	-0.29	-0.65	-0.65	0.78	0.78	0.03	0.03

**Figure 12.40 Simplistic up-sampling with nearest neighbor interpolation**

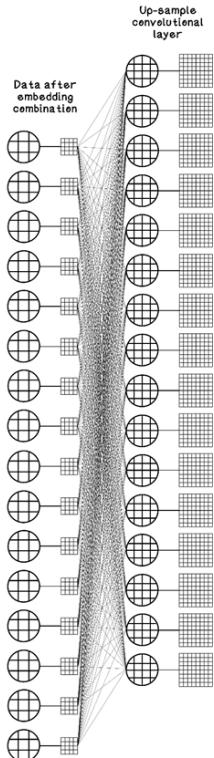
For a more sophisticated reconstruction, the U-Net uses a *transposed convolution* (figure 12.41). Unlike simple interpolation, a transposed convolution is a learnable layer similar to the other convolutional layers you've seen. It doesn't just copy pixels, it uses a set of trainable filters to intelligently project the data from the low-resolution feature map onto a larger feature map.

It is like hiring a skilled artist. You give the artist the low-resolution feature map, and they use their learned skills to paint a large detailed version. They don't just copy the lines, they intelligently add the right brushstrokes and textures to create a high-quality result.

Here's how it works in the network:

- It takes one pixel at a time from the small feature map,
- Instead of reading a patch to produce one pixel (like a normal convolution), it uses a learnable filter to "paint" a larger patch of pixels onto the bigger output map,
- It repeats this for every single pixel in the small map. Where the "painted" patches overlap, their values are added together.

The filter's weights are learned during the training process. The network figures out the absolute best way to add detail when up-sampling, allowing it to reconstruct a much cleaner and more coherent image.



**Figure 12.41** The up-sampling layer in our U-Net

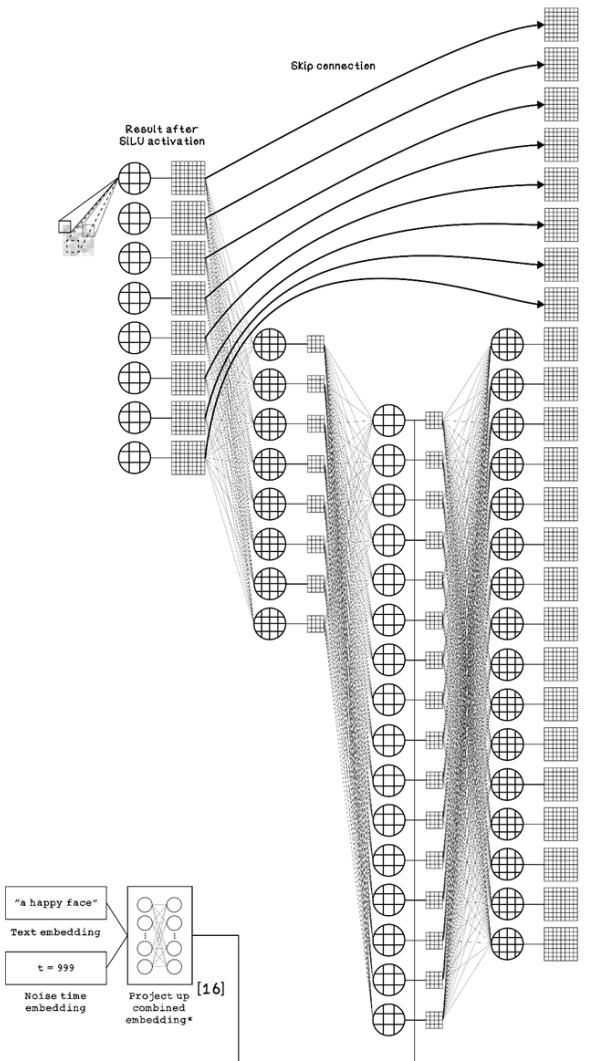
After this up-sampling convolution step, we have 16 feature maps that are  $8 \times 8$  in dimensions. Remember these maps now have the generalized image data, text label embedding, timestep embedding, and have been upscaled to the actual dimensions of the images that we are working with.

## SKIP CONNECTION

Up until now, the network can get very good at understanding the high-level context of the “what” an image contains, but it loses the precise spatial information about the “where” of fine details. The *skip connection* is one of the U-Net’s superpowers that solves this problem. It acts as a shortcut for feeding the decoder a direct copy of the high-resolution details that it needs to reconstruct the image accurately.

In Figure 12.42, notice that the skip connection takes the rich detailed feature maps from the early stage of the encoder and sends them directly across to the corresponding layer in the decoder. This decoder layer now has  $8 \times 8 \times 8$  tensor  $h_1$ , and stacks it with the  $16 \times 8 \times 8$  up-sampled tensor from the previous step. This concatenation process creates one  $24 \times 8 \times 8$  tensor.

This combined tensor is the key to a high-quality reconstruction. It provides the decoder with both the abstract information from the deep layers and bridge, and the precise information from the early layers, allowing the model to generate a final image that is not only conceptually correct but also has well-defined details.



**Figure 12.42 How skip connection feature maps are combined with the up-sampled feature maps**

## PYTHON CODE SAMPLE

This code performs transposed convolution for upsampling in the decoder path of the U-Net. It increases the spatial resolution of the input tensor by using learned  $4 \times 4$  filters to project each input pixel into a larger output space. At each location, the pixel value is multiplied by a filter and the result is added to a larger output region, effectively “painting” detail back into the image. Overlapping contributions are summed, and a bias is added to each output channel. This operation enables the model to reconstruct higher-resolution images from compressed feature maps.

```
def upsample_transposed_convolution(input_tensor, num_output_channels):
    input_channels = input_tensor.shape[0]
    input_height = input_tensor.shape[1]
    input_width = input_tensor.shape[2]

    filter_size = 4
    stride = 2
    padding = 1

    output_height = (input_height - 1) * stride + filter_size - 2 * padding
    output_width = (input_width - 1) * stride + filter_size - 2 * padding

    filters = np.random.randn(num_output_channels, input_channels, filter_size,
filter_size) * 0.01
    biases = np.random.randn(num_output_channels) * 0.01

    output_tensor = np.zeros((num_output_channels, output_height, output_width))

    for oc in range(num_output_channels):
        for ic in range(input_channels):
            current_filter = filters[oc, ic]

            for i in range(input_height):
                for j in range(input_width):
                    value = input_tensor[ic, i, j]

                    for m in range(filter_size):
                        for n in range(filter_size):

                            out_y = i * stride + m - padding
                            out_x = j * stride + n - padding

                            if 0 <= out_y < output_height and 0 <= out_x <
output_width:
                                output_tensor[oc, out_y, out_x] += value *
current_filter[m, n]

            output_tensor[oc] += biases[oc]

    return output_tensor
```

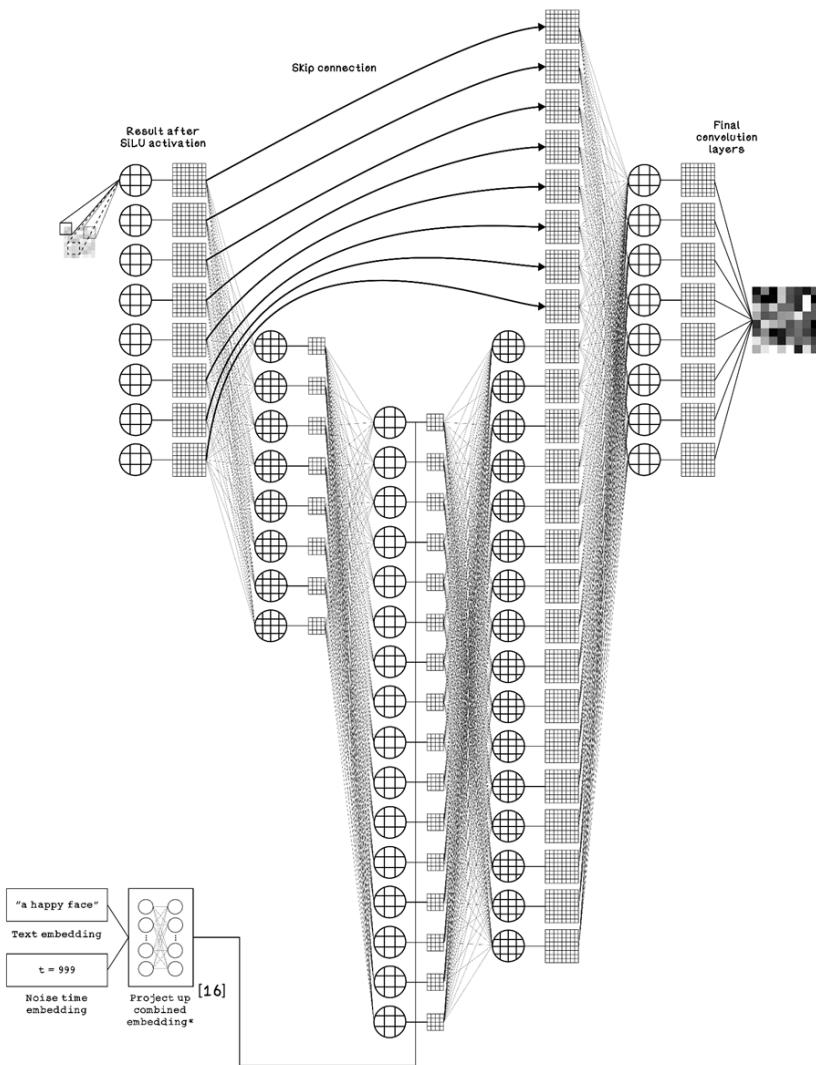
## FINAL CONVOLUTIONS

Before we look at the final layers, it's important to remember the U-Net's single, specific job in this process. It receives three key inputs: 1) a noisy image, 2) the text prompt, and 3) a timestep (which indicates how much noise the image contains). Its goal is to analyze this information and output a prediction of the exact noise pattern that was added to the image for that specific timestep.

The rich  $24 \times 8 \times 8$  tensor, which now contains both the high-level concepts from the decoder and the fine-grained details from the skip connection, is passed through a final set of convolutional layers (figure 12.43). This is the last stage where the model must distill all of this complex information into a single useful output.

- *Blending and distilling:* The first of these final layers takes the 24-node tensor and performs a blending operation. It must learn to synthesize the two distinct types of information it has received (the generalized and the detailed). This convolutional layer reduces the channels from 24 down to 8 while keeping the resulting feature maps  $8 \times 8$ , effectively distilling the most important combined features into a manageable tensor.
- *The final prediction:* This is the last convolutional layer and has a single vital job: take the 32 blended feature maps and collapse them into one final  $8 \times 8$  image. The final output is the model's pixel-by-pixel prediction of the noise that was added at the very beginning of the process.

It's important to remember that this entire U-Net architecture has been trained for this one specific regression task, to predict the exact noise pattern for a given image, at a given timestep, guided by a text prompt. This final matrix isn't the desired clean image, it is the key that unlocks it as the model learns over many epochs.



**Figure 12.43** The final convolutions to produce a noise prediction

We have now completed the entire *forward-pass* process for the diffusion model. After one epoch, the model would have processed a noise step, and produced a prediction for the initial noise at timestep 0 (likely a very wrong prediction if this is the first epoch).

## PYTHON CODE SAMPLE

This code defines the skip connection convolution in the decoder stage of a U-Net. It merges the upsampled feature maps with the corresponding encoder feature maps (the “skip connection”) by concatenating them along the channel axis. Then, a  $3 \times 3$  convolution is applied across the combined tensor using multiple learnable filters. Each filter processes all input channels to produce refined output feature maps that blend high-level abstract information with fine spatial details. This operation enables precise reconstruction of the image by preserving both the “what” and the “where” from earlier layers.

```

def apply_padding_3d(tensor, padding):
    return np.pad(tensor, ((0, 0), (padding, padding), (padding, padding)),
mode='constant', constant_values=0)

def skip_convolution(upsampled_tensor, skip_tensor, num_output_channels):

    input_tensor = np.concatenate([upsampled_tensor, skip_tensor], axis=0)

    input_channels = input_tensor.shape[0]
    output_height = input_tensor.shape[1]
    output_width = input_tensor.shape[2]

    filter_size = 3
    stride = 1
    padding = 1

    padded_input = apply_padding_3d(input_tensor, padding)

    filters = np.random.randn(num_output_channels, input_channels, filter_size,
filter_size) * 0.01
    biases = np.random.randn(num_output_channels) * 0.01

    output_tensor = np.zeros((num_output_channels, output_height, output_width))

    for oc in range(num_output_channels):
        current_bias = biases[oc]

        for i in range(output_height):
            for j in range(output_width):

                accumulation_sum = 0

                for ic in range(input_channels):
                    current_filter = filters[oc, ic]

                    region = padded_input[ic, i : i + filter_size, j : j +
filter_size]

                    accumulation_sum += np.sum(region * current_filter)

                output_tensor[oc, i, j] = accumulation_sum + current_bias

    return output_tensor

```

This code performs the final output convolution in the U-Net, reducing a multi-channel tensor to a single-channel image. It applies one  $3 \times 3$  convolutional filter per input channel, computes the dot product over each patch, sums the results across channels, and adds a scalar bias. The resulting 2D matrix represents the model's prediction of the noise present in the original image. This output is used during training to calculate the loss, or during inference to denoise the image step-by-step toward a clean result.

```
final_output_convolution(input_tensor):

    let input_channels equal number of channels in input_tensor
    let filter_size equal 3
    let stride equal 1
    let padding equal 1

    let filter equal single  $3 \times 3$  filter with input_channels depth

    let bias equal single scalar value

    let height equal height of input_tensor
    let width equal width of input_tensor

    let output_image equal zero matrix of shape [height, width]

    for i from 0 to height - 1:
        for j from 0 to width - 1:
            let sum equal 0

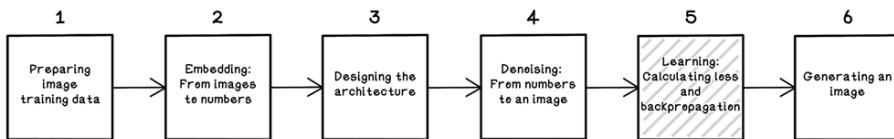
            for c from 0 to input_channels - 1:
                let region equal  $3 \times 3$  patch from input_tensor[c] centered at (i, j)
                let value equal dot product of region and filter[c]
                sum equal sum + value

            set output_image[i][j] equal sum + bias

    return output_image
```

## 12.7 Learning: Calculating loss and backpropagation

After the U-Net makes its prediction, the learning process begins. This is where the model determines how wrong the prediction was and uses that information to improve itself (figure 12.44). This is a critical two-step cycle: calculating a single loss score, and then using that score to update every weight parameter in the network through backpropagation.



**Figure 12.44 Learning in the Diffusion model training lifecycle**

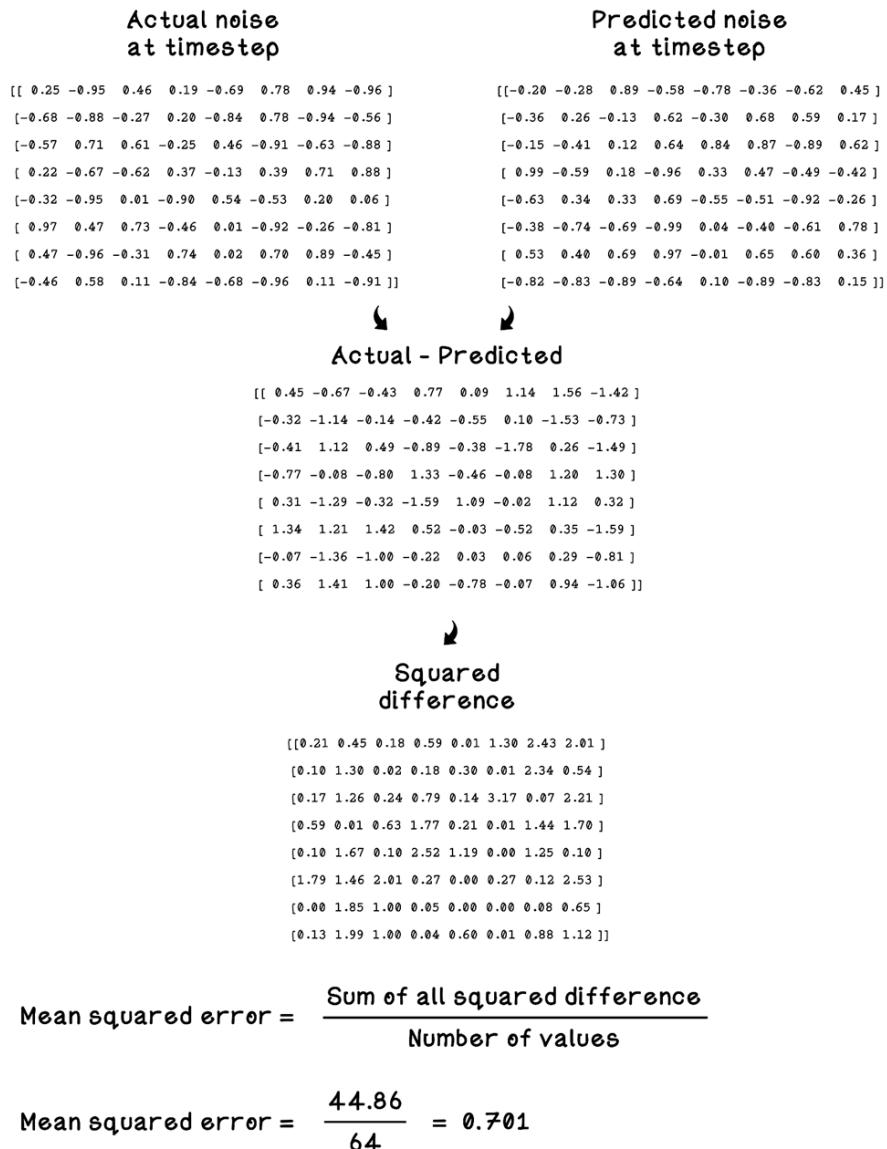
### 12.7.1 Calculating loss

The first step is to get a single number that measures exactly how wrong the model's prediction was. In our diffusion model, the U-Net predicts a noise pattern. Because we added the noise ourselves, we know precisely what the actual noise looks like at a specific timestep. We can therefore calculate the loss by directly comparing the predicted noise to the actual noise.

#### LOSS WITH MEAN SQUARED ERROR (MSE)

Since the U-Net performs a regression task (its goal is to predict the exact continuous values of the noise that was added to an image), Mean Squared Error (MSE) is an ideal method for calculating loss since it measures the average difference between the *predicted noise* and the *actual noise*, pixel by pixel.

MSE is a loss function where the goal is to predict a continuous numerical value. This is perfect for our diffusion model, where the U-Net must predict the exact floating-point values for each pixel in the noise pattern. MSE measures the average of the squares of the errors, that is, the average squared difference between the *predicted values* and the *actual values* (figure 12.45).



**Figure 12.45 Calculations for finding the loss**

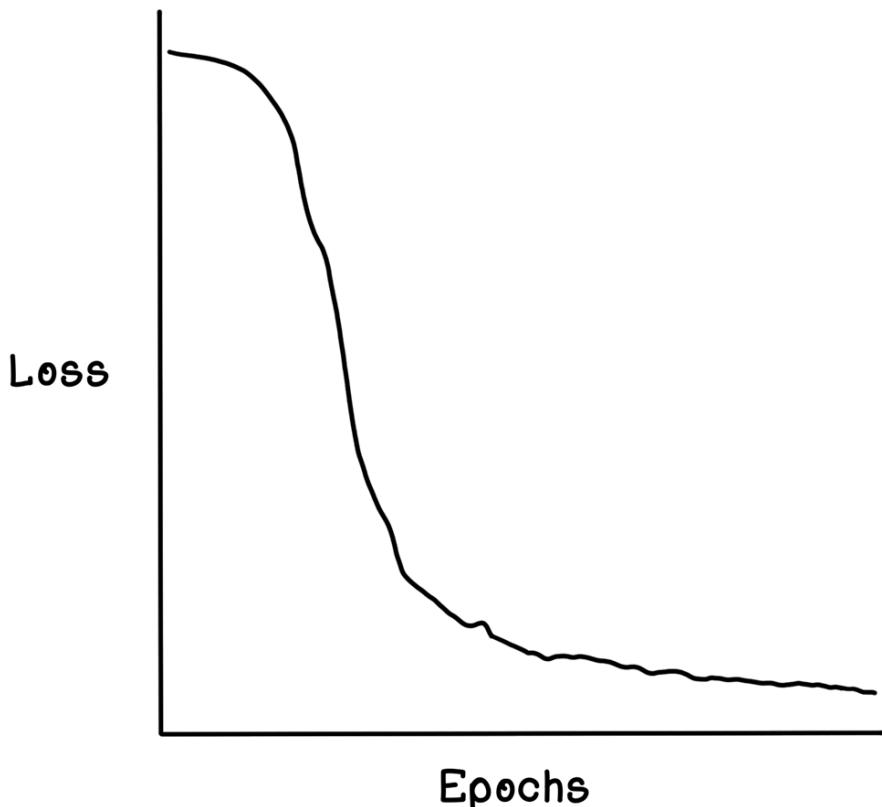
From calculating the loss using the mean squared error formula, we have a loss of **0.701**. We can interpret it as quite a big loss, because:

- $\text{Loss} = 0$  is a perfect score. It means the predicted noise matrix was identical to the actual noise matrix, with no error.

- A *low loss*, like 0.01 indicates that, on average, the squared difference between the predicted and actual pixel values is tiny - the model is performing well and its predictions are very close to the truth.
- A *high loss*, like 2.0 indicates that, on average, the predictions are far from the actual values - the model is performing poorly.

The absolute value of the loss at a specific time isn't as important as its trend over time. The primary use of the loss value during training is to see if it is consistently decreasing. A steadily decreasing loss means that the model is successfully learning from the training data with each backpropagation step. If it increases, it means that the model is getting worse over time. And if it stagnates over many epochs, then the model is likely not to improve with continuing for more epochs.

Figure 12.46 illustrates a common and ideal trend line for loss in diffusion models.



**Figure 12.46 The ideal loss-over-time for a Diffusion model**

## 12.7.2 Backpropagation

The loss score is the signal that is used in backpropagation. In this step, the model learns from its mistakes. The algorithm works backward from the loss value through every layer of the network—including the feed-forward layers for embedding the text label and timesteps and the convolutional layers. It calculates how much each individual weight contributed to the final error and then “nudges” every weight in a direction that will make the loss smaller on the next attempt.

As explored in Chapter 9, we use the chain rule. The algorithm moves backward from the final layer of the network to the one before it, and so on, all the way to the beginning. At each layer, it calculates the gradient for each weight. The gradient tells the model which direction to “nudge” the weight to decrease the error. Once the gradient for a specific weight is known, the model updates the weight.

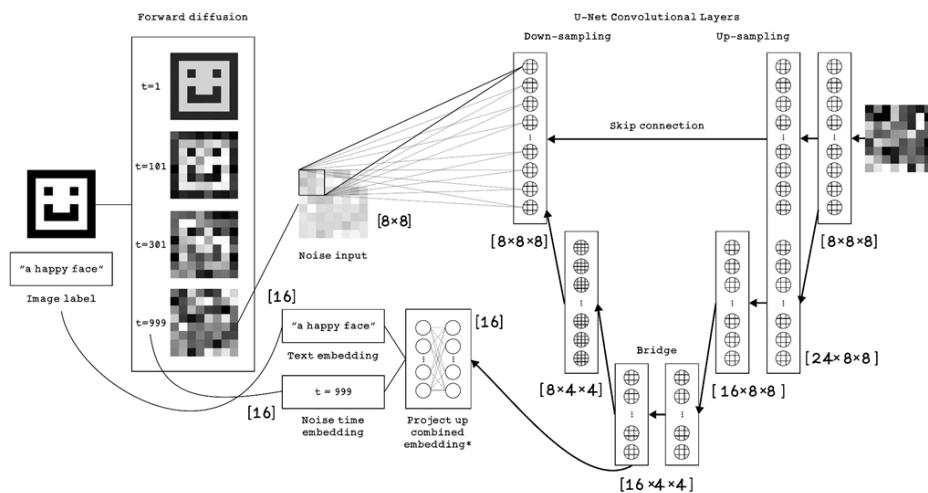
For example, if a weight's original value is **0.5**, learning rate is **0.01**, and the calculated gradient is **-0.5**, the update would be determined as shown in figure 12.47:

**new weight = old weight - (learning rate × gradient)**

```
new weight = 0.5000 - (0.01 * -0.5)
new weight = 0.5000 - (-0.005)
new weight = 0.5050
```

**Figure 12.47 Adjusting weights based on loss**

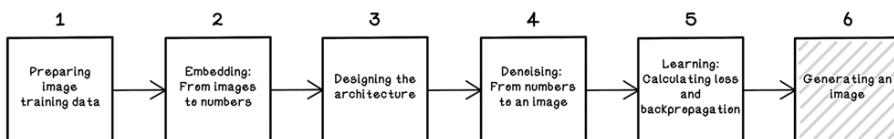
This adjustment, when applied to all weights over thousands or millions of training steps, is what allows the U-Net to refine its understanding and improve its ability to generate images. In Figure 12.48, every arrow indicates the layers and their respective weights that will be updated during backpropagation.



**Figure 12.48** The components whose weights are updated with backpropagation

## 12.8 Generating an image

Now that the U-Net has been trained, we can use it for its main purpose: generating a new image from a text prompt (figure 12.49). This process, often called *inference*, works by applying the diffusion process in reverse. We start with pure noise and use the trained model to denoise it step-by-step until a clean image emerges.



**Figure 12.49** Generating an image in the Diffusion model lifecycle

### PYTHON CODE SAMPLE

This code outlines how to calculate loss and perform backpropagation in a diffusion model. It begins by computing the Mean Squared Error (MSE) between the predicted noise and the actual noise across all pixels, producing a scalar loss value. Then, it performs backpropagation: iterating backward through each model layer to compute gradients of the loss with respect to each weight using the chain rule. Each weight is updated using gradient descent, scaled by the learning rate. This process gradually tunes the model to more accurately predict noise over many training steps.

```

def calculate_loss_and_backprop(predicted_noise, actual_noise, learning_rate):

    error = predicted_noise - actual_noise

    loss = np.mean(np.square(error))

    dummy_weights = ['conv1_w', 'resblock_b_w', 'attn_v_w', 'proj_up_w']

    backprop_and_update(dummy_weights, loss, learning_rate)

    return loss

```

### 12.8.1 Starting with a blank canvas (of pure noise)

Unlike training, the inference process doesn't start with a pre-existing image. Instead, we begin with a blank canvas filled entirely with random noise (figure 12.50). We create an 8 x 8 matrix of random numbers drawn from a *standard normal distribution*. Remember, drawing numbers from the standard normal distribution will result in statistically predictable random numbers.

This noisy canvas is what an image looks like at the final timestep ( $t = 1000$ ) of the forward diffusion process. Like our early analogy, think of this noisy image as a block of marble. The final, coherent image is already hidden inside, and the U-Net's job is to act as a sculptor, carefully chiseling away the noise to reveal a masterpiece within.

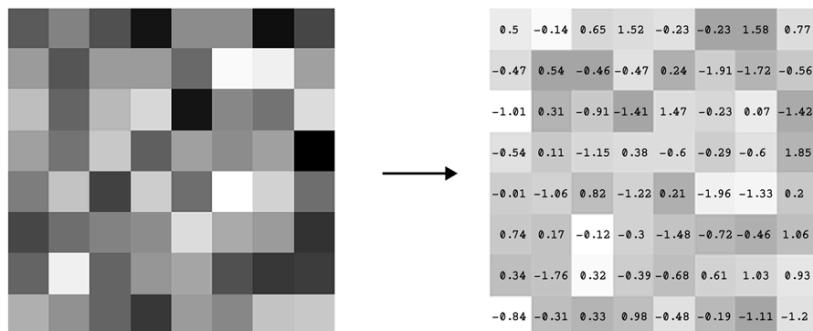


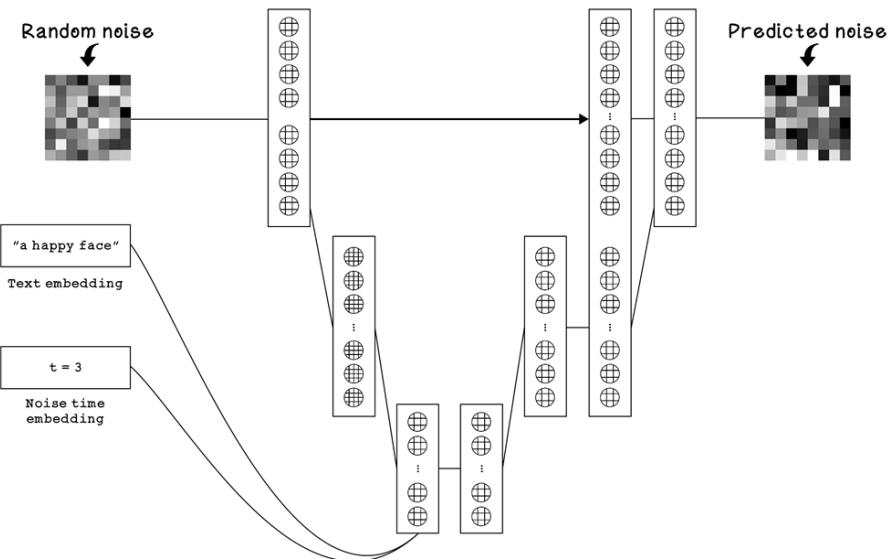
Figure 12.50 Starting with pure noise for making a prediction

## 12.8.2 Denoising the data

The model works backward from timestep 1000 (or the total timesteps in forward diffusion) to timestep 0. This happens inside a loop that iteratively cleans the image, with each step making the image slightly less noisy, and over time, making the image more true to its predicted form.

At each step in the loop, the following happens (figure 12.51):

- *Predict the noise:* The U-Net is given three inputs: the current noisy image, the current timestep,  $t$ , and the text prompt embedding, for example “a happy face”. With this information, the U-Net predicts the noise pattern that it thinks is present in the image at that specific timestep.



**Figure 12.51 How a denoised image is predicted**

- *Subtract the noise:* We then use this prediction to take one small step toward a cleaner image. Remember during the forward diffusion process, we calculated a noise schedule. We use the same schedule constants (alphas and betas) to calculate exactly how much of the predicted noise to subtract from the current image. We have to use the same alphas and betas because the denoising (reverse) process is designed to be the mathematical mirror of the noising (forward) process. The schedule of alphas and betas acts as a precise, shared “recipe” for both adding and removing noise.

Here's the alphas and betas that were previously calculated.

**Table 12.4 A subset of values for the noise schedule**

Timestep (t)	Beta	Alpha calculation (1 - beta)	Alpha	Alpha-Bar calculation	Alpha-Bar
1	0.00010	1 - 0.00010	0.99990	0.99990	0.99990
2	0.00012	1 - 0.00012	0.99988	0.99990 × 0.99988	0.99978
3	0.00014	1 - 0.00014	0.99986	0.99990 × 0.99988 × 0.99986	0.99964
...					
1000	0.02000	1 - 0.02000	0.98000		0.00004

Figure 12.52 is the denoising calculation for timestep 1000. We first need to calculate the timestep scale, which determines how much of the predicted noise we must subtract from the current image in question.

## Timestep scale for t = 1000

$$\text{timestep scale} = \frac{1 - \alpha}{\sqrt{1 - \bar{\alpha}}}$$

$$\text{timestep scale} = \frac{1 - 0.98}{\sqrt{1 - 0.00004}}$$

$$\text{timestep scale} = 0.01998$$

**Figure 12.52 Calculations for finding timestep scale for timestep 1000**

Typically, the scale is larger closer to the initial noise timestep, and smaller closer to the final image generated step. This means the model takes a bigger, more aggressive step, subtracting a significant portion of the predicted noise at the beginning. Then, taking tiny, cautious steps, subtracting only a minuscule amount of the predicted noise - like our sculptor chiseling out fine details.

**EXERCISE: WHAT IS THE TIMESTEP SCALE FOR T = 3?**

Calculate the timestep scale for timestep 3.

SOLUTION: WHAT IS THE TIMESTEP SCALE FOR T = 3?

Timestep scale for t = 3

$$\text{timestep scale} = \frac{1 - \alpha}{\sqrt{1 - \bar{\alpha}}}$$

$$\text{timestep scale} = \frac{1 - 0.99986}{\sqrt{1 - 0.99964}}$$

$$\text{timestep scale} = 0.00738$$

The calculated timestep scale is then used together with the current input matrix and predicted noise to produce the new image from the network. This is done by simply subtracting the predicted noise at the respective scale (figure 12.53).

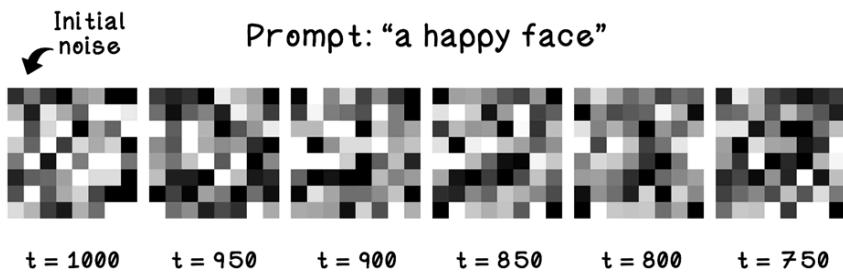


`new image = current image - (timestep scale * predicted noise)`

**Figure 12.53 Generating a new image by removing noise**

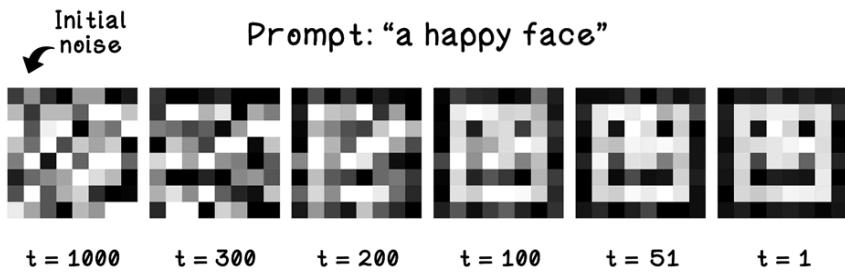
- *Repeat:* This cycle of predicting and subtracting a small amount of noise is repeated hundreds of times. With each iteration, the image becomes progressively less noisy and more defined, gradually revealing the final result that is guided by the text prompt.

Figure 12.54 shows the chaotic nature of the denoising process at the start. Notice how the noise changes drastically but still resembles nothing close to “a happy face”.



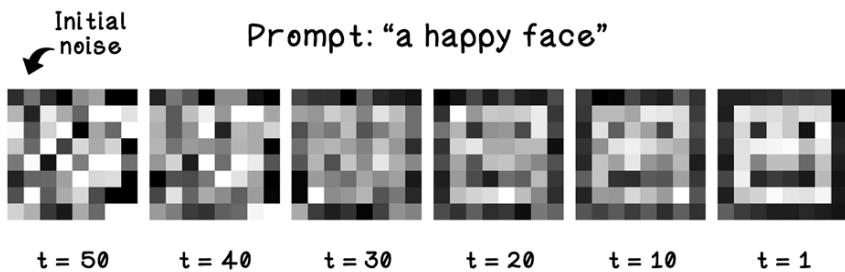
**Figure 12.54 The denoised images from timestep 1000 to timestep 750**

However, in the timesteps towards the end, closer to the final timestep, notice that a resemblance of “a happy face” starts appearing from as soon as timestep 100, and gradually denoises to reveal an almost perfect image at timestep 1 (figure 12.55).



**Figure 12.55** The denoised images from timestep 1000 to timestep 1

As an interesting experiment, we can tinker with the timesteps for the inference stage. In Figure 12.56, we're only cycling through 50 timesteps. This means that there are fewer opportunities to remove noise, and since denoising models excel at removing small amounts of noise at a time, we can see that the final image is poorer than when we ran it over 1000 timesteps.



**Figure 12.56** The denoised images when we generate with only 50 timesteps instead of 1000 timesteps

## PYTHON CODE SAMPLE

This code describes the inference process for generating an image from a text prompt using a trained diffusion model. It starts with pure Gaussian noise and iteratively denoises it over a defined number of timesteps. At each step, it generates a timestep embedding, combines it with the text embedding, and passes these through the U-Net to predict the noise present in the image. It then uses the learned noise schedule (*alphas* and *betas*) to subtract the predicted noise, gradually refining the image. Optionally, random noise is re-added at each step to maintain realism. After all timesteps, the result is a fully generated image guided by the input prompt.

```

def generate_image_with_prompt(prompt, total_timesteps):

    alpha_schedule, beta_schedule, alpha_bar_schedule =
define_schedules(total_timesteps)

    text_embedding = create_text_embedding(prompt, EMBEDDING_SIZE)

    current_image = np.random.randn(IMAGE_H, IMAGE_W)

    for t in range(total_timesteps, 0, -1):

        idx = t - 1

        timestep_embedding = create_timestep_embedding(t, EMBEDDING_SIZE)
        combined_embedding = inject_embeddings_into_bridge(None, text_embedding,
timestep_embedding)

        predicted_noise = run_unet_forward_pass(current_image,
combined_embedding, t)

        alpha_t = alpha_schedule[idx]
        alpha_bar_t = alpha_bar_schedule[idx]
        beta_t = beta_schedule[idx]

        noise_scale = 1.0 / np.sqrt(alpha_t)
        residual_scale = (1.0 - alpha_t) / np.sqrt(1.0 - alpha_bar_t)

        denoised_image = noise_scale * (current_image - residual_scale *
predicted_noise)

        if t > 1:
            z = np.random.randn(*current_image.shape)
            sigma_t = np.sqrt(beta_t)
            denoised_image += sigma_t * z

        current_image = denoised_image

    return current_image

```

## 12.9 Controlling the diffusion model

Congratulations, you've completed the entire training and inference cycle for a generative image model using diffusion. It's amazing and interesting that something as counterintuitive as learning how noise can be removed can produce the fantastical images that we see generated by modern image generation models. Large-scale models are trained with billions of images, have mode channels to support color images, have thousands of nodes in their U-net layers, and perform drastically more matrix manipulation operations than we have in this small example. Even so, the core principles remain the same, but are just scaled. Let's explore ways that we can control the effectiveness of training and quality of the images that are generated by a model.

### 12.9.1 Training data composition and diversity

Before a diffusion model ever touches a pixel, we need to decide what kind of visual world it should learn from. The quality, diversity, and structure of your image dataset are crucial to how flexible, creative, and useful the model will be. Just like an LLM trained on narrow legal text might struggle with poetry, an image model trained solely on X-rays won't know how to paint a dog in a sunflower field, not because it's unintelligent, but because it's never seen one.

The dataset defines the model's visual vocabulary. That includes:

- What kinds of objects exist
- What relationships those objects have
- What textures, styles, lighting, and colors appear
- What prompts those images were paired with

In essence, the dataset sets the imaginative boundaries of your model. If your dataset is filled with architectural blueprints, don't expect it to generate whimsical landscapes. But if you feed it a variety of paintings, photos, doodles, and diagrams, you're giving it the essence for creative synthesis.

- A *narrow dataset* like chest X-rays, satellite images, or architecture diagrams trains the model to be expert-level in one thing. This is great for niche tasks, but not general-purpose image generation.
- A *broad dataset* teaches the model about the messy, diverse real world, cities, people, animals, art, memes. These models generalize much better.

### 12.9.2 Timesteps and noise schedule

Diffusion models are trained to do one main thing: recover a clean image from a noisy one. But how that noise is added, how much, how fast, and over how many steps, plays an important role in how well the model learns to denoise and generate sharp, coherent images. This process is controlled by the noise schedule - the carefully crafted plan that determines how much noise gets added at each timestep during training.

There are a few popular choices for the noise schedule:

- *Linear*: Noise is added at a steady, constant rate. Simple, but not always ideal. Early steps may be too subtle, and later ones too aggressive.
- *Cosine*: Starts gently and accelerates over time. This results in smoother degradation, especially in the early timesteps, giving the model a more consistent training signal. Many modern models prefer this.
- *Sigmoid or custom*: More exotic schedules that try to tailor the noise curve to the dataset. These can offer improvements but are harder to tune manually.

The core idea is that how you destroy the image determines how easy it is to learn to reverse that destruction. If your noise schedule is too harsh early on, the model might struggle to learn. Too gentle, and the later steps become computationally expensive with diminishing returns.

### **12.9.3 Attention layers and cross-attention injection**

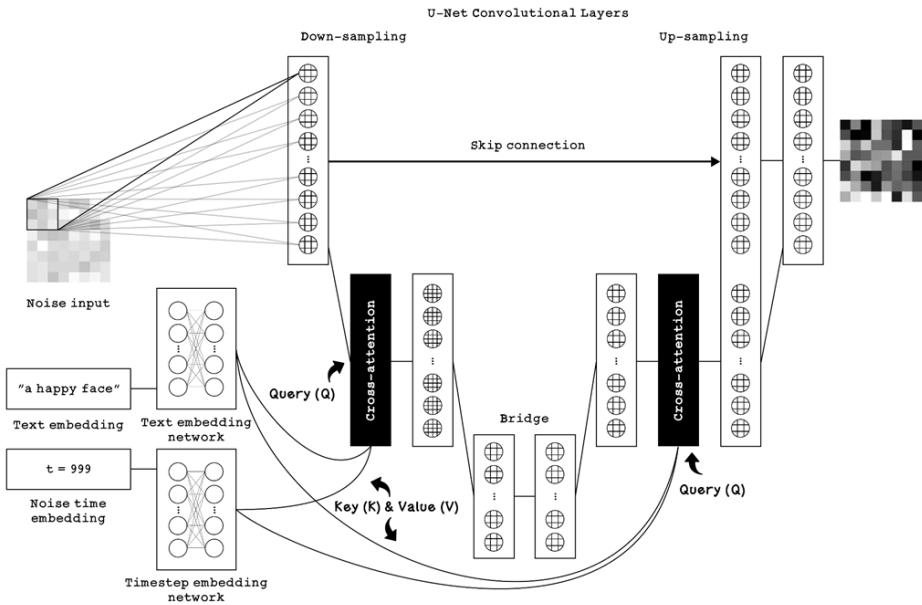
Up until now, we've manually translated the prompt "a happy face" into a vector of numbers that we used to train the U-Net at the level of the Bridge. But in a large-scale model, the words and their relationships to the vast model space are much more complex and interconnected. This is where the powerful tool of attention plays a role. As in LLMs where self-attention helps tokens find semantic relationships between each other, with U-Nets, the text label embedding is applied through cross-attention with the noise pixels and becomes associated with the patterns found.

- The image features act as the *Queries(Q)*
- The text embedding acts as *Keys(K)* and *Values(V)*

The model learns to "attend" to relevant words while refining each region of the image. This lets the model decide things like:

- Which parts of the text label / text prompt apply to this layer?
- Which word is relevant to this spatial feature?
- Should this patch be yellow because of the word "sunlight"?
- Should that patch be rounded because it's a "snail"?

Figure 12.57 illustrates how cross-attention can be applied in the model architecture that we just explored through this chapter. Notice that it is an additional layer of abstraction between down-sampling and up-sampling layers.



**Figure 12.57 How cross-attention fits into the U-Net architecture**

In large-scale models that have many of these layers, cross-attention is applied at each stage.

- *Deeper layers* (low resolution layers at the bottom of the diagram) focus on layout and composition, like “where is the snail, where is the lava?”
- *Shallower layers* (high resolution layers at the top of the diagram) refine details, like “what does the shell look like?”, “how many brush strokes should I simulate for Van Gogh’s style?”

Cross-attention is the bridge between language and vision. By injecting attention at each stage, the model builds a coherent scene that aligns with the text prompt.

#### 12.9.4 Training epochs

Diffusion models are powerful, but they’re also computationally expensive and sensitive to overtraining. Let them run too long, and you risk that the model memorizes the training set instead of learning how to generalize. If you let them stop too early, the images they generate may look noisy, dull, or “half-baked”.

Finding the balanced moment to stop training is a key part of controlling the overall quality. We want to find where the model has learned enough to generate high-quality images, without slipping into overfitting.

With LLMs, we measure the next-token prediction loss. Diffusion models require alternative signals to tell us when training is going well. Here are the most common metrics used to guide training duration:

- *Validation loss*: Measures how well the model predicts the noise on a validation set of image-noise pairs it's never seen before. If validation loss stops improving, or worse, starts increasing, the model is overfitting.
- *FID score (Fréchet Inception Distance score)*: A statistical comparison between the generated images and real ones. Lower FID means your model is producing images that look more like real photos. This is slower to compute but provides a strong perceptual quality signal.

The trick is not to wait for training to collapse. Instead, you want to track these signals as early indicators of whether the model is still improving or starting to stall.

As with other deep learning models that are computationally expensive and long, it's always a good idea to save checkpoints. This is saving the model's state, including weights, and points in the training process. With a diffusion model, monitor the validation loss or FID score, and save a new "best model" only when performance improves. This gives you a single golden checkpoint without eating massive storage space.

## 12.10 Inpainting and Outpainting

Beyond creating images from scratch, diffusion models are powerful tools for editing existing ones. Inpainting and outpainting allow a user to selectively add, remove, or change parts of an image.

- *Inpainting*: This is like having a magical eraser and redraw tool. A user can mask out, or "erase", an unwanted object or area in a picture. They can then provide a new text prompt describing what should be in its place. The diffusion model performs its denoising process, but only within the masked region. It uses the surrounding pixels as context to ensure the newly generated content blends in seamlessly with the rest of the image.
- *Outpainting*: This is often called "uncropping". It's like infinitely expanding the canvas. A user takes an existing image and extends its borders, creating a blank area around it. This blank area acts as the mask. The model then looks at the original image and the text prompt and "hallucinates" what should exist beyond the original borders, creating a wider, panoramic scene that makes sense given the starting picture.

To support these features, the key difference is that the U-Net needs to be aware of which parts of the image to change (the mask) and which parts to use as context. During the denoising process, after each step where the U-Net predicts the noise, the model uses the mask to “paste back” the original, known pixels from the unmasked region. This ensures that only the area you want to change is progressively denoised, while the context from the original image remains locked in place, resulting in a seamless blend.

## 12.11 LoRA (Low-Rank Adaptation)

Fine-tuning a massive, multi-gigabyte diffusion model for a specific art style or character, for example, is computationally expensive and would result in a new, equally massive file for every single concept that we want to focus on. LoRA (Low-Rank Adaptation) is an incredibly efficient technique that solves this problem.

Think of the main diffusion model as a master artist with a vast library of skills. A LoRA is a tiny set of instruction notes (often just 10 - 100 MB) that you give to the artist. These notes don't re-teach the artist how to paint, they provide the small, specific adjustments needed to draw in a new style or capture a particular character's likeness.

Technically, LoRA works by freezing the model's original, massive weight matrices and training two much smaller, “low-rank” matrices that capture the changes needed for the new style. During generation, the outputs from these small matrices are added to the outputs of the original model, effectively “adapting” its style on the fly. This allows users to have a single base model and thousands of small LoRA files to rapidly switch between countless different art styles, characters, and concepts.

## 12.12 High-Resolution Fix and Upscalers

Diffusion models are typically trained on images of a fixed size (for example 512 x 512). Generating directly at very high resolutions can cause strange artifacts, like repeated heads or a distorted composition. To create large, detailed images, a two-stage process is often used.

- *High-resolution fix:* This is a technique used during the initial text-to-image generation. First, the model generates a smaller, coherent image (for example at 512 x 512) to get the overall composition right. Then, it upscales this image slightly and runs a few final diffusion steps to add detail at the higher resolution without breaking the underlying structure.
- *Dedicated upscalers:* After a good image has been generated, a separate, specialized AI model called an upscaler can be used. These models are specifically trained for one task: to take a low-resolution image and intelligently add new pixels to increase its size (for example, 2x or 4x) while preserving and often enhancing the clarity and detail of the image.

## 12.13 ControlNets and IP-Adapters

While a text prompt provides general guidance, ControlNets and IP-Adapters offer much more direct and fine-grained control over the final image's composition and style.

- ControlNets act as structural guides. They are auxiliary neural networks that run alongside the main U-Net. A user provides a “control map”, such as a human pose skeleton, a Canny edge map (a simple line-art outline of an image), or a depth map, along with their text prompt. Think of this input image as the blueprint. The ControlNet reads this blueprint and injects guidance signals directly into the layers of the main diffusion model. This forces the generated image to strictly conform to the structure of your blueprint (the lines or pose) while using the text prompt to fill in the colors, lighting, and textures.
- IP-Adapters (Image Prompt Adapters) function as a style and character reference sheet. An IP-Adapter is a small module that allows you to provide a reference image along with your text prompt. The adapter extracts the core stylistic elements or character features from the reference image and injects this information into the U-Net's attention layers. This allows you to generate entirely new compositions based on your text prompt that perfectly mimic the style or feature the character from your reference image.

## 12.14 Refining Aesthetics with Human Feedback

A diffusion model can learn to accurately create an image from a prompt, but it doesn't inherently know what makes an image beautiful, aesthetically pleasing, or even if the image is as accurate as it can be. This is where a process similar to Reinforcement Learning with Human Feedback (RLHF), which is used to align LLMs, comes into play.

Think of it as sending the model to art school.

- *Collect human feedback:* First, the model generates several images for the same prompt, varying the initial noisy image fed into the model. Thousands of human labelers then review these images and rank them from best to worst based on aesthetic appeal.
- *Train a reward model:* A separate “aesthetics predictor” model is trained on this dataset of human preferences. Its only job is to learn to predict which images a human would find most visually appealing, outputting a single “aesthetics score”.
- *Fine-tune with reinforcement learning:* Finally, the main diffusion model is fine-tuned. It generates an image, the aesthetics predictor gives it a score, and this score is used as a reward to update the diffusion model's weights. Over time, the model learns to generate images that not only match the prompt but are also consistently scored higher for aesthetic quality.

## 12.15 Use cases for image generation

Image generation models are a significant leap in AI's ability to create, modify, and conceptualize visual content. Because they are trained on vast and diverse datasets of images and their corresponding text descriptions, they learn the intricate relationships between concepts and visual aesthetics. This deep understanding allows them to act as a powerful visual synthesizer, translating complex text prompts into unique, high-quality images. It's like having a world-class artist and photographer on call, ready to create any scene imaginable. This unlocks the potential for a wide variety of tasks that were previously expensive, time-consuming, or required specialized skills.

### 12.15.1 Creative ideation and concept art

For artists, designers, and creators, the most difficult step can be overcoming the "blank canvas". An image generation model can act as a powerful brainstorming partner, rapidly generating a wide range of visual ideas based on a simple prompt. The process becomes one of augmentation, where the human guides the creative direction and refines the AI-generated concepts, rather than starting from scratch.

- *Entertainment industry (Film & Gaming)*: Concept artists can generate dozens of variations for characters, environments, creatures, and props in minutes instead of days. This allows directors and creative leads to visualize and approve a direction much earlier in the production pipeline.
- *Graphic design & illustration*: Designers can use these models to create initial drafts for logos, posters, book covers, and other illustrations. This provides a rich starting point that can then be professionally refined and customized.

### 12.15.2 Commercial design and advertising

Creating high-quality visual assets for commercial use often requires expensive and logistically complex photoshoots. Image generation models offer a fast and cost-effective alternative for creating unique, on-brand imagery.

- *Marketing and advertising*: Creative teams can produce multiple versions of ad visuals for A/B testing, generate social media content tailored to different platforms, or create entire campaign aesthetics without needing a single photographer, model, or location.
- *Product visualization*: Companies can generate realistic mockups of products in various lifestyle settings. A new brand of handbag can be shown in a city, at the beach, or in a cafe, allowing marketing teams to create compelling visuals before the product is even manufactured.
- *Architectural & interior design*: Architects and designers can turn blueprints or simple descriptions into photorealistic renders of buildings and interior spaces, helping clients visualize the final result long before construction begins.

### 12.15.3 Content creation and media

Image generation models can create perfectly tailored visual assets on demand, freeing content creators from the limitations of generic stock photo libraries.

- *Custom stock photography*: Bloggers, journalists, and corporate presenters can generate a specific image that perfectly matches their content—for example, "a scientist in a lab looking at a blue liquid in a beaker with a concerned expression"—instead of spending hours searching for a suitable-but-not-perfect stock photo.
- *Fashion design*: Designers can visualize entire clothing collections on a diverse range of virtual models, experimenting with different fabrics, colors, and styles instantly, dramatically speeding up the design and prototyping phase.

### 12.15.4 Personalization and Photo Editing

These tools empower individuals to become creators and editors, regardless of their technical skill.

- *Personalized content*: Users can create custom avatars for their social media profiles, unique banners for their channels, or simply generate art for personal projects and enjoyment.
- *Intelligent photo editing*: Using inpainting, a user can easily remove an unwanted person or object from a personal photograph by simply masking it out and letting the AI fill in the background. With outpainting, they can take a favorite photo and expand its borders, allowing the AI to creatively extend the scene.

## 12.16 Summary of Generative Image Models

Counterintuitively, Diffusion starts with noise and learns how to remove tiny bits to reveal a clear image

```
1 1 1 1 1 1 1 1
1 -1 -1 -1 1 -1 -1 1
1 -1 1 1 -1 1 -1 1
1 -1 -1 -1 1 -1 -1 1
1 -1 1 -1 -1 1 -1 1
1 -1 1 -1 -1 1 -1 1
1 -1 1 1 -1 1 -1 1
1 -1 1 1 1 1 -1 1
1 -1 -1 -1 -1 -1 -1 1
1 -1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
```

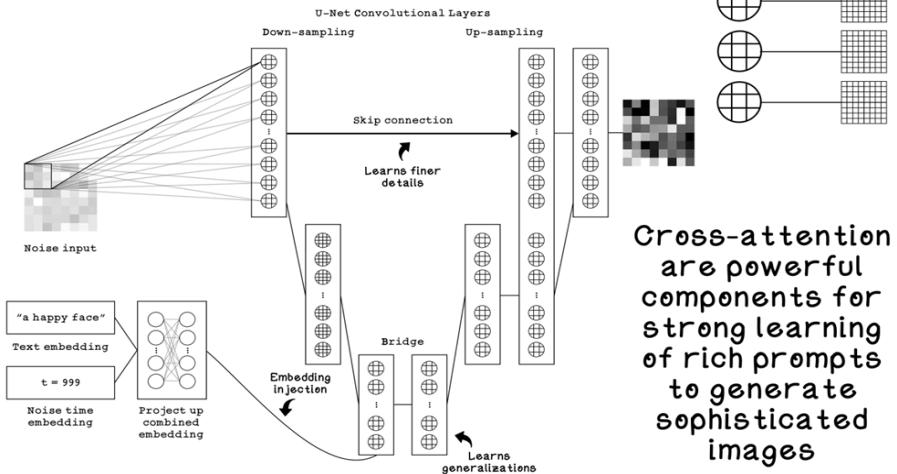
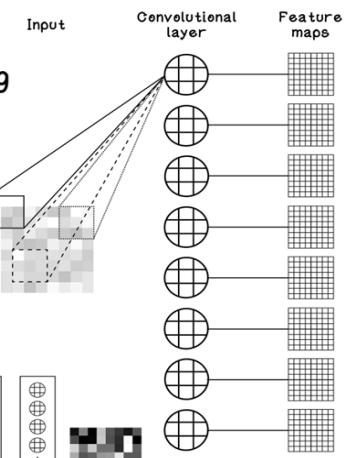
= [[1, 1, 1, 1, 1, 1, 1, 1],  
 [1,-1,-1,-1,-1,-1,-1, 1],  
 [1,-1, 1,-1,-1, 1,-1, 1],  
 [1,-1,-1,-1, 1,-1,-1, 1],  
 [1,-1, 1,-1,-1, 1,-1, 1],  
 [1,-1,-1,-1, 1,-1,-1, 1],  
 [1,-1, 1,-1,-1, 1,-1, 1],  
 [1,-1,-1,-1,-1,-1,-1, 1],  
 [1,-1, 1, 1,-1, 1,-1, 1],  
 [1,-1,-1,-1,-1,-1,-1, 1],  
 [1, 1, 1, 1, 1, 1, 1, 1]]

a happy face

Images are nothing but numbers represented as matrices

CNNs are crucial for learning shapes, colors, textures, subjects, and more

Down-sampling, embedding injection, upsampling, and skip connections are the core superpowers of the U-Net



Cross-attention are powerful components for strong learning of rich prompts to generate sophisticated images