"Andriy's long-awaited sequel in his "The Hundred-Page" series of machine learning textbooks is a masterpiece of concision."

— Bob van Luijt, CEO and Co-Founder of Weaviate

"Andriy has this almost supernatural talent for shrinking epic AI concepts down to bite-sized, 'Ah, now I get it!' moments."

— Jorge Torres, CEO at MindsDB

"Andriy paints for us, in 100 marvelous strokes, the journey from linear algebra basics to the implementation of transformers."

— Florian Douetteau, Co-founder and CEO at Dataiku

"Andriy's book is an incredibly concise, clear, and accessible introduction to machine learning."

— Andre Zayarni, Co-founder and CEO at Qdrant

"This is one of the most comprehensive yet concise handbooks out there for truly understanding how LLMs work under the hood."

— Jerry Liu, Co-founder and CEO at LlamaIndex

Featuring a foreword by **Tomáš Mikolov** and back cover text by **Vint Cerf**

The Hundred-Page Language Models Book

Andriy Burkov

ISBN 978-1-7780427-2-0

Publisher: True Positive Inc.

To my family, with love

"Language is the source of misunderstandings."
***—Antoine de Saint-Exupéry**, The Little Prince*

"In mathematics you don't understand things. You just get used to them."
—John von Neumann

"Computers are useless. They can only give you answers."
— Pablo Picasso

The book is distributed on the "read first, buy later" principle

Contents

Chapter 1. Machine Learning Basics

This chapter introduces the fundamental concepts of machine learning. Starting with the evolution of artificial intelligence, it defines a model and presents the four-step machine learning process. The chapter then covers essential mathematical foundations, including vectors and matrices, before examining neural networks. It concludes with key optimization techniques, focusing on gradient descent and automatic differentiation.

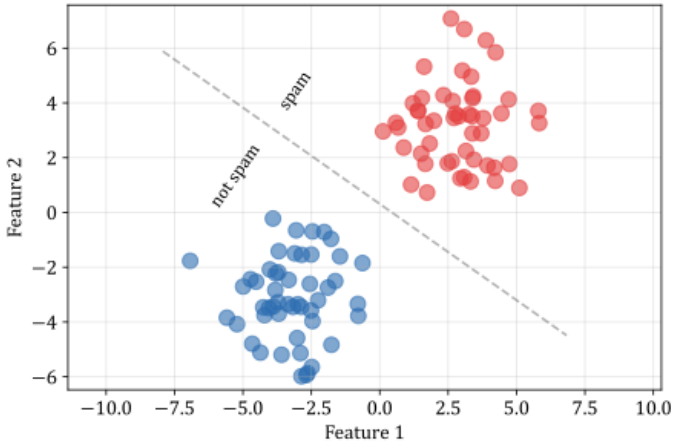
1.1. AI and Machine Learning

The term **artificial intelligence** (AI) was first introduced in 1955 during a workshop led by John McCarthy. Researchers at the workshop aimed to explore how machines could use language, form concepts, solve problems like humans, and improve over time.

1.1.1. Early Progress

The field's first major breakthrough came in 1956 with the **Logic Theorist**. Created by Allen Newell, Herbert Simon, and Cliff Shaw, it was the first program engineered to perform automated reasoning, and has been later described as "the first artificial intelligence program."

Frank Rosenblatt's **Perceptron** (1958) was an early **neural network** designed to recognize patterns by adjusting its internal parameters based on examples. Perceptron learned a **decision boundary**—a dividing line that separates examples of different classes (e.g., spam versus not spam):

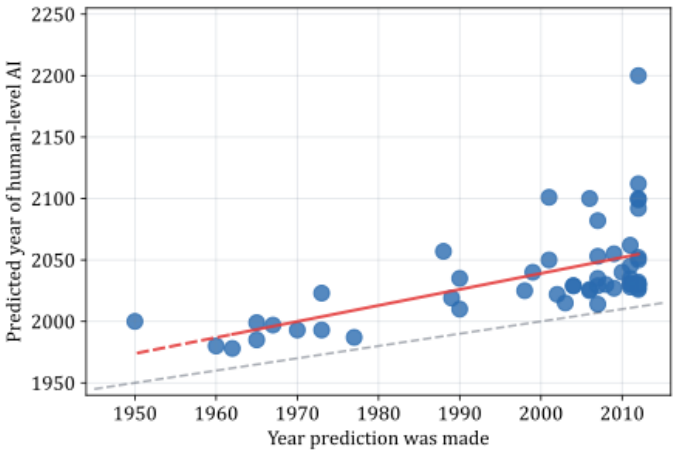


Around the same time, in 1959, Arthur Samuel coined the term **machine learning**. In his paper, "Some Studies in Machine Learning Using the Game of Checkers," he described machine learning as "programming computers to learn from experience."

Another notable development of the mid-1960s was **ELIZA**. Developed in 1967 by Joseph Weizenbaum and being the first chatbot in history, ELIZA gave the illusion of understanding language by matching patterns in users' text and generating preprogrammed responses. Despite its simplicity, it illustrated the lure of building machines that could appear to think or understand.

Optimism about near-future breakthroughs ran high during this period. Herbert Simon, a future Turing Award recipient, exemplified this enthusiasm when he predicted in 1965 that "machines will

be capable, within twenty years, of doing any work a man can do." Many experts shared this optimism, forecasting that truly human-level AI—often called **artificial general intelligence** (AGI)—was just a few decades away. Interestingly, these predictions maintained a consistent pattern: decade after decade, AGI remained roughly 25 years on the horizon:



1.1.2. AI Winters

As researchers tried to deliver on early promises, they encountered unforeseen complexity. Numerous high-profile projects failed to meet ambitious goals. As a consequence, funding and enthusiasm waned significantly between 1975 and 1980, a period now known as the first **AI winter**.

> During the first AI winter, even the term "AI" became somewhat taboo. Many researchers rebranded their work as "informatics," "knowledge-based systems," or "pattern recognition" to avoid association with AI's perceived failures.

In the 1980s, a resurgence of interest in **expert systems**—rule-based software designed to replicate specialized human knowledge—promised to capture and automate domain expertise. These expert systems were part of a broader branch of AI research known as **symbolic AI**, often referred to as **good old-fashioned AI** (GOFAI), which had been a dominant approach since AI's earliest days. GOFAI methods relied on explicitly coded rules and symbols to represent knowledge and logic, and while they worked well in narrowly defined areas, they struggled with scalability and adaptability.

From 1987 to 2000, AI entered its second winter, when the limitations of symbolic methods caused funding to diminish, once again leading to numerous research and development projects being put on hold or canceled.

Despite these setbacks, new techniques continued to evolve. In particular, **decision trees**, first introduced in 1963 by John Sonquist and James Morgan and then advanced by Ross Quinlan's **ID3** algorithm in 1986, split data into subsets through a tree-like structure. Each node in a tree represents a question about the data, each branch is an answer, and each leaf provides a prediction. While easy to interpret, decision trees were prone to **overfitting**, where they adapted too closely to training data, reducing their ability to perform well on new, unseen data.

1.1.3. The Modern Era

In the late 1990s and early 2000s, incremental improvements in hardware and the availability of larger datasets (thanks to the widespread use of the Internet) started to lift AI from its second winter. Leo Breiman's **random forest** algorithm (2001) addressed overfitting in decision trees by creating multiple trees on random subsets of the data and then combining their outputs—dramatically improving predictive accuracy.

Support vector machines (SVMs), introduced in 1992 by Vladimir Vapnik and his colleagues, were another significant step forward. SVMs identify the optimal hyperplane that separates data points of different classes with the widest margin. The introduction of **kernel methods** allowed SVMs to manage complex, non-linear patterns by mapping data into higher-dimensional spaces, making it easier to find a suitable separating hyperplane. These innovations placed SVMs at the center of machine learning research in the early 2000s.

A turning point arrived around 2012, when more advanced versions of neural networks called **deep neural networks** began outperforming other techniques in fields like speech and image recognition. Unlike the simple Perceptron, which used only a single "layer" of learnable parameters, this **deep learning** approach stacked multiple layers to tackle much more complex problems. Surging computational power, abundant data, and algorithmic advancements converged to produce remarkable breakthroughs. As academic and commercial interest soared, so did AI's visibility and funding.

Today, AI and machine learning remain intimately entwined. Research and industry efforts continue to seek ever more capable models that learn complex tasks from data. Although predictions of achieving human-level AI "in just 25 years" have consistently failed to materialize, AI's impact on everyday applications is undeniable.

Throughout this book, AI refers broadly to techniques that enable machines to solve problems once considered solvable only by humans, with machine learning being its key subfield focusing on creating algorithms learning from collections of examples. These examples can come from nature, be designed by humans, or be generated by other algorithms. The process involves gathering a dataset and building a model from it, which is then used to solve a problem.

> I will use "learning" and "machine learning" interchangeably to save keystrokes.

Let's examine what exactly we mean by a model and how it forms the foundation of machine learning.

1.2. Model

A **model** is typically represented by a mathematical equation:

$$y = f(x)$$

Here, x is the input, y is the output, and f represents a function of x. A **function** is a named rule that describes how one set of values is related to another. Formally, a function f maps inputs from the **domain** to outputs in the **codomain**, ensuring each input has exactly one output. The function uses a specific rule or formula to transform the input into the output.

In machine learning, the goal is to compile a **dataset** of **examples** and use them to build f, so when f is applied to a new, unseen x, it produces a y that gives meaningful insight into x.

To estimate a house's price based on its area, the dataset might include (area, price) pairs such as {(150,200), (200,600), ... }. Here, the area is measured in m^2, and the price is in thousands.

> Curly brackets denote a set. A set containing N elements, ranging from x_1 to x_N, is expressed as $\{x_i\}_{i=1}^{N}$.

Imagine we own a house with an area of 250 m^2 (about 2691 square feet). To find a function f that returns a reasonable price for this house, testing every possible function is infeasible. Instead, we select a specific *structure* for f and focus on functions that match this structure.

Let's define the structure for f as:

$$f(x) \stackrel{\text{def}}{=} wx + b, \tag{1.1}$$

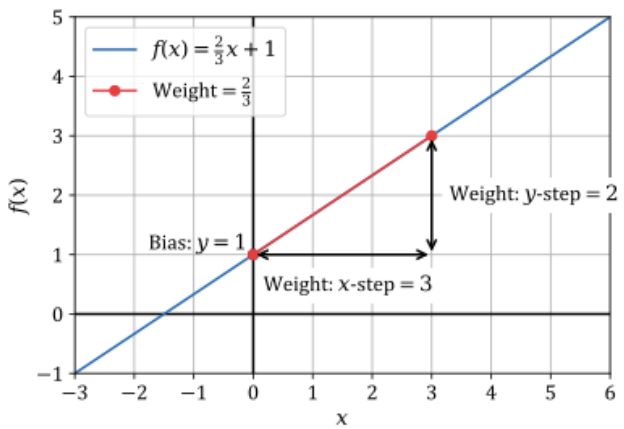
which is a **linear function** of x. The formula $wx + b$ is a **linear transformation** of x.

> The notation $\stackrel{\text{def}}{=}$ means "equals by definition" or "is defined as."

For linear functions, determining f requires only two values: w and b. These are called the **parameters** or **weights** of the model.

In other texts, w might be referred to as the **slope**, **coefficient**, or **weight term**. Similarly, b may be called the **intercept**, **constant term**, or **bias**. In this book, we'll stick to "weight" for w and "bias" for b, as these terms are widely used in machine learning. When the meaning is clear, "parameters" and "weights" will be used interchangeably.

For instance, when $w = \frac{2}{3}$ and $b = 1$, the linear function is shown below:



Here, the bias shifts the graph vertically, so the line crosses the y-axis at $y = 1$. The weight determines the slope, meaning the line rises by 2 units for every 3 units it moves to the right.

> Mathematically, the function $f(x) = wx + b$ is an **affine transformation**, not a linear one, since true linear transformations require $b = 0$. However, in machine learning, we often call such models "linear" whenever the parameters appear linearly in the equation—meaning w and b are only multiplied by inputs or constants and added,

> without multiplying each other, being raised to powers, or appearing inside functions like e^w.

Even with a simple model like $f(x) = wx + b$, the parameters w and b can take infinitely many values. To find the best ones, we need a way to measure optimality. A natural choice is to minimize the average prediction error when estimating house prices from area. Specifically, we want $f(x) = wx + b$ to generate predictions that match the actual prices as closely as possible.

Let our dataset be $\{(x_i, y_i)\}_{i=1}^{N}$, where N is the size of the dataset and $\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$ are individual examples, with each x_i being the **input** and corresponding y_i being the **target**. When examples contain both inputs and targets, the learning process is called **supervised**. This book focuses on supervised machine learning.

> Other machine learning types include **unsupervised learning**, where models learn patterns from inputs alone, and **reinforcement learning**, where models learn by interacting with environments and receiving rewards or penalties for their actions.

When $f(x)$ is applied to x_i, it generates a predicted value $\tilde{y}_i$. We can define the prediction error $\text{err}(\tilde{y}_i, y_i)$ for a given example (x_i, y_i) as:

$$\text{err}(\tilde{y}_i, y_i) \stackrel{\text{def}}{=} (\tilde{y}_i - y_i)^2 \tag{1.2}$$

This expression, called **squared error**, equals 0 when $\tilde{y}_i = y_i$. This makes sense: no error if predicted price matches the actual price. The further $\tilde{y}_i$ deviates from y_i, the larger the error becomes. Squaring ensures the error is always positive, whether the prediction overshoots or undershoots.

We define w^* and b^* as the optimal parameter values for w and b in our function f, when they minimize the average price prediction error across our dataset. This error is calculated using the following expression:

$$\frac{\text{err}(\tilde{y}_1, y_1) + \text{err}(\tilde{y}_2, y_2) + \cdots + \text{err}(\tilde{y}_N, y_N)}{N}$$

Let's rewrite the above expression by expanding each $\text{err}(\cdot)$:

$$\frac{(\tilde{y}_1 - y_1)^2 + (\tilde{y}_2 - y_2)^2 + \cdots + (\tilde{y}_N - y_N)^2}{N}$$

Let's assign the name $J(w, b)$ to our expression, turning it into a function:

$$J(w, b) \stackrel{\text{def}}{=} \frac{(wx_1 + b - y_1)^2 + (wx_2 + b - y_2)^2 + \cdots + (wx_N + b - y_N)^2}{N} \tag{1.3}$$

In the equation defining $J(w, b)$, which represents the average prediction error, the values of x_i and y_i for each i from 1 to N are known since they come from the dataset. The unknowns are w and b. To determine the optimal w^* and b^*, we need to minimize $J(w, b)$. As this function is quadratic in two variables, calculus guarantees it has a single minimum.

The expression in Equation 1.3 is referred to as the **loss function** in the machine learning problem of **linear regression**. In this case, the loss function is the **mean squared error** or **MSE**.

To find the optimum (minimum or maximum) of a function, we calculate its **first derivative**. When we reach the optimum, the first derivative equals zero. For functions of two or more variables, like

the loss function $J(w, b)$, we compute **partial derivatives** with respect to each variable. We denote these as $\frac{\partial J}{\partial w}$ for w and $\frac{\partial J}{\partial b}$ for b.

To determine w^* and b^*, we solve the following system of two equations:

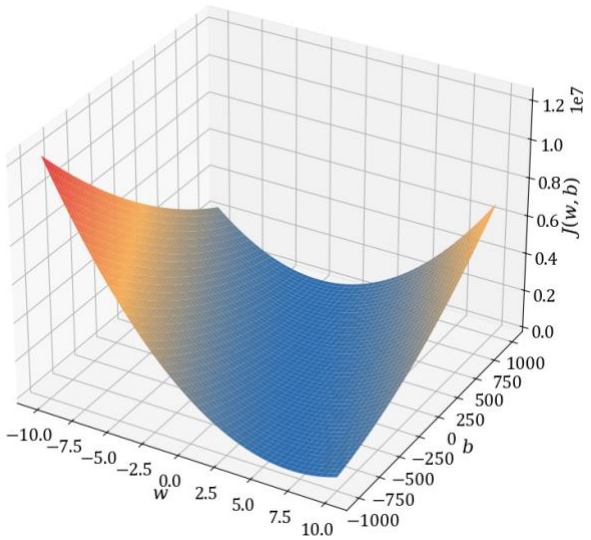
$$\begin{cases} \frac{\partial J}{\partial w} & = 0 \\ \frac{\partial J}{\partial b} & = 0 \end{cases}$$

We set the partial derivatives to zero because when this occurs, we are at an optimum.

Fortunately, the MSE function's structure and the model's linearity allow us to solve this system of equations analytically. To illustrate, consider a dataset with three examples: $(x_1, y_1) = (150,200)$, $(x_2, y_2) = (200,600)$, and $(x_3, y_3) = (260,500)$. For this dataset, the loss function is:

$$J(w, b) \stackrel{\text{def}}{=} \frac{(150w + b - 200)^2 + (200w + b - 600)^2 + (260w + b - 500)^2}{3}$$

Let's plot it:



> Navigate to the book's wiki, from the file thelmbook.com/py/1.1 retrieve the code used to generate the above plot, run the code, and rotate the graph to observe the minimum.

Now we need to derive the expressions for $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$. Notice that $J(w, b)$ is a composition of the following functions:

- Functions $d_1 \stackrel{\text{def}}{=} 150w + b - 200$, $d_2 \stackrel{\text{def}}{=} 200w + b - 600$, $d_3 \stackrel{\text{def}}{=} 260w + b - 500$ are linear functions of w and b;
- Functions $\text{err}_1 \stackrel{\text{def}}{=} d_1^2$, $\text{err}_2 \stackrel{\text{def}}{=} d_2^2$, $\text{err}_3 \stackrel{\text{def}}{=} d_3^2$ are quadratic functions of d_1, d_2, and d_3;

- Function $J \overset{\text{def}}{=} \frac{1}{3}(\text{err}_1 + \text{err}_2 + \text{err}_3)$ is a linear function of err_1, err_2, and err_3.

> A **composition of functions** means the output of one function becomes the input to another. For example, with two functions f and g, you first apply g to x, then apply f to the result. This is written as $f\big(g(x)\big)$, which means you calculate $g(x)$ first and then use that result as the input for f.

In our loss function $J(w, b)$, the process starts by computing the linear functions for d_1, d_2, and d_3 using the current values of w and b. These outputs are then passed into the quadratic functions err_1, err_2, and err_3. The final step is averaging these results to compute J.

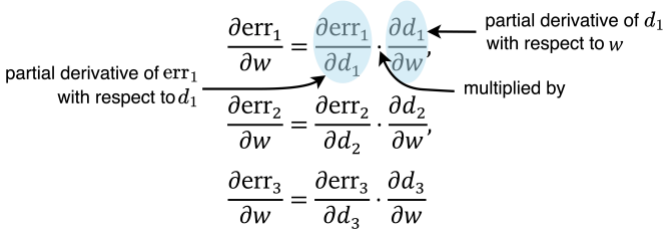
Using the sum rule and the constant multiple rule of differentiation, $\frac{\partial J}{\partial w}$ is given by:

$$\frac{\partial J}{\partial w} = \frac{1}{3}\left(\frac{\partial \text{err}_1}{\partial w} + \frac{\partial \text{err}_2}{\partial w} + \frac{\partial \text{err}_3}{\partial w}\right),$$

where $\frac{\partial \text{err}_1}{\partial w}$, $\frac{\partial \text{err}_2}{\partial w}$, and $\frac{\partial \text{err}_3}{\partial w}$ are the partial derivatives of err_1, err_2, and err_3 with respect to w.

> The **sum rule** of differentiation states that the derivative of the sum of two functions equals the sum of their derivatives: $\frac{\partial}{\partial x}[f(x) + g(x)] = \frac{\partial}{\partial x}f(x) + \frac{\partial}{\partial x}g(x)$.
>
> The **constant multiple rule** of differentiation states that the derivative of a constant multiplied by a function equals the constant times the derivative of the function: $\frac{\partial}{\partial x}[c \cdot f(x)] = c \cdot \frac{\partial}{\partial x}f(x)$.

By applying the chain rule of differentiation, the partial derivatives of err_1, err_2, and err_3 with respect to w are:



> The **chain rule** of differentiation states that the derivative of a **composite function** $f\big(g(x)\big)$, written as $\frac{\partial}{\partial x}\big[f\big(g(x)\big)\big]$, is the product of the derivative of f with respect to g and the derivative of g with respect to x, or: $\frac{\partial}{\partial x}\big[f\big(g(x)\big)\big] = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$.

Then,

$$\frac{\partial \text{err}_1}{\partial d_1} \xrightarrow{\frac{\partial \text{err}_1}{\partial w} = 2d_1 \cdot 150 = 300 \cdot (150w + b - 200),} \xleftarrow{\frac{\partial d_1}{\partial w}}$$
$$\frac{\partial \text{err}_2}{\partial w} = 2d_2 \cdot 200 = 400 \cdot (200w + b - 600),$$
$$\frac{\partial \text{err}_3}{\partial w} = 2d_3 \cdot 260 = 520 \cdot (260w + b - 500)$$

Therefore,

$$\begin{aligned}\frac{\partial J}{\partial w} &= \frac{1}{3}\big(300 \cdot (150w + b - 200) + 400 \cdot (200w + b - 600) + 520 \cdot (260w + b - 500)\big) \\ &= \frac{1}{3}(260200w + 1220b - 560000)\end{aligned}$$

Similarly, we find $\frac{\partial J}{\partial b}$:

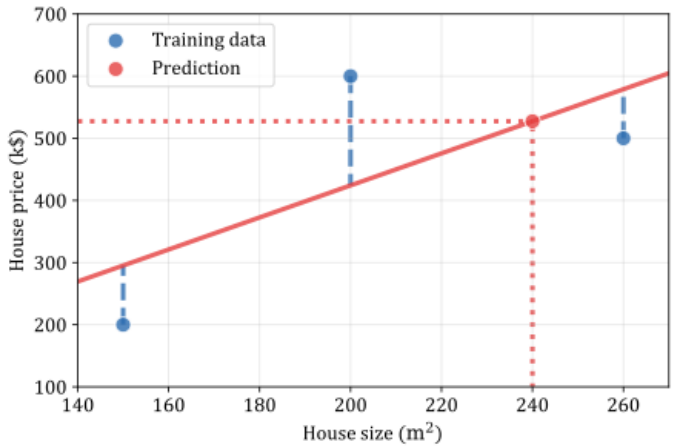
$$\begin{aligned}\frac{\partial J}{\partial b} &= \frac{1}{3}\big(2 \cdot (150w + b - 200) + 2 \cdot (200w + b - 600) + 2 \cdot (260w + b - 500)\big) \\ &= \frac{1}{3}(1220w + 6b - 2600)\end{aligned}$$

Setting the partial derivatives to 0 results in the following system of equations:

$$\begin{cases}\frac{1}{3}(260200w + 1220b - 560000) = 0 \\ \frac{1}{3}(1220w + 6b - 2600) = 0\end{cases}$$

Simplifying the system and using substitution to solve for the variables gives the optimal values: $w^* = 2.58$ and $b^* = -91.76$.

The resulting model $f(x) = 2.58x - 91.76$ is shown in the plot below. It includes the three examples (blue dots), the model itself (red solid line), and a prediction for a new house with an area of 240 m^2 (dotted orange lines).



A vertical blue dashed line shows the square root of the model's prediction error compared to the actual price.[1] Smaller errors mean the model **fits** the data better. The loss, which aggregates these errors, measures how well the model aligns with the dataset.

When we calculate the loss using our model's training dataset (called the **training set**), we obtain the **training loss**. For our model, this training loss is defined by Equation 1.3. Using our learned parameter values, we can now compute the loss for the training set:

$$\begin{aligned} J(2.58, -91.76) &= \frac{(2.58 \cdot 150 - 91.76 - 200)^2}{3} + \frac{(2.58 \cdot 200 - 91.76 - 600)^2}{3} \\ &+ \frac{(2.58 \cdot 260 - 91.76 - 500)^2}{3} \\ &= 15403.19. \end{aligned}$$

The square root of this value is approximately 124.1, indicating an average prediction error of around $124,100. The interpretation of whether a loss value is high or low depends on the specific business context and comparative benchmarks. Neural networks and other non-linear models, which we explore later in this chapter, typically achieve lower loss values.

1.3. Four-Step Machine Learning Process

At this stage, you should clearly understand the four steps involved in supervised learning:

1. **Collect a dataset**: For example, $(x_1, y_1) = (150,200)$, $(x_2, y_2) = (200,600)$, and $(x_3, y_3) = (260,500)$.
2. **Define the model's structure**: For example, $y = wx + b$.
3. **Define the loss function**: Such as Equation 1.3.
4. **Minimize the loss**: Minimize the loss function on the dataset.

In our example, we minimized the loss manually by solving a system of two equations with two variables. This approach works for small systems. However, as models grow in complexity—such as large language models with billions of parameters—manual approach becomes infeasible. Let's now introduce new concepts that will help us address this challenge.

1.4. Vector

To predict a house price, knowing its area alone isn't enough. Factors like the year of construction or the number of bedrooms and bathrooms also matter. Suppose we use two attributes: (1) area and (2) number of bedrooms. In this case, the input $\mathbf{x}$ becomes a **feature vector**. This vector includes two **features**, also called **dimensions** or **components**:

$$\mathbf{x} \stackrel{\text{def}}{=} \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix}$$

In this book, the vectors are represented with lowercase bold letters, such as $\mathbf{x}$ or $\mathbf{w}$. For a given house $\mathbf{x}$, $x^{(1)}$ represents its size in square meters, and $x^{(2)}$ is the number of bedrooms.

[1] It's the square root of the error because our error was defined as the square of the difference between the predicted price and the real price of the house. Taking a square root of this error makes it easier to interpret the error value.

A vector is usually represented as a column of numbers, called a **column vector**. However, in text, it is often written as its **transpose**, $\mathbf{x}^\top$. Transposing a column vector converts it into a **row vector**. For example, $\mathbf{x}^\top \stackrel{\text{def}}{=} \left[x^{(1)}, x^{(2)}\right]$ or $\mathbf{x} \stackrel{\text{def}}{=} \left[x^{(1)}, x^{(2)}\right]^\top$.

The **dimensionality** of the vector, or its **size**, refers to the number of components it contains. Here, $\mathbf{x}$ has two components, so its dimensionality is 2.

With two features, our linear model needs three parameters: the weights $w^{(1)}$ and $w^{(2)}$, and the bias b. The weights can be grouped into a vector:

$$\mathbf{w} \stackrel{\text{def}}{=} \begin{bmatrix} w^{(1)} \\ w^{(2)} \end{bmatrix}$$

The linear model can then be written compactly as:

$$y = \mathbf{w} \cdot \mathbf{x} + b, \tag{1.4}$$

where $\mathbf{w} \cdot \mathbf{x}$ is a **dot product** of two vectors (also known as **scalar product**). It is defined as:

$$\mathbf{w} \cdot \mathbf{x} \stackrel{\text{def}}{=} \sum_{j=1}^{D} w^{(j)} x^{(j)}$$

The dot product combines two vectors of the same dimensionality to produce a **scalar**, a number like 22, 0.67, or -10.5. Scalars in this book are denoted by italic lowercase or uppercase letters, such as x or D. The expression $\mathbf{w} \cdot \mathbf{x} + b$ generalizes the idea of a **linear transformation** to vectors.

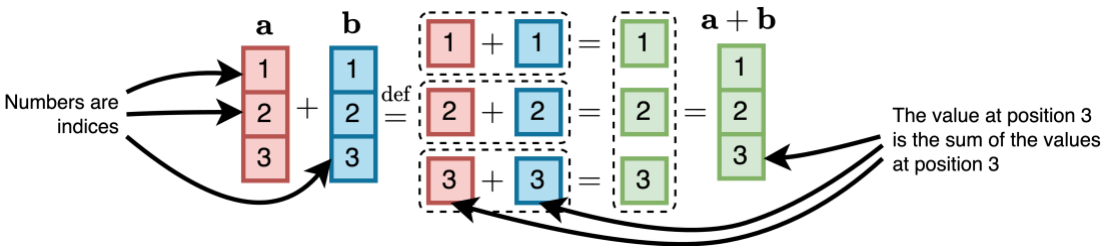
The equation above uses **capital-sigma notation**, where D represents the dimensionality of the input, and j runs from 1 to D. For example, in the 2-dimensional house scenario, $\sum_{j=1}^{2} w^{(j)} x^{(j)} \stackrel{\text{def}}{=} w^{(1)}x^{(1)} + w^{(2)}x^{(2)}$.

Although the capital-sigma notation suggests the dot product might be implemented as a loop, modern computers handle it much more efficiently. Optimized **linear algebra libraries** like **BLAS** and **cuBLAS** compute the dot product using low-level, highly optimized methods. These libraries leverage hardware acceleration and parallel processing, achieving speeds far beyond a simple loop.

The **sum of two vectors a** and **b**, both with the same dimensionality D, is defined as:

$$\mathbf{a} + \mathbf{b} \stackrel{\text{def}}{=} \left[a^{(1)} + b^{(1)}, a^{(2)} + b^{(2)}, \ldots, a^{(D)} + b^{(D)}\right]^\top$$

The calculation for a sum of two 3-dimensional vectors is illustrated below:

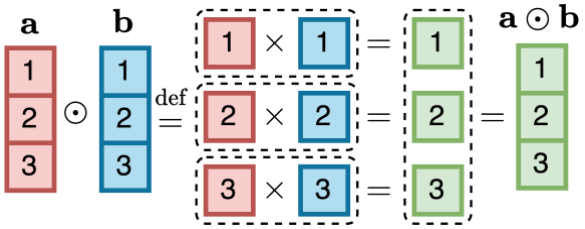


> In this chapter's illustrations, the numbers in the cells indicate the position of an element within an input or output matrix, or a vector. They do not represent actual values.

The **element-wise product** of two vectors **a** and **b** of dimensionality D, is defined as:

$$\mathbf{a} \odot \mathbf{b} \stackrel{\text{def}}{=} [a^{(1)} \cdot b^{(1)}, a^{(2)} \cdot b^{(2)}, \ldots, a^{(D)} \cdot b^{(D)}]^\top$$

The computation of the element-wise product for two 3-dimensional vectors is shown below:



The **norm** of a vector **x**, denoted $\| \mathbf{x} \|$, represents its **length** or **magnitude**. It is defined as the square root of the sum of the squares of its components:

$$\| \mathbf{x} \| \stackrel{\text{def}}{=} \sqrt{\sum_{j=1}^{D} (x^{(j)})^2}$$

For a 2-dimensional vector **x**, the norm is:

$$\| \mathbf{x} \| = \sqrt{(x^{(1)})^2 + (x^{(2)})^2}$$

The cosine of the angle θ between two vectors **x** and **y** is defined as:

$$\cos(\theta) = \frac{\mathbf{x} \cdot \mathbf{y}}{\| \mathbf{x} \| \| \mathbf{y} \|} \tag{1.5}$$

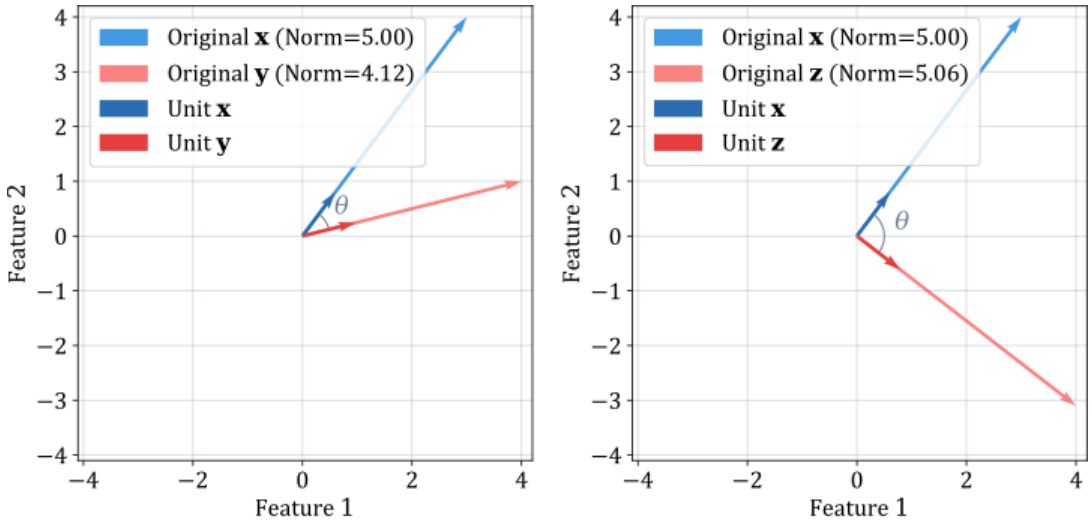
The cosine of the angle between two vectors quantifies their similarity. For instance, two houses with similar areas and bedroom counts will have a cosine similarity close to 1, otherwise the value will be lower. **Cosine similarity** is widely used to compare words or documents represented as **embedding vectors**. This will be discussed further in Section 2.2.

A **zero vector** has all components equal to zero. A **unit vector** has a length of 1. To convert any non-zero vector **x** into a unit vector $\hat{\mathbf{x}}$, you divide the vector by its norm:

$$\hat{\mathbf{x}} = \frac{\mathbf{x}}{\| \mathbf{x} \|}$$

Dividing a vector by a number results in a new vector where each component of the original vector is divided by that number.

A unit vector preserves the direction of the original vector but has a length of 1. The figure below demonstrates this with 2-dimensional examples. On the left, aligned vectors have $\cos(\theta) = 0.78$. On the right, nearly orthogonal vectors have $\cos(\theta) = -0.02$.



> Unit vectors are valuable because their dot product equals the cosine of the angle between them, and computing dot products is efficient. When documents are represented as unit vectors, finding similar ones becomes fast by calculating the dot product between the query vector and document vectors. This is how vector search engines and libraries like Milvus, Qdrant, and Weaviate operate.

As dimensions increase, the number of parameters in a linear model becomes too large to solve manually. Furthermore, in high-dimensional spaces, we cannot visually verify if data follows a linear pattern. Even if we could visualize beyond three dimensions, we would still need more flexible models to handle data that linear models cannot fit.

The next section covers non-linear models, focusing on neural networks. These are key to understanding large language models, a specific type of neural network architecture.

1.5. Neural Network

A **neural network** differs from a linear model in two key ways: (1) it applies fixed non-linear functions to the outputs of trainable linear functions, and (2) its structure is deeper, combining multiple functions hierarchically through layers. Let's illustrate these differences.

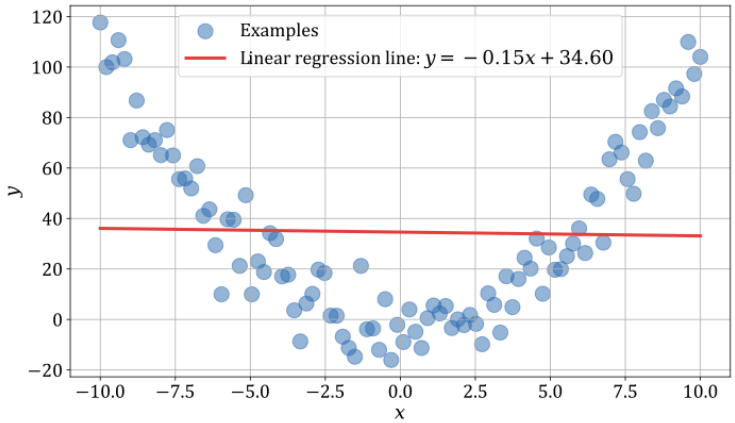
Linear models like $wx + b$ or $\mathbf{w} \cdot \mathbf{x} + b$ cannot solve many machine learning problems effectively. Even if we combine them into a **composite function** $f_2(f_1(x))$, a composite function of linear functions remains linear. This is straightforward to verify.

Let's define $y_1 = f_1(x) \stackrel{\text{def}}{=} a_1 x$ and $y_2 = f_2(y_1) \stackrel{\text{def}}{=} a_2 y_1$. Here, f_2 depends on f_1, making it a composite function. We can rewrite f_2 as:

$$y_2 = a_2 y_1 = a_2(a_1 x) = (a_2 a_1)x$$

Since a_1 and a_2 are constants, we can define $a_3 \stackrel{\text{def}}{=} a_1 a_2$, so $y_2 = a_3 x$, which is linear.

A straight line often fails to capture patterns in one-dimensional data, as demonstrated when **linear regression** is applied to non-linear data:



To address this, we add non-linearity. For a one-dimensional input, the model becomes:

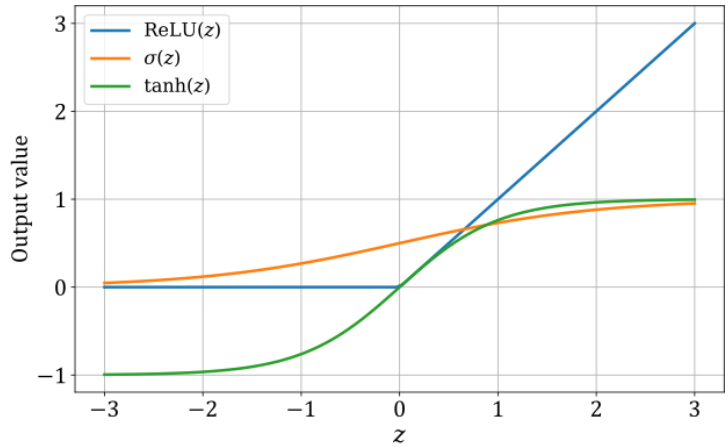
$$y = \phi(wx + b)$$

The function ϕ is a fixed non-linear function, known as an **activation**. Common choices are:

1) **ReLU** (**rectified linear unit**): $\mathrm{ReLU}(z) \stackrel{\text{def}}{=} \max(0, z)$, which outputs non-negative values and is widely used in neural networks;
2) **Sigmoid**: $\sigma(z) \stackrel{\text{def}}{=} \frac{1}{1+e^{-z}}$, which outputs values between 0 and 1, making it suitable for **binary classification** (e.g., classifying spam emails as 1 and non-spam as 0);
3) **Tanh** (**hyperbolic tangent**): $\tanh(z) \stackrel{\text{def}}{=} \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$; outputs values between -1 and 1.

In these equations, e denotes **Euler's number**, approximately 2.72.

These functions are widely used due to their mathematical properties, simplicity, and effectiveness in diverse applications. This is what they look like:

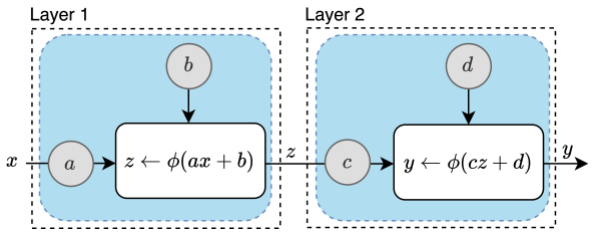


The structure $\phi(wx + b)$ enables learning non-linear models but can't capture all non-linear curves. By nesting these functions, we build more expressive models. For instance, let $f_1(x) \stackrel{\text{def}}{=} \phi(ax + b)$ and $f_2(z) \stackrel{\text{def}}{=} \phi(cz + d)$. A **composite model** combining f_1 and f_2 is:

$$y = f_2\big(f_1(x)\big) = \phi(c\phi(ax + b) + d)$$

Here, the input x is first transformed linearly using parameters a and b, then passed through the non-linear function ϕ. The result is further transformed linearly with parameters c and d, followed by another application of ϕ.

Below is the graph representation of the composite model $y = f_2\big(f_1(x)\big)$:



A **computational graph** represents the structure of a model. The computational graph above shows two non-linear **units** (blue rectangles), often referred to as **artificial neurons**. Each unit contains two trainable parameters—a weight and a bias—represented by grey circles. The left arrow ← denotes that the value on the right is assigned to the variable on the left. This graph illustrates a basic neural network with two **layers**, each containing one unit. Most neural networks in practice are built with more layers and multiple units per layer.

Suppose we have a two-dimensional input, an **input layer** with three units, and an **output layer** with a single unit. The computational graph appears as follows:

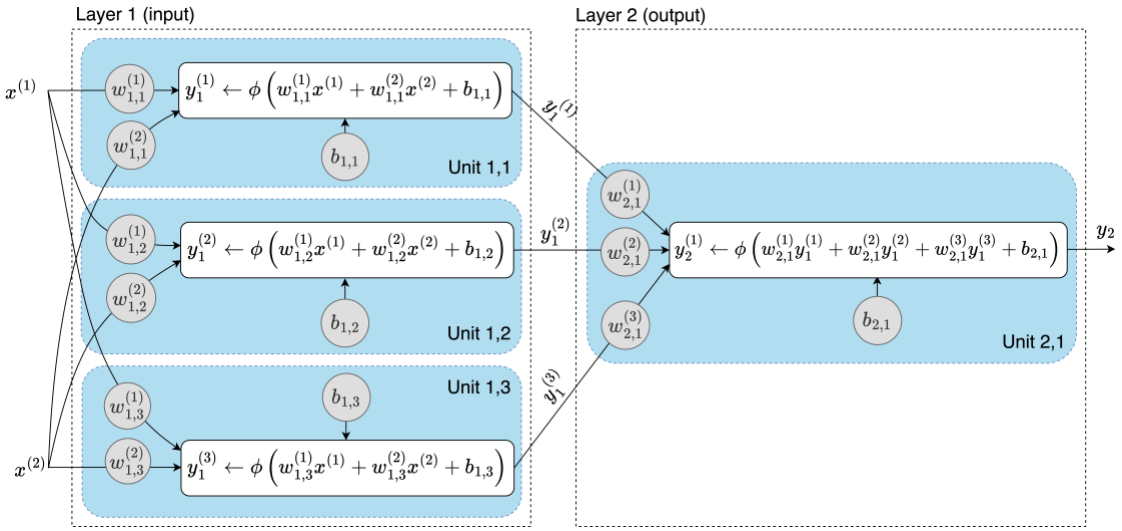


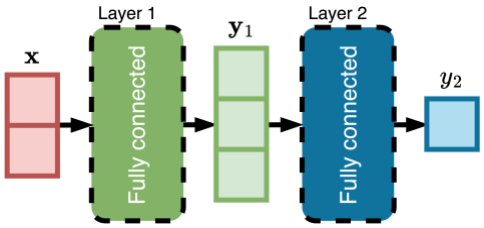
Figure 1.1: A neural network with two layers.

This structure represents a **feedforward neural network** (FNN), where information flows in one direction—left to right—without loops. When units in each layer connect to all units in the subsequent layer, as shown above, we call it a **multilayer perceptron** (MLP). A layer where each unit connects to all units in both adjacent layers is termed a **fully connected layer**, or **dense layer**.

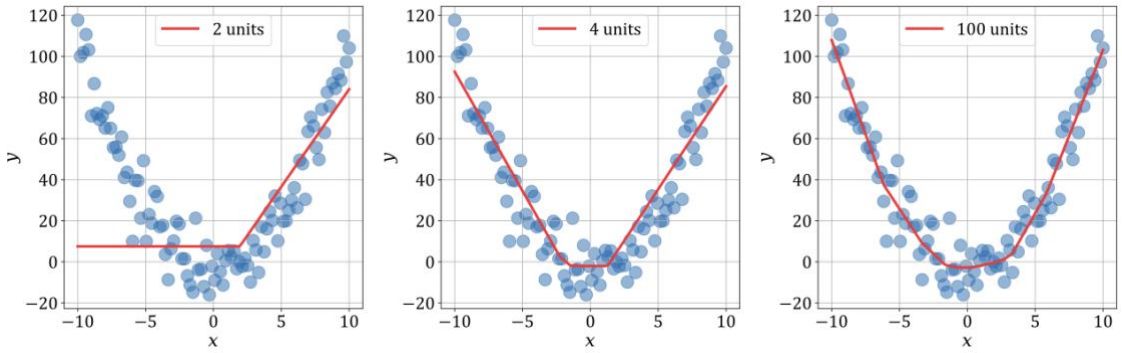
In Chapter 3, we will explore recurrent neural networks (RNNs). Unlike FNNs, RNNs have loops, where outputs from a layer are used as inputs to the same layer.

> **Convolutional neural networks** (**CNN**) are feedforward neural networks with convolutional layers that are not fully connected. While initially designed for image processing, they are effective for tasks like document classification in text data. To learn more about CNNs refer to the additional materials in the book's wiki.

To simplify diagrams, individual neural units can be replaced with squares. Using this approach, the above network can be represented more compactly as follows:



If you think this simple model is too weak, look at the figure below. It contains three plots demonstrating how increasing model size improves performance. The left plot shows a model with 2 units: one input, one output, and ReLU activations. The middle plot is a model with 4 units: three inputs and one output. The right plot shows a much larger model with 100 units:



> The ReLU activation function, despite its simplicity, was a breakthrough in machine learning. Neural networks before 2012 relied on smooth activations like tanh and sigmoid, which made training deep models increasingly difficult. We will return to this subject in Chapter 4 on the Transformer neural network architecture.

Increasing the number of parameters helps the model approximate the data more accurately. Experiments consistently show that adding more units per layer or increasing the number of layers in a neural network improves its capacity to fit high-dimensional datasets, such as natural language, voice, sound, image, and video data.

1.6. Matrix

Neural networks can handle high-dimensional datasets but require substantial memory and computation. Calculating a layer's transformation naïvely would involve iterating over thousands of parameters per unit across thousands of units and dozens of layers, which is both slow and resource-intensive. Using **matrices** makes the computations more efficient.

A **matrix** is a two-dimensional array of numbers arranged into rows and columns, which generalizes the concept of vectors to higher dimensionalities. Formally, a matrix $\mathbf{A}$ with m rows and n columns is written as:

$$\mathbf{A} \stackrel{\text{def}}{=} \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

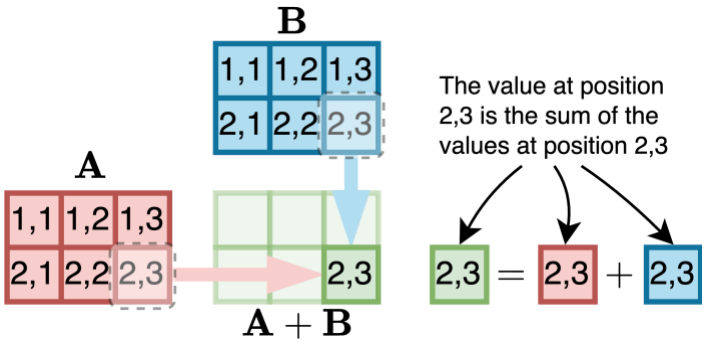
Here, $a_{i,j}$ represents the element in the i-th row and j-th column of the matrix. The dimensions of the matrix are expressed as $m \times n$ (read as "m by n").

Matrices are fundamental in machine learning. They compactly represent data and weights and enable efficient computation through operations such as addition, multiplication, and transposition. In this book, matrices are represented with uppercase bold letters, such as $\mathbf{X}$ or $\mathbf{W}$.

The **sum of two matrices** $\mathbf{A}$ and $\mathbf{B}$ of the same dimensionality is defined element-wise as:

$$(\mathbf{A} + \mathbf{B})_{i,j} \stackrel{\text{def}}{=} a_{i,j} + b_{i,j}$$

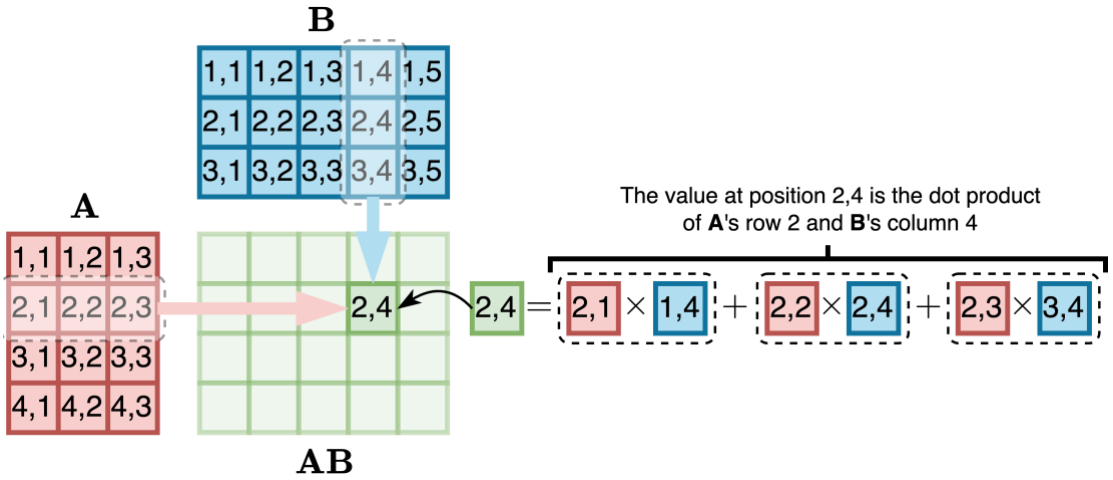
For example, for two 2×3 matrices $\mathbf{A}$ and $\mathbf{B}$, the addition works like this:



The **product of a matrices** $\mathbf{A}$ with dimensions $m \times n$ and $\mathbf{B}$ with dimensions $n \times p$ is a matrix $\mathbf{C}$ with dimensions $m \times p$ such that the value in row i and column k is given by:

$$(\mathbf{C})_{i,k} = \sum_{j=1}^{n} a_{i,j}\, b_{j,k}$$

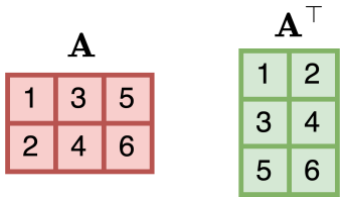
For example, for a 4×3 matrix $\mathbf{A}$ and a 3×5 matrix $\mathbf{B}$, the product is a 4×5 matrix:



Transposing a matrix A swaps its rows and columns, resulting in $\mathbf{A}^\top$, where:

$$(\mathbf{A}^\top)_{i,j} = a_{j,i}$$

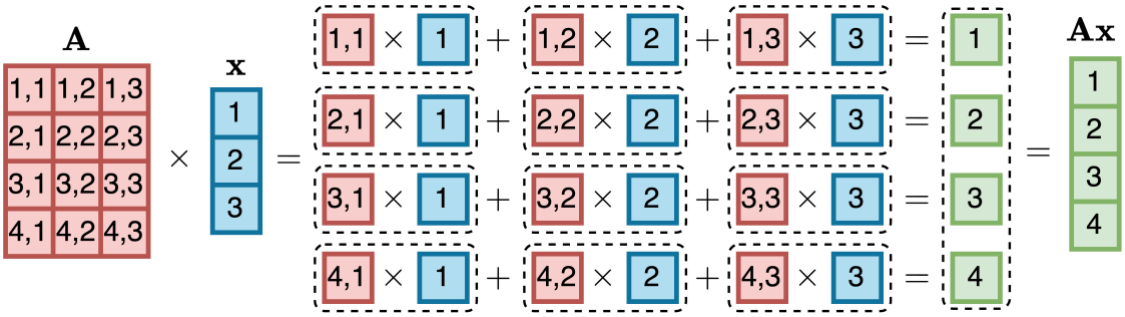
For example, for a 2×3 matrix **A**, its transpose $\mathbf{A}^\top$ look like this:



Matrix-vector multiplication is a special case of matrix multiplication. When an $m \times n$ matrix **A** is multiplied by a vector **x** of size n, the result is a vector $\mathbf{y} = \mathbf{Ax}$ with m components. Each element y_i of the resulting vector **y** is computed as:

$$y_i = \sum_{j=1}^{n} a_{i,j}\, x^{(j)}$$

For example, a 4×3 matrix **A** multiplied by a 3D vector **x** produces a 4-dimensional vector:



The weights and biases in fully connected layers of neural networks can be compactly represented using matrices and vectors, enabling the use of highly optimized linear algebra libraries. As a result, matrix operations form the backbone of neural network training and inference.

Let's express the model in Figure 1.1 using matrix notation. Let $\mathbf{x}$ be the 2D input feature vector. For the first layer, the weights and biases are represented as a 3×2 matrix $\mathbf{W}_1$ and a 3D vector $\mathbf{b}_1$, respectively. The 3D output $\mathbf{y}_1$ of the first layer is given by:

$$\mathbf{y}_1 = \phi(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{1.6}$$

The second layer also uses a weight matrix and a bias. The output y_2 of the second layer is computed using the output $\mathbf{y}_1$ from the first layer. The weight matrix for the second layer is a 1×3 matrix $\mathbf{W}_2$. The bias for the second layer is a scalar $b_{2,1}$. The model output corresponds to the output of the second layer:

$$y_2 = \phi\left(\mathbf{W}_2\mathbf{y}_1 + b_{2,1}\right) \tag{1.7}$$

Equation 1.6 and Equation 1.7 capture the operations from input to output in the neural network, with each layer's output serving as the input for the next.

1.7. Gradient Descent

Neural networks are typically large and composed of non-linear functions, which makes solving for the minimum of the loss function analytically infeasible. Instead, the gradient descent algorithm is widely used to minimize the loss, including in large language models.

Consider a practical example: **binary classification**. This task assigns input data to one of two classes, like deciding if an email is spam or not, or detecting whether a website connection request is a DDoS attack.

Our training dataset $\mathcal{D}$ is $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $\mathbf{x}_i$ are vectors of input features, and y_i are the labels. Each y_i, indexed from 1 to N, takes a value of 0 for "not spam" or 1 for "spam." A well-trained model should output $\tilde{y}$ close to 1 for spam inputs $\mathbf{x}$ and close to 0 for non-spam inputs. We can define the model as follows:

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b), \tag{1.8}$$

where $\mathbf{x} = \left[x^{(j)}\right]_{j=1}^D$ and $\mathbf{w} = \left[w^{(j)}\right]_{j=1}^D$ are D-dimensional vectors, b is a scalar, and σ is the **sigmoid** defined in Section 1.5.

This model, called **logistic regression**, is commonly used for binary classification tasks. Unlike **linear regression**, which produces outputs ranging from $-\infty$ to ∞, logistic regression always outputs values between 0 and 1. It can serve either as a standalone model or as the output layer in a larger neural network.

> Despite being over 80 years old, logistic regression remains one of the most widely used algorithms in production machine learning systems.

A common choice for the **loss function** in this case is **binary cross-entropy**, also called **logistic loss**. For a single example i, the binary cross-entropy loss is defined as:

$$\text{loss}(\tilde{y}_i, y_i) \stackrel{\text{def}}{=} -\left[y_i \log(\tilde{y}_i) + (1 - y_i)\log(1 - \tilde{y}_i)\right] \tag{1.9}$$

In this equation, y_i represents the actual label of the i-th example in the dataset, and $\tilde{y}_i$ is the **prediction score**, a value between 0 and 1 that the model outputs for input vector $\mathbf{x}_i$. The function log denotes the **natural logarithm**.

Loss functions are usually designed to penalize incorrect predictions while rewarding accurate ones. To see why logistic loss works for logistic regression, consider two extreme cases:

1. **Perfect prediction**, when $y_i = 0$ and $\tilde{y}_i = 0$:

$$\text{loss}(0,0) = -[0 \cdot \log(0) + (1-0) \cdot \log(1-0)] = -\log(1) = 0$$

   Here, the loss is zero which is good because the prediction matches the label.

2. **Opposite prediction**, when $y_i = 0$ and $\tilde{y}_i = 1$:

$$\text{loss}(1,0) = -[0 \cdot \log(1) + (1-0) \cdot \log(1-1)] = -\log(0)$$

The logarithm of 0 is undefined, and as a approaches 0, $-\log(a)$ approaches infinity, representing a severe loss for completely wrong predictions. However, since $\tilde{y}_i$, the sigmoid's output, always remains strictly between 0 and 1, without reaching them, the loss stays finite.

For an entire dataset $\mathcal{D}$, the loss is given by the average loss for all examples in the dataset:

$$\text{loss}_{\mathcal{D}} \overset{\text{def}}{=} -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\tilde{y}_i) + (1-y_i)\log(1-\tilde{y}_i)] \tag{1.10}$$

To simplify the gradient descent derivation, we'll stick to a single example, i, and rewrite the equation by substituting the prediction score $\tilde{y}_i$ with the model's expression for it:

$$\text{loss}(\tilde{y}_i, y_i) = -\big[y_i \log\big(\sigma(z_i)\big) + (1-y_i)\log\big(1-\sigma(z_i)\big)\big], \text{where } z_i = \mathbf{w} \cdot \mathbf{x}_i + b$$

To minimize $\text{loss}(\tilde{y}_i, y_i)$, we calculate the partial derivatives with respect to each weight $w^{(j)}$ and the bias b. We will use the **chain rule** because we have a **composition** of three functions:

- **Function 1**: $z_i \overset{\text{def}}{=} \mathbf{w} \cdot \mathbf{x}_i + b$, a linear function with the weights $\mathbf{w}$ and the bias b;
- **Function 2**: $\tilde{y}_i = \sigma(z_i) \overset{\text{def}}{=} \frac{1}{1+e^{-z_i}}$, the sigmoid function applied to z_i;
- **Function 3**: $\text{loss}(\tilde{y}_i, y_i)$, as defined in Equation 1.9, which depends on $\tilde{y}_i$.

> Notice that $\mathbf{x}_i$ and y_i are given: $\mathbf{x}_i$ is the feature vector for example i, and $y_i \in \{0,1\}$ is its label. The notation $y_i \in \{0,1\}$ means that y_i belongs to the set $\{0,1\}$ and, in this case, indicates that y_i can only be 0 or 1.

Let's denote $\text{loss}(\tilde{y}_i, y_i)$ as l_i. For weights $w^{(j)}$, the application of the chain rule gives us:

$$\frac{\partial \text{l}_i}{\partial w^{(j)}} = \frac{\partial \text{l}_i}{\partial \tilde{y}_i} \cdot \frac{\partial \tilde{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial w^{(j)}} = (\tilde{y}_i - y_i) \cdot x_i^{(j)}$$

For the bias b, we have:

$$\frac{\partial \text{l}_i}{\partial b} = \frac{\partial \text{l}_i}{\partial \tilde{y}_i} \cdot \frac{\partial \tilde{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial b} = \tilde{y}_i - y_i$$

> This is where the beauty of machine learning math shines: the activation function—sigmoid—and loss function—cross-entropy—both arise from e, Euler's number. Their functional properties serve distinct purposes: sigmoid ranges between 0 and 1, ideal for binary classification, while cross-entropy spans from 0 to ∞, great as a penalty. When combined, the exponential and logarithmic components elegantly cancel, yielding a linear function—prized for its computational simplicity and numerical stability. The book's wiki provides the full derivation.

The partial derivatives with respect to $w^{(j)}$ and b for a single example $(\mathbf{x}_i, y_i)$ can be extended to the entire dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$ by averaging the contributions from all examples. This follows from the **sum rule** and the **constant multiple rule** of differentiation:

$$\begin{aligned} \frac{\partial \text{loss}}{\partial w^{(j)}} &= \frac{1}{N}\sum_{i=1}^{N}\left[(\tilde{y}_i - y_i)\cdot x_i^{(j)}\right] \\ \frac{\partial \text{loss}}{\partial b} &= \frac{1}{N}\sum_{i=1}^{N}\left[\tilde{y}_i - y_i\right] \end{aligned} \tag{1.11}$$

Here, loss denotes the average loss for the entire dataset. Averaging the losses for individual examples ensures that each example contributes equally to the overall loss, regardless of the total number of examples.

The **gradient** is a vector that contains all the partial derivatives. The gradient of the loss function, denoted as ∇loss, is defined as follows:

$$\nabla\text{loss} \stackrel{\text{def}}{=} \left(\frac{\partial \text{loss}}{\partial w^{(1)}}, \frac{\partial \text{loss}}{\partial w^{(2)}}, \ldots, \frac{\partial \text{loss}}{\partial w^{(D)}}, \frac{\partial \text{loss}}{\partial b}\right)$$

If a gradient's component is positive, this means that increasing the corresponding parameter will increase the loss. Therefore, to minimize the loss, we should decrease that parameter.

The **gradient descent** algorithm uses the gradient of the loss function to iteratively update the weights and bias, aiming to minimize the loss function. Here's how it operates:

0. **Initialize parameters**: Start with random values of parameters $w^{(j)}$ and b.
1. **Compute the predictions**: For each training example $(\mathbf{x}_i, y_i)$, compute the predicted value $\tilde{y}_i$ using the model:
$$\tilde{y}_i \leftarrow \sigma(\mathbf{w}\cdot\mathbf{x}_i + b)$$
2. **Compute the gradient**: Calculate the partial derivatives of the loss function with respect to each weight $w^{(j)}$ and the bias b using Equation 1.11.
3. **Update the weights and bias**: Adjust the weights and bias in the direction that decreases the loss function. This adjustment involves taking a small step in the opposite direction of the gradient. The step size is controlled by the learning rate η (explained below):
$$w^{(j)} \leftarrow w^{(j)} - \eta\frac{\partial \text{loss}}{\partial w^{(j)}}$$
$$b \leftarrow b - \eta\frac{\partial \text{loss}}{\partial b}$$
4. **Calculate the loss**: Calculate the logistic loss by substituting the updated values of $w^{(j)}$ and b into Equation 1.10.

5. **Continue the iterative process**: Repeat steps 1-4 for a set number of **iterations** (also called **steps**) or until the loss value converges to a minimum.

Here's a bit more detail to clarify the steps:

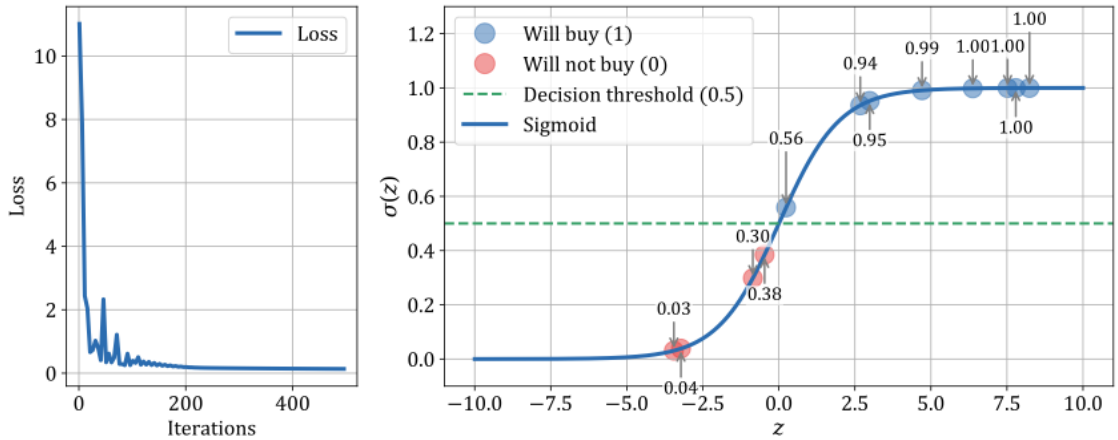
- Gradients are subtracted from parameters because they point in the direction of steepest ascent in the loss function. Since our goal is to minimize loss, we move in the opposite direction—hence, the subtraction.
- The **learning rate** η is a positive value close to 0 and serves as a **hyperparameter**—not learned by the model but set manually. It controls the step size of each update, and finding its optimal value requires experimentation.
- **Convergence** occurs when subsequent iterations yield minimal decreases in loss. The learning rate η is crucial here: too small, and progress crawls; too large, and we risk overshooting the minimum or even seeing the loss increase. Choosing an appropriate η is therefore essential for effective gradient descent.

Let's illustrate the process with a simple dataset of 12 examples:

$$\{((22,25),0),((25,35),0),((47,80),1),((52,95),1),((46,82),1),((56,90),1),\\((23,27),0),((30,50),1),((40,60),1),((39,57),0),((53,95),1),((48,88),1)\}$$

In this dataset, $\mathbf{x}_i$ contains two features: age (in years) and income (in thousands of dollars). The objective is to predict whether a person will buy a product, with label y_i being either 0 (will not buy) or 1 (will buy).

The loss evolution across gradient descent steps and the resulting trained model are shown in the figures below:



The left plot shows the loss decreasing steadily during gradient descent optimization. The right plot displays the trained model's sigmoid function, with training examples positioned by their z-values ($z_i = \mathbf{w}^* \cdot \mathbf{x}_i + b^*$), where $\mathbf{w}^*$ and b^* are the learned weights and bias.

The 0.5 threshold was chosen based on the plot's clear separation: all "will-buy" examples (blue dots) lie above it, while all "will-not-buy" examples (red dots) fall below. For new inputs $\mathbf{x}$, generate $\tilde{y} = \sigma(\mathbf{w}^* \cdot \mathbf{x} + b^*)$. If $\tilde{y} < 0.5$, predict "will not buy;" otherwise, predict "will buy."

1.8. Automatic Differentiation

Gradient descent optimizes model parameters but requires partial derivative equations. Until now, we calculated these derivatives by hand for each model. As models grow more complex, particularly in neural networks with multiple layers, manual derivation becomes impractical.

This is where **automatic differentiation** (or **autograd**) comes in. Built into machine learning frameworks like PyTorch and TensorFlow, this feature computes partial derivatives directly from Python code defining the model. This eliminates manual derivation, even for very complex models.

> Modern automatic differentiation systems can handle derivatives for millions of variables efficiently. Manual computation of these derivatives would be unfeasible—writing the equations alone could take years.

To use gradient descent in PyTorch, first install it with `pip3` like this:

```
$ pip3 install torch
```

Now that PyTorch is installed, let's import the dependencies:

```
import torch
import torch.nn as nn
import torch.optim as optim
```

The `torch.nn` module contains building blocks for creating models. When you use these components, PyTorch automatically handles derivative calculations. For optimization algorithms like gradient descent, the `torch.optim` module has what you need. Here's how to implement logistic regression in PyTorch:

```
model = nn.Sequential(
    nn.Linear(n_inputs, n_outputs), ❶
    nn.Sigmoid() ❷
)
```

Our model leverages PyTorch's **sequential API**, which is well-suited for simple feedforward neural networks where data flows sequentially through layers. Each layer's output naturally becomes the input for the subsequent layer. The more versatile **module API**, which we'll use in the next chapter, enables the creation of models with multiple inputs, outputs, or loops.

The input layer, defined in line ❶ using `nn.Linear`, has input dimensionality `n_inputs` matching the size of our feature vector $\mathbf{x}$, while the output dimensionality `n_outputs` determines the layer's unit count. For our buy/no-buy **classifier**—a model assigning classes to inputs—we set `n_inputs` to 2 since $\mathbf{x} = \left[x^{(1)}, x^{(2)}\right]^{\mathsf{T}}$. With the output z being scalar, `n_outputs` becomes 1. Line ❷ transforms z through the sigmoid function to produce the output score.

We then proceed to define our dataset, create the model instance, establish the binary cross-entropy loss function, and set up the gradient descent algorithm:

```
inputs = torch.tensor([
    [22, 25], [25, 35], [47, 80], [52, 95], [46, 82], [56, 90],
    [23, 27], [30, 50], [40, 60], [39, 57], [53, 95], [48, 88]
```

```
], dtype=torch.float32) ❶

labels = torch.tensor([
    [0], [0], [1], [1], [1], [1], [0], [1], [1], [0], [1], [1]
], dtype=torch.float32) ❷

model = nn.Sequential(
    nn.Linear(inputs.shape[1], 1),
    nn.Sigmoid()
)
optimizer = optim.SGD(model.parameters(), lr=0.001) ❸
criterion = nn.BCELoss() # binary cross-entropy loss
```

In the above code block, we defined `inputs` and `labels`. The `inputs` form a matrix with 12 rows and 2 columns, while the `labels` are a vector with 12 components. The `shape` attribute of the `inputs` tensor return its dimensionality:

```
>>> inputs.shape
torch.Size([12, 2])
```

Tensors are PyTorch's core data structures—multi-dimensional arrays optimized for computation on both CPU and GPU. Supporting automatic differentiation and flexible data reshaping, tensors form the foundation for neural network operations. In our example, the `inputs` tensor contains 12 examples with 2 features each, while the `labels` tensor holds 12 examples with single labels. Following standard convention, examples are arranged in rows and their features in columns.

> If you're not familiar with tensors, there's an introductory chapter on tensors available on the book's wiki.

When creating tensors in PyTorch, specifying `dtype=torch.float32` in line ❶ sets 32-bit floating-point precision explicitly. This precision setting is essential for neural network computations, including weight adjustments, activation functions, and gradient calculations.

> The 32-bit floating-point precision is not the only option for neural networks. **Quantization**, an advanced technique that uses lower-precision data types like 16-bit or 8-bit floats and integers, helps reduce model size and improve computational efficiency. For more information, refer to resources on model optimization and deployment available on the book's wiki.

The `optim.SGD` class in line ❸ implements gradient descent by taking a list of model parameters and learning rate as inputs.[2] Since our model inherits from `nn.Module`, we can access all trainable parameters through its `parameters` method.

PyTorch provides the **binary cross-entropy** loss function through `nn.BCELoss()`.

[2] While 0.001 is a common default learning rate, optimal values vary by problem and dataset. Finding the best rate involves systematically testing different values and comparing model performance.

Now, we have everything we need to start the training loop:

```
for step in range(500):
    optimizer.zero_grad() ❶
    loss = criterion(model(inputs), labels) ❷
    loss.backward() ❸
    optimizer.step() ❹
```

Line ❷ calculates the binary cross-entropy loss (Equation 1.10) by evaluating model predictions against training labels. Line ❸ then uses backpropagation to compute the gradient of this loss with respect to the model parameters.

Backpropagation applies differentiation rules, particularly the chain rule, to compute gradients through deep composite functions. This algorithm forms the backbone of neural network training. When PyTorch operates on tensors, it builds a computational graph as shown in Figure 1.1 from Section 1.5. This graph tracks all operations performed on the tensors. The `loss.backward()` call prompts PyTorch to traverse this graph and compute gradients via the chain rule, eliminating the need for manual gradient derivation and implementation.

The flow of data from input to output through the computational graph constitutes the **forward pass**, while the computation of gradients from output to input through backpropagation represents the **backward pass**.

> PyTorch accumulates gradients in the `.grad` attribute of parameters like weights and biases. While this feature enables multiple gradient computations before parameter updates—useful for recurrent neural networks (covered in Section 3)—our implementation doesn't require gradient accumulation. Line ❶ therefore clears the gradients at each step's beginning.

Finally, in line ❹, parameter values are updated by subtracting the product of the learning rate and the loss function's partial derivatives, completing step 3 of the gradient descent algorithm discussed earlier.

> The reader might wonder why labels are floats and not integers in this binary classification problem. The reason lies in how PyTorch's `BCELoss` function operates. Since the model's output layer uses a sigmoid activation function that produces floating-point values between 0 and 1, `BCELoss` expects both predictions and target labels to be floating-point numbers in the same range. If we were to use integer types like `torch.long`, we would encounter an error because `BCELoss` isn't designed to handle integer types and its internal calculations expect floating-point numbers. This is specific to `BCELoss`—other loss functions like `CrossEntropyLoss` that we will use later actually require integer labels instead.

One of automatic differentiation's key advantages is its flexibility with model switching—as long as you're using PyTorch's components, you can readily swap between different architectures. For instance, you could replace logistic regression with a basic two-layer FNN, defined through the sequential API:

```
model = nn.Sequential(
    nn.Linear(features.shape[1], 100),
    nn.Sigmoid(),
    nn.Linear(100, labels.shape[1]),
    nn.Sigmoid()
)
```

In this setup, each of the 100 units in the first layer contains 2 weights and 1 bias, while the output layer's single unit has 100 weights and 1 bias. The automatic differentiation system handles gradient computation internally, so the remaining code stays unchanged.

In the next chapter, we examine representing and processing text data. We start with basic methods like bag-of-words and word embeddings for converting documents into numerical formats, then introduce count-based language modeling.