Andriy Burkov

THE HUNDRED-PAGE LANGUAGE MODELS BOOK

hands-on with PyTorch

"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

Foreword

First time I got involved in language modeling was already two decades ago. I wanted to improve some of my data compression algorithms and found out about the n-gram statistics. Very simple concept, but so hard to beat! Then I quickly gained another motivation—since my childhood, I was interested in artificial intelligence. I had a vision of machines that would understand patterns in our world that are hidden from our limited minds. It would be so exciting to talk with such super-intelligence. And I realized that language modeling could be a way towards such AI.

I started searching for others sharing this vision and did find the works of Solomonoff, Schmidhuber and the Hutter prize competition organized by Matt Mahoney. They all did write about AI completeness of language modeling and I knew I had to try to make it work. But the world was very different than it is today. Language modeling was considered a dead research direction, and I've heard countless times that I should give up as nothing will ever beat n-grams on large data.

I've completed my master's thesis on neural language models, as these models were quite like what I previously developed for data compression, and I did believe the distributed representations that could be applied to any language is the right way to go. This infuriated a local linguist who declared my ideas to be a total nonsense as language modeling has to be addressed from the linguistics point of view, and each language had to be treated differently.

However, I did not give up and did continue working on my vision of AI-complete language models. Just the summer before starting my PhD, I did come up with the idea to generate text from these neural models. I was amazed by how much better this text was than text generated from n-grams models. That was summer 2007 and I quickly realized the only person excited about this at the Brno University of Technology was actually me. But I did not give up anyways.

In the following years, I did develop a number of algorithms to make neural language models more useful. To convince others about their qualities, I published open-source toolkit RNNLM in 2010. It had the first implementations ever of neural text generation, gradient clipping, dynamic evaluation, model adaptation (nowadays called fine-tuning) and other tricks such as hierarchical softmax or splitting infrequent words into subword units. However, the result

I was the most proud of was when I could demonstrate in my PhD thesis that neural language models not only beat n-grams on large datasets—something widely considered to be impossible at the time—but the improvements were actually increasing with the amount of training data. This happened for the first time after something like fifty years of language modeling research and I still remember the disbelief in faces of famous researchers when I showed them my work.

Fast forward some fifteen years, and I'm amazed by how much the world has changed. The mindset completely flipped—what used to be some obscure technology in a dead research direction is now thriving and gets the attention of CEOs of the largest companies in the world. Language models are everywhere today. With all this hype, I think it is needed more than ever to actually understand this technology.

Young students who want to learn about language modeling are flooded with information. Thus, I was delighted when I learned about Andriy's project to write a short book with only one hundred pages that would cover some of the most important ideas. I think the book is a good start for anyone new to language modeling who aspires to improve on state of the art—and if someone tells you that everything that could have been invented in language modeling has already been discovered, don't believe it.

> **Tomáš Mikolov**, Senior Researcher at Czech Institute of Informatics, Robotics and Cybernetics, the author of **word2vec** and **FastText**

Preface

My interest in text began in the late 1990s during my teenage years, building dynamic websites using Perl and HTML. This early experience with coding and organizing text into structured formats sparked my fascination with how text could be processed and transformed. Over the years, I advanced to building web scrapers and text aggregators, developing systems to extract structured data from webpages. The challenge of processing and understanding text led me to explore more complex applications, including designing chatbots that could understand and address user needs.

The challenge of extracting meaning from words intrigued me. The complexity of the task only fueled my determination to "crack" it, using every tool at my disposal—ranging from regular expressions and scripting languages to text classifiers and named entity recognition models.

The rise of large language models (LLMs) transformed everything. For the first time, computers could converse with us fluently and follow verbal instructions with remarkable precision. However, like any tool, their immense power comes with limitations. Some are easy to spot, but others are more subtle, requiring deep expertise to handle properly. Attempting to build a skyscraper without fully understanding your tools will only result in a pile of concrete and steel. The same holds true for language models. Approaching large-scale text processing tasks or creating reliable products for paying users requires precision and knowledge—guesswork simply isn't an option.

Who This Book Is For

I wrote this book for those who, like me, are captivated by the challenge of understanding language through machines. Language models are, at their core, just mathematical functions. However, their true potential isn't fully appreciated in theory—you need to implement them to see their power and how their abilities grow as they scale. This is why I decided to make this book hands-on.

This book serves software developers, data scientists, machine learning engineers, and anyone curious about language models. Whether your goal is to integrate existing models into applications or to train your own, you'll find practical guidance alongside theoretical foundations.

Given its hundred-page format, the book makes certain assumptions about readers. You should have programming experience, as all hands-on examples use Python.

While familiarity with PyTorch and tensors—PyTorch's fundamental data types—is beneficial, it's not mandatory. If you're new to these tools, the book's wiki (thelmbook.com/wiki) provides a concise introduction with examples and resource links for further learning. This wiki format ensures content remains current and addresses reader questions beyond publication.

College-level math knowledge helps, but you needn't remember every detail or have machine learning experience. The book introduces concepts systematically, beginning with notations, definitions, and fundamental vector and matrix operations. From there, it progresses through simple neural networks to more advanced topics. Mathematical concepts are presented intuitively, with clear diagrams and examples that facilitate understanding.

What This Book Is Not

This book is focused on understanding and implementing language models. It will *not* cover:

- **Large-scale training**: This book won't teach you how to train massive models on distributed systems or how to manage training infrastructure.
- **Production deployment**: Topics like model serving, API development, scaling for high traffic, monitoring, and cost optimization are not covered. The code examples focus on understanding the concepts rather than production readiness.
- **Enterprise applications**: This book won't guide you through building commercial LLM applications, handling user data, or integrating with existing systems.

If you're interested in learning the mathematical foundations of language models, understanding how they work, implementing core components yourself, or learning to work effectively with LLMs, this book is for you. But if you're primarily looking to deploy models in production or build scalable applications, you may want to supplement this book with other resources.

Book Structure

To make this book engaging and to deepen the reader's understanding, I decided to discuss language modeling as a whole, including approaches that are often overlooked in modern literature. While Transformer-based LLMs dominate the spotlight, earlier approaches like count-based methods and recurrent neural networks (RNNs) remain effective for some tasks.

Learning the math of the Transformer architecture from scratch may seem overwhelming for someone starting from scratch. By revisiting these foundational methods, my goal is to gradually build up the reader's intuition and mathematical understanding, making the transition to modern Transformer architectures feel like a natural progression rather than an intimidating leap.

The book is divided into six chapters, progressing from fundamentals to advanced topics:

- **Chapter 1** covers machine learning basics, including key concepts like AI, models, neural networks, and gradient descent. Even if you're familiar with these topics, the chapter provides important foundations for understanding language models.
- **Chapter 2** introduces language modeling fundamentals, exploring text representation methods like bag of words and word embeddings, as well as count-based language models and evaluation techniques.
- **Chapter 3** focuses on recurrent neural networks, covering their implementation, training, and application as language models.
- **Chapter 4** provides a detailed exploration of the Transformer architecture, including key components like self-attention, position embeddings, and practical implementation.
- **Chapter 5** examines large language models (LLMs), discussing why scale matters, finetuning techniques, practical applications, and important considerations around hallucinations, copyright, and ethics.
- **Chapter 6** concludes with further reading on advanced topics like mixture of experts, model compression, preference-based alignment, and vision language models, providing direction for continued learning.

Most chapters contain working code examples you can run and modify. While only essential code appears in the book, complete code is available as Jupyter notebooks on the book's website, with notebooks referenced in relevant

sections. All code in notebooks remains compatible with the latest stable versions of Python, PyTorch, and other libraries.

The notebooks run on Google Colab, which at the time of writing offers free access to computing resources including GPUs and TPUs. These resources, though, aren't guaranteed and have usage limits that may vary. Some examples might require extended GPU access, potentially involving wait times for availability. If the free tier proves limiting, Colab's pay-as-you-go option lets you purchase compute credits for reliable GPU access. While these credits are relatively affordable by North American standards, costs may be significant depending on your location.

For those familiar with the Linux command line, GPU cloud services provide another option through pay-per-time virtual machines with one or more GPUs. The book's wiki maintains current information on free and paid notebook or GPU rental services.

`Verbatim` terms and blocks indicate code, code fragments, or code execution outputs. **Bold** terms link to the book's term index, and occasionally highlight algorithm steps.

In this book, we use `pip3` to ensure the packages are installed for Python 3. On most modern systems, you can use `pip` instead if it's already set up for Python 3.

Should You Buy This Book?

Like my previous two books, this one is distributed on the *read first, buy later* principle. I firmly believe that paying for content before consuming it means buying a pig in a poke. At a dealership, you can see and try a car. In a department store, you can try on clothes. Similarly, you should be able to read a book before paying for it.

The *read first, buy later* principle means you can freely download the book, read it, and share it with friends and colleagues. If you find the book helpful or useful in your work, business, or studies—or if you simply enjoy reading it—then buy it.

Acknowledgements

The high quality of this book would be impossible without volunteering editors. I especially thank Erman Sert, Viet Hoang Tran Duong, Alex Sherstinsky, Kelvin Sundli, and Mladen Korunoski for their systematic contributions.

I am also grateful to Alireza Bayat Makou, Taras Shalaiko, Domenico Siciliani, Preethi Raju, Srikumar Sundareshwar, Mathieu Nayrolles, Abhijit Kumar, Giorgio Mantovani, Abhinav Jain, Steven Finkelstein, Ryan Gaughan, Ankita Guha, Harmanan Kohli, Daniel Gross, Kea Kohv, Marcus Oliveira, Tracey Mercier, Prabin Kumar Nayak, Saptarshi Datta, Gurgen R. Hayrapetyan, Sina Abdidizaji, Federico Raimondi Cominesi, Santos Salinas, Anshul Kumar, Arash Mirbagheri, Roman Stanek, Jeremy Nguyen, Efim Shuf, Pablo Llopis, Marco Celeri, Tiago Pedro, and Manoj Pillai for their help.

If this is your first time exploring language models, I envy you a little—it's truly magical to discover how machines learn to understand the world through natural language.

I hope you enjoy reading this book as much as I enjoyed writing it.

Now grab your tea or coffee, and let's begin!

Chapter 1. Machine Learning Basics

This chapter starts with a brief overview of how artificial intelligence has evolved, explains what a machine learning model is, and presents the four steps of the machine learning process. Then, it covers some math basics like vectors and matrices, introduces neural networks, and wraps up with optimization methods like gradient descent and automatic differentiation.

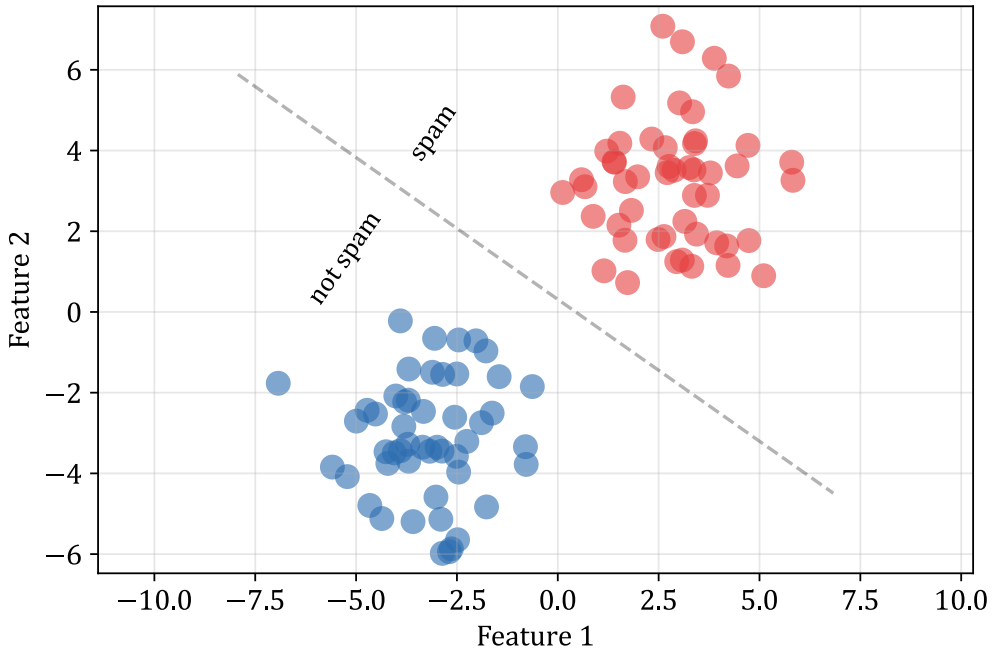
1.1. AI and Machine Learning

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."

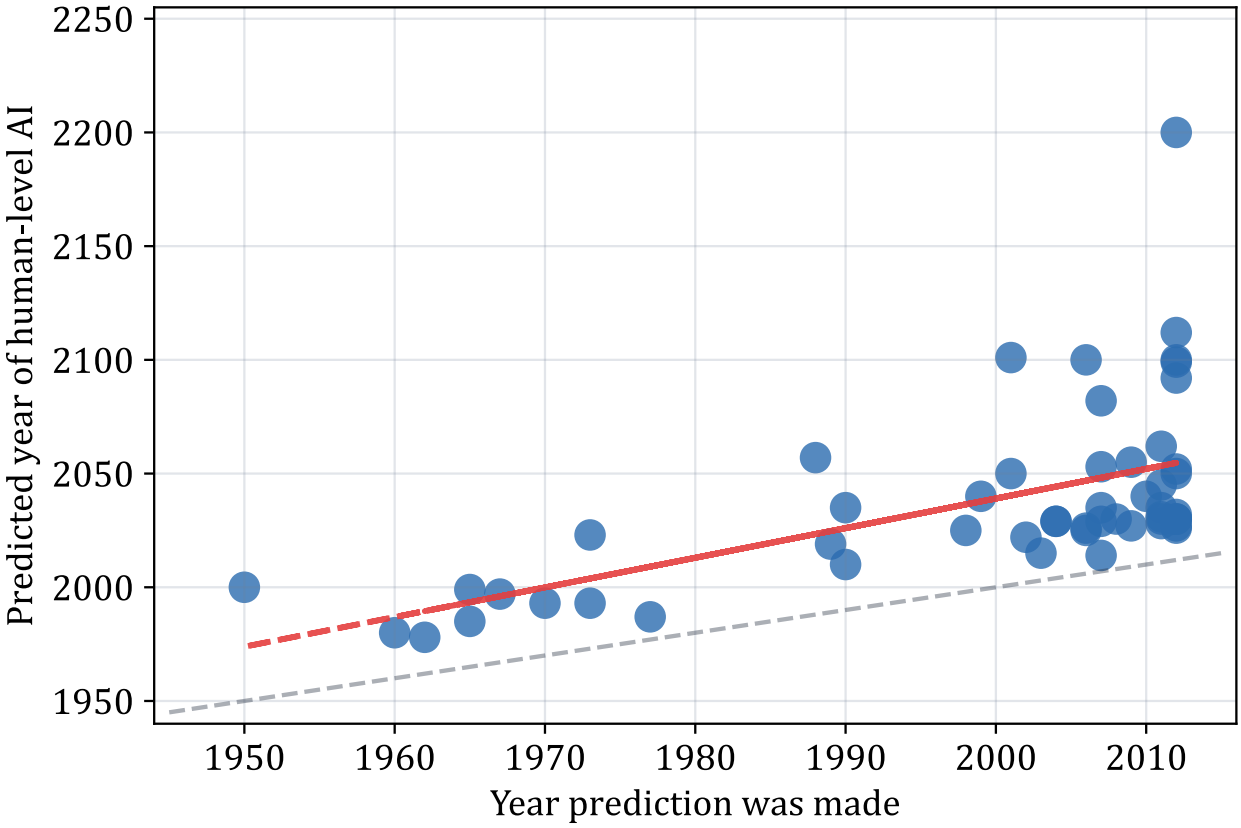
Frank Rosenblatt's **Perceptron** (1958) was an early **neural network** designed to recognize patterns by adjusting its internal parameters based on examples. Perceptron learned a **decision boundary**—a dividing line that separates examples of different classes (e.g., spam versus not spam):



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 $\{(150,200), (200,600), \ldots\}$. 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 $250\,\text{m}^2$ (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:

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

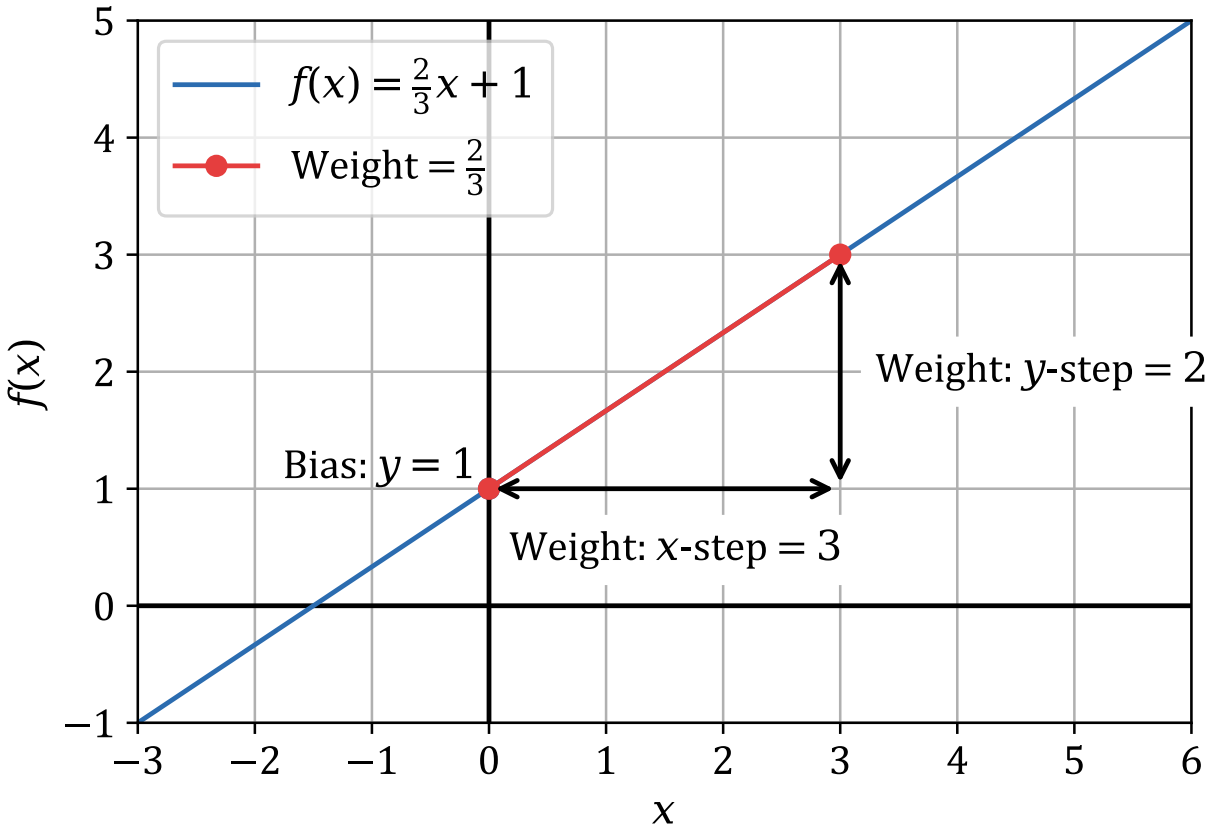
which is a **linear function** of x. The formula $wx + b$ is a **linear transformation** of x.

> The notation $\overset{\text{def}}{=}$ 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:

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

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:

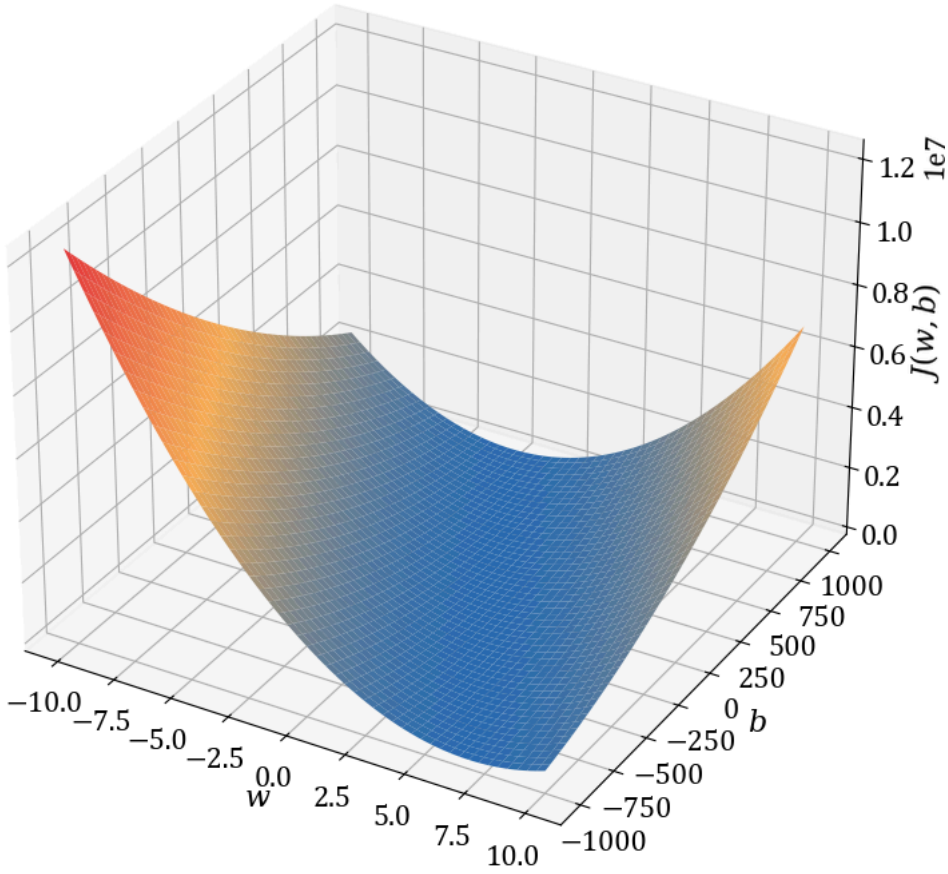
$$\begin{cases} \frac{\partial J}{\partial w} = 0 \\ \frac{\partial J}{\partial b} = 0 \end{cases}$$

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:

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

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

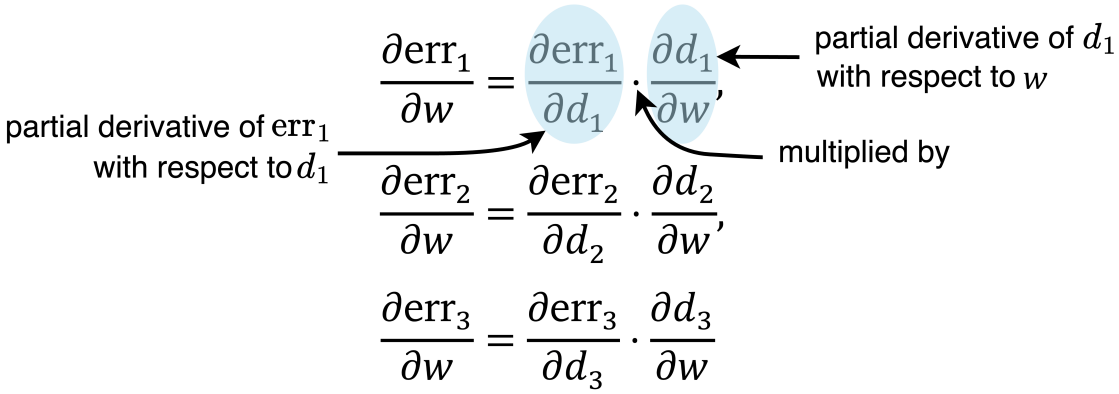
$$\frac{\partial J}{\partial w} = \frac{1}{3}\left(\frac{\partial \text{err}_1}{\partial w} + \frac{\partial \text{err}_2}{\partial w} + \frac{\partial \text{err}_3}{\partial w}\right),$$

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:



$$\frac{\partial \text{err}_1}{\partial w} = \frac{\partial \text{err}_1}{\partial d_1} \cdot \frac{\partial d_1}{\partial w},$$

$$\frac{\partial \text{err}_2}{\partial w} = \frac{\partial \text{err}_2}{\partial d_2} \cdot \frac{\partial d_2}{\partial w},$$

$$\frac{\partial \text{err}_3}{\partial w} = \frac{\partial \text{err}_3}{\partial d_3} \cdot \frac{\partial d_3}{\partial w}$$

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

Then,

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

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 $\frac{\partial J}{\partial b}$:

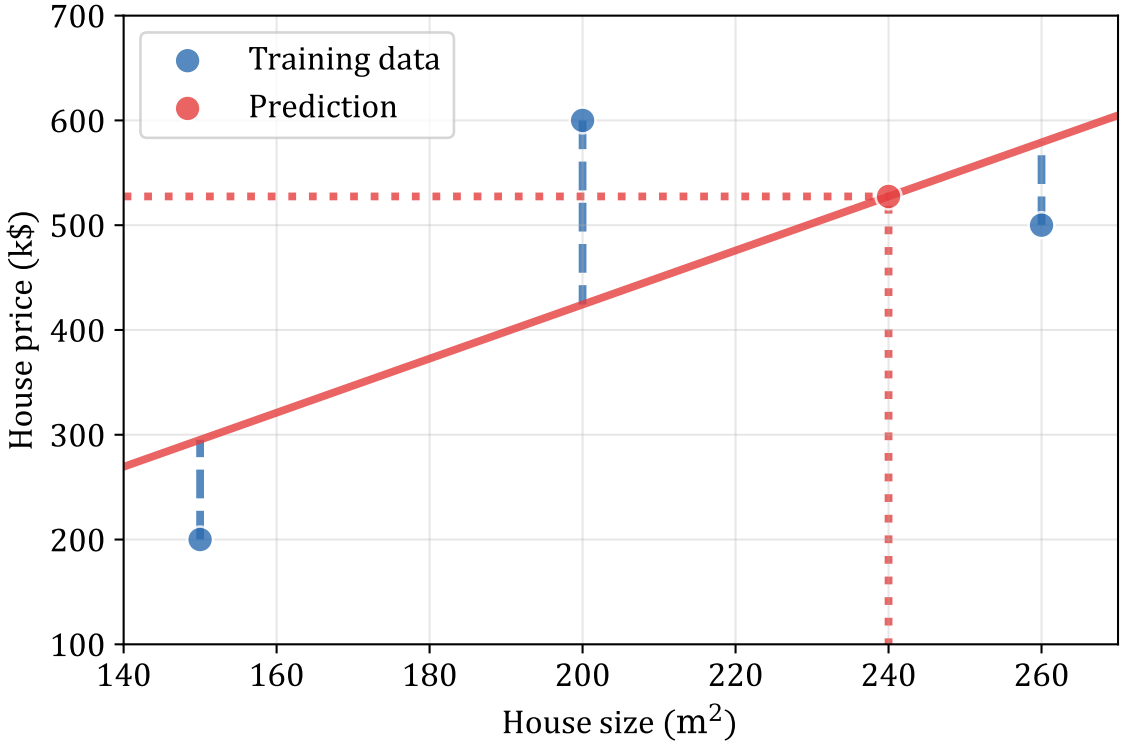
$$\begin{aligned} \frac{\partial J}{\partial b} &= \frac{1}{3}\big(2 \cdot (150w + b - 200) + 2 \cdot (200w + b - 600) + 2 \cdot (260w + b - 500)\big) \\ &= \frac{1}{3}(1220w + 6b - 2600) \end{aligned}$$

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:

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

[1] It's the square root of the error because our error, as defined in Equation 1.2, is the square of the difference between the predicted price and the real price of the house. It's common practice to take the square root of the mean squared error because it expresses the error in the same units as the target variable (price in this case). This makes it easier to interpret the error value.

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**:

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

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.

> 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: