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 Luij**t, 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.

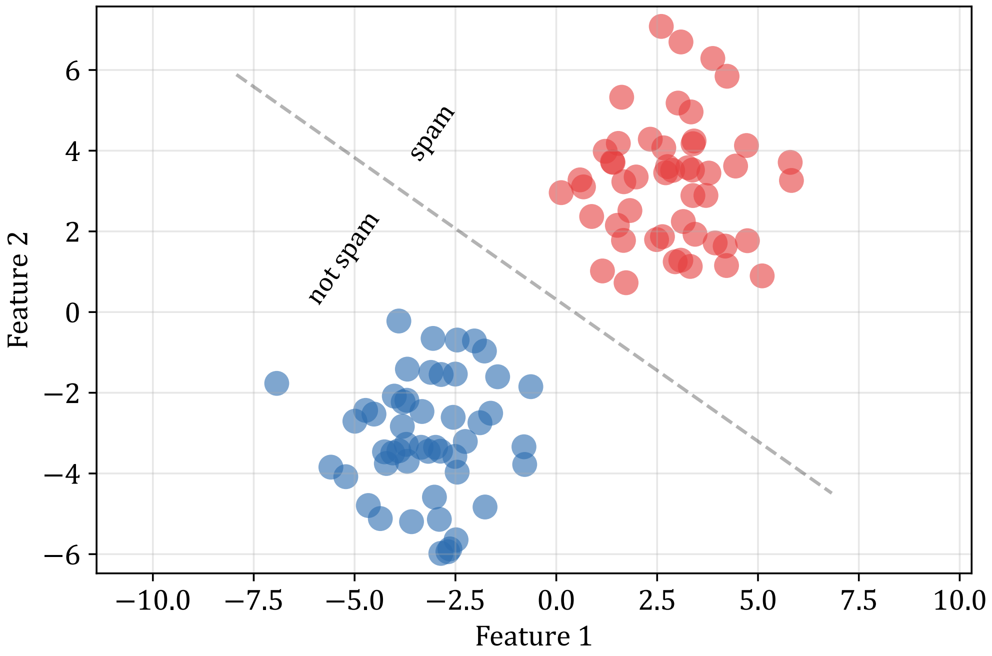
1.1. AI and Machine Learning

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."

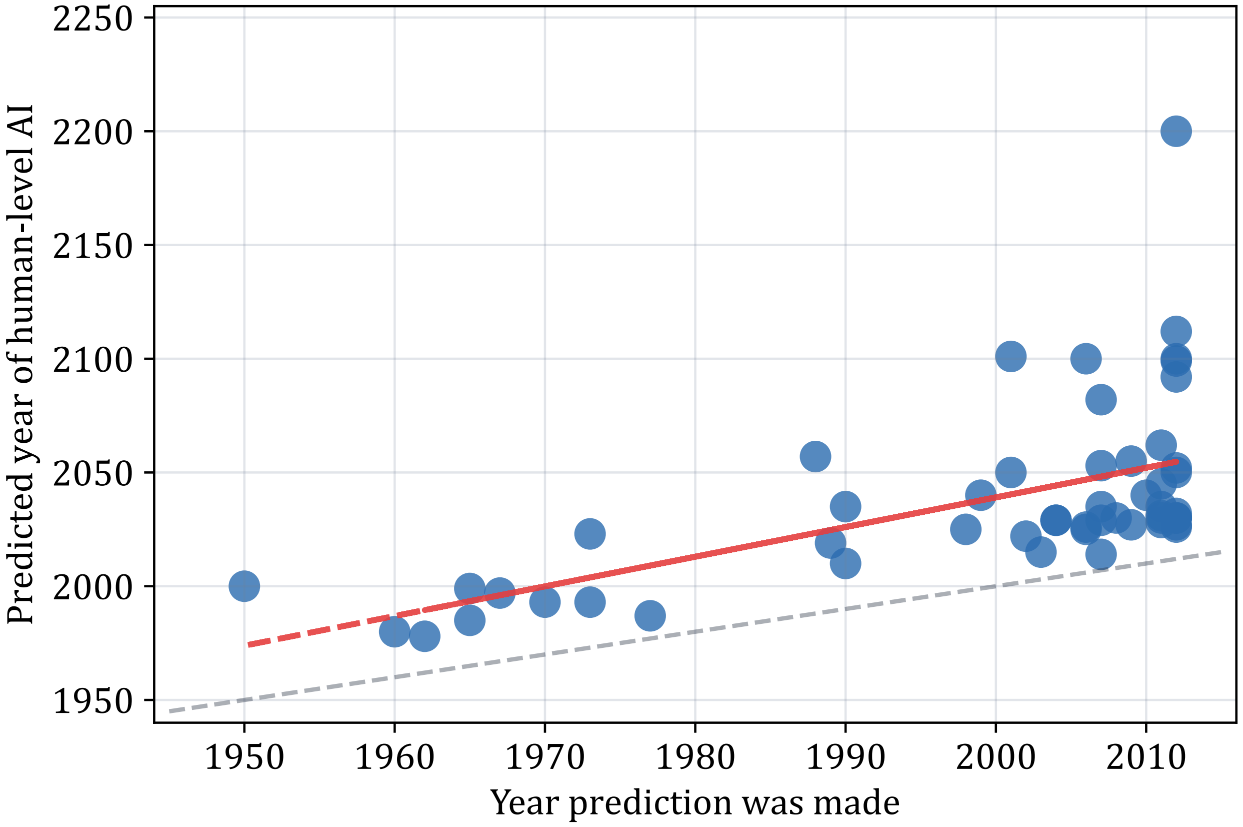
Frank Rosenblatt's **Perceptron** (1958) was an early **neural network** designed to recognize patterns by adjusting its internal parameters based on examples. Perceptron learned a **decision boundary**—a dividing line that separates examples of different classes (e.g., spam versus not spam):



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

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

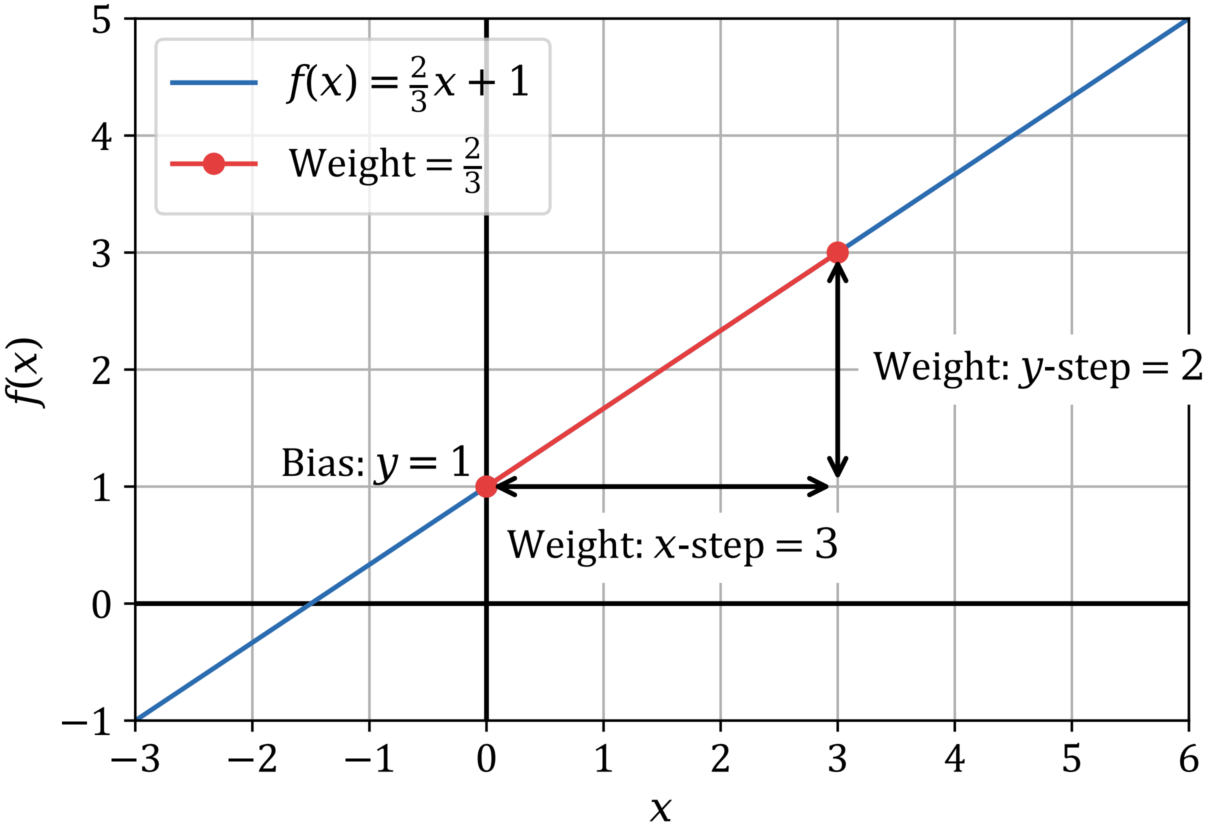
which is a **linear function** of x. The formula $wx + b$ is a **linear transformation** of x.

> 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 $\{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ 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 $\mathrm{err}(\tilde{y}_i, y_i)$ for a given example (x_i, y_i) as:

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

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:

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

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

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

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

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

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:

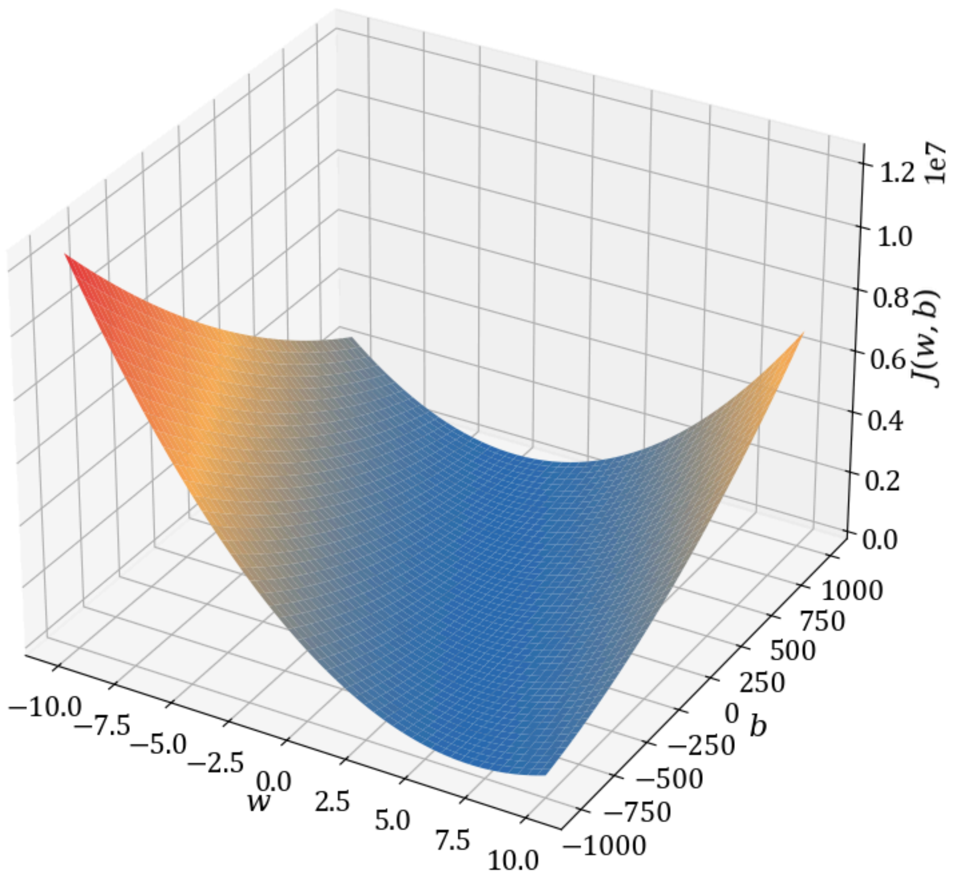
$$\begin{cases} \dfrac{\partial J}{\partial w} & = 0 \\ \dfrac{\partial J}{\partial b} & = 0 \end{cases}$$

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 $d_1 \overset{\text{def}}{=} 150w + b - 200$, $d_2 \overset{\text{def}}{=} 200w + b - 600$, $d_3 \overset{\text{def}}{=} 260w + b - 500$ 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 $f(g(x))$, 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:

$$\frac{\partial \text{err}_1}{\partial w} = \underbrace{\frac{\partial \text{err}_1}{\partial d_1}}_{\text{partial derivative of } \text{err}_1 \text{ with respect to } d_1} \underbrace{\cdot}_{\text{multiplied by}} \overbrace{\frac{\partial d_1}{\partial w}}^{\text{partial derivative of } d_1 \text{ with respect to } w},$$

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

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

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,

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

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

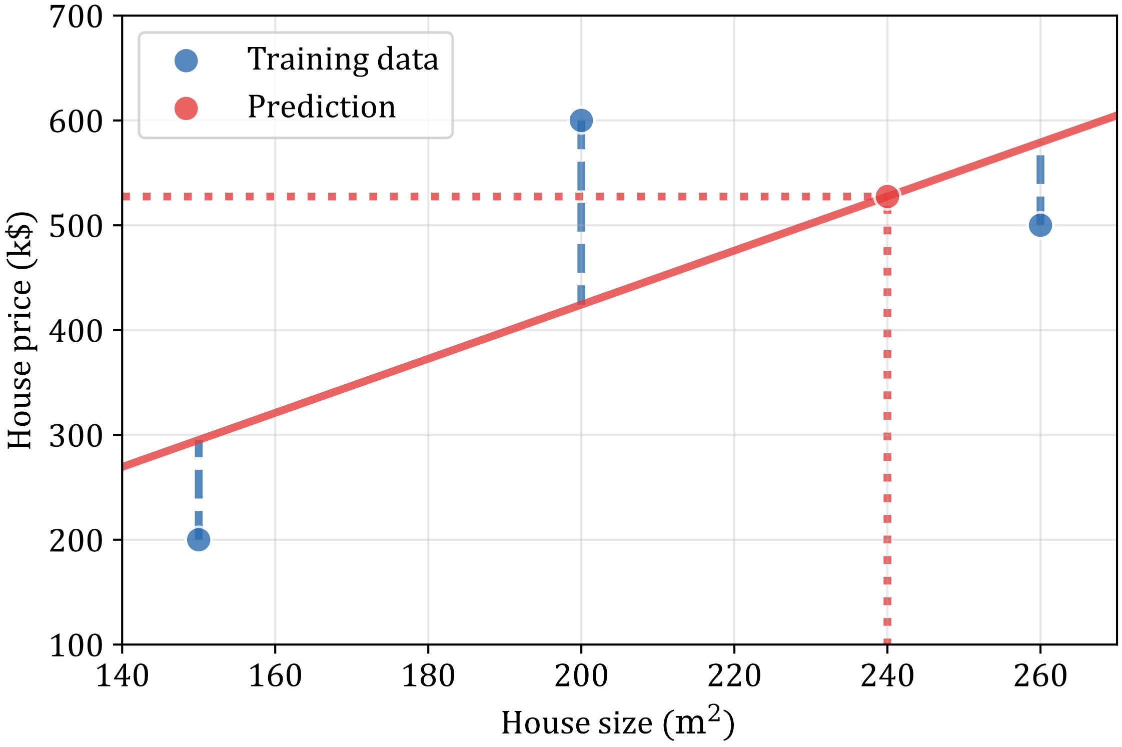
$$
\begin{aligned}
\frac{\partial J}{\partial b} &= \frac{1}{3}\big(2 \cdot (150w + b - 200) + 2 \cdot (200w + b - 600) + 2 \cdot (260w + b - 500)\big) \\
&= \frac{1}{3}(1220w + 6b - 2600)
\end{aligned}
$$

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:

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

[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 **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 **x** or **w**. For a given house **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.

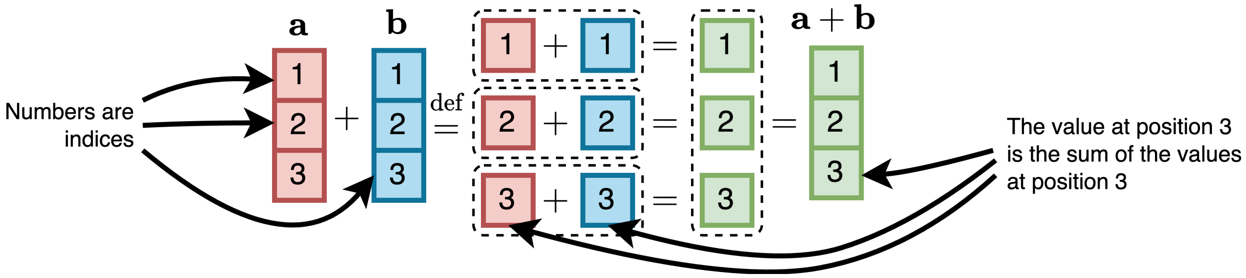
The equation above uses **capital-sigma notation**, where D represents the dimensionality of the input, and j runs from 1 to D. For example, in the 2-dimensional house scenario, $\sum_{j=1}^{2} w^{(j)} x^{(j)} \stackrel{\text{def}}{=} w^{(1)}x^{(1)} + w^{(2)}x^{(2)}$.

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:

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

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



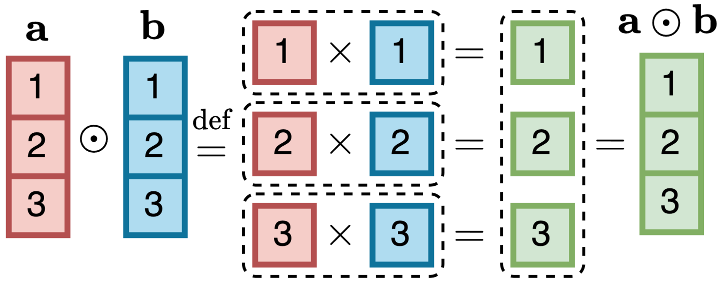
Numbers are indices
a
b
a + b
def
The value at position 3 is the sum of the values at position 3

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:

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

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 $\mathbf{x}$ and $\mathbf{y}$ is defined as:

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

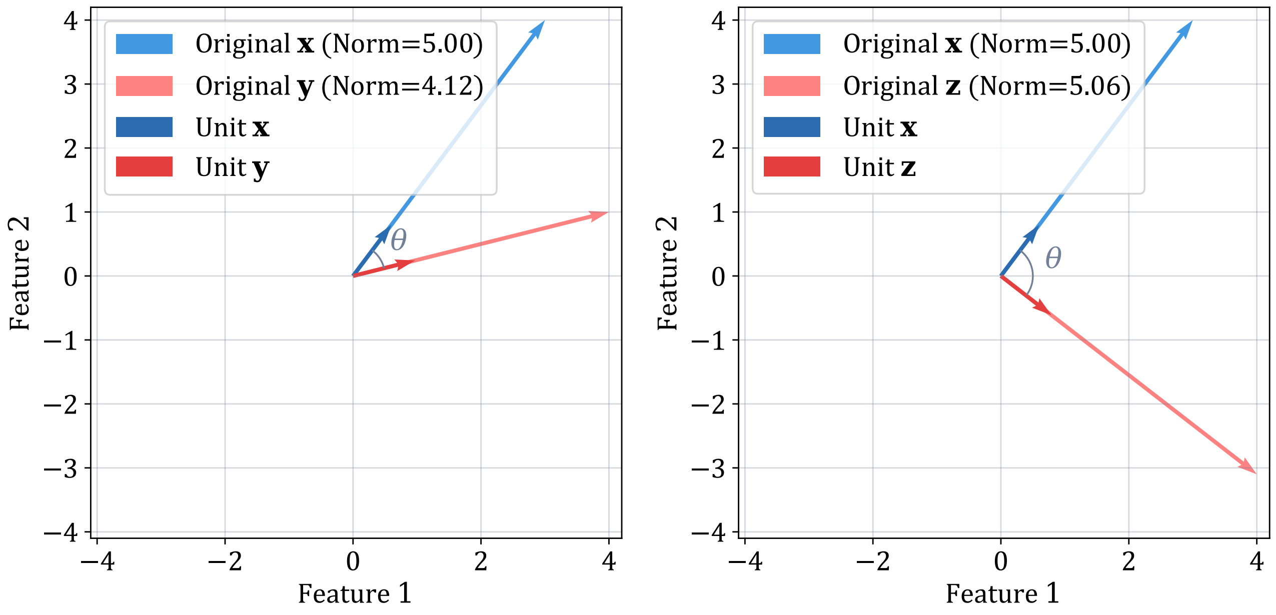
The cosine of the angle between two vectors quantifies their similarity. For instance, two houses with similar areas and bedroom counts will have a cosine similarity close to 1, otherwise the value will be lower. **Cosine similarity** is widely used to compare words or documents represented as **embedding 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 $\mathbf{x}$ 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.

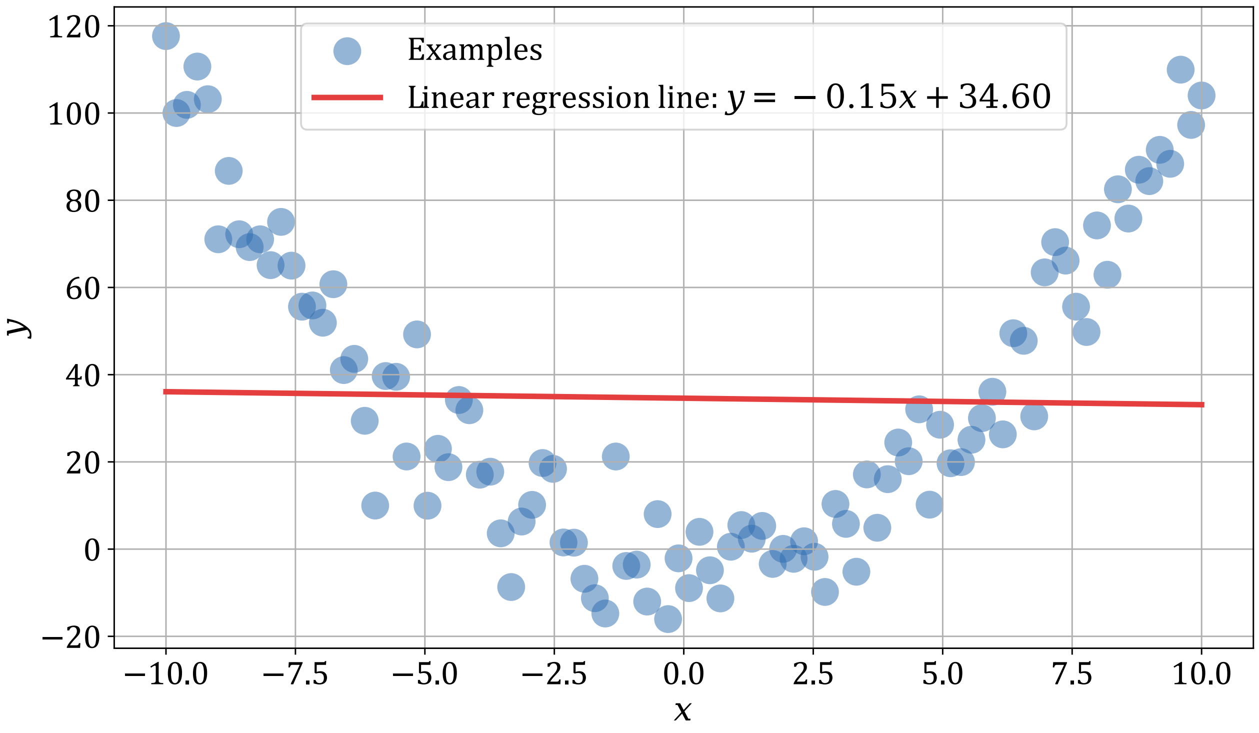
Linear models like $wx + b$ or $\mathbf{w} \cdot \mathbf{x} + b$ cannot solve many machine learning problems effectively. Even if we combine them into a **composite function** $f_2(f_1(x))$, a composite function of linear functions remains linear. This is straightforward to verify.

Let's define $y_1 = f_1(x) \overset{\text{def}}{=} a_1 x$ and $y_2 = f_2(y_1) \overset{\text{def}}{=} a_2 y_1$. 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 $a_3 \overset{\text{def}}{=} a_1 a_2$, 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:

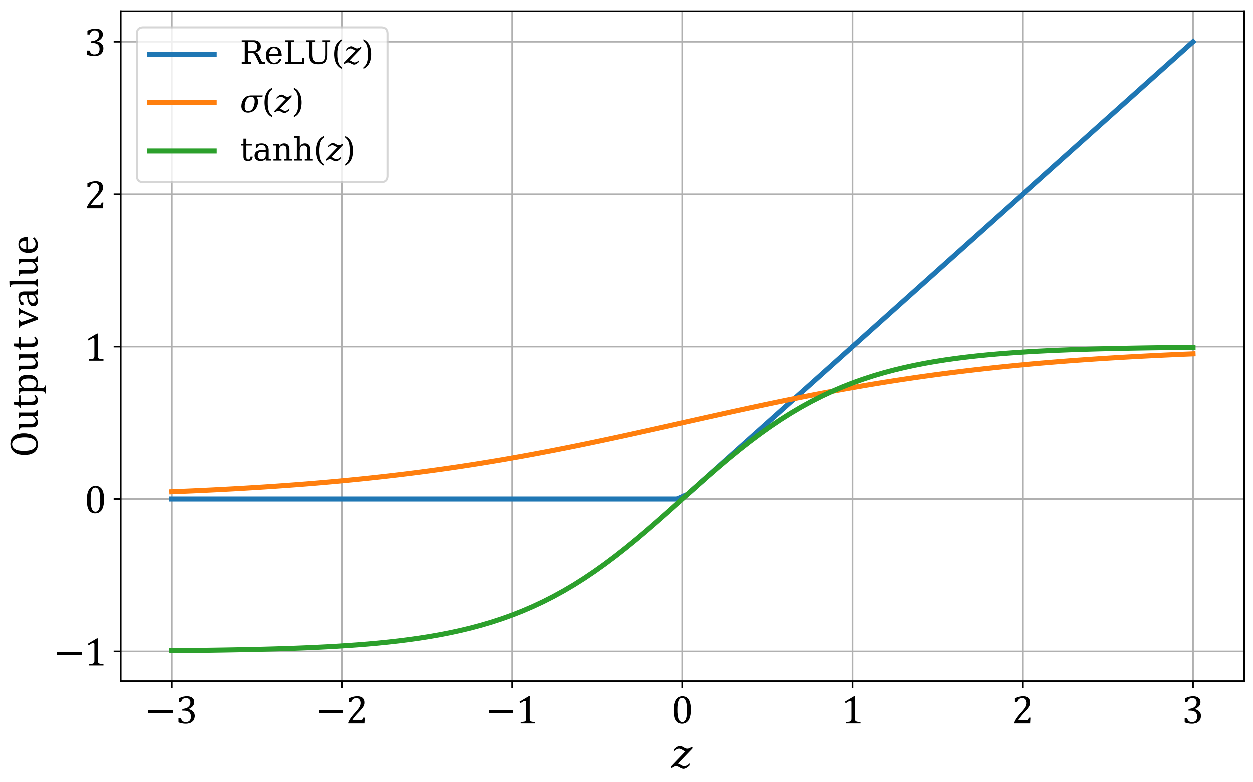
$$y = \phi(wx + b)$$

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

1) **ReLU** (**rectified linear unit**): $\text{ReLU}(z) \overset{\text{def}}{=} \max(0, z)$, which outputs non-negative values and is widely used in neural networks;
2) **Sigmoid**: $\sigma(z) \overset{\text{def}}{=} \frac{1}{1+e^{-z}}$, 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**): $\tanh(z) \overset{\text{def}}{=} \frac{e^z - e^{-z}}{e^z + e^{-z}}$; 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:

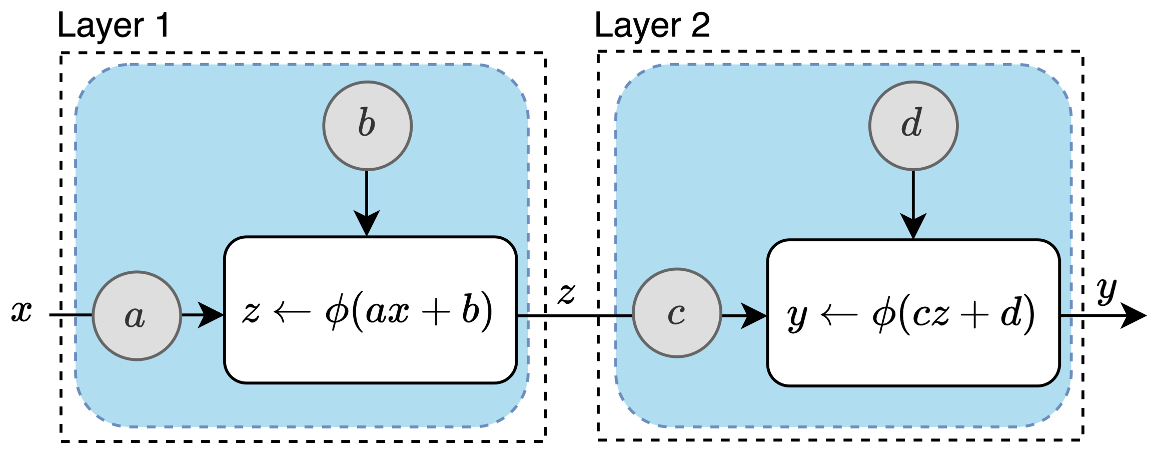


The structure $\phi(wx + b)$ enables learning non-linear models but can't capture all non-linear curves. By nesting these functions, we build more expressive models. For instance, let $f_1(x) \overset{\text{def}}{=} \phi(ax + b)$ and $f_2(z) \overset{\text{def}}{=} \phi(cz + d)$. A **composite model**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

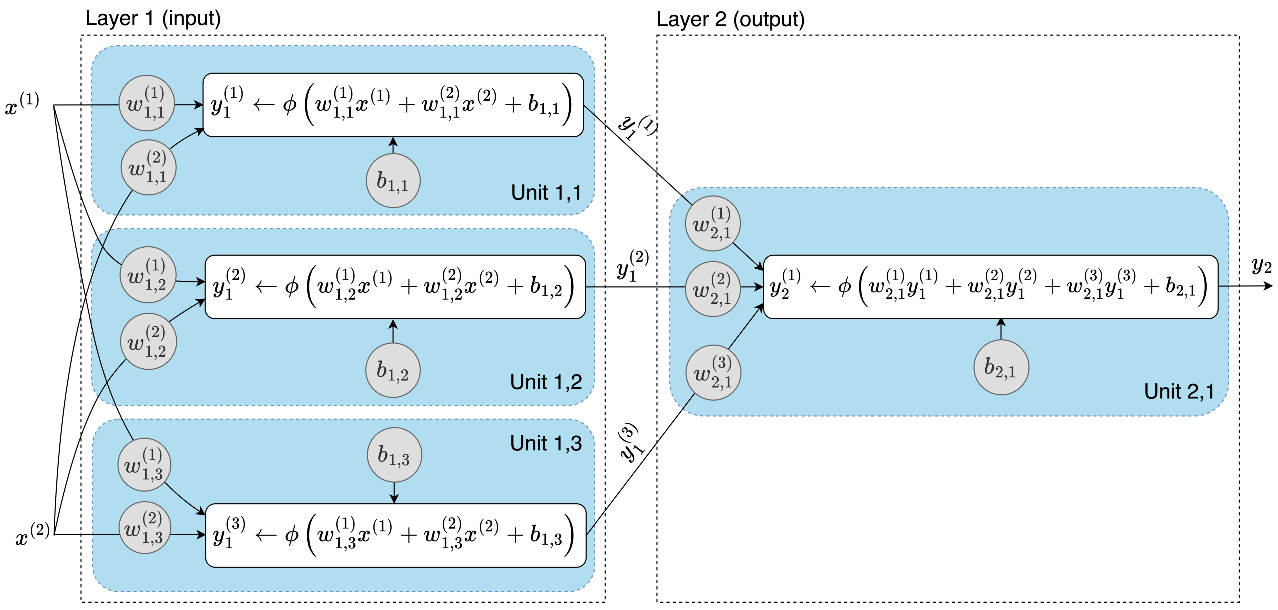

Layer 1 (input)
Layer 2 (output)
$x^{(1)}$
$x^{(2)}$
$w_{1,1}^{(1)}$
$w_{1,1}^{(2)}$
$y_1^{(1)} \leftarrow \phi\left(w_{1,1}^{(1)}x^{(1)} + w_{1,1}^{(2)}x^{(2)} + b_{1,1}\right)$
$b_{1,1}$
Unit 1,1
$w_{1,2}^{(1)}$
$w_{1,2}^{(2)}$
$y_1^{(2)} \leftarrow \phi\left(w_{1,2}^{(1)}x^{(1)} + w_{1,2}^{(2)}x^{(2)} + b_{1,2}\right)$
$b_{1,2}$
Unit 1,2
$w_{1,3}^{(1)}$
$w_{1,3}^{(2)}$
$y_1^{(3)} \leftarrow \phi\left(w_{1,3}^{(1)}x^{(1)} + w_{1,3}^{(2)}x^{(2)} + b_{1,3}\right)$
$b_{1,3}$
Unit 1,3
$y_1^{(1)}$
$y_1^{(2)}$
$y_1^{(3)}$
$w_{2,1}^{(1)}$
$w_{2,1}^{(2)}$
$w_{2,1}^{(3)}$
$y_2^{(1)} \leftarrow \phi\left(w_{2,1}^{(1)}y_1^{(1)} + w_{2,1}^{(2)}y_1^{(2)} + w_{2,1}^{(3)}y_1^{(3)} + b_{2,1}\right)$
$b_{2,1}$
Unit 2,1
y_2

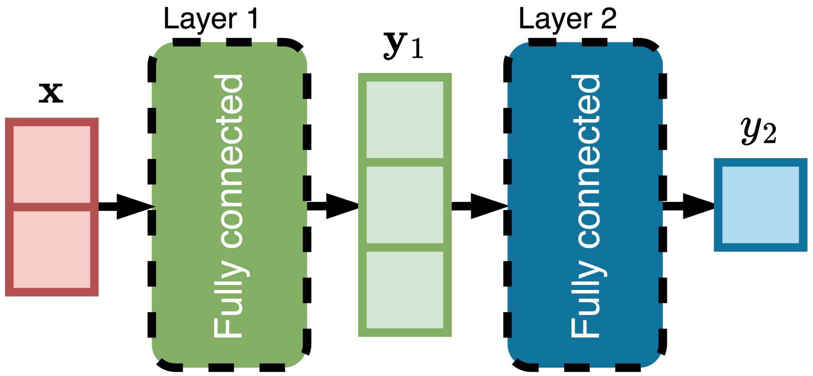
Figure 1.1: A neural network with two layers.

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**.

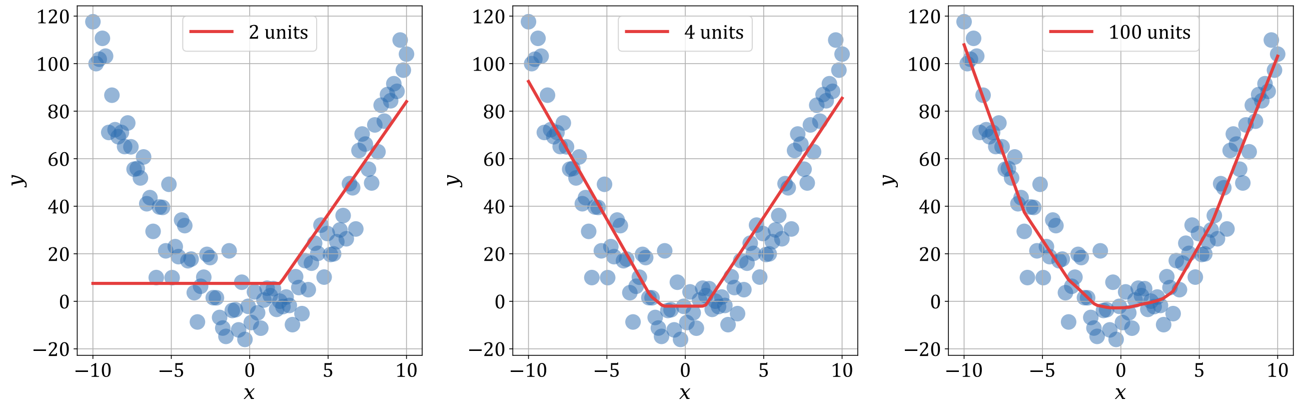
In Chapter 3, we will explore recurrent neural networks (RNNs). Unlike FNNs, RNNs have loops, where outputs from a layer are used as inputs to the same layer.

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:

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

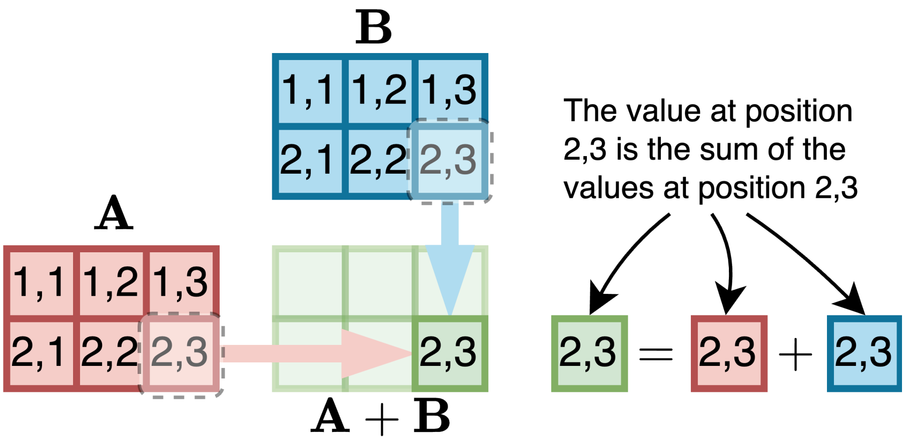
Here, $a_{i,j}$ represents the element in the i-th row and j-th column of the matrix. The dimensions of the matrix are expressed as $m \times n$ (read as "m by n").

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 **X** or **W**.

The **sum of two matrices A** and **B** of the same dimensionality is defined element-wise as:

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

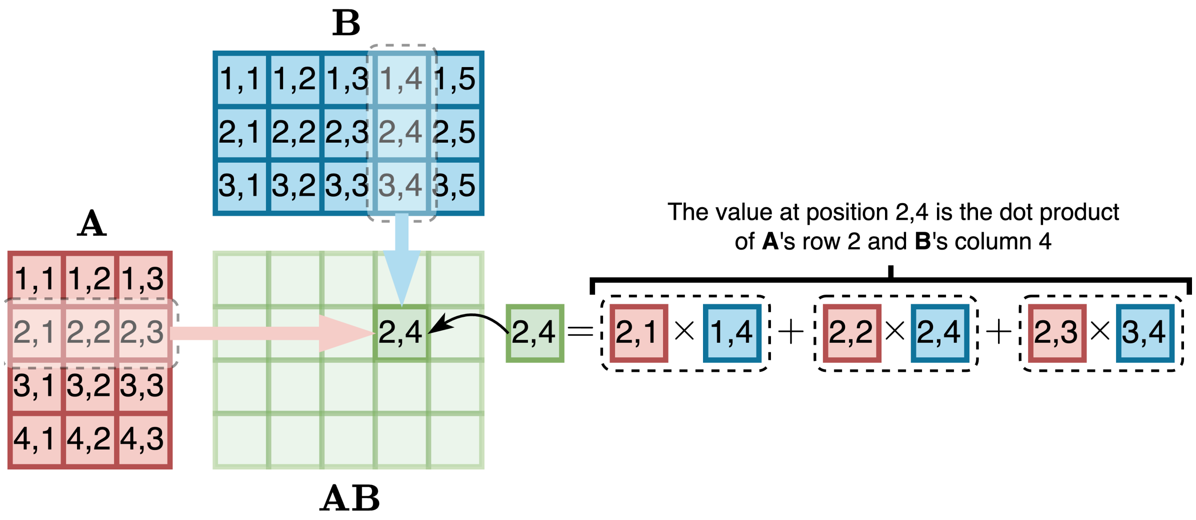
For example, for two 2×3 matrices **A** and **B**, the addition works like this:


B
A
A + B
The value at position 2,3 is the sum of the values at position 2,3
2,3 = 2,3 + 2,3

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

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

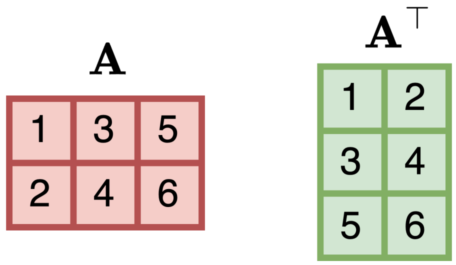
For example, for a 4×3 matrix **A** and a 3×5 matrix **B**, the product is a 4×5 matrix:


B
A
AB
The value at position 2,4 is the dot product of A's row 2 and B's column 4
2,4 = 2,1 × 1,4 + 2,2 × 2,4 + 2,3 × 3,4

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

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

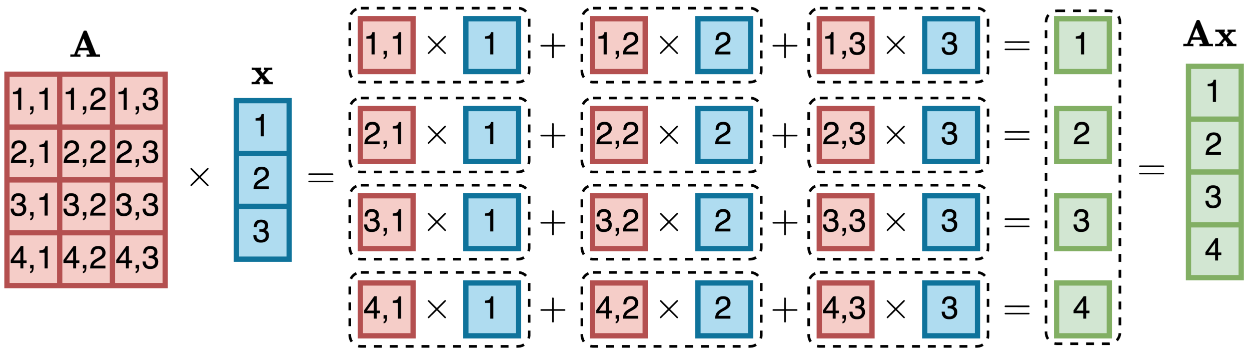
For example, for a 2×3 matrix **A**, its transpose $\mathbf{A}^\top$ look like this:



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

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

For example, a 4×3 matrix $\mathbf{A}$ multiplied by a 3D vector $\mathbf{x}$ 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:

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

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:

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

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:

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

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:

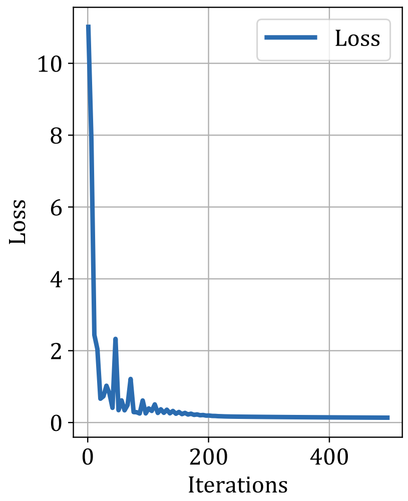
- Gradients are subtracted from parameters because they point in the direction of steepest ascent in the loss function. Since our goal is to minimize loss, we move in the opposite direction—hence, the subtraction.
- The **learning rate** η is a positive value close to 0 and serves as a **hyperparameter**—not 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.

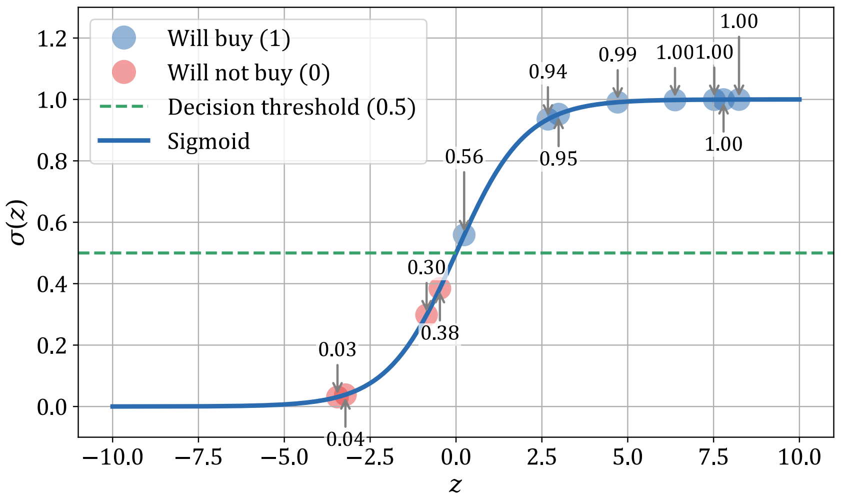
Let's illustrate the process with a simple dataset of 12 examples:

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

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 ➊ 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 ➌ 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.

[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.

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

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

```
for step in range(500):
    optimizer.zero_grad() ➊
    loss = criterion(model(inputs), labels) ➋
    loss.backward() ➌
    optimizer.step() ➍
```

Line ➋ calculates the binary cross-entropy loss (Equation 1.10) by evaluating model predictions against training labels. Line ➌ 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 ➊ therefore clears the gradients at each step's beginning.

Finally, in line ➍, 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.

Chapter 2. Language Modeling Basics

Language modeling requires transforming text into numbers that computers can process. In this chapter, we'll explore how to convert words and documents into numerical formats, introduce the fundamentals of language modeling, and study count-based models as our first architecture. Finally, we'll cover techniques for measuring language model performance.

Let's begin with one of the oldest yet effective techniques for converting text into usable data for machine learning: bag of words.

2.1. Bag of Words

Suppose you have a collection of documents and want to predict the main topic of each one. When topics are defined in advance, this task is called **classification**. With only two possible topics, it's known as **binary classification**, as explained in Section 1.7. With more than two topics, we have **multiclass classification**.

In multiclass classification, the dataset consists of pairs $\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, where $y_i \in \{1, \dots, C\}$, N represents the number of examples, and C denotes the number of possible classes. Each $\mathbf{x}_i$ could be a text document, with y_i being an integer indicating its topic—for example, 1 for "music," 2 for "science," or 3 for "cinema."

Machines don't process text like humans. To use machine learning on text, we first need to convert documents into numbers. Each document becomes a **feature vector**, where each feature is a **scalar**.

A common and effective approach to convert a collection of documents into feature vectors is the **bag of words** (**BoW**). Here's how it works for a collection of 10 simple documents:

ID	Text
1	Movies are fun for everyone.
2	Watching movies is great fun.
3	Enjoy a great movie today.
4	Research is interesting and important.
5	Learning math is very important.
6	Science discovery is interesting.

ID	Text
7	Rock is great to listen to.
8	Listen to music for fun.
9	Music is fun for everyone.
10	Listen to folk music!

A collection of text documents used in machine learning is called a **corpus**. The bag of words method applied to a corpus involves two key steps:

1. **Create a vocabulary**: List all unique words in the corpus to create the **vocabulary**.
2. **Vectorize documents**: Convert each document into a feature vector, where each dimension represents a word from the vocabulary. The value indicates the word's presence, absence, or frequency in the document.

For the 10-document corpus, the vocabulary is built by listing all unique words in alphabetical order. This involves removing punctuation, converting words to lowercase, and eliminating duplicates. After processing, we get:

```
vocabulary = ["a", "and", "are", "discovery", "enjoy", "every
one", "folk", "for", "fun", "great", "important", "interestin
g", "is", "learning", "listen", "math", "movie", "movies", "m
usic", "research", "rock", "science", "to", "today", "very",
"watching"]
```

Splitting a document into small indivisible parts is called **tokenization**, and each part is a **token**. There are different ways to tokenize. We tokenized our 10-document corpus by words. Sometimes, it's useful to break words into smaller units, called **subwords,** to keep the vocabulary size manageable. For instance, instead of including "interesting" in the vocabulary, we might split it into "interest" and "-ing." One method for subword tokenization, which we'll cover in this chapter, is byte-pair encoding. The choice of tokenization method depends on the language, dataset, and model, and the best one is found experimentally.

> A count of all English word **surface forms**—like *do*, *does*, *doing*, and *did*—reveals several million possibilities. Languages with more

complex morphology have even greater numbers. A Finnish noun alone can take 2,000–3,000 different forms to express various case and number combinations. Using subwords offers a practical solution, as storing every surface form in the vocabulary would consume excessive memory and computational resources.

Words are a type of token, so "token" and "word" are often used interchangeably as the smallest indivisible units of a document. In this book, when a distinction is important, context will make it clear. While the bag-of-words approach can handle both words and subwords, it was originally designed for words—hence the name.

Feature vectors can be organized into a **document-term matrix** (DTM). Here, rows represent documents, and columns represent tokens. Below is a partial document-term matrix for a 10-document corpus. It includes only a subset of tokens to fit within the page width:

Doc	a	and	…	fun	…	listen	math	…	science	…	watching
1	0	0	…	1	…	0	0	…	0	…	0
2	0	0	…	1	…	0	0	…	0	…	1
3	1	0	…	0	…	0	0	…	0	…	0
4	0	1	…	0	…	0	0	…	0	…	0
5	0	0	…	0	…	0	1	…	0	…	0
6	0	0	…	0	…	0	0	…	1	…	0
7	0	0	…	0	…	1	0	…	0	…	0
8	0	0	…	1	…	1	0	…	0	…	0
9	0	0	…	1	…	0	0	…	0	…	0
10	0	0	…	0	…	1	0	…	0	…	0

In the DTM above, 1 means the token appears in the document, while 0 means it does not. For instance, the feature vector $\mathbf{x}_2$ for document 2 (*"Watching movies is great fun."*) is:

$$\mathbf{x}_2 = [0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1]^\top.$$

In natural languages, word frequencies follow **Zipf's Law,** stating that a word's frequency is inversely proportional to its rank in the frequency table—for instance, the second most frequent word appears half as often as the most frequent one. Consequently, document-term matrices are usually **sparse,** containing mostly zeros.

A neural network can be trained to predict a document's topic using these feature vectors. Let's do that. The first step is to assign labels to the documents, a process known as **labeling**. Labeling can be done manually or assisted by an algorithm. When algorithms are used, human validation is often needed to confirm accuracy. Here, we will manually label the documents by reading each one and choosing the most suitable topic from the three options.

Doc	Text	Class ID	Class Name
1	Movies are fun for everyone.	1	Cinema
2	Watching movies is great fun.	1	Cinema
3	Enjoy a great movie today.	1	Cinema
4	Research is interesting and important.	3	Science
5	Learning math is very important.	3	Science
6	Science discovery is interesting.	3	Science
7	Rock is great to listen to.	2	Music
8	Listen to music for fun.	2	Music
9	Music is fun for everyone.	2	Music
10	Listen to folk music!	2	Music

Advanced **chat language models** enable highly accurate automated document labeling through a panel of expert models. Using three LLMs, when two or more assign the same label to a document, that label is adopted. If all three disagree, either a human can decide, or a fourth model can break the tie. In many business contexts, manual labeling is becoming obsolete, as LLMs offer faster and often more reliable labeling.

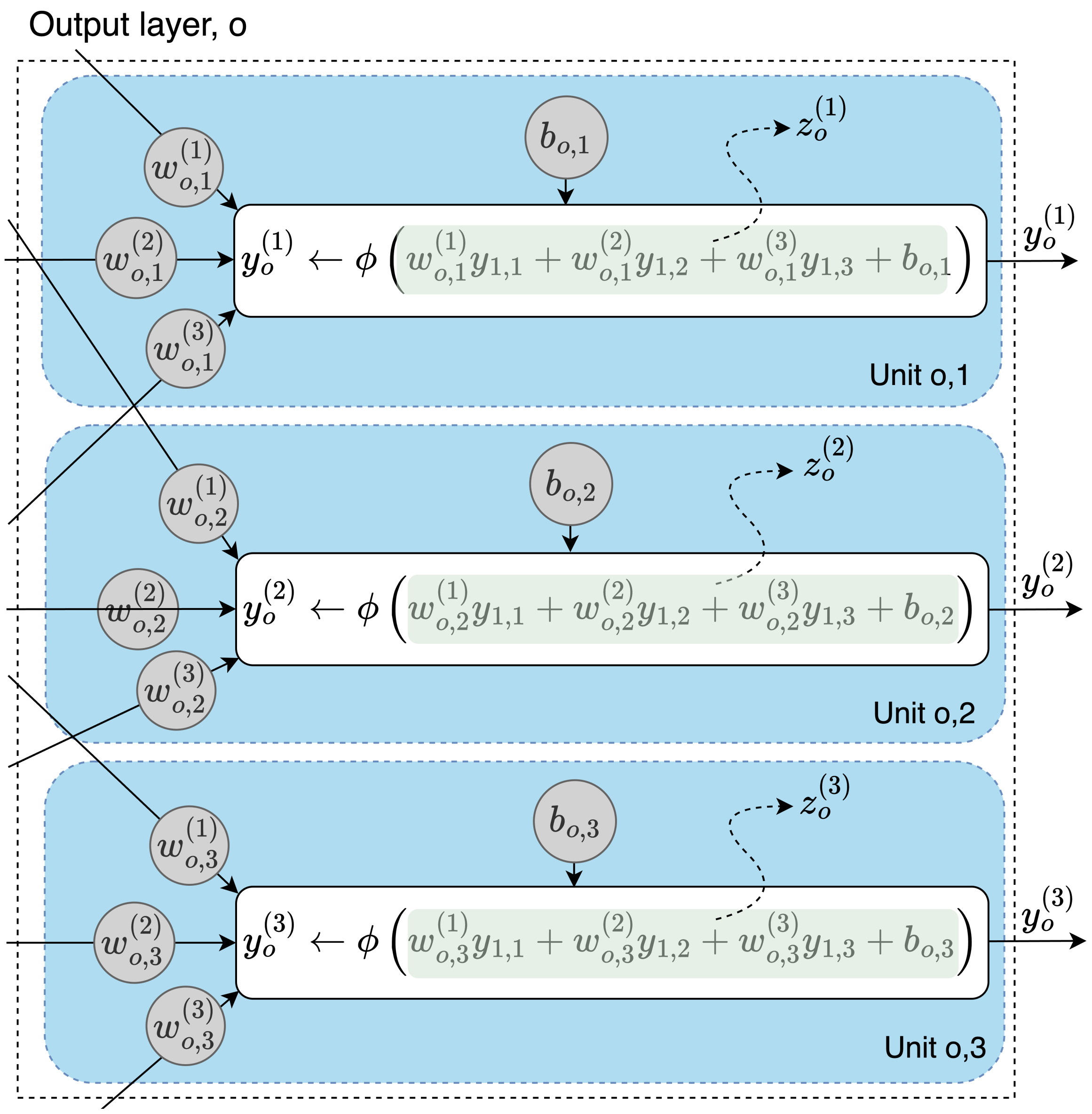
We have three classes: 1 for cinema, 2 for music, and 3 for science.[3] While binary classifiers typically use the **sigmoid** activation function with the **binary cross-entropy** loss, as discussed in Section 1.7, tasks involving three or more classes generally employ the softmax activation function paired with the cross-entropy loss.

The **softmax** function is defined as:

$$\text{softmax}(\mathbf{z}, k) \overset{\text{def}}{=} \frac{e^{z^{(k)}}}{\sum_{j=1}^{D} e^{z^{(j)}}}$$

Here, **z** is a D-dimensional vector of logits, k is the index for which the softmax is computed, and e is **Euler's number**. **Logits** are the raw outputs of a neural network, prior to applying an activation function, as shown below:

[3] Class labels in classification are arbitrary and unordered. You can assign numbers to the classes in any way, and the model's performance won't change as long as the mapping is consistent for all examples.



Output layer, o
$b_{o,1}$
$z_o^{(1)}$
$w_{o,1}^{(1)}$
$w_{o,1}^{(2)}$
$w_{o,1}^{(3)}$
$y_o^{(1)} \leftarrow \phi\left(w_{o,1}^{(1)}y_{1,1} + w_{o,1}^{(2)}y_{1,2} + w_{o,1}^{(3)}y_{1,3} + b_{o,1}\right)$
$y_o^{(1)}$
Unit o,1
$b_{o,2}$
$z_o^{(2)}$
$w_{o,2}^{(1)}$
$w_{o,2}^{(2)}$
$w_{o,2}^{(3)}$
$y_o^{(2)} \leftarrow \phi\left(w_{o,2}^{(1)}y_{1,1} + w_{o,2}^{(2)}y_{1,2} + w_{o,2}^{(3)}y_{1,3} + b_{o,2}\right)$
$y_o^{(2)}$
Unit o,2
$b_{o,3}$
$z_o^{(3)}$
$w_{o,3}^{(1)}$
$w_{o,3}^{(2)}$
$w_{o,3}^{(3)}$
$y_o^{(3)} \leftarrow \phi\left(w_{o,3}^{(1)}y_{1,1} + w_{o,3}^{(2)}y_{1,2} + w_{o,3}^{(3)}y_{1,3} + b_{o,3}\right)$
$y_o^{(3)}$
Unit o,3

The figure shows the output layer of a neural network, labeled as o. The logits $z_o^{(k)}$, for $k \in \{1,2,3\}$, are the values in light green. These represent the outputs of the units before the activation function is applied. The vector $\mathbf{z}$ is expressed as $\mathbf{z}_o = \left[z_o^{(1)}, z_o^{(2)}, z_o^{(3)}\right]^\top$.

For instance, the softmax for unit $o,2$ in the figure is calculated as:

$$\text{softmax}(\mathbf{z}_o, 2) = \frac{e^{z_o^{(2)}}}{e^{z_o^{(1)}} + e^{z_o^{(2)}} + e^{z_o^{(3)}}}$$

Softmax transforms a vector into a **discrete probability distribution** (DPD), ensuring that $\sum_{k=1}^{D} \text{softmax}(\mathbf{z}, k) = 1$. A DPD assigns probabilities to values in a finite set, with their sum equaling 1. A **finite set** contains a countable number of distinct elements. For instance, in a classification task with classes 1, 2, and 3, these classes constitute a finite set. The softmax function maps each class to a probability, with these probabilities summing to 1.

Let's compute the probabilities step by step. Assume we have three logits, $\mathbf{z} = [2.0, 1.0, 0.5]^\top$, representing a document's classification into cinema, music, or science.

First, calculate $e^{z^{(k)}}$ for each logit:

$$\begin{aligned} e^{z^{(1)}} &= e^{2.0} &&\approx 7.39, \\ e^{z^{(2)}} &= e^{1.0} &&\approx 2.72, \\ e^{z^{(3)}} &= e^{0.5} &&\approx 1.65 \end{aligned}$$

Next, sum these values: $\sum_{j=1}^{3} e^{z^{(j)}} = 7.39 + 2.72 + 1.65 \approx 11.76$.

Now use the softmax formula, $\text{softmax}(\mathbf{z}, k) = \frac{e^{z^{(k)}}}{\sum_{j=1}^{3} e^{z^{(j)}}}$, to compute the probabilities:

$$\Pr(\text{cinema}) = \frac{7.39}{11.76} \approx 0.63,$$
$$\Pr(\text{music}) = \frac{2.72}{11.76} \approx 0.23,$$
$$\Pr(\text{science}) = \frac{1.65}{11.76} \approx 0.14$$

> Neural network softmax outputs are better characterized as "probability scores" rather than true statistical probabilities, despite summing to one and resembling class likelihoods. Unlike logistic regression or Naïve Bayes models, neural networks don't generate genuine class probabilities. For simplicity, though, I'll refer to these probability scores as "probabilities" throughout this book.

The **cross-entropy** loss measures how well predicted probabilities match the true distribution. The true distribution is typically a **one-hot vector** with a single element equal to 1 (the correct class) and 0 elsewhere. For example, a one-hot encoding with 3 classes looks like:

Class	One-hot vector
1	$[1,0,0]^\top$
2	$[0,1,0]^\top$
3	$[0,0,1]^\top$

The cross-entropy loss for a single example is:

$$\text{loss}(\tilde{\mathbf{y}}, \mathbf{y}) = -\sum_{k=1}^{C} y^{(k)} \log\left(\tilde{y}^{(k)}\right),$$

where C is the number of classes, $\mathbf{y}$ is the one-hot encoded true label, and $\tilde{\mathbf{y}}$ is the predicted probabilities. Here, $y^{(k)}$ and $\tilde{y}^{(k)}$ represent the k^{th} elements of $\mathbf{y}$ and $\tilde{\mathbf{y}}$, respectively.

Since $\mathbf{y}$ is one-hot encoded, only the term corresponding to the correct class contributes to the summation. The summation thus simplifies by retaining only that single term. Let's simplify it. Suppose the correct class is c, so $y^{(c)} = 1$ and $y^{(k)} = 0$ for all $k \neq c$. In the summation, only the term where $k = c$ will be non-zero. The equation simplifies to:

$$\text{loss}(\tilde{\mathbf{y}}, \mathbf{y}) = -\log\left(\tilde{y}^{(c)}\right) \tag{2.1}$$

This simplified form indicates that the loss corresponds to the negative logarithm of the probability assigned to the correct class. For N examples, the average loss is:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N} \log\left(\hat{y}_i^{(c_i)}\right),$$

where c_i is the correct class index for the i^{th} example.

When used with softmax in the output layer, cross-entropy loss guides the network to assign high probabilities to correct classes while reducing probabilities for incorrect ones.

For a document classification example with three classes (cinema, music, and science), the network generates three logits. These logits are passed through the softmax function to convert them into probabilities for each class. The cross-entropy loss is then calculated between these scores and the true one-hot encoded labels.

Let's illustrate this by training a simple two-layer neural network to classify documents into three classes. We first import dependencies, set a random seed, and define the dataset:

```
import re, torch, torch.nn as nn

torch.manual_seed(42) ❶

docs = [
    "Movies are fun for everyone.",
    "Watching movies is great fun.",
    ...
    "Listen to folk music!"
]

labels = [1, 1, 1, 3, 3, 3, 2, 2, 2, 2]
num_classes = len(set(labels))
```

Setting the random seed in line ❶ ensures consistent random number generation across PyTorch runs. This guarantees **reproducibility**, allowing you to attribute performance changes to code or hyperparameter modifications rather than random variations. Reproducibility is also essential for teamwork, enabling collaborators to examine issues under identical conditions.

Next, we convert documents into a bag of words using two methods: `tokenize`, which splits input text into lowercase words, and `get_vocabulary`, which constructs the vocabulary:

```
def tokenize(text):
    return re.findall(r"\w+", text.lower()) ❶

def get_vocabulary(texts):
    tokens = {token for text in texts for token in tokenize(t
ext)} ❷
    return {word: idx for idx, word in enumerate(sorted(token
s))} ❸

vocabulary = get_vocabulary(docs)
```

In line ➊, the regular expression `\w+` extracts individual words from the text. A **regular expression** is a sequence of characters used to define a search pattern. The pattern `\w+` matches sequences of "word characters," such as letters, digits, and underscores.

The `findall` function from Python's `re` module applies the regular expression and returns a list of all matches in the input string. In this case, it extracts all words.

In line ➋, the corpus is converted into a set of tokens by iterating through each document and extracting words using the same regular expression. In line ➌, these tokens are sorted alphabetically and mapped to unique indices, forming a vocabulary.

Once the vocabulary is built, the next step is to define the feature extraction function that converts a document into a feature vector:

```
def doc_to_bow(doc, vocabulary):
    tokens = set(tokenize(doc))
    bow = [0] * len(vocabulary)
    for token in tokens:
        if token in vocabulary:
            bow[vocabulary[token]] = 1
    return bow
```

The `doc_to_bow` function takes a document string and a vocabulary and returns the bag-of-words representation of the document.

Now, let's transform our documents and labels into numbers:

```
vectors = torch.tensor(
    [doc_to_bow(doc, vocabulary) for doc in docs],
    dtype=torch.float32
)
labels = torch.tensor(labels, dtype=torch.long) - 1 ➊
```

The `vectors` tensor with shape `(10, 26)` represents 10 documents as rows and 26 vocabulary tokens as columns, while the `labels` tensor of shape `(10,)` contains the class label for each document. The labels use integer indices rather than one-hot encoding since PyTorch's cross-entropy loss function (`nn.CrossEntropyLoss`) expects this format.

Line ❶ uses `torch.long` to cast labels to 64-bit integers. The `-1` adjustment converts our original classes 1, 2, 3 to indices 0, 1, 2, which aligns with PyTorch's expectation that class indices begin at 0 for models and loss functions like `CrossEntropyLoss`.

PyTorch provides two APIs for model definition: the **sequential API** and the **module API**. While we used the straightforward `nn.Sequential` API to define our model in Section 1.8, we'll now explore building a multilayer perceptron using the more versatile `nn.Module` API:

```
input_dim = len(vocabulary)
hidden_dim = 50
output_dim = num_classes

class SimpleClassifier(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x) ❶
        x = self.relu(x) ❷
        x = self.fc2(x) ❸
        return x

model = SimpleClassifier(input_dim, hidden_dim, output_dim)
```

The `SimpleClassifier` class implements a **feedforward neural network** with two layers. Its constructor defines the network components:

1. A fully connected layer, `self.fc1`, maps the input of size `input_dim` (equal to the vocabulary size) to 50 (`hidden_dim`) outputs.
2. A ReLU activation function introduces non-linearity.
3. A second fully connected layer, `self.fc2`, reduces the 50 intermediate outputs to `output_dim`, the number of unique labels.

The `forward` method describes the **forward pass**, where inputs flow through the layers:

- In line ❶, the input `x` of shape `(10, 26)` is passed to the first fully connected layer, transforming it to shape `(10, 50)`.
- In line ❷, output from this layer is fed through the ReLU activation function, keeping the shape `(10, 50)`.
- In line ❸, the result is sent to the second fully connected layer, reducing it from shape `(10, 50)` to `(10, 3)`, producing the model's final output with logits.

The `forward` method is called automatically when you pass input data to the model instance, like this: `model(input)`.

> While `SimpleClassifier` omits a final softmax layer, this is intentional—PyTorch's `CrossEntropyLoss` combines softmax and cross-entropy loss internally for stability. This design eliminates the need for an explicit softmax in the model's forward pass.

With our model defined, the next steps, as outlined in Section 1.8, are to define the loss function, choose the gradient descent algorithm, and set up the training loop:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

for step in range(3000):
    optimizer.zero_grad()
    loss = criterion(model(vectors), labels)
    loss.backward()
    optimizer.step()
```

As you can see, the training loop is identical to the one in Section 1.8. Once the training is complete, we can test the model on a new document:

```
new_docs = [
    "Listening to rock music is fun.",
    "I love science very much."
]
class_names = ["Cinema", "Music", "Science"]

new_doc_vectors = torch.tensor(
    [doc_to_bow(new_doc, vocabulary) for new_doc in new_docs]
,
    dtype=torch.float32
)

with torch.no_grad(): ❶
    outputs = model(new_doc_vectors) ❷
    predicted_ids = torch.argmax(outputs, dim=1) + 1 ❸

for i, new_doc in enumerate(new_docs):
    print(f'{new_doc}: {class_names[predicted_ids[i].item() -
1]}')
```

Output:

```
Listening to rock is fun.: Music
I love scientific research.: Science
```

The `torch.no_grad()` statement in line ❶ disables the default gradient tracking. While gradients are essential during **training** to update model parameters, they're unnecessary during **testing** or **inference**. Since these phases don't involve parameter updates, disabling gradient tracking conserves memory and speeds up computation. Note that the terms "testing," "inference," and "evaluation" are often used interchangeably when referring to generating predictions on unseen data.

In line ❷, the model processes all inputs simultaneously during inference, just as it does during training. This parallel processing approach leverages vectorized operations, substantially reducing computation time compared to processing inputs one by one.

We only care about the final label, not the logits returned by the model. In line ❸, `torch.argmax` identifies the highest logit's index, corresponding to the predicted class. Adding 1 compensates for the earlier shift from 1-based to 0-based indexing.

While the bag-of-words approach offers simplicity and practicality, it has notable limitations. Most significantly, it fails to capture token order or context. Consider how "the cat chased the dog" and "the dog chased the cat" yield identical representations, despite conveying opposite meanings.

N-grams provide one solution to this challenge. An n-gram consists of n consecutive tokens from text. Consider the sentence “Movies are fun for everyone”—its bigrams (2-grams) include “Movies are,” “are fun,” “fun for,” and “for everyone.” By preserving sequences of tokens, n-grams retain contextual information that individual tokens cannot capture.

However, using n-grams comes at a cost. The vocabulary expands considerably, increasing the computational cost of model training. Additionally, the model requires larger datasets to effectively learn weights for the expanded set of possible n-grams.

Another limitation of bag-of-words is how it handles out-of-vocabulary words. When a word appears during inference that wasn’t present during training—and thus isn’t in the vocabulary—it can’t be represented in the feature vector. Similarly, the approach struggles with synonyms and near-synonyms. Words like “movie” and “film” are processed as completely distinct terms, forcing the model to learn separate parameters for each. Since labeled data is often costly to obtain, resulting in rather small labeled datasets, it would be more efficient if the model could recognize and collectively process words with similar meanings.

Word embeddings address this by mapping semantically similar words to similar vectors.

2.2. Word Embeddings

Consider document 3 (“Enjoy a great movie today.”) from earlier. We can break down this bag of words (BoW) into one-hot vectors representing individual words:

BoW	1	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
enjoy	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
great	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
movie	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
today	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

As we see, a bag-of-word vector of a document is a sum of one-hot vectors of its words. Now, let's examine the one-hot vectors and the BoW vector for the text "Films are my passion.":

BoW	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
films	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
are	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
my	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
pas-sion	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

There are two key problems here. First, even when a word exists in the training data and vocabulary, one-hot encoding reduces it to a single 1 in a vector of zeros, giving the classifier almost no meaningful information to learn from.

Second, in the above document, most one-hot encoded word vectors add no value since three out of four become **zero vectors**—representing words missing from the vocabulary.

A better approach would let the model understand that "films," though unseen in training, shares semantic meaning with "movies." This would allow the feature vector for "films" to be processed similarly to "movies." Such an approach requires word representations that capture semantic relationships between words.

Word embeddings overcome the limitations of the bag-of-words model by representing words as **dense vectors** rather than **sparse** one-hot vectors. These lower-dimensional representations contain mostly non-zero values, with similar words having embeddings that exhibit high **cosine similarity**. The embeddings are learned from vast unlabeled datasets spanning millions to hundreds of millions of documents.

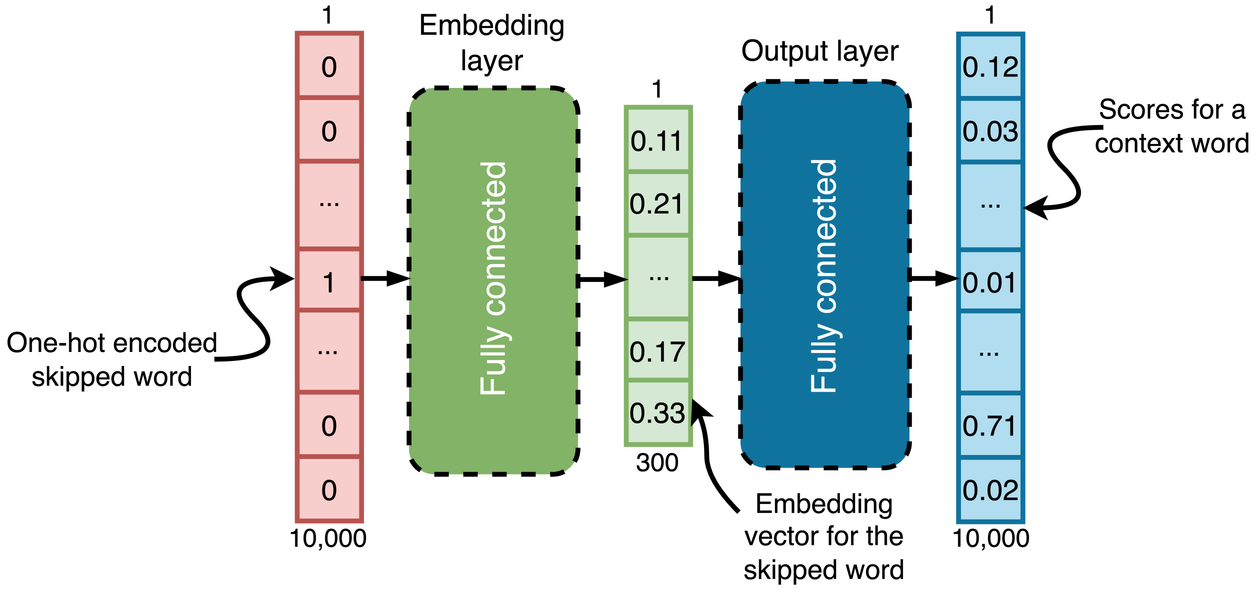
Word2vec, a widely-used embedding learning algorithm, exists in two variants. We'll examine the skip-gram formulation.

Skip-grams are word sequences where one word is omitted. For example, in *"Professor Alan Turing's * advanced computer science,"* the missing word (marked as *) might be "research," "work," or "theories"—words that fit contextually despite not being exact synonyms. Training a model to predict these skipped words from their surrounding context helps it learn semantic relationships between words. The process can also work in reverse: the skipped word can be used to predict its context words. This is the basis of the **skip-gram algorithm**.

The skip-gram size specifies how many context words are included. For a size of five, this means two words before and two after the skipped word. Here are examples of skip-grams of size five from our sentence, with different words skipped (marked as *):

Skip-gram	Skipped word
professor alan * research advanced	turing's
alan turing's * advanced computer	research
turing's research * computer science	advanced

If the corpus vocabulary contains 10,000 words, the skip-gram model with an embedding layer of 300 units, is depicted below:



This is a skip-gram model with a skip-gram size of 5 and the embedding layer of 300 units. As you can see, the model uses a one-hot encoded skipped word to predict a context word, processing the input through two consecutive fully connected layers. It doesn't predict all context words at once but makes separate predictions for each.

Here's how it works for the skip-gram `*professor alan * research advanced"* and the skipped word "turing's". We transform the skip-gram into 4 training pairs:

Skipped word (input)	Context word (target)	Position
turing's	professor	−2
turing's	alan	−1
turing's	research	+1
turing's	advanced	+2

For each pair of skipped and context words, say (turing's, professor), the model:

1. Takes "turing's" as input,
2. Converts it to a one-hot vector,
3. Passes it through the embedding layer to get the word embedding,
4. Passes the word embedding through the output layer, and
5. Outputs probabilities for "professor."

For a given context word, the output layer produces a probability vector across the vocabulary. Each value represents how likely that vocabulary word is to be the context word.

A curious reader might notice: if the input for each training pair remains constant—say, "turing's"—why would the output differ? That's a great observation! The output will indeed be identical for the same input. However, the loss calculation varies depending on each context word.

> When using **chat language models**, you may notice that the same question often yields different answers. While this might suggest the model is non-deterministic, that's not accurate. An LLM is fundamentally a neural network, similar to a skip-gram model but with far more parameters. The apparent randomness comes from the way these models are *used* to generate text. During generation, words are sampled based on their predicted probabilities. Though higher-probability words are more likely to be chosen, lower-probability ones may still be selected. This sampling process creates the variations we observe in responses. We will talk about sampling in Chapter 5.

The skip-gram model uses **cross-entropy** as its loss function, just as in the three-class text classifier discussed earlier but handles 10,000 classes—one for each word in the vocabulary. For each skip-gram in the training set, the model computes losses separately for each context word, such as the four words surrounding "turing's," then averages these losses to receive feedback on all context word predictions simultaneously.

This training approach enables the model to capture meaningful word relationships, even when working with the same input across different training pairs.

Here's an example. For the input word "turing's," suppose the model assigns these probabilities to different vocabulary words: professor (0.1), alan (0.15), research (0.2), advanced (0.05). When training the model, each input-target word pair contributes to the loss function. For example, when "turing's" appears with "professor" in the training data, the loss works to increase the score of 0.1. Similarly, when paired with "alan," the loss works to increase 0.15, with "research" to increase 0.2, and with "advanced" to increase 0.05.

During backpropagation, the model adjusts its weights to make these scores higher for the given context words. For instance, the updated scores might be: professor: 0.11, alan: 0.17, research: 0.22, advanced: 0.07, while the scores for other vocabulary words decrease slightly.

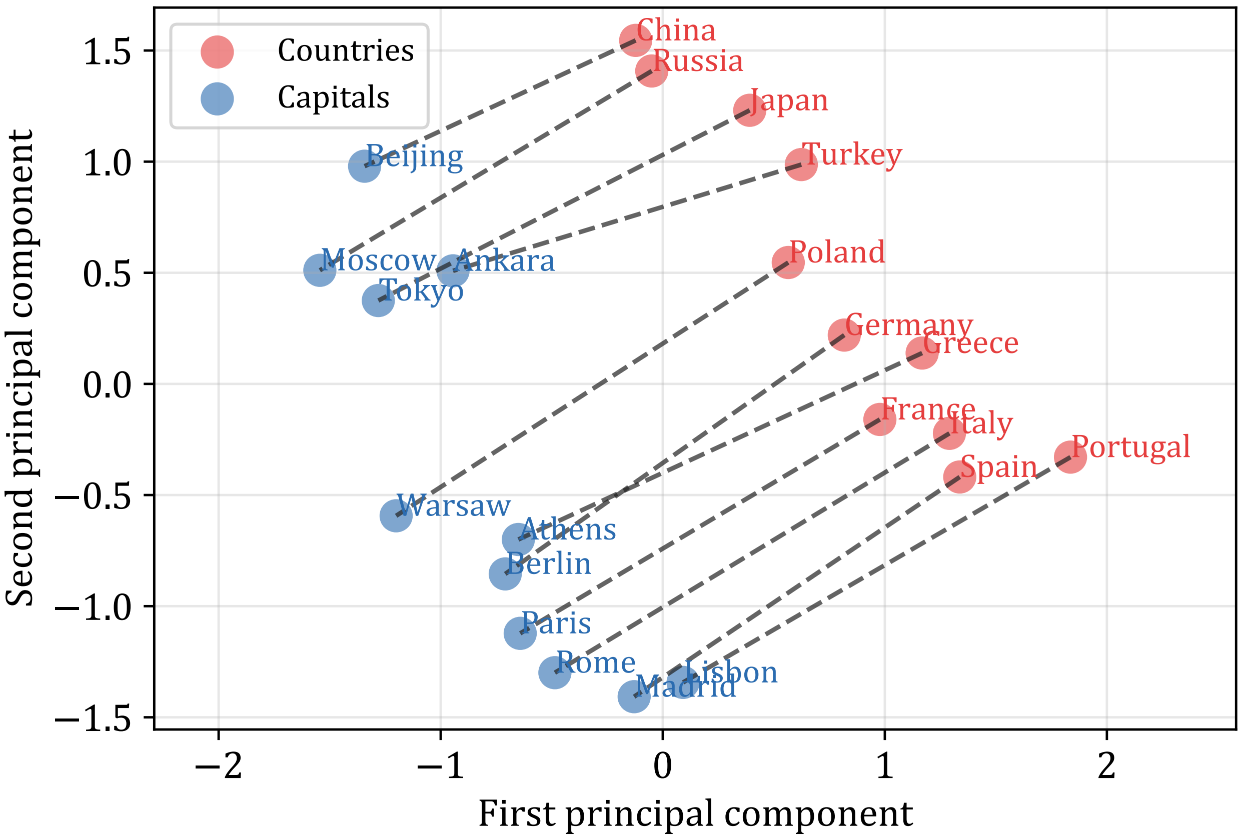
Once training is complete, the output layer is discarded. The embedding layer then serves as the new output layer. When given a one-hot encoded input word, the model produces a 300-dimensional vector—this is the word embedding.

Word2vec is just one method for learning word embeddings from large text corpora. Other methods, such as **GloVe** and **FastText**, offer alternative approaches, focusing on capturing global co-occurrence statistics or subword information to create more robust embeddings.

Using word embeddings to represent texts offers clear advantages over bag of words. One advantage is **dimensionality reduction**, which compresses the word representation from the size of the vocabulary (as in one-hot encoding) to a small vector, typically between 100 and 1000 dimensions. This makes it feasible to process very large corpora in machine learning tasks.

Semantic similarity is another advantage of word embeddings. Words with similar meanings are mapped to vectors that are close to each other in the embedding space. For example, consider word2vec embeddings trained by

Google on a news corpus containing about 100 billion words.[4] In the graph below, “Moscow” and “Beijing,” or “Russia” and “China,” are represented by points located near one another. This reflects their semantic relationships:



The graph shows a 2D projection of 300-dimensional embedding vectors for countries and their capitals. Words with related meanings cluster together, while nearly parallel lines connect cities to their respective countries, revealing their semantic relationships.

The skip-gram model captures semantic similarity when words occur in similar contexts, even without direct co-occurrence. For instance, if the model produces different probabilities for “films” and “movies,” the loss function drives it to predict similar ones, since context words frequently overlap. Through backpropagation, the embedding layer outputs for these words converge.

[4] These embeddings can be found online by using the “GoogleNews-vectors-negative300.bin.gz” query. A backup is available on the book’s wiki at thelmbook.com/data/word-vectors.

> Before word embeddings, **WordNet** (created at Princeton in 1985) attempted to capture word relationships by organizing words into sets of synonyms and recording semantic links between them. While effective, these hand-crafted mappings couldn't scale to large vocabularies or capture the subtle patterns in word usage that naturally emerge from embedding-based approaches.

Because directly visualizing 300-dimensional vectors isn't possible, we used a **dimensionality reduction** technique called **principal component analysis** (**PCA**) to project them onto two dimensions, known as first and second **principal components**.

Dimensionality reduction algorithms compress high-dimensional vectors while maintaining their relationships. The first and second principal components in the above graph preserved the semantic connections between words, revealing their relationships.

> For resources on PCA and other dimensionality reduction methods, check the recommended material listed on the book's wiki.

Word embeddings capture the meaning of words and their relationships to other words. They are fundamental to many natural language processing (NLP) tasks. Neural language models, for example, encode documents as matrices of word embeddings. Each row corresponds to a word's embedding vector, and its position in the matrix reflects the word's position in the document.

> The discovery that word2vec embeddings support meaningful arithmetic operations (like "king − man + woman ≈ queen") was a pivotal moment, revealing that neural networks could encode semantic relationships in a space where vector operations produced changes in word meaning. This made the invention of neural networks capable of doing complex math on words, like large language models do, only a matter of time.

Modern language models, though, often use subwords—tokens smaller than complete words. Before moving on to language models—the main topic of this book—let's first examine byte-pair encoding, a widely used subword tokenization method.

2.3. Byte-Pair Encoding

Byte-pair encoding (**BPE**) is a tokenization algorithm that addresses the challenges of handling out-of-vocabulary words by breaking words into smaller units called **subwords**.

Initially a data compression technique, BPE was adapted for NLP by treating words as sequences of characters. It merges the most frequent symbol pairs—characters or subwords—into new subword units. This continues until the vocabulary reaches the target size.

Below is the basic BPE algorithm:

1. **Initialization**: Use a text corpus. Split each word in the corpus into individual characters. For example, the word "hello" becomes "h e l l o". The initial vocabulary consists of all unique characters in the corpus.
2. **Iterative merging**:
    - **Count adjacent symbol pairs**: Treat each character as a **symbol**. Go through the corpus and count every pair of adjacent symbols. For example, in "h e l l o", the pairs are "h e", "e l", "l l", "l o".
    - **Select the most frequent symbol pair**: Identify the pair with the highest count in the entire corpus. For instance, if "l l" occurs most frequently, select it.
    - **Merge the selected pair**: Replace all occurrences of the most frequent symbol pair with a new single **merged symbol**. For example, "l l" would be replaced with a new merged symbol "ll". The word "h e l l o" now becomes "h e ll o".
    - **Update the vocabulary**: Add the new merged symbol to the vocabulary. The vocabulary now includes the original characters and the new symbol "ll".
3. **Repeat**: Continue the iterative merging until the vocabulary reaches the desired size.

The algorithm is simple, but implementing it directly on large corpora is inefficient. Recomputing symbol pairs or updating the entire corpus after each merge is computationally expensive.

A more efficient approach initializes the vocabulary with all unique words in the corpus and their counts. Pair counts are calculated using these word counts, and the vocabulary is updated iteratively by merging the most popular pairs. Let's write the code:

```
from collections import defaultdict

def initialize_vocabulary(corpus):
    vocabulary = defaultdict(int)
    charset = set()
    for word in corpus:
        word_with_marker = '_' + word ❶
        characters = list(word_with_marker) ❷
        charset.update(characters) ❸
        tokenized_word = ' '.join(characters) ❹
        vocabulary[tokenized_word] += 1 ❺
    return vocabulary, charset
```

The function generates a vocabulary that represents words as sequences of characters and tracks their counts. Given a `corpus` (a list of words), it returns two outputs: `vocabulary`, a dictionary mapping each word—tokenized with spaces between characters—to its count, and `charset`, a set of all unique characters present in the corpus.

Here's how it works:

- Line ❶ adds a word boundary marker "_" to the start of each word to differentiate subwords at the beginning from those in the middle. For example, "_re" in "restart" is distinct from "re" in "agree." This helps rebuild sentences from tokens generated using the model. When a token starts with "_", it marks the beginning of a new word, requiring a space to be added before it.
- Line ❷ splits each word into individual `characters`.
- Line ❸ updates `charset` with any new characters encountered in the word.
- Line ❹ joins `characters` with spaces to create a tokenized version of the word. For example, the word "hello" becomes `_ h e l l o`.

- Line ❺ adds `tokenized_word` to `vocabulary` with its count incremented.

After the initialization, BPE iteratively merges the most frequent pairs of tokens (bigrams) in the `vocabulary`. By removing spaces between these pairs, it forms progressively longer tokens.

```
def get_pair_counts(vocabulary):
    pair_counts = defaultdict(int)
    for tokenized_word, count in vocabulary.items():
        tokens = tokenized_word.split() ❶
        for i in range(len(tokens) - 1):
            pair = (tokens[i], tokens[i + 1]) ❷
            pair_counts[pair] += count ❸
    return pair_counts
```

The function counts how often adjacent token pairs appear in the tokenized vocabulary words. The input `vocabulary` maps tokenized words to their counts, and the output is a dictionary of token pairs and their total counts.

For each `tokenized_word` in `vocabulary`, we split it into tokens in line ❶. A nested loop forms adjacent token pairs in line ❷ and increments their count by the word's count in line ❸.

```
def merge_pair(vocabulary, pair):
    new_vocabulary = {}
    bigram = re.escape(' '.join(pair)) ❶
    pattern = re.compile(r"(?<!\S)" + bigram + r"(?!\S)") ❷
    for tokenized_word, count in vocabulary.items():
        new_tokenized_word = pattern.sub("".join(pair), token
ized_word) ❸
        new_vocabulary[new_tokenized_word] = count
    return new_vocabulary
```

The function merges the input token pair in all tokenized words from the vocabulary. It returns a new vocabulary where every occurrence of the `pair` is merged into a single token. For example, if the pair is `('e', 'l')` and a tokenized word is `"_ h e l l o"`, merging `'e'` and `'l'` removes the space between them, resulting in `"_ h el l o"`.

In line ❶, the `re.escape` function automatically adds backslashes to special characters in a string (like `.`, `*`, or `?`), so they are interpreted as literal characters rather than having their special meaning in regular expressions.

The regular expression in line ❷ matches only whole token pairs. It ensures the `bigram` is not part of a larger word by checking for the absence of non-whitespace characters immediately before and after the match. For instance `"good morning"` matches in `"this is good morning"`, but not in `"thisisgood morning"`, where `"good"` is part of `"thisisgood"`.

> The expressions `(?<!\S)` and `(?!\S)` are regex **negative lookbehind** and **negative lookahead** assertions that ensure a `bigram` stands alone. The lookbehind checks that no non-whitespace character precedes the `bigram`, meaning it follows whitespace or the start of text. The lookahead similarly ensures no non-whitespace follows the `bigram`, meaning it precedes whitespace or the end of text. Together, these prevent the `bigram` from being part of longer words.

Finally, in line ❸, the function uses `pattern.sub()` to replace all occurrences of the matched pattern with the joined pair, creating the new tokenized word.

The function below implements the BPE algorithm, merging the most frequent token pairs iteratively until no merges remain or the target vocabulary size is reached:

```
def byte_pair_encoding(corpus, vocab_size):
    vocabulary, charset = initialize_vocabulary(corpus)
    merges = []
    tokens = set(charset)
    while len(tokens) < vocab_size: ❶
        pair_counts = get_pair_counts(vocabulary)
        if not pair_counts: ❷
            break
        most_frequent_pair = max(pair_counts, key=pair_counts
.get) ❸
        merges.append(most_frequent_pair)
        vocabulary = merge_pair(vocabulary, most_frequent_pai
r) ❹
```

```
        new_token = ''.join(most_frequent_pair) ➎
        tokens.add(new_token) ➏

    return vocabulary, merges, charset, tokens
```

This function processes a corpus to produce the components needed for a tokenizer. It initializes the vocabulary and character set, creates an empty `merges` list for storing merge operations, and sets `tokens` to the initial character set. Over time, `tokens` grows to include all unique tokens the tokenizer will be able to generate.

The loop in line ➊ continues until the number of tokens supported by the tokenizer reaches `vocab_size` or no pairs remain to merge. Line ➋ checks if there are no more valid pairs, in which case the loop exits. Line ➌ finds the most frequent token pair, which is merged throughout the vocabulary in line ➍ to create a new token in line ➎. This new token is added to the `tokens` set in line ➏, and the merge is recorded in `merges`.

The function returns four outputs: the updated vocabulary, the list of merge operations, the original character set, and the final set of unique tokens.

The function below tokenizes a word using a trained tokenizer:

```
def tokenize_word(word, merges, vocabulary, charset, unk_toke
n="<UNK>"):
    word = '_' + word
    if word in vocabulary:
        return [word]
    tokens = [char if char in charset else unk_token for char
in word]

    for left, right in merges:
        i = 0
        while i < len(tokens) - 1:
            if tokens[i:i+2] == [left, right]:
                tokens[i:i+2] = [left + right]
            else:
                i += 1
    return tokens
```

This function tokenizes a `word` using `merges`, `vocabulary`, and `charset` from `byte_pair_encoding`. The `word` is first prefixed. If the prefixed `word` exists in the `vocabulary`, it returns it as the only token. Otherwise, the `word` is split into characters, with any not in `charset` replaced by `unk_token`. These characters are then iteratively merged using the order of rules in `merges`.

To tokenize a text, we first split it into words based on spaces and then tokenize each word individually. The thelmbook.com/nb/2.1 notebook contains code for training a BPE tokenizer by using a news corpus. The tokenized version of the sentence *"Let's proceed to the language modeling chapter."* using the tokenizer trained in the notebook, is:

```
["_Let", "'", "s", "_proceed", "_to", "_the", "_language", "_
model", "ing", "_part", "."]
```

Here, "let's," and "modeling," were broken into subwords. This indicates their relative rarity in the training data and a small target vocabulary size (I set 5000 tokens).

The `tokenize_word` algorithm is inefficient due to nested loops: it iterates over all merges in line ❹ while checking every token pair in line ❺. However, since modern language models have vocabularies exceeding 100,000 tokens, most input words exist in the vocabulary, bypassing subword tokenization. The notebook's optimized version uses caching and precomputed data structures to eliminate these nested loops, reducing tokenization time from 0.0549 to 0.0037 seconds. While actual performance varies by system, the optimized approach consistently delivers better speed.

For languages without spaces, like Chinese, or for multilingual models, the initial space-based tokenization is typically skipped. Instead, the text is split into individual characters. From there, BPE proceeds as usual, merging the most frequent character or token pairs to form subwords.

We're now ready to examine the core ideas of language modeling. We'll begin with traditional count-based methods and cover neural network-based techniques in later chapters.

2.4. Language Model

A **language model** predicts the next token in a sequence by estimating its conditional probability based on previous tokens. It assigns a probability to all possible next tokens, enabling the selection of the most likely one. This capability supports tasks like text generation, machine translation, and speech recognition. Trained on large unlabeled text corpora, language models learn statistical patterns in language, allowing them to be used to generate human-like text.

Formally, for a sequence $\mathbf{s}$ of L tokens $(t_1, t_2, \dots, t_L)$, a language model computes:

$$\Pr\big(t = t_{L+1} | \mathbf{s} = (t_1, t_2, \dots, t_L)\big) \tag{2.2}$$

Here, Pr represents the conditional probability distribution over the vocabulary for the next token. A **conditional probability** quantifies the likelihood of one event occurring given that another has already occurred. In language models, it reflects the probability of a specific token being the next one, given the preceding sequence of tokens. This sequence is often referred to as the **input sequence**, **context**, or **prompt**.

The following notations are equivalent to Equation 2.2:

$$\Pr(t_{L+1} | t_1, t_2, \dots, t_L) \text{ or } \Pr(t_{L+1} | \mathbf{s}) \tag{2.3}$$

We will select different notations, ranging from concise to detailed, based on the context.

For any token t and sequence $\mathbf{s}$, the conditional probability satisfies $\Pr(t|\mathbf{s}) \geq 0$, meaning probabilities are always non-negative. Furthermore, the probabilities for all possible next tokens in the vocabulary $\mathcal{V}$ must sum to 1: $\sum_{t \in \mathcal{V}} P\,(t|\mathbf{s}) = 1$. This ensures the model outputs a valid **discrete probability distribution** over the vocabulary.

To illustrate, let's consider an example with a vocabulary $\mathcal{V}$ containing 5 words: "are," "cool," "language," "models," and "useless." For the sequence $\mathbf{s} =$ (language, models, are), a language model could output the following probabilities for each possible next word in $\mathcal{V}$:

$$\begin{aligned}
\Pr\big(t = \text{are} | \mathbf{s} = (\text{language, models, are})\big) &= 0.01 \\
\Pr\big(t = \text{cool} | \mathbf{s} = (\text{language, models, are})\big) &= 0.77 \\
\Pr\big(t = \text{language} | \mathbf{s} = (\text{language, models, are})\big) &= 0.02 \\
\Pr\big(t = \text{models} | \mathbf{s} = (\text{language, models, are})\big) &= 0.15 \\
\Pr\big(t = \text{useless} | \mathbf{s} = (\text{language, models, are})\big) &= 0.05
\end{aligned}$$

The illustration demonstrates how the language model assigns probabilities across its vocabulary for each potential next word, with “cool” receiving the highest probability. These probabilities sum to 1, forming a valid discrete probability distribution.

This type of model is an **autoregressive language model**, also known as a **causal language model**. **Autoregression** involves predicting an element in a sequence using only its predecessors. Such models excel at text generation and include Transformer-based **chat language models** (chat LMs) and all language models discussed in this book.

In contrast, **masked language models**, such as BERT—a pioneering Transformer-based model—use a different approach. These models predict intentionally masked tokens within sequences, utilizing both preceding and following context. This bidirectional approach particularly suits tasks like text classification and named entity recognition.

Before neural networks became standard for language modeling, traditional methods relied on statistical techniques. These count-based models, still used in smartphone autocomplete, estimate the probability of word sequences based on word or n-gram frequency counts learned from a corpus. To understand these methods better, let's implement a simple count-based language model.

2.5. Count-Based Language Model

We’ll focus on a trigram model ($n = 3$) to illustrate how this works. In a trigram model, the probability of a token is calculated based on the two preceding tokens:

$$\Pr(t_i|t_{i-2}, t_{i-1}) = \frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}, \tag{2.4}$$

where $C(\cdot)$ denotes the count of occurrences of an n-gram in the training data.

For instance, if the trigram “language models rock” appears 50 times in the corpus and “language models” appears 200 times overall, then:

$$\Pr(\text{rock}|\text{language, models}) = \frac{50}{200} = 0.25$$

This means that “rock” follows “language models” 25% of the time in our training data.

Equation 2.4 is the **maximum likelihood estimate** (MLE) of a token's probability given its context. It measures the relative frequency of a trigram compared to all trigrams sharing the same two-token history. With a larger training corpus, the MLE becomes more reliable for n-grams that occur frequently. This aligns with a basic statistical principle: larger datasets yield more accurate estimates.

However, a limited-size corpus poses a problem: some n-grams we may encounter in practice might not appear in the training data. For instance, if the trigram "language models sing" never appears in our corpus, its probability would be zero according to the MLE:

$$\Pr(\text{sing}|\text{language, models}) = \frac{0}{200} = 0$$

This is problematic because it assigns a zero probability to any sequence containing an unseen n-gram, even if it's a valid phrase. To solve this, several techniques exist, one of which is **backoff**. The idea is simple: if a higher-order n-gram (e.g., trigram) is not observed, we "back off" to a lower-order n-gram (e.g., bigram). The probability $\Pr(t_i|t_{i-2}, t_{i-1})$ is given by one of the following expressions, depending on whether the condition is true:

Expression	**Condition**
$\frac{C(t_{i-2}, t_{i-1}, t_i)}{C(t_{i-2}, t_{i-1})}$	if $C(t_{i-2}, t_{i-1}, t_i) > 0$
$\Pr(t_i|t_{i-1})$	if $C(t_{i-2}, t_{i-1}, t_i) = 0$ and $C(t_{i-1}, t_i) > 0$
$\Pr(t_i)$	otherwise

Here, $C(t_{i-2}, t_{i-1}, t_i)$ is the count of the trigram (t_{i-2}, t_{i-1}, t_i), $C(t_{i-2}, t_{i-1})$ and $C(t_{i-1}, t_i)$ are the counts of the bigrams (t_{i-2}, t_{i-1}) and (t_{i-1}, t_i) respectively.

The bigram probability and unigram probability are computed as:

$$\Pr(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}, \quad \Pr(t_i) = \frac{C(t_i) + 1}{W + V},$$

where $C(t_i)$ is the count of the token t_i, W is the total number of tokens in the corpus, and V is the vocabulary size.

Adding 1 to $C(t_i)$, known as **add-one smoothing** or **Laplace smoothing**, addresses zero probabilities for tokens not present in the corpus. If we used the actual frequency $\Pr(t_i) = \frac{C(t_i)}{W}$, any token not found in the corpus would have a zero probability, creating problems when the model encounters valid but unseen tokens. Laplace smoothing solves this by adding 1 to each token count, ensuring all tokens, including unseen ones, receive a small, non-zero probability. The denominator is adjusted by adding V to account for the extra counts introduced in the numerator.

Now, let's implement a language model with backoff in the `CountLanguageModel` class (we'll implement Laplace smoothing in the next section):

```
class CountLanguageModel:
    def __init__(self, n): ❶
        self.n = n
        self.ngram_counts = [{} for _ in range(n)] ❷
        self.total_unigrams = 0

    def predict_next_token(self, context): ❸
        for n in range(self.n, 1, -1): ❹
            if len(context) >= n - 1: ❺
                context_n = tuple(context[-(n - 1):]) ❻
                counts = self.ngram_counts[n - 1].get(context
_n)
                if counts:
                    return max(counts.items(), key=lambda x:
x[1])[0]
        unigram_counts = self.ngram_counts[0].get(())
        if unigram_counts:
            return max(unigram_counts.items(), key=lambda x:
x[1])[0]
        return None
```

In line ❶, the model is initialized with an `n` parameter, defining the maximum n-gram order (e.g., `n=3` for trigrams). The `ngram_counts` list in line ❷ stores n-gram frequency dictionaries for unigrams, bigrams, trigrams, etc., populated during training. For `n=3`, given the corpus *"Language models are powerful. Language models are useful."* in lowercase with punctuation removed, `self.ngram_counts` would contain:

```
ngram_counts[0] = {(): {"language": 2, "models": 2, "are": 2,
"powerful": 1, "useful": 1}}

ngram_counts[1] = {("language",): {"models": 2}, ("models",):
{"are": 2}, ("are",): {"powerful": 1, "useful": 1}, ("powerfu
l",): {"language": 1}}

ngram_counts[2] = {("language", "models"): {"are": 2}, ("mode
ls", "are"): {"powerful": 1, "useful": 1}, ("are", "powerful"
): {"language": 1}, ("powerful", "language"): {"models": 1}}
```

The `predict_next_token` method uses backoff to predict the next token. Starting from the highest n-gram order in line ❹, it checks if the context contains enough tokens for this n-gram order in line ❺. If so, it extracts the context in line ❻ and attempts to find a match in `ngram_counts`. If no match is found, it backs off to lower-order n-grams or defaults to unigram counts. For instance, given `context=["language", "models", "are"]` and `n=3`:

- First iteration: `context_n = ("models", "are")`
- Second iteration (if needed): `context_n = ("are",)`
- Last resort: unigram counts with empty tuple key `()`

If a matching context is found, the method returns the token with the highest count for that context. For input `["language", "models"]` it will return `"are"`, the token with highest count among values for the key `("language", "models")` in `ngram_counts[2]`. However, for the input `["english", "language"]` it will not find the key `("english", "language")` in `ngram_counts[2]`, so it will backoff to `ngram_counts[1]` and return `"models"`, the token with highest count among values for the key `("language",)`.

Now, let's define the method that trains our model:

```
def train(model, tokens):
    model.total_unigrams = len(tokens)
    for n in range(1, model.n + 1): ❶
        counts = model.ngram_counts[n - 1]
        for i in range(len(tokens) - n + 1):
            context = tuple(tokens[i:i + n - 1]) ❷
            next_token = tokens[i + n - 1] ❸
            if context not in counts:
                counts[context] = defaultdict(int)
            counts[context][next_token] = counts[context][nex
t_token] + 1
```

The `train` method takes a model (an instance of `CountLanguageModel`) and a list of tokens (the training corpus) as input. It updates the n-gram counts in the model using these tokens.

In line ❶, the method iterates over n-gram orders from 1 to `model.n` (inclusive). For each `n`, it generates n-grams of that order from the token sequence and counts their occurrences.

Lines ❷ and ❸ extract contexts and their subsequent tokens to build a nested dictionary where each context maps to a dictionary of following tokens and their counts. These counts are stored in `model.ngram_counts`, which the `predict_next_token` method later uses to make predictions based on context.

Now, let's train the model:

```
set_seed(42)
n = set_hyperparameters()
data_url = "https://www.thelmbook.com/data/brown"
train_corpus, test_corpus = download_and_prepare_data(data_ur
l)

model = CountLanguageModel(n)
train(model, train_corpus)

perplexity = compute_perplexity(model, test_corpus)
print(f"\nPerplexity on test corpus: {perplexity:.2f}")

contexts = [
    "i will build a",
    "the best place to",
    "she was riding a"
]

for context in contexts:
    words = tokenize(context)
    next_word = model.predict_next_token(words)
    print(f"\nContext: {context}")
    print(f"Next token: {next_word}")
```

The full implementation of this model, including the methods to retrieve and process the training data, can be found in the thelmbook.com/nb/2.2 notebook. Within the `download_and_prepare_data` method, the corpus is downloaded, converted to lowercase, tokenized into words, and divided into **training** and **test** partitions with a 90/10 split. Let's take a moment to understand why this last step is critical.

In machine learning, using the entire dataset for training leaves no way to evaluate whether the model **generalizes** well. A frequent issue is **overfitting**, where the model excels on training data but struggles to make accurate predictions on unseen, new data.

Partitioning the dataset into training and test sets is a standard practice to control overfitting. It involves two steps: (1) shuffling the data and (2) splitting it into two subsets. The larger subset, called the training data, is used for training the model, while the smaller subset, called the test data, is used to evaluate the model's performance on unseen examples.

> The test set requires sufficient size to reliably estimate model performance. A test ratio of 0.1 to 0.3 (10% to 30% of the entire dataset) is common, though this varies with dataset size. For very large datasets, even a smaller test set ratio results in enough examples to provide reliable performance estimates.

The training data comes from the **Brown Corpus**, a collection of over 1 million words from American English texts published in 1961. This corpus is frequently used in linguistic studies.

When you run the code, you will see the following output:

```
Perplexity on test corpus: 299.06

Context: i will build a
```

```
Next word: wall

Context: the best place to
Next word: live

Context: she was riding a
Next word: horse
```

Ignore the perplexity number for now; we'll discuss it shortly. Count-based language models can produce reasonable immediate continuations, making them good for autocomplete systems. However, they have notable limitations. These models generally work with word-tokenized corpora, as their n-gram size is typically small (up to $n = 5$). Extending beyond this would require too much memory and lead to slower processing. Subword tokenization, while more efficient, results in many n-grams that represent only fragments of words, degrading the quality of next-word predictions.

Word-level tokenization creates another significant drawback: count-based models cannot handle out-of-vocabulary (OOV) words. This is similar to the issue seen in the **bag-of-words** approach discussed in Section 2.1. For example, consider the context: *"according to WHO, COVID-19 is a."* If "COVID-19" wasn't in the training data, the model would back off repeatedly until it relies only on *"is a,"* severely limiting the context for meaningful predictions.

Count-based models are also unable to capture long-range dependencies in language. While modern Transformer models can handle thousands of tokens, training a count-based model with a context of 1000 tokens would require storing counts for all n-grams from $n = 1$ to $n = 1000$, requiring prohibitive amounts of memory.

Additionally, these models cannot be adapted for downstream tasks after training, as their n-gram counts are fixed, and any adjustment requires retraining on new data.

These limitations have led to the development of advanced methods, particularly neural network-based language models, which have largely replaced count-based models in modern natural language processing. Approaches like recurrent neural networks and transformers, which we'll discuss in the next two chapters, handle longer contexts effectively, producing coherent and context-aware text. Before exploring these methods, let's look at how to evaluate a language model's quality.

2.6. Evaluating Language Models

Evaluating language models measures their performance and allows comparing models. Several metrics and techniques are commonly used. Let's look at the main ones.

2.6.1. Perplexity

Perplexity is a widely used metric for evaluating language models. It measures how well a model predicts a text. Lower perplexity values indicate a better model—one that is more confident in its predictions. Perplexity is defined as the exponential of the average **negative log-likelihood** per token in the test set:

$$\text{Perplexity}(\mathcal{D}, k) = \exp\left(-\frac{1}{D}\sum_{i=1}^{D} \log \Pr\left(t_i | t_{\max(1,i-k)}, \dots, t_{i-1}\right)\right) \quad (2.5)$$

Here, $\mathcal{D}$ represents the test set, D is the total number of tokens in it, t_i is the i^{th} token, and $\Pr\left(t_i | t_{\max(1,i-k)}, \dots, t_{i-1}\right)$ is the probability the model assigns to t_i given its preceding context window of size k, where $\max(1, i - k)$ ensures we start from the first token when there aren't enough preceding tokens to fill the context window. The notations $\exp(x)$ and e^x, where e is **Euler's number**, are equivalent.

The negative log-likelihood (NLL) in Equation 2.5 is the negative logarithm of the probabilities our language model assigns. When a model processes text like "*language models are*" and assigns a probability of 0.77 to the next word "cool", the NLL would be $-\log(0.77)$. It's called "negative" log-likelihood because we take the negative of the logarithm, and "likelihood" refers to these conditional probabilities the model computes. In language modeling, NLL serves two purposes: it acts as a loss function during training to help models learn better probability distributions (which we'll see in the next chapter when training a recurrent neural network language model), and as shown in the perplexity formula, it helps us evaluate how well models predict text.

Perplexity can be understood more intuitively through its geometric mean formulation. The geometric mean of a set of numbers is the D^{th} root of their product (where D is the number of values), and perplexity is the geometric mean of the inverse probabilities:

$$\text{Perplexity}(\mathcal{D}, k) = \left(\prod_{i=1}^{D} \frac{1}{\Pr\left(t_i | t_{\max(1, i-k)}, \ldots, t_{i-1}\right)}\right)^{\frac{1}{D}}$$

This form shows that perplexity represents the weighted average factor by which the model is "perplexed" when predicting each token. A perplexity of 10 means that, on average, the model is as uncertain as if it had to choose uniformly between 10 possibilities at each step.

> If a language model assigns equal probability to every token in a vocabulary of size V, its perplexity equals V. This provides an intuitive upper bound for perplexity—a model cannot be more uncertain than when it assigns equal likelihood to all possible tokens.

While both formulations of perplexity shown above are mathematically equivalent (proof available on the book's wiki), the exponential form is computationally more convenient as it transforms products into sums through the logarithm, making calculations more numerically stable.

Let's calculate perplexity using the example text with word-level tokenization and ignoring punctuation: *"We are evaluating a language model for English."* To keep things simple, we assume a context of up to three words. We begin by determining the probability of each word based on its preceding context of three words as provided by the model. Here are the probabilities:

$$\begin{aligned}
\Pr(\text{We}) &= 0.10 \\
\Pr(\text{are} \mid \text{We}) &= 0.20 \\
\Pr(\text{evaluating} \mid \text{We, are}) &= 0.05 \\
\Pr(\text{a} \mid \text{We, are, evaluating}) &= 0.50 \\
\Pr(\text{language} \mid \text{are, evaluating, a}) &= 0.30 \\
\Pr(\text{model} \mid \text{evaluating, a, language}) &= 0.40 \\
\Pr(\text{for} \mid \text{a, language, model}) &= 0.15 \\
\Pr(\text{English} \mid \text{language, model, for}) &= 0.25
\end{aligned}$$

Using the probabilities, we compute the negative log-likelihood for each word:

$$\begin{aligned}
-\log\big(P(\text{We})\big) &= -\log(0.10) \approx 2.30\\
-\log\big(P(\text{are} \mid \text{We})\big) &= -\log(0.20) \approx 1.61\\
-\log\big(P(\text{evaluating} \mid \text{We, are})\big) &= -\log(0.05) \approx 3.00\\
-\log\big(P(\text{a} \mid \text{We, are, evaluating})\big) &= -\log(0.50) \approx 0.69\\
-\log\big(P(\text{language} \mid \text{are, evaluating, a})\big) &= -\log(0.30) \approx 1.20\\
-\log\big(P(\text{model} \mid \text{evaluating, a, language})\big) &= -\log(0.40) \approx 0.92\\
-\log\big(P(\text{for} \mid \text{a, language, model})\big) &= -\log(0.15) \approx 1.90\\
-\log\big(P(\text{English} \mid \text{language, model, for})\big) &= -\log(0.25) \approx 1.39
\end{aligned}$$

Next, we sum these values and divide the sum by the number of words (8) to get the average:

$$(2.30 + 1.61 + 3.00 + 0.69 + 1.20 + 0.92 + 1.90 + 1.39)/8 \approx 1.63$$

Finally, we exponentiate the average negative log-likelihood to obtain the perplexity:

$$e^{1.63} \approx 5.10$$

So, the model's perplexity on this text, using a 3-word context, is about 5.10. This means that, on average, the model acts as if it selects from roughly 5 equally likely options for each prediction.

Now, let's calculate the perplexity of the count-based model from the previous section. To do this, the model must be updated to return the probability of a token given a specific context. Add this function to the `CountLanguageModel` we implemented earlier:

```
def get_probability(self, token, context):
    for n in range(self.n, 1, -1): ❶
        if len(context) >= n - 1:
            context_n = tuple(context[-(n - 1):])
            counts = self.ngram_counts[n - 1].get(context_n)
            if counts: ❷
                total = sum(counts.values()) ❸
                count = counts.get(token, 0)
                if count > 0:
                    return count / total ❹
    unigram_counts = self.ngram_counts[0].get(()) ❺
    count = unigram_counts.get(token, 0)
    V = len(unigram_counts)
    return (count + 1) / (self.total_unigrams + V) ❻
```

The `get_probability` function is similar to `predict_next_token`. Both loop through n-gram orders in reverse (line ❶) and extract the relevant context (`context_n`). If `context_n` matches in the n-gram counts (line ❷), the function retrieves the token counts. If no match exists, it backs off to lower-order n-grams and, finally, unigrams (line ❺).

Unlike `predict_next_token`, which returns the most probable token directly, `get_probability` calculates a token's probability. In line ❸, `total` is the sum of counts for tokens following the context, acting as the denominator. Line ❹ divides the token count by `total` to compute its probability. If no higher-order match exists, line ❻ uses **add-one smoothing** with unigram counts.

The `compute_perplexity` method computes a language model's perplexity for a token sequence. It takes three arguments: the model, the token sequence, and the context size:

```
def compute_perplexity(model, tokens, context_size):
    if not tokens:
        return float('inf')
    total_log_likelihood = 0
    num_tokens = len(tokens)
    for i in range(num_tokens): ❶
        context_start = max(0, i - context_size)
        context = tuple(tokens[context_start:i]) ❷
        word = tokens[i]
        probability = model.get_probability(word, context)
        total_log_likelihood += math.log(probability) ❸
    average_log_likelihood = total_log_likelihood / num_token
s ❹
    perplexity = math.exp(-average_log_likelihood) ❺
    return perplexity
```

In line ❶, the function iterates through each token in the sequence. For every token:

- Line ❷ extracts its context, using up to `context_size` tokens before it. The expression `max(0, i - context_size)` ensures indices stay within bounds like in Equation 2.5.
- In line ❸, the log of the token's probability is added to the cumulative log-likelihood. The `get_probability` method from the model handles the probability calculation.

Once all tokens are processed, line ❹ computes the average log-likelihood by dividing the total log-likelihood by the number of tokens.

Finally, in line ❺, the perplexity is computed as the exponential of the negative average log-likelihood, as described in Equation 2.5.

By applying this method to the `test_corpus` sequence from the thelm-book.com/nb/2.2 notebook, we observe the following output:

```
Perplexity on test corpus: 299.06
```

This perplexity is very high. For example, **GPT-2** has a perplexity of about 20, while modern LLMs achieve values below 5. Later, we'll compute perplexities for RNN and Transformer-based models and compare them with the perplexity of this count-based model.

2.6.2. ROUGE

Perplexity is a standard metric used to evaluate language models trained on large, unlabeled datasets by measuring how well they predict the next token in context. These models are referred to as **pretrained models** or **base models**. As we'll discuss in the chapter on large language models, their ability to perform specific tasks or answer questions comes from **supervised finetuning**. This additional training uses a labeled dataset where input contexts are matched with target outputs, such as answers or task-specific results. This enables problem-solving capabilities.

Perplexity is not an ideal metric for evaluating a finetuned model. Instead, metrics are needed that compare the model's output to reference texts, often called **ground truth**. A common choice is **ROUGE** (Recall-Oriented Understudy for Gisting Evaluation). ROUGE is widely used for tasks like summarization and machine translation. It evaluates text quality by measuring overlaps, such as tokens or n-grams, between the generated text and the reference.

ROUGE has several variants, each focusing on different aspects of text similarity. Here, we'll discuss three widely used ones: ROUGE-1, ROUGE-N, and ROUGE-L.

ROUGE-N evaluates the overlap of n-grams between the generated and reference texts, with N indicating the length of the n-gram. One of the most commonly used versions is **ROUGE-1**.

ROUGE-1 measures the overlap of unigrams (single tokens) between the generated and reference texts. As a recall-focused metric (hence the "R" in ROUGE), it assesses how much of the reference text is captured in the generated output.

Recall is the ratio of matching tokens to the total number of tokens in the reference text:

$$\text{recall} \overset{\text{def}}{=} \frac{\text{Number of matching tokens}}{\text{Total number of tokens in reference texts}}$$

Formally, ROUGE-1 is defined as:

$$\text{ROUGE-1} \overset{\text{def}}{=} \frac{\sum_{(g,r)\in\mathcal{D}} \sum_{t\in r} \text{count}\,(t,g)}{\sum_{(g,r)\in\mathcal{D}} \text{length}\,(r)}$$

Here, $\mathcal{D}$ is the dataset of (generated text, reference text) pairs, count(t, g) counts how often a token t from the reference text r appears in the generated text g, and the denominator is the total token count across all reference texts.

To understand this calculation, consider a simple example:

Reference text	**Generated text**
Large language models are very important for text processing.	Large language models are useful in processing text.

Let's use word-level tokenization and calculate:

- **Matching words**: large, language, models, are, processing, text (6 words)
- **Total words in the reference text**: 9
- **ROUGE-1**: $\frac{6}{9} \approx 0.67$

A ROUGE-1 score of 0.67 means roughly two-thirds of the words from the reference text appear in the generated text. However, this number alone has little value. ROUGE scores are only useful for *comparing* how different language models perform on the same test set, as they indicate which model more effectively captures the content of the reference texts.

ROUGE-N extends the ROUGE metric from unigrams to n-grams while using the same formula.

ROUGE-L relies on the **longest common subsequence** (LCS). This is the longest sequence of tokens appearing in both generated and reference texts in the same order, without being adjacent.

Let g and r be the generated and reference texts with lengths L_g and L_r. Then:

$$\text{recall}_{\text{LCS}} \stackrel{\text{def}}{=} \frac{\text{LCS}(g,r)}{L_r}, \quad \text{precision}_{\text{LCS}} \stackrel{\text{def}}{=} \frac{\text{LCS}(g,r)}{L_g}$$

Here, $\text{LCS}(L_g, L_r)$ represents the number of tokens in the LCS between the generated text g and the reference text r. The **recall** measures the proportion of the reference text captured by the LCS, while the **precision** measures the proportion of the generated text that matches the reference. Recall and precision are combined into a single metric as follows:

$$\text{ROUGE-L} \stackrel{\text{def}}{=} (1+\beta^2) \times \frac{\text{recall}_{\text{LCS}} \times \text{precision}_{\text{LCS}}}{\text{recall}_{\text{LCS}} + \beta^2 \times \text{precision}_{\text{LCS}}}$$

Here, β controls the trade-off between precision and recall in the ROUGE-L score. Since ROUGE favors recall, β is usually set high, such as 8.

Let's revisit the two texts used to illustrate ROUGE-L. For these sentences, there are two valid longest common subsequences, each with a length of 5 words:

LCS 1	**LCS 2**
Large, language, models, are, text	Large, language, models, are, processing

Both subsequences are the longest sequences of tokens that appear in the same order in both sentences, but not necessarily consecutively. When multiple LCS options exist, ROUGE-L can use any of them since their lengths are identical.

Here's how the calculations work here. The length of the LCS is 5 words. The reference text is 9 words long, and the generated text is 8 words long. Thus, recall and precision are:

$$\text{recall}_{\text{LCS}} = \frac{5}{9} \approx 0.56, \quad \text{precision}_{\text{LCS}} = \frac{5}{8} \approx 0.63$$

With $\beta = 8$, ROUGE-L is then given by:

$$\text{ROUGE-L} = \frac{(1 + 8^2) \cdot 0.56 \cdot 0.63}{0.56 + 8^2 \cdot 0.63} \approx 0.56$$

ROUGE scores range from 0 to 1, where 1 means a perfect match between generated and reference texts. However, even excellent summaries or translations rarely approach 1 in practice.

Choosing the right ROUGE metric depends on the task:

- ROUGE-1 and ROUGE-2 are standard starting points. ROUGE-1 checks overall content similarity using unigram overlap, while ROUGE-2 evaluates local fluency and phrase accuracy using bigram matches.
- ROUGE-L is preferred over ROUGE-1 or ROUGE-2 when evaluating text quality in terms of sentence structure and flow, particularly in summarization and translation tasks, since it captures the longest sequence of matching words that appear in the same relative order, better reflecting grammatical coherence.
- In cases where preserving longer patterns is key—such as maintaining technical terms or idioms—higher-order metrics like ROUGE-3 or ROUGE-4 might be more relevant.

A combination of metrics, such as ROUGE-1, ROUGE-2, and ROUGE-L, often gives a more balanced evaluation, covering both content overlap and structural flexibility.

Keep in mind, though, that ROUGE has limitations. It measures lexical overlap but not semantic similarity or factual correctness. To address these gaps, ROUGE is often paired with human evaluations or other methods for a fuller assessment of text quality.

2.6.3. Human Evaluation

Automated metrics are useful, but human evaluation is still necessary to assess language models. Humans can evaluate qualities that automated metrics often miss, like fluency and accuracy. Two common approaches for human evaluation are Likert scale ratings and Elo ratings.

Likert scale ratings involve assigning scores to outputs using a fixed, typically symmetric scale. Raters judge the quality by selecting a score, often from -2 to 2, where each scale point corresponds to a descriptive label. For instance, -2 could mean "Strongly Disagree" or "Poor," while 2 might mean "Strongly Agree" or "Excellent." The scale is symmetric because it includes equal levels of agreement and disagreement around a neutral midpoint, making positive and negative responses easier to interpret.

Likert scales are flexible for evaluating different aspects of language model outputs, such as fluency, coherence, relevance, and accuracy. For example, a rater could separately rate a sentence's grammatical correctness and its relevance to a prompt, both on a scale from -2 to 2.

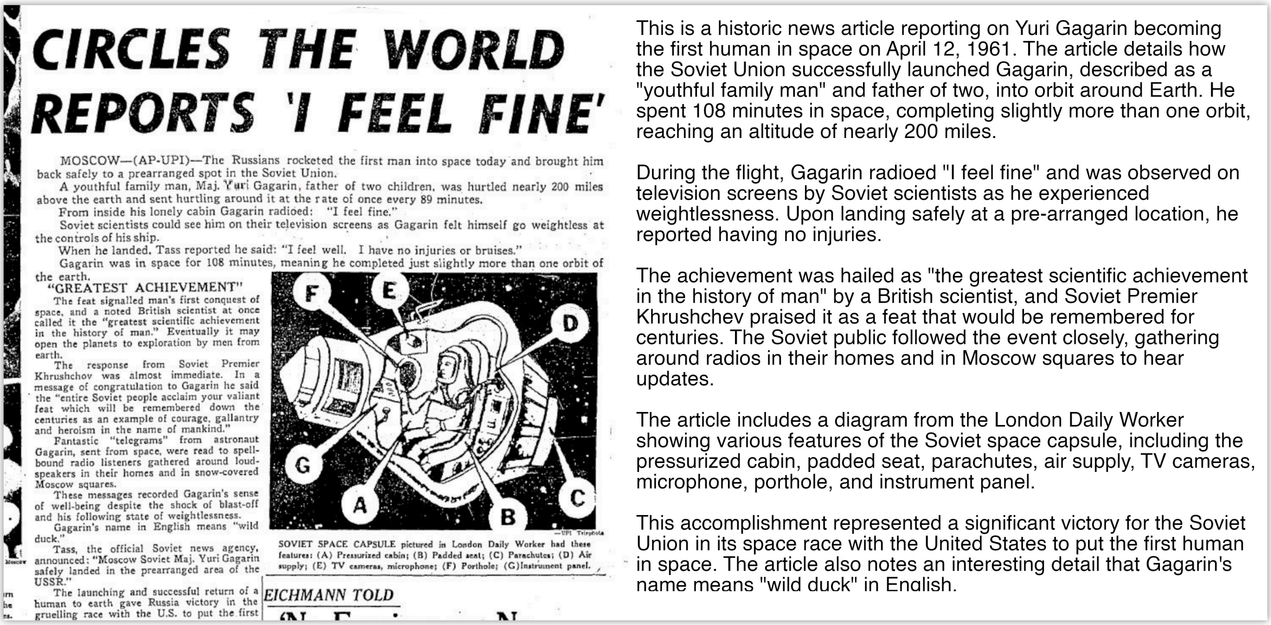
However, the method has limitations. One issue is **central tendency bias**, where some raters avoid extreme scores and stick to the middle of the scale. Another challenge is inconsistency in how raters interpret the scale—some may reserve a 2 for exceptional outputs, while others may assign it to any high-quality response.

To mitigate these issues, researchers often involve multiple raters, phrase similar questions in different ways for the same rater, and use detailed rubrics that clearly define each scale point.

Let's illustrate Likert scale evaluation using a scenario where machine-generated summaries of news articles are assessed.

Human raters compare a model-generated summary to the original article. They rate it on three aspects using 5-point Likert scales: coherence, informativeness, and factual accuracy.

For example, consider the news article on the left and the generated summary on the right:


F
E
D
G
A
B
C
—UPI Telephoto

CIRCLES THE WORLD
REPORTS 'I FEEL FINE'

MOSCOW—(AP-UPI)—The Russians rocketed the first man into space today and brought him back safely to a prearranged spot in the Soviet Union.

A youthful family man, Maj. Yuri Gagarin, father of two children, was hurtled nearly 200 miles above the earth and sent hurtling around it at the rate of once every 89 minutes.

From inside his lonely cabin Gagarin radioed: "I feel fine."

Soviet scientists could see him on their television screens as Gagarin felt himself go weightless at the controls of his ship.

When he landed, Tass reported he said: "I feel well. I have no injuries or bruises."

Gagarin was in space for 108 minutes, meaning he completed just slightly more than one orbit of the earth.

"GREATEST ACHIEVEMENT"

The feat signalled man's first conquest of space, and a noted British scientist at once called it the "greatest scientific achievement in the history of man." Eventually it may open the planets to exploration by men from earth.

The response from Soviet Premier Khrushchov was almost immediate. In a message of congratulation to Gagarin he said the "entire Soviet people acclaim your valiant feat which will be remembered down the centuries as an example of courage, gallantry and heroism in the name of mankind."

Fantastic "telegrams" from astronaut Gagarin, sent from space, were read to spell-bound radio listeners gathered around loud-speakers in their homes and in snow-covered Moscow squares.

These messages recorded Gagarin's sense of well-being despite the shock of blast-off and his following state of weightlessness.

Gagarin's name in English means "wild duck."

Tass, the official Soviet news agency, announced: "Moscow Soviet Maj. Yuri Gagarin safely landed in the prearranged area of the USSR."

The launching and successful return of a human to earth gave Russia victory in the gruelling race with the U.S. to put the first

SOVIET SPACE CAPSULE pictured in London Daily Worker had these features: (A) Pressurized cabin; (B) Padded seat; (C) Parachutes; (D) Air supply; (E) TV cameras, microphone; (F) Porthole; (G) Instrument panel.

EICHMANN TOLD

This is a historic news article reporting on Yuri Gagarin becoming the first human in space on April 12, 1961. The article details how the Soviet Union successfully launched Gagarin, described as a "youthful family man" and father of two, into orbit around Earth. He spent 108 minutes in space, completing slightly more than one orbit, reaching an altitude of nearly 200 miles.

During the flight, Gagarin radioed "I feel fine" and was observed on television screens by Soviet scientists as he experienced weightlessness. Upon landing safely at a pre-arranged location, he reported having no injuries.

The achievement was hailed as "the greatest scientific achievement in the history of man" by a British scientist, and Soviet Premier Khrushchev praised it as a feat that would be remembered for centuries. The Soviet public followed the event closely, gathering around radios in their homes and in Moscow squares to hear updates.

The article includes a diagram from the London Daily Worker showing various features of the Soviet space capsule, including the pressurized cabin, padded seat, parachutes, air supply, TV cameras, microphone, porthole, and instrument panel.

This accomplishment represented a significant victory for the Soviet Union in its space race with the United States to put the first human in space. The article also notes an interesting detail that Gagarin's name means "wild duck" in English.

Raters assess the summary based on how effectively it meets these three criteria.

The scale for assessing coherence—that is, how well-organized, readable, and logically connected the summary is—might look like this:

Very poor	Poor	Acceptable	Good	Excellent
-2	-1	0	1	2

The scale for assessing informativeness, that is, how well the summary captures the essence and main points of the original article, might look this way:

Not informative	Slightly	Moderately	Very	Extremely informative
-2	-1	0	1	2

The scale for assessing factual accuracy—that is, how precisely the summary represents facts and data from the original article—might look like this:

Very low	Some inaccuracies	Mostly Accurate	Very accurate	Perfect
-2	-1	0	1	2

In this example, raters would select one option for each of the three aspects. The use of descriptive anchors at each point on the scale helps standardize understanding among raters.

After collecting ratings from multiple raters across various summaries, researchers analyze the data through several approaches:

- Calculate average scores for each aspect across all summaries and raters to get an overall performance measure.
- Compare scores across different versions of the model to track improvements.
- Analyze the correlation between different aspects (e.g., is high coherence associated with high factual accuracy?).

> Although Likert scale ratings were originally intended for humans, the rise of advanced **chat LMs** means raters can now be either humans or language models.

Pairwise comparison is a method where two outputs are evaluated side-by-side, and the better one is chosen based on specific criteria. This simplifies decision-making, especially when outputs are of similar quality or changes are minor.

The method builds on the principle that relative judgments are easier than absolute ones. Binary choices often produce more consistent and reliable results than absolute ratings.

In practice, raters compare pairs of outputs, such as translations, summaries, or answers, and decide which is better based on criteria like coherence, informativeness, or factual accuracy.

For example, in machine translation evaluation, raters compare pairs of translations for each source sentence, selecting which one better preserves the original meaning in the target language. By repeating this process across many pairs, evaluators can compare different models or versions.

Pairwise comparison helps rank models or model versions by having each rater evaluate multiple pairs, with each model compared against others several times. This repetition minimizes individual biases, resulting in more reliable evaluations. A related approach is **ranking**, where a rater orders multiple responses by quality. Ranking requires less effort than pairwise comparisons while still capturing relative quality.

Results from pairwise comparisons are typically analyzed statistically to determine significant differences between models. A common method for this analysis is the Elo rating system.

Elo ratings, originally created by Arpad Elo in 1960 for ranking chess players, can be adapted for language model evaluation. The system assigns ratings based on "wins" and "losses" in direct comparisons, quantifying relative model performance.

In language model evaluation, all models typically start with an initial rating, often set to 1500. When two models are compared, the probability of one model "winning" is calculated using their current ratings. After each comparison, their ratings are updated based on the actual result versus the expected result.

The probability of model A with rating $\text{Elo}(A)$ winning against model B with rating $\text{Elo}(B)$ is:

$$\Pr(A \text{ wins}) = \frac{1}{1 + 10^{\left(\text{Elo}(B) - \text{Elo}(A)\right)/400}}$$

> The value 400 in the Elo formula acts as a scaling factor, creating a logarithmic relationship between rating differences and winning probabilities. Arpad Elo chose this number ensuring that a 400-point rating difference reflects 10: 1 odds in favor of the higher-rated chess player. While originally designed for chess, this scaling factor has proven effective in other contexts, including language model evaluation.

After a match, ratings are updated using the formula:

$$\text{Elo}(A) \leftarrow \text{Elo}(A) + k \times \left(\text{score}(A) - \Pr(A \text{ wins})\right),$$

where k (typically between 4 and 32) controls the maximum rating change, and $\text{score}(A)$ reflects the outcome: 1 for a win, 0 for a loss, and 0.5 for a draw.

Consider an example with three models: LM_1, LM_2, and LM_3. We'll evaluate them based on their ability to generate coherent text continuations. Assume their initial ratings are:

$$\begin{aligned} \text{Elo}(\text{LM}_1) &= 1500 \\ \text{Elo}(\text{LM}_2) &= 1500 \\ \text{Elo}(\text{LM}_3) &= 1500 \end{aligned}$$

We'll use $k = 32$ for this example.

Consider this prompt: *"The scientists were shocked when they discovered…"*

Continuation by LM_1: *"…a new species of butterfly in the Amazon rainforest. Its wings were unlike anything they had ever seen before."*

Continuation by LM_2: *"…that the ancient artifact they unearthed was emitting a faint, pulsating light. They couldn't explain its source."*

Continuation by LM_3: *"…the results of their experiment contradicted everything they thought they knew about quantum mechanics."*

Let's say we conduct pairwise comparisons and get the following results:

1. LM_1 vs LM_2: LM_1 wins
   - $\Pr(\mathrm{LM}_1 \text{ wins}) = 1/\left(1 + 10^{((1500-1500)/400)}\right) = 0.5$
   - New rating for $\mathrm{LM}_1 \leftarrow 1500 + 32(1 - 0.5) = 1516$
   - New rating for $\mathrm{LM}_2 \leftarrow 1500 + 32(0 - 0.5) = 1484$
2. LM_1 vs LM_3: LM_3 wins
   - $\Pr(\mathrm{LM}_1 \text{ wins}) = 1/\left(1 + 10^{((1500-1516)/400)}\right) \approx 0.523$
   - New rating for $\mathrm{LM}_1 \leftarrow 1516 + 32(0 - 0.523) \approx 1499$
   - New rating for $\mathrm{LM}_3 \leftarrow 1500 + 32(1 - 0.477) \approx 1517$
3. LM_2 vs LM_3: LM_3 wins
   - $\Pr(\mathrm{LM}_2 \text{ wins}) = 1/\left(1 + 10^{((1517-1484)/400)}\right) \approx 0.453$
   - New rating for $\mathrm{LM}_2 \leftarrow 1484 + 32(0 - 0.453) \approx 1470$
   - New rating for $\mathrm{LM}_3 \leftarrow 1517 + 32(1 - 0.547) \approx 1531$

Final ratings after these comparisons:

$$\begin{aligned} \mathrm{Elo}(\mathrm{LM}_1) &= 1499 \\ \mathrm{Elo}(\mathrm{LM}_2) &= 1470 \\ \mathrm{Elo}(\mathrm{LM}_3) &= 1531 \end{aligned}$$

Elo ratings quantify how models perform relative to each other. In this case, LM_3 is the strongest, followed by LM_1, with LM_2 ranking last.

Performance isn't judged from a single match. Instead, multiple pairwise matches are used. This limits the effects of random fluctuations or biases in individual comparisons, giving a better estimate of each model's performance.

A variety of prompts or inputs ensures evaluation across different contexts and tasks. When human raters are involved, several raters assess each comparison to reduce individual bias.

To avoid order effects, both the sequence of comparisons and the presentation of outputs are randomized. Elo ratings are updated after every comparison.

How many matches are needed until the results are reliable? There's no universal number that applies to all cases. As a general guideline, some researchers suggest that each model should participate in at least 100–200 comparisons before considering the Elo ratings stable and ideally 500+ comparisons for high confidence. However, for high-stakes evaluations or when comparing very similar models, thousands of comparisons may be necessary.

> Statistical methods can be used to calculate a **confidence interval** for a model's Elo rating. Explaining these techniques is beyond the scope of this book. For those interested, the **Bradley–Terry model** and **bootstrap resampling** are good starting points. Both are well-documented, with resources linked on the book's wiki.

Elo ratings provide a continuous scale for ranking models, making it easier to track incremental improvements. The system rewards wins against strong opponents more than wins against weaker ones, and it can handle incomplete comparison data, meaning not every model needs to be compared against every other model. However, the choice of k significantly affects rating volatility; a poorly chosen k can undermine the evaluation's stability.

To address these limitations, Elo ratings are often used alongside other evaluation methods. For instance, researchers might use Elo ratings for ranking models in pairwise comparisons, while collecting Likert scale ratings to assess absolute quality. This combined approach yields a more comprehensive view of a language model's performance.

Now that we've covered language modeling and evaluation methods, let's explore a more advanced model architecture: recurrent neural networks (RNNs). RNNs made significant progress in processing text. They introduced the ability to maintain context over long sequences, allowing for the creation of more powerful language models.

Chapter 3. Recurrent Neural Network

In this chapter, we explore recurrent neural networks, a fundamental architecture that revolutionized sequential data processing. While transformers have become dominant in many applications, understanding RNNs first provides an ideal stepping stone—their elegant design introduces key sequential processing concepts that make Transformer mathematics more intuitive. We'll examine RNNs' structure and applications in language modeling, building essential foundations for more advanced architectures.

3.1. Elman RNN

A **recurrent neural network**, or **RNN**, is a neural network designed for sequential data. Unlike **feedforward neural networks**, RNNs include loops in their connections, enabling information to carry over from one step in the sequence to the next. This makes them well-suited for tasks like time series analysis, natural language processing, and other sequential data problems.

To illustrate the sequential nature of RNNs, let's consider a neural network with a single unit. Consider the input document *"Learning from text is cool."* Ignoring case and punctuation, the matrix representing this document would be as follows:

Word	Embedding vector
learning	$[0.1,0.2,0.6]^\top$
from	$[0.2,0.1,0.4]^\top$
text	$[0.1,0.3,0.3]^\top$
is	$[0.0,0.7,0.1]^\top$
cool	$[0.5,0.2,0.7]^\top$
PAD	$[0.0,0.0,0.0]^\top$

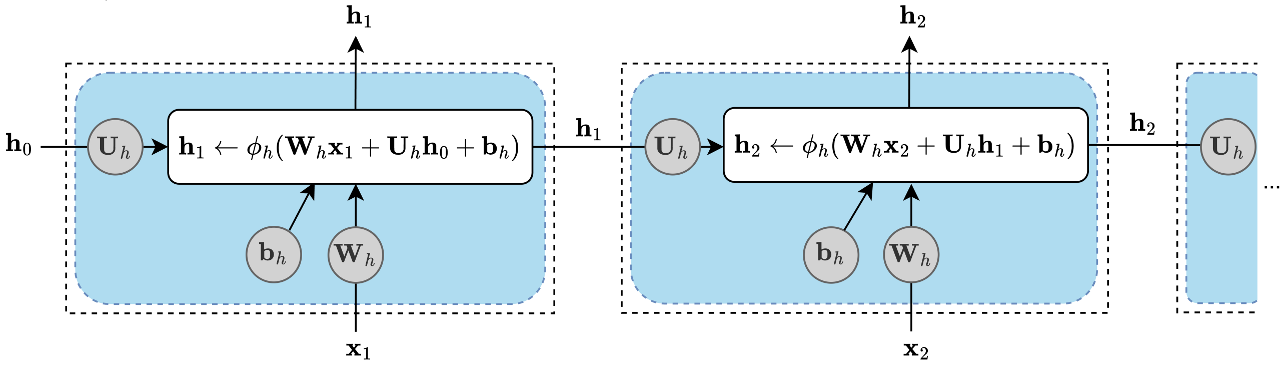
Each row of the matrix represents a word's embedding learned during neural network training. The order of words is preserved. The matrix dimensions are (sequence length, embedding dimensionality). Sequence length specifies the maximum number of words in a document. Shorter documents are padded with padding tokens (like PAD in this example), while longer ones are truncated. **Padding** uses dummy embeddings, usually **zero vectors**.

More formally, the matrix would look like this:

$$\mathbf{X} = \begin{bmatrix} 0.1 & 0.2 & 0.6 \\ 0.2 & 0.1 & 0.4 \\ 0.1 & 0.3 & 0.3 \\ 0.0 & 0.7 & 0.1 \\ 0.5 & 0.2 & 0.7 \\ 0.0 & 0.0 & 0.0 \end{bmatrix}$$

Here, we have five 3D embedding vectors, $\mathbf{x}_1, \dots, \mathbf{x}_5$, representing each word in the document. For instance, $\mathbf{x}_1 = [0.1, 0.2, 0.6]^\top$, $\mathbf{x}_2 = [0.2, 0.1, 0.4]^\top$, and so on. The sixth vector is a padding vector.

The **Elman RNN**, introduced by Jeffrey Locke Elman in 1990 as the **simple recurrent neural network**, processes a sequence of embedding vectors one at a time, as illustrated below:

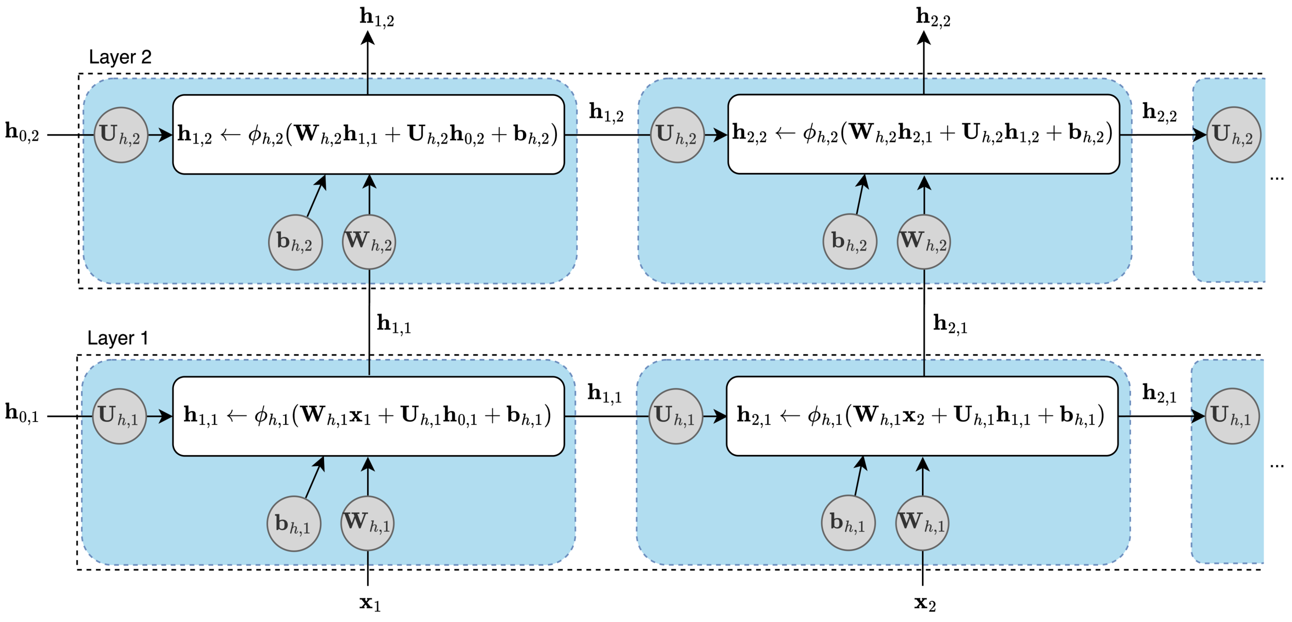


$\mathbf{h}_1$
$\mathbf{h}_2$
$\mathbf{h}_0$
$\mathbf{U}_h$
$\mathbf{h}_1 \leftarrow \phi_h(\mathbf{W}_h\mathbf{x}_1 + \mathbf{U}_h\mathbf{h}_0 + \mathbf{b}_h)$
$\mathbf{h}_1$
$\mathbf{U}_h$
$\mathbf{h}_2 \leftarrow \phi_h(\mathbf{W}_h\mathbf{x}_2 + \mathbf{U}_h\mathbf{h}_1 + \mathbf{b}_h)$
$\mathbf{h}_2$
$\mathbf{U}_h$
...
$\mathbf{b}_h$
$\mathbf{W}_h$
$\mathbf{b}_h$
$\mathbf{W}_h$
$\mathbf{x}_1$
$\mathbf{x}_2$

At each time step t, the current input embedding $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ are combined by multiplying them with trainable weight matrices $\mathbf{W}_h$ and $\mathbf{U}_h$, adding a bias vector $\mathbf{b}_h$, and producing the updated hidden state $\mathbf{h}_t$. Unlike MLP units, which output scalars, an RNN unit outputs vectors and acts as an entire layer. The initial hidden state $\mathbf{h}_0$ is usually a **zero vector**.

A **hidden state** is a memory vector that captures information from previous steps in a sequence. Updated at each step using current input and past state, it helps neural networks use context from earlier words to predict the next word in a sentence.

To deepen the network, we add a second RNN layer. The first layer's outputs, $\mathbf{h}_t$, become inputs to the second, whose outputs are the network's final outputs:



Layer 2
Layer 1
$\mathbf{h}_{1,2}$
$\mathbf{h}_{2,2}$
$\mathbf{h}_{0,2}$
$\mathbf{U}_{h,2}$
$\mathbf{h}_{1,2} \leftarrow \phi_{h,2}(\mathbf{W}_{h,2}\mathbf{h}_{1,1} + \mathbf{U}_{h,2}\mathbf{h}_{0,2} + \mathbf{b}_{h,2})$
$\mathbf{h}_{2,2} \leftarrow \phi_{h,2}(\mathbf{W}_{h,2}\mathbf{h}_{2,1} + \mathbf{U}_{h,2}\mathbf{h}_{1,2} + \mathbf{b}_{h,2})$
$\mathbf{b}_{h,2}$
$\mathbf{W}_{h,2}$
$\mathbf{h}_{1,1}$
$\mathbf{h}_{2,1}$
$\mathbf{h}_{0,1}$
$\mathbf{U}_{h,1}$
$\mathbf{h}_{1,1} \leftarrow \phi_{h,1}(\mathbf{W}_{h,1}\mathbf{x}_1 + \mathbf{U}_{h,1}\mathbf{h}_{0,1} + \mathbf{b}_{h,1})$
$\mathbf{h}_{2,1} \leftarrow \phi_{h,1}(\mathbf{W}_{h,1}\mathbf{x}_2 + \mathbf{U}_{h,1}\mathbf{h}_{1,1} + \mathbf{b}_{h,1})$
$\mathbf{b}_{h,1}$
$\mathbf{W}_{h,1}$
$\mathbf{x}_1$
$\mathbf{x}_2$
...

Figure 3.1: A two-layer Elman RNN. The first layer's outputs serve as inputs to the second layer.

3.2. Mini-Batch Gradient Descent

Before coding the RNN model, we need to discuss the shape of the input data. In Section 1.7, we used the entire dataset for each gradient descent step. Here, and for training all future models, we'll adopt **mini-batch gradient descent**, a widely used method for large models and datasets. Mini-batch gradient descent calculates derivatives over smaller data subsets, which speeds up learning and reduces memory usage.

With mini-batch gradient descent, the data shape is organized as (batch size, sequence length, embedding dimensionality). This structure divides the training set into fixed-size mini-batches, each containing sequences of embeddings with consistent lengths. (From this point on, "batch" and "mini-batch" will be used interchangeably.)

For example, if the batch size is 2, the sequence length is 4, and the embedding dimensionality is 3, the mini-batch can be represented as:

$$\text{batch}_1 = \begin{bmatrix} \text{seq}_{1,1} & \text{seq}_{1,2} & \text{seq}_{1,3} & \text{seq}_{1,4} \\ \text{seq}_{2,1} & \text{seq}_{2,2} & \text{seq}_{2,3} & \text{seq}_{2,4} \end{bmatrix}$$

Here, $\text{seq}_{i,j}$, for $i \in \{1,2\}$ and $j \in \{1, \ldots, 4\}$ is an embedding vector.

Let's have the following embeddings for each sequence:

$$\text{seq}_1\text{:}\begin{bmatrix}[0.1,0.2,0.3]\\ [0.4,0.5,0.6]\\ [0.7,0.8,0.9]\\ [1.0,1.1,1.2]\end{bmatrix}$$

$$\text{seq}_2\text{:}\begin{bmatrix}[1.3,1.4,1.5]\\ [1.6,1.7,1.8]\\ [1.9,2.0,2.1]\\ [2.2,2.3,2.4]\end{bmatrix}$$

The mini-batch will look like this:

$$\text{batch}_1 = \begin{bmatrix}[0.1,0.2,0.3] & [0.4,0.5,0.6] & [0.7,0.8,0.9] & [1.0,1.1,1.2]\\ [1.3,1.4,1.5] & [1.6,1.7,1.8] & [1.9,2.0,2.1] & [2.2,2.3,2.4]\end{bmatrix}$$

During each step of gradient descent, we:

1. Select a mini-batch from the training set,
2. Pass it through the neural network,
3. Compute the loss,
4. Calculate gradients,
5. Update model parameters,
6. Repeat from step 1.

Mini-batch gradient descent often achieves faster **convergence** compared to using the entire training set per step. It efficiently handles large models and datasets by using modern hardware's parallel processing capabilities. In PyTorch, models require the first dimension of the input data to be the batch dimension, even if there's only one example in the batch.

3.3. Programming an RNN

Let's implement an Elman RNN unit:

```
import torch
import torch.nn as nn

class ElmanRNNUnit(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.Uh = nn.Parameter(torch.randn(emb_dim, emb_dim))
❶
```

```
        self.Wh = nn.Parameter(torch.randn(emb_dim, emb_dim))
❷
        self.b = nn.Parameter(torch.zeros(emb_dim)) ❸

    def forward(self, x, h):
        return torch.tanh(x @ self.Wh + h @ self.Uh + self.b)
❹
```

In the constructor:

- Lines ❶ and ❷ initialize `self.Uh` and `self.Wh`, the weight matrices for the hidden state and input vector, with random values.
- Line ❸ sets `self.b`, the bias vector, to zero.

In the `forward` method, line ❹ handles the computation for each time step. It processes the current input `x` and the previous hidden state `h`, both shaped (`batch_size`, `emb_dim`), combines them with the weight matrices and bias, and applies the **tanh** activation. The output is the new hidden state, also of shape (`batch_size`, `emb_dim`).

The `@` character is the **matrix multiplication** operator in PyTorch. We use `x @ self.Wh` rather than `self.Wh @ x` because of the way PyTorch handles batch dimensions in matrix multiplication. When working with batched inputs, `x` has a shape of (`batch_size`, `emb_dim`), while `self.Wh` has a shape of (`emb_dim`, `emb_dim`). Remember from Section 1.6 that for two matrices to be multipliable, the number of columns in the left matrix must be the same as the number of rows in the right matrix. This is satisfied in `x @ self.Wh`.

Now, let's define the class `ElmanRNN`, which implements a two-layer Elman RNN using `ElmanRNNUnit` as its core building block:

```
class ElmanRNN(nn.Module):
    def __init__(self, emb_dim, num_layers):
        super().__init__()
        self.emb_dim = emb_dim
        self.num_layers = num_layers
        self.rnn_units = nn.ModuleList(
            [ElmanRNNUnit(emb_dim) for _ in range(num_layers)
]
        ) ❶
    def forward(self, x):
        batch_size, seq_len, emb_dim = x.shape ❷
        h_prev = [
            torch.zeros(batch_size, emb_dim, device=x.device)
❸
            for _ in range(self.num_layers)
        ]
        outputs = []
        for t in range(seq_len): ❹
            input_t = x[:, t]
            for l, rnn_unit in enumerate(self.rnn_units):
                h_new = rnn_unit(input_t, h_prev[l])
                h_prev[l] = h_new    # Update hidden state
                input_t = h_new      # Input for next layer
            outputs.append(input_t)  # Collect outputs
        return torch.stack(outputs, dim=1) ❺
```

In line ❶ of the constructor, we initialize the RNN layers by creating a `ModuleList` containing `ElmanRNNUnit` instances—one per layer. Using `ModuleList` instead of a regular Python list ensures the parent module (`ElmanRNN`) properly registers all RNN unit parameters. This guarantees that calling `.parameters()` or `.to(device)` on the parent module includes parameters from all modules in the `ModuleList`.

In the `forward` method:

- Line ❷ extracts `batch_size`, `seq_len`, and `emb_dim` from the input tensor `x`.
- Line ❸ initializes the hidden states `h_prev` for all layers with zero tensors. Each hidden state in the list has the shape (`batch_size`, `emb_dim`).

> We store hidden states for each layer in a list instead of a multidimensional tensor because we need to modify them during processing. In-place modifications of tensors can disrupt PyTorch's automatic differentiation system, which might result in incorrect gradient calculations.

- Line ❹ iterates over time steps `t` in the input sequence. For each `t`:

    - Extract the input at time `t`: `input_t = x[:, t]`.
    - For each layer `l`:
        - Compute the new hidden state `h_new` from `input_t` and `h_prev[l]`.
        - Update the hidden state: `h_prev[l] = h_new` (updates in place).
        - Set `input_t = h_new` to pass to the next layer.
    - Append the output of the last layer: `outputs.append(input_t)`.
- Once all time steps are processed, line ❺ converts the `outputs` list into a tensor by stacking it along the time dimension. The resulting tensor has the shape (`batch_size`, `seq_len`, `emb_dim`).

3.4. RNN as a Language Model

An RNN-based language model uses `ElmanRNN` as its building block:

```
class RecurrentLanguageModel(nn.Module):
    def __init__(self, vocab_size, emb_dim, num_layers, pad_i
dx):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size,
            emb_dim,
            padding_idx=pad_idx
        ) ❶
        self.rnn = ElmanRNN(emb_dim, num_layers)
        self.fc = nn.Linear(emb_dim, vocab_size)

    def forward(self, x):
        embeddings = self.embedding(x)
        rnn_output = self.rnn(embeddings)
        logits = self.fc(rnn_output)
        return logits
```

The `RecurrentLanguageModel` class integrates three components: an embedding layer, the `ElmanRNN` defined earlier, and a final linear layer.

In the constructor, line ❶ defines the embedding layer. This layer transforms input token indices into dense vectors. The `padding_idx` parameter ensures that padding tokens are represented by zero vectors. (We'll cover the embedding layer in the next section.)

Next, we initialize the custom `ElmanRNN`, specifying the embedding dimensionality and the number of layers. Finally, we add a fully connected layer, which converts the RNN's output into vocabulary-sized logits for each token in the sequence.

In the `forward` method:

- We pass the input `x` through the embedding layer. Input `x` has shape (`batch_size`, `seq_len`), and the output `embeddings` have shape (`batch_size`, `seq_len`, `emb_dim`).
- We then pass the embedded input through our `ElmanRNN`, obtaining `rnn_output` with shape (`batch_size`, `seq_len`, `emb_dim`).
- Finally, we apply the fully connected layer to the RNN output, producing logits for each token in the vocabulary at each position in the sequence. The output logits have shape (`batch_size`, `seq_len`, `vocab_size`).

3.5. Embedding Layer

An **embedding layer**, implemented as `nn.Embedding` in PyTorch, maps token indices from a vocabulary to dense, fixed-size vectors. It acts as a learnable lookup table, where each token is assigned a unique embedding vector. During training, these vectors are adjusted to capture meaningful numerical representations of the tokens.

Let's see how an embedding layer works. Imagine a vocabulary with five tokens, indexed from 0 to 4. We want each token to have a 3D embedding vector. To begin, we create an embedding layer:

```
import torch
import torch.nn as nn

vocab_size = 5  # Number of unique tokens
emb_dim = 3     # Size of each embedding vector
emb_layer = nn.Embedding(vocab_size, emb_dim)
```

The embedding layer initializes the embedding matrix **E** with random values. In this case, the matrix has 5 rows (one for each token) and 3 columns (the embedding dimensionality):

$$\mathbf{E} = \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ -0.3 & 0.8 & -0.5 \\ 0.7 & 0.1 & -0.2 \\ -0.6 & 0.5 & 0.4 \\ 0.9 & -0.7 & 0.3 \end{bmatrix}$$

Each row in **E** represents the embedding vector for a specific token in the vocabulary.

Now, let's input a sequence of token indices:

```
token_indices = torch.tensor([0, 2, 4])
```

The embedding layer retrieves the rows of **E** corresponding to the input indices:

$$\text{Embeddings} = \begin{bmatrix} 0.2 & -0.4 & 0.1 \\ 0.7 & 0.1 & -0.2 \\ 0.9 & -0.7 & 0.3 \end{bmatrix}$$

This output is a matrix whose number of rows equals the input sequence length and whose number of columns equals the embedding dimensionality:

```
embeddings = embedding_layer(token_indices)
print(embeddings)
```

The output might look like this:

```
tensor([[ 0.2, -0.4,  0.1],
        [ 0.7,  0.1, -0.2],
        [ 0.9, -0.7,  0.3]])
```

The embedding layer can manage padding tokens as well. Padding ensures sequences in a mini-batch have the same length. To prevent the model from updating embeddings for padding tokens during training, the layer maps them to a zero vector that remains unchanged. For example, we can define the padding index as follows:

```
emb_layer = nn.Embedding(vocab_size, emb_dim, padding_idx=0)
```

With this configuration, the embedding for token 0 (padding token) is always $[0,0,0]^\top$.

Given the input:

```
token_indices = torch.tensor([0, 2, 4])
embeddings = emb_layer(token_indices)
print(embeddings)
```

The result would be:

```
tensor([[ 0.0,  0.0,  0.0],  # Padding token
        [ 0.7,  0.1, -0.2],  # Token 2 embedding
        [ 0.9, -0.7,  0.3]]) # Token 4 embedding
```

With modern language models, vocabularies often include hundreds of thousands of tokens, and embedding dimensions are typically several thousands. This makes the embedding matrix a significant part of the model, sometimes containing up to 2 billion parameters.

3.6. Training an RNN Language Model

Start by importing libraries and defining utility functions:

```
import torch, torch.nn as nn

def set_seed(seed):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed) ❶
    torch.backends.cudnn.deterministic = True ❷
    torch.backends.cudnn.benchmark = False ❸
```

The `set_seed` function enforces **reproducibility** by setting the Python random seed, the PyTorch CPU seed, and, in line ❶, the CUDA seed for all GPUs (Graphics Processing Units). CUDA is NVIDIA's parallel computing platform and API that enables significant performance improvements in computing by leveraging the power of GPUs. Using `torch.cuda.manual_seed_all` ensures consistent GPU-based random behavior, while lines ❷ and ❸ disable CUDA's auto-tuner and enforce deterministic algorithms, guaranteeing identical results across different GPU models.

With the model class ready, we'll train our neural language model. First, we install the `transformers` package—an open-source library providing APIs and tools to easily download, train and use pretrained models from the **Hugging Face Hub**:

```
$ pip3 install transformers
```

The package offers a Python API for training that works with both **PyTorch** and **TensorFlow**. For now, we only need it to get a tokenizer.

Now we import `transformers`, set the tokenizer, define the hyperparameter values, prepare the data, and instantiate the model, loss function, and optimizer objects:

```
from transformers import AutoTokenizer

device = torch.device("cuda" if torch.cuda.is_available() els
e "cpu") ❶
tokenizer = AutoTokenizer.from_pretrained(
    "microsoft/Phi-3.5-mini-instruct"
) ❷
vocab_size = len(tokenizer) ❸

emb_dim, num_layers, batch_size, learning_rate, num_epochs =
get_hyperparameters()

data_url = "https://www.thelmbook.com/data/news"
train_loader, test_loader = download_and_prepare_data(
    data_url, batch_size, tokenizer) ❹

model = RecurrentLanguageModel(
    vocab_size, emb_dim, num_layers, tokenizer.pad_token_id
)
initialize_weights(model) ❺
model.to(device)

criterion = nn.CrossEntropyLoss(ignore_index=tokenizer.pad_to
ken_id) ❻
optimizer = torch.optim.AdamW(model.parameters(), lr=learning
_rate)
```

Line ❶ detects a CUDA device if it's available. Otherwise, it defaults to CPU.

CUDA is not the only GPU acceleration framework available for training neural networks—PyTorch also provides native support for checking availability of MPS (Apple Metal) through its `is_available()` method. In this book, though, we will use CUDA as it remains the most widely used platform for machine learning acceleration.

Most models on the Hugging Face Hub include the tokenizer that was used to train them. Line ❷ initializes the **Phi 3.5 mini** tokenizer. It was trained on a large text corpus using the **byte-pair encoding** algorithm and has a vocabulary size of 32,064.

Line ❸ retrieves the tokenizer's vocabulary size. Line ❹ downloads and prepares the dataset—a collection of news sentences from online articles—tokenizing them and creating `DataLoader` objects. We'll explore `DataLoader` shortly. For now, think of them as iterators over batches.

Line ❺ initializes the model parameters. Initial parameter values can greatly influence the training process. They can affect how quickly training progresses and the final loss value. Certain initialization techniques, like **Xavier initialization**, have shown good results in practice. The `initialize_weights` function, implementing this method, is defined in the notebook.

Line ❻ creates the loss function with the `ignore_index` parameter. This ensures the loss is not calculated for padding tokens.

Now, let's look at the training loop:

```
for epoch in range(num_epochs): ❶
    model.train() ❷
    for batch in train_loader: ❸
        input_seq, target_seq = batch
        input_seq = input_seq.to(device) ❹
        target_seq = target_seq.to(device) ❺
        batch_size_current, seq_len = input_seq.shape ❻
        optimizer.zero_grad()
        output = model(input_seq)
        output = output.reshape(batch_size_current * seq_len,
vocab_size) ❼
```

```
        target = target_seq.reshape(batch_size_current * seq_
len) ❽
        loss = criterion(output, target) ❾
        loss.backward()
        optimizer.step()
```

Line ❶ iterates over epochs. An **epoch** is a single pass through the entire dataset. Training for multiple epochs can improve the model, especially with limited training data. The number of epochs is a **hyperparameter** that you adjust based on the model's performance on the test set.

Line ❷ calls `model.train()` at the start of each epoch to set the model in training mode. This is important for models that have layers behaving differently during training vs. **evaluation**.

> Although our RNN model doesn't use such layers, calling `model.train()` ensures the model is properly configured for training. This avoids unexpected behavior and keeps consistency, especially if future changes add layers dependent on the mode.

Line ❸ iterates over batches. Each batch is a tuple: one tensor contains input sequences, and the other contains target sequences. Lines ❹ and ❺ move these tensors to the same device as the model. If the model and data are on different devices, PyTorch raises an error.

Line ❻ retrieves the batch size and sequence length from `input_seq` (`target_seq` has the same shape). These dimensions are needed to reshape the model's output tensor (`batch_size_current`, `seq_len`, `vocab_size`) and target tensor (`batch_size_current`, `seq_len`) into compatible shapes for the **cross-entropy** loss function. In line ❼, the output is reshaped to (`batch_size_current * seq_len`, `vocab_size`), and in line ❽, the target is flattened to `batch_size_current * seq_len`, allowing the loss calculation in line ❾ to process all tokens in the batch simultaneously and return the average loss per token.

This concludes the training loop implementation. The full RNN language model training implementation is in the thelmbook.com/nb/3.1 notebook. Now, let's examine the DataLoader and Dataset classes that make this batch processing possible.

3.7. Dataset and DataLoader

As mentioned earlier, the `download_and_prepare_data` function returns two *loader* objects: `train_loader` and `test_loader`. I asked you to think of them as iterators over batches of data. But what are they, exactly?

These classes were designed to manage data efficiently during training. While this book doesn't focus on data loading and manipulation, a brief explanation is important for clarity.

The `Dataset` class serves as an interface to your actual data source. By implementing its `__len__` method, you can get the size of the dataset. By defining `__getitem__`, you can access individual examples. These examples can come from many "physical" sources: files, databases, or even data generated on the fly.

Let's look at an example. Assume we have a **JSONL** file called `data.jsonl`, where each line is a **JSON** object containing two input features and a label. Here's how a couple of lines might look:

```
{"feature1": 1.0, "feature2": 2.0, "label": 3.0}
{"feature1": 4.0, "feature2": 5.0, "label": 9.0}
...
```

Here's how you can create a custom `Dataset` to read this file:

```
import json
import torch
from torch.utils.data import Dataset

class JSONDataset(Dataset):
    def __init__(self, file_path):
        self.data = []
        with open(file_path, 'r') as f:
            for line in f:
                item = json.loads(line)
                features = [item['feature1'], item['feature2'
]]
                label = item['label']
                self.data.append((features, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        features, label = self.data[idx]
        features = torch.tensor(features, dtype=torch.float32
)
        label = torch.tensor(label, dtype=torch.long)
        return features, label
```

In this example:

- `__init__` reads the file and stores the data in memory,
- `__len__` returns the total number of examples,
- `__getitem__` retrieves a single example and converts it to tensors.

We can access individual examples like this:

```
dataset = JSONDataset('data.jsonl')
features, label = dataset[0]
```

A `DataLoader` is used with a `Dataset` to manage tasks like batching, shuffling, and loading data in parallel. For example:

```
from torch.utils.data import DataLoader

dataset = JSONLDataset('data.jsonl') ❶

data_loader = DataLoader(
    dataset,
    batch_size=32, # Number of examples per batch
    shuffle=True,  # Shuffle data at every epoch
    num_workers=0  # Number of subprocesses for data loading
) ❷

num_epochs = 5
for epoch in range(num_epochs):
    for batch_features, batch_labels in data_loader: ❸
        print(f"Batch features shape: {batch_features.shape}")
        print(f"Batch labels shape: {batch_labels.shape}")
        # Feed batch_features and batch_labels into your mode
l
```

Line ❶ creates a `Dataset` instance. Line ❷ then wraps the dataset in a `DataLoader`. Finally, line ❸ iterates over the `DataLoader` for five epochs. With `shuffle=True`, the data is shuffled before batching in each epoch. This prevents the model from learning the order of the training data.

With `num_workers=0`, data loading happens in the main process. This simple setup may not be the most efficient, especially for large datasets. Using a positive value for `num_workers` makes PyTorch spawn that many worker processes, enabling parallel data loading. This can significantly speed up training by preventing data loading from becoming a bottleneck.

Output:

```
Batch features shape: torch.Size([32, 2])
Batch labels shape: torch.Size([32])
```

By using a well-designed `Dataset` with a `DataLoader`, you can scale your training pipeline to handle large datasets, optimize data loading with parallel workers, and experiment with different batching strategies. This approach streamlines the training process, letting you concentrate on model design and optimization.

3.8. Training Data and Loss Computation

When studying neural language models, a key aspect is understanding the structure of a training example. The text corpus is split into overlapping input and target sequences. Each input sequence aligns with a target sequence shifted by one token. This setup trains the model to predict the next word at each position in the sequence.

For instance, take the sentence *"We train a recurrent neural network as a language model."* After tokenizing it with the Phi 3.5 mini tokenizer, we get:

```
["_We", "_train", "_a", "_rec", "urrent", "_neural", "_networ
k", "_as", "_a", "_language", "_model", "."]
```

To create one training example, we convert the sentence into input and target sequences by shifting tokens forward by one position:

```
Input: ["_We", "_train", "_a", "_rec", "urrent", "_neural", "
_network", "_as", "_a", "_language", "_model"]
```

```
Target: ["_train", "_a", "_rec", "urrent", "_neural", "_netwo
rk", "_as", "_a", "_language", "_model", "."]
```

A training example doesn't need to be a complete sentence. Modern language models process sequences up to their **context window** length—the maximum tokens (like 8192) they can handle at once. The window limits how far apart the model can connect relationships in text. Training splits text into window-sized chunks, each target sequence shifted one token forward from its input.

During training, the RNN processes one token at a time, updating its hidden states layer by layer. At each step, it generates logits aimed at predicting the next token in the sequence. Each logit corresponds to a vocabulary token and is converted into probabilities using **softmax.** These probabilities are then used to compute the loss.

Each training example results in multiple predictions and losses. For example, the model first processes "_We" and tries to predict "_train" by assigning probabilities to all vocabulary tokens. The loss is computed using the probability of "_train," as defined in Equation 2.1. Next, the model processes "_train" to predict "_a," generating another loss. This continues for every token in the input sequence. For the above example, the model makes 11 predictions and calculates 11 losses.

The losses are averaged across the tokens in a training example and all examples in the batch. The average loss expression is then used in backpropagation to update the model's parameters.

Let's break down the loss calculation for each position with some made-up numbers:

- **Position 1**:
    - Target token: "_train"
    - Logit for "_train": -0.5
    - After applying softmax to the logits, suppose the probability of "_train" is 0.1
    - Contribution to the total loss by Equation 2.1 is $-\log(0.1) = 2.30$
- **Position 2**:
    - Target token: "_a"
    - Logit for "_a": 3.2
    - After softmax, the probability for "_a": 0.05

    - Contribution to loss: $-\log(0.05) = 2.99$
- **Position 3**:
    - The probability for "_rec": 0.02
    - Contribution to loss: $-\log(0.02) = 3.91$
- **Position 4**:
    - The probability for "urrent": 0.34
    - Contribution to loss: $-\log(0.34) = 1.08$

We continue until calculating the loss contribution for the final token, the period:

- **Position 11**:
    - Target token: "."
    - Logit for ".": -1.2
    - After softmax, the probability for ".": 0.11
    - Contribution to loss: $-\log(0.11) = 2.21$

The final loss is calculated by taking the average of these values:

$$\frac{(2.30 + 2.99 + 3.91 + 1.08 + \cdots + 2.21)}{11} = 2.11 \text{ (hypothetically)}$$

During training, the objective is to minimize this loss. This involves improving the model so that it assigns higher probabilities to the correct target tokens at each position.

The full code for training the RNN-based language model can be found in thelmbook.com/nb/3.1. I used the following hyperparameter values: `emb_dim = 128`, `num_layers = 2`, `batch_size = 128`, `learning_rate = 0.001`, and `num_epochs = 1`.

Here are three continuations for the prompt *"The President"* generated at later training steps:

```
The President refused to comment on the best news in the five
on BBC .
The President has been a `` very serious '' and `` unacceptab
le '' .
The President 's office is not the first time to be able to t
ake the lead .
```

> When Elman introduced RNNs in 1990, his experiments used sequences averaging 3.92 words, limited by the hardware of the time. By 2014, advances in computing and improved activation functions made it possible to train RNNs on sequences hundreds of words long, turning them from an academic idea into a practical tool.

At training start, our model produced nearly random tokens, but gradually improved, reaching a perplexity of 72.41—better than the count-based model's 299.06 but far behind GPT-2's 20 and modern LLMs' sub-5 scores.

Three key factors explain this performance gap:

1. The model is small, with just 8,292,619 parameters, mostly in the embedding layer.
2. The context size we used was relatively short—30 tokens.
3. The Elman RNN's hidden state gradually "forgets" information from earlier tokens.

Long short-term memory (**LSTM**) networks improved upon RNNs but still struggled with very long sequences. Transformers later superseded both architectures, becoming dominant in natural language processing by 2023 through better handling of long contexts and improved parallel computation enabling larger models.

> Interest in RNNs was reignited in 2024 with the invention of the **minLSTM** and **xLSTM** architectures, which achieve performance comparable to Transformer-based models. This resurgence reflects a broader trend in AI research: no model type is ever permanently obsolete. Researchers often revisit and refine older ideas, adapting them to address modern challenges and leverage current hardware capabilities.

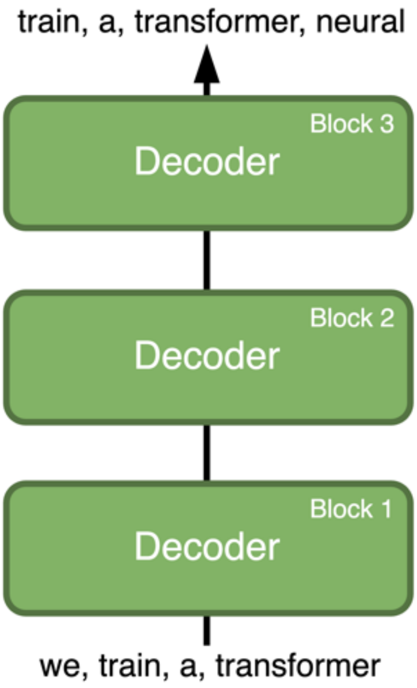
With this, we've completed our study of recurrent neural networks and their applications in language modeling. In the remainder of the book, we'll examine transformer neural networks and language modeling based on them. We'll investigate their approach to tasks like question answering, document classification, and other practical applications.

Chapter 4. Transformer

Transformer models have greatly advanced NLP. They overcome RNNs' limitations in managing long-range dependencies and enable parallel processing of input sequences. There are three main Transformer architectures: encoder-decoder, initially formulated for machine translation; encoder-only, typically used for classification; and decoder-only, commonly found in chat LMs.

In this chapter, we'll explore the decoder-only Transformer architecture in detail, as it is the most widely used approach for training **autoregressive language models**.

The transformer architecture introduces two key innovations: self-attention and positional encoding. Self-attention enables the model to assess how each word relates to all others during prediction, while positional encoding captures word order and sequential patterns. Unlike RNNs, transformers process all tokens simultaneously, using positional encoding to maintain sequential context despite parallel processing of each token. This chapter explores these fundamental elements in detail.



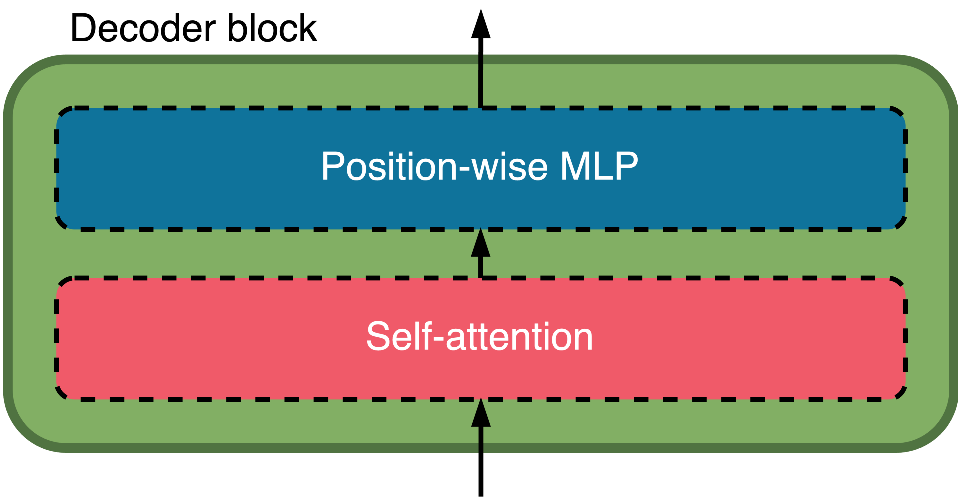
A **decoder-only Transformer** (referred to simply as "decoder" from here on) is made up of multiple identical[5] layers, known as decoder blocks, stacked vertically as shown on the right.

As you can see, training a decoder involves pairing each input sequence with a target sequence that is shifted forward by one token — the same method used for RNN-based language models.

4.1. Decoder Block

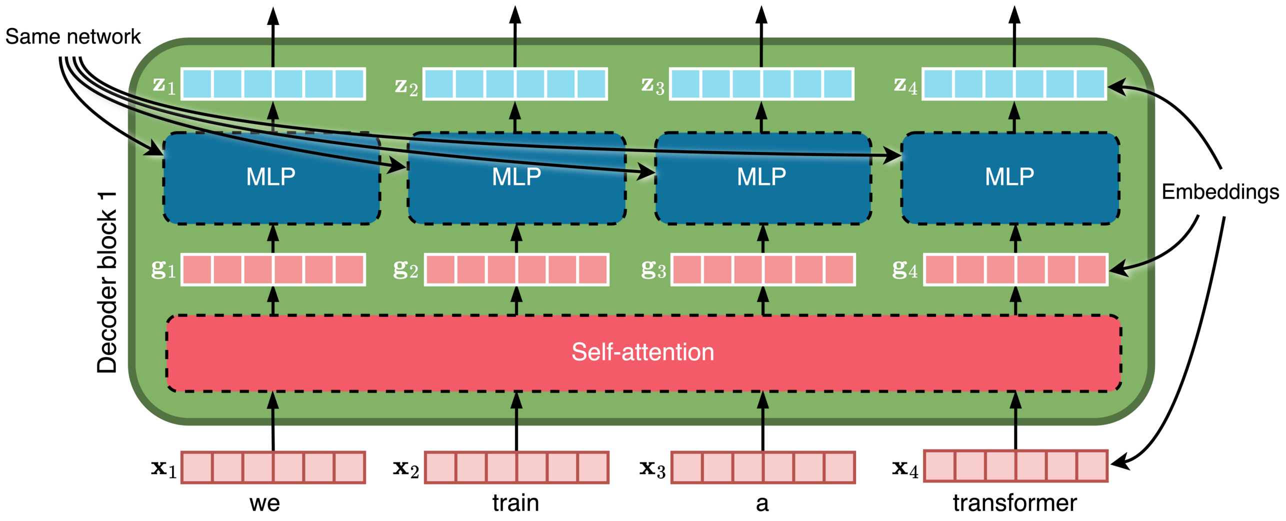
Each decoder block has two sub-layers: self-attention and a position-wise multilayer perceptron (MLP) as shown below:

[5] Decoder blocks share the same architecture but have distinct trainable parameters unique to each block.



The illustration simplifies certain aspects to avoid introducing too many new concepts at once. We'll introduce the missing details step by step.

Let's take a closer look at what happens in a decoder block, starting with the first one:



The first decoder block processes input token embeddings. For this example, we use 6-dimensional input and output embeddings, though in practice these dimensions grow larger with parameter count and token vocabulary. The **self-attention layer**, transforms each input embedding vector $\mathbf{x}_t$ into a new vector $\mathbf{g}_t$ for every token t, from 1 to L, where L represents the input length.

> Here, we simplified each unit as a square, following the same approach we used for the four-unit network in Section 1.5. While our earlier chapters showed information in a neural network flowing from left to right, we've now shifted to a bottom-to-top orientation—the standard

convention for high-level language model diagrams in the literature. We'll maintain this vertical orientation from now on.

After self-attention, the position-wise MLP independently processes each vector $\mathbf{g}_t$ one at a time. Each decoder block has its own MLP with unique parameters, and within a block, this same MLP is applied independently to each position's vector, taking one $\mathbf{g}_t$ as input and producing one $\mathbf{z}_t$ as output. When the MLP finishes processing each position sequentially, the number of output vectors $\mathbf{z}_t$ equals the number of input tokens $\mathbf{x}_t$.

The output vectors $\mathbf{z}_t$ then serve as inputs to the next decoder block. This process repeats through each decoder block, preserving a number of output vectors equal to the number of input tokens $\mathbf{x}_t$.

4.2. Self-Attention

To see how **self-attention** works, let's start with an intuitive comparison. Transforming $\mathbf{g}_t$ into $\mathbf{z}_t$ is straightforward: a position-wise MLP takes an input vector and outputs a new vector by applying a learned transformation. This is what feedforward networks are designed to do. However, self-attention can seem more complex.

Consider a 5-token example: ["we," "train," "a," "transformer," "model"], and assume a decoder with a maximum input sequence length of 4.

In each decoder block, the self-attention function relies on three tensors of trainable parameters: $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$. Here, Q stands for "query," K for "key," and V for "value."

Let's assume these tensors are 6×6. This means each of the four 6-dimensional input vectors will be transformed into four 6-dimensional output vectors. Let's use the second token, $\mathbf{x}_2$, representing the word "train," as our illustrative example. To compute the output $\mathbf{g}_2$ for $\mathbf{x}_2$, the self-attention layer works in six steps.

4.2.1. Step 1 of Self-Attention

Compute matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ as shown below:

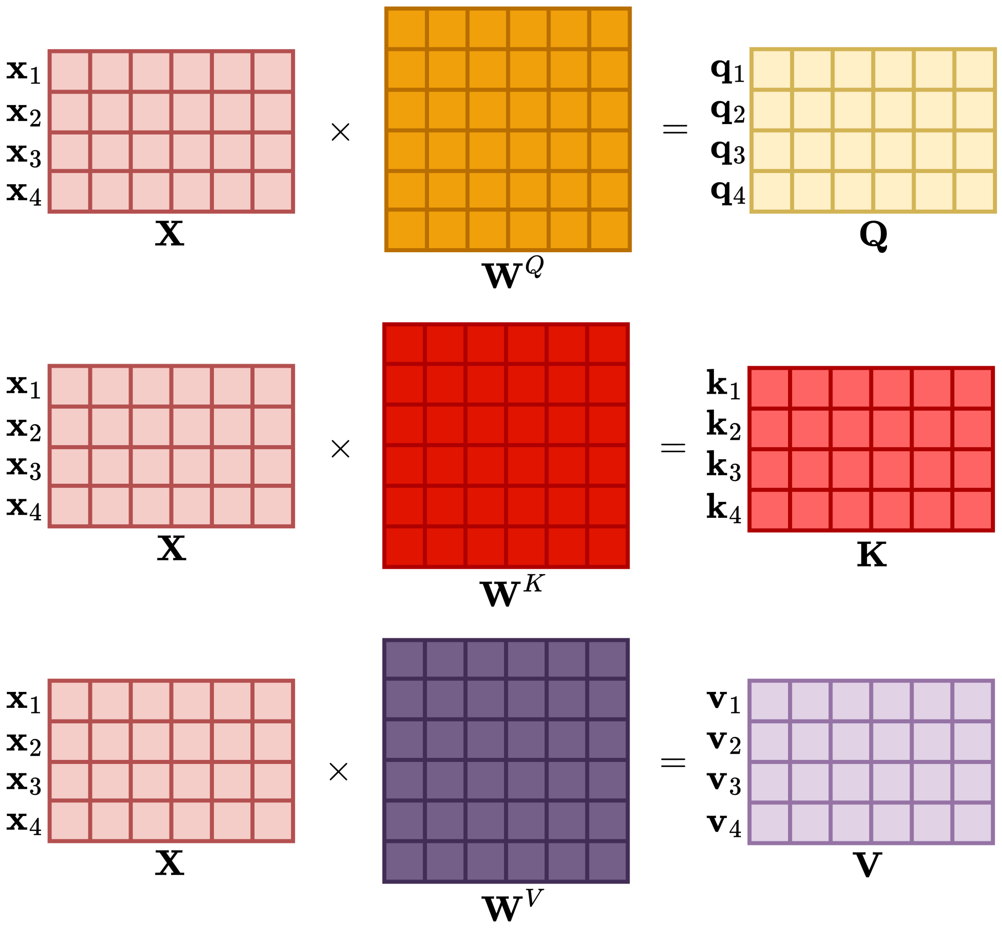


Figure 4.1: Matrix multiplication in the self-attention layer.

In the illustration, we combined the four input embeddings $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$, and $\mathbf{x}_4$ into a matrix $\mathbf{X}$. Then, we multiplied $\mathbf{X}$ by the weight matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, and $\mathbf{W}^V$ to create matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$. These matrices hold 6-dimensional query, key, and value vectors, respectively. Since the process generates the same number of query, key, and value vectors as input embeddings, each input embedding $\mathbf{x}_t$ corresponds to a query vector $\mathbf{q}_t$, a key vector $\mathbf{k}_t$, and a value vector $\mathbf{v}_t$.

4.2.2. Step 2 of Self-Attention

Taking the second token $\mathbf{x}_2$ as our example, we compute **attention scores** by taking the dot product of its query vector $\mathbf{q}_2$ with each key vector $\mathbf{k}_t$. Let's assume the resulting scores are:

$$\mathbf{q}_2 \cdot \mathbf{k}_1 = 4.90, \quad \mathbf{q}_2 \cdot \mathbf{k}_2 = 17.15, \quad \mathbf{q}_2 \cdot \mathbf{k}_3 = 9.80, \quad \mathbf{q}_2 \cdot \mathbf{k}_4 = 12.25$$

In vector format:

$$\mathbf{scores}_2 = [4.90, 17.15, 9.80, 12.25]^\top$$

4.2.3. Step 3 of Self-Attention

To obtain the **scaled scores**, we divide each attention score by the square root of the key vector's dimensionality. In our example, since the key vector has a dimensionality of 6, we divide all scores by $\sqrt{6} \approx 2.45$, yielding:

$$\mathbf{scaled\_scores}_2 = \left[\frac{4.9}{2.45}, \frac{17.15}{2.45}, \frac{9.8}{2.45}, \frac{12.25}{2.45}\right]^\top = [2,7,4,5]^\top$$

4.2.4. Step 4 of Self-Attention

We then apply the **causal mask** to the scaled scores. (If the reason for using the causal mask isn't clear yet, it will be explained in detail soon.) For the second input position, the causal mask is:

$$\mathbf{causal\_mask}_2 \overset{\text{def}}{=} [0,0,-\infty,-\infty]^\top$$

We add the scaled scores to the causal mask, resulting in the **masked scores**:

$$\mathbf{masked\_scores}_2 = \mathbf{scaled\_scores}_2 + \mathbf{causal\_mask}_2 = [2,7,-\infty,-\infty]^\top$$

4.2.5. Step 5 of Self-Attention

We apply the **softmax** function to the masked scores to produce the **attention weights**:

$$\mathbf{attention\_weights}_2 = \text{softmax}([2,7,-\infty,-\infty]^\top)$$

Since scores of $-\infty$ become zero after applying the exponential function, the attention weights for the third and fourth positions will be zero. The remaining two weights are calculated as:

$$\mathbf{attention\_weights}_2 = \left[\frac{e^2}{e^2+e^7}, \frac{e^7}{e^2+e^7}, 0, 0\right]^\top \approx [0.0067,0.9933,0,0]^\top$$

> Dividing attention scores by the square root of the key dimensionality helps prevent the dot products from growing too large in magnitude as the dimensionality increases, which could lead to extremely small gradients after applying **softmax** (due to very large negative or positive values pushing the softmax outputs to 0 or 1).

4.2.6. Step 6 of Self-Attention

We compute the output vector $\mathbf{g}_2$ for the input embedding $\mathbf{x}_2$ by taking a weighted sum of the value vectors $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, and $\mathbf{v}_4$ using the attention weights from the previous step:

$$\mathbf{g}_2 \approx 0.0067 \cdot \mathbf{v}_1 + 0.9933 \cdot \mathbf{v}_2 + 0 \cdot \mathbf{v}_3 + 0 \cdot \mathbf{v}_4$$

As you can see, the decoder's output for position 2 depends only on (or, we can say "attends only to") the inputs at positions 1 and 2, with position 2 having a much stronger influence. This effect comes from the **causal mask**, which restricts the model from attending to future positions when generating an output for a given position. This property is essential for maintaining the **autoregressive** nature of language models, ensuring that predictions for each position rely solely on previous and current inputs, not future ones.

> While this token primarily attends to itself in our example, attention patterns vary across different contexts. A token may attend strongly to other tokens providing relevant semantic or syntactic information, depending on sentence structure.

The vectors $\mathbf{q}_t$, $\mathbf{k}_t$, and $\mathbf{v}_t$ can be interpreted as follows: each input position (token or embedding) seeks information about other positions. For example, a token like "I" might look for a name in another position, allowing the model to process "I" and the name in a similar way. To enable this, each position t is assigned a query $\mathbf{q}_t$.

The self-attention mechanism calculates a **dot product** between $\mathbf{q}_t$ and every key $\mathbf{k}_p$ across all positions p. A larger dot-product indicates greater similarity between the vectors. If position p's key $\mathbf{k}_p$ aligns closely with position t's query $\mathbf{q}_t$, then position p's value $\mathbf{v}_p$ contributes more significantly to the final result.

> The concept of attention emerged before the Transformer. In 2014, Dzmitry Bahdanau, while studying under Yoshua Bengio, addressed a fundamental challenge in machine translation: enabling an RNN to focus on the most relevant parts of a sentence. Drawing from his own experience learning English—where he moved his focus between different parts of the text—Bahdanau developed a mechanism for the RNN to "decide" which input words were most important at each

translation step. This mechanism, which Bengio then termed attention, became a cornerstone of modern neural networks.

The process used to calculate $\mathbf{g}_2$ is repeated for each position in the input sequence, resulting in a set of output vectors: $\mathbf{g}_1$, $\mathbf{g}_2$, $\mathbf{g}_3$, and $\mathbf{g}_4$. Each position has its own causal mask, so when calculating $\mathbf{g}_1$, $\mathbf{g}_3$, and $\mathbf{g}_4$, a different causal mask is applied for each position. The full causal mask for all positions is shown below:

$$\mathbf{M} \overset{\text{def}}{=} \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

As you can see, the first token attends only to itself, the second to itself and the first, the third to itself and the first two, and the last to itself and all preceding tokens.

The general formula for computing attention for all positions is:

$$\mathbf{G} = \text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \overset{\text{def}}{=} \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} + \mathbf{M}\right)\mathbf{V}$$

Here, $\mathbf{Q}$ and $\mathbf{V}$ are $L \times d_k$ query and value matrices. $\mathbf{K}^\top$ is the $d_k \times L$ transposed key matrix. d_k is the dimensionality of the key, query, and value vectors, and L is the sequence length.

While we computed the attention scores explicitly for $\mathbf{x}_2$ earlier, the matrix multiplication $\mathbf{Q}\mathbf{K}^\top$ calculates the scores for all positions at once. This method makes the process much faster.

This completes the definition of self-attention.

4.3. Position-Wise Multilayer Perceptron

After the masked self-attention layer, each output vector $\mathbf{g}_t$ is individually processed by a **multilayer perceptron** (MLP). The MLP applies a sequence of additional transformations:

$$\mathbf{z}_t = \mathbf{W}_2\big(\text{ReLU}(\mathbf{W}_1\mathbf{g}_t + \mathbf{b}_1)\big) + \mathbf{b}_2$$

Here, $\mathbf{W}_1$, $\mathbf{W}_2$, $\mathbf{b}_1$, and $\mathbf{b}_2$ are learned parameters. The resulting vector $\mathbf{z}_t$ is then either passed to the next decoder block or, if it's the final decoder block, used to generate the output vector.

This component is a position-wise multilayer perceptron, which is why I use that term. The literature may refer to it as a feedforward network, dense layer, or fully connected layer, but these names can be misleading. The entire Transformer is a feedforward neural network. Additionally, dense or fully connected layers typically incorporate one weight matrix, one bias vector, and an output non-linearity. The position-wise MLP in a Transformer, however, utilizes two weight matrices, two bias vectors, and omits an output non-linearity.

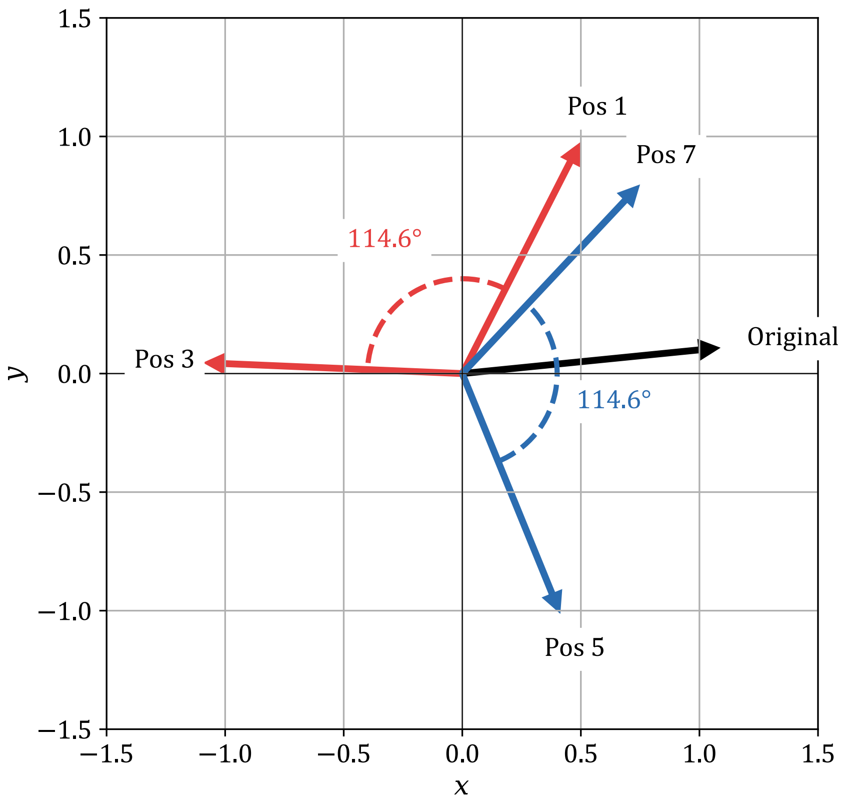
4.4. Rotary Position Embedding

The Transformer architecture, as described so far, does not inherently account for word order. The **causal mask** ensures that each token cannot attend to tokens on its right, but rearranging tokens on the left does not affect the attention weights of a given token. This is unlike RNNs, where hidden states are computed sequentially, each depending on the previous one. Changing word order in RNNs alters the hidden states and, consequently, the output. In contrast, Transformers calculate attention across all tokens at once, without sequential dependency.

To handle word order, Transformers need to incorporate positional information. A widely used method for this is **rotary position embedding** (RoPE), which applies position-dependent rotations to the query and key vectors in the attention mechanism. One key benefit of RoPE is its ability to generalize effectively to sequences longer than those seen during training. This allows models to be trained on shorter sequences—saving time and computational resources—while still supporting much longer contexts at inference.

RoPE encodes positional information by rotating the query and key vectors. This rotation occurs before the attention computation. The illustration on the next page shows how it works in 2D. The black arrow labeled "Original" shows a position-less key or query vector in self-attention. RoPE embeds positional information by rotating this vector according to the token's position.[6] The colored arrows show the resulting rotated vectors for positions 1, 3, 5, and 7.

[6] In practice, RoPE operates by rotating pairs of adjacent dimensions within query and key vectors, rather than rotating the entire vectors themselves, as we will explore shortly.



A key property of RoPE is that the angle between any two rotated vectors encodes the distance between their positions in the sequence. For example, the angle between positions 1 and 3 is the same as the angle between positions 5 and 7, since both pairs are two positions apart.
So, how do we rotate vectors? We use matrix multiplication! **Rotation matrices** are widely used in fields like computer graphics to rotate 3D scenes—one of the original purposes of GPUs (the "G" in GPU stands for graphical) before they were applied to neural network training.

In two dimensions, the rotation matrix for an angle θ is:

$$\mathbf{R}_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Let's rotate the two-dimensional vector $\mathbf{q} = [2,1]^\top$. To do this, we multiply $\mathbf{q}$ by the rotation matrix $\mathbf{R}_\theta$. The result is a new vector, representing $\mathbf{q}$ rotated counterclockwise by an angle θ.

For a 45° rotation ($\theta = \pi/4$ radians), we can use the special values $\cos(\theta) = \sin(\theta) = \frac{\sqrt{2}}{2}$. This gives us the rotation matrix:

$$\mathbf{R}_{45^\circ} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}$$

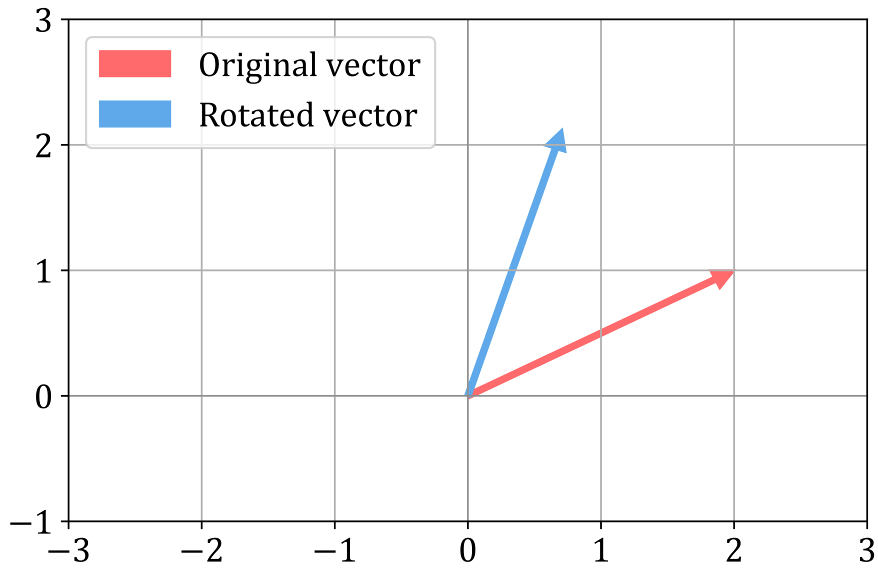
To find the rotated vector, we multiply $\mathbf{R}_{45^\circ}$ by $\mathbf{q}$:

$$\mathbf{q}\text{rotated} = \mathbf{R}_{45^\circ} \cdot \mathbf{q} = \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

Computing this multiplication step by step:

$$\mathbf{q}_{\text{rotated}} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 2 - \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 2 + \frac{\sqrt{2}}{2} \cdot 1 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2}(2-1) \\ \frac{\sqrt{2}}{2}(2+1) \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \cdot 1 \\ \frac{\sqrt{2}}{2} \cdot 3 \end{bmatrix} = \begin{bmatrix} \frac{\sqrt{2}}{2} \\ \frac{3\sqrt{2}}{2} \end{bmatrix}$$

The figure below illustrates $\mathbf{q}$ and its rotated version for $\theta = 45^\circ$:



For a position t, RoPE rotates each pair of dimensions in the query and key vectors defined as:

$$\begin{aligned} \mathbf{q}_t &= \left[q_t^{(1)}, q_t^{(2)}, \ldots, q_t^{(d_q-1)}, q_t^{(d_q)}\right]^\top \\ \mathbf{k}_t &= \left[k_t^{(1)}, k_t^{(2)}, \ldots, k_t^{(d_k-1)}, k_t^{(d_k)}\right]^\top \end{aligned}$$

Here, d_q and d_k are the (even) dimensionality of the query and key vectors. RoPE rotates pairs of dimensions indexed as $(2p-1,\ 2p)$, where each pair's index p ranges from 1 to $d_q/2$.

To split the dimensions of $\mathbf{q}_t$ into $d_q/2$ pairs, we group them like this:

$$\left[q_t^{(1)}, q_t^{(2)}\right]^\top, \left[q_t^{(3)}, q_t^{(4)}\right]^\top, \dots, \left[q_t^{(d_q-1)}, q_t^{(d_q)}\right]^\top$$

When we write $\mathbf{q}_t(p)$, it represents the pair $\left[q_t^{(2p-1)}, q_t^{(2p)}\right]$. For example, $\mathbf{q}_t(3)$ corresponds to:

$$\left[q_t^{(2\cdot3-1)}, q_t^{(2\cdot3)}\right] = \left[q_t^{(5)}, q_t^{(6)}\right]$$

Each pair p undergoes a rotation based on the token position t and a **rotation frequency** θ_p:

$$\operatorname{RoPE}\big(\mathbf{q}_t(p)\big) \stackrel{\text{def}}{=} \begin{bmatrix} \cos(\theta_p t) & -\sin(\theta_p t) \\ \sin(\theta_p t) & \cos(\theta_p t) \end{bmatrix} \begin{bmatrix} q_t^{(2p-1)} \\ q_t^{(2p)} \end{bmatrix}$$

Applying the **matrix-vector multiplication** rule, the rotation results in the following 2D vector:

$$\begin{aligned} &\operatorname{RoPE}\big(\mathbf{q}_t(p)\big) \\ &= \Big[q_t^{(2p-1)}\cos(\theta_p t) - q_t^{(2p)}\sin(\theta_p t),\ q_t^{(2p-1)}\sin(\theta_p t) \\ &+ q_t^{(2p)}\cos(\theta_p t)\Big]^\top, \end{aligned}$$

where θ_p is the rotation frequency for the p^{th} pair. It is defined as:

$$\theta_p \stackrel{\text{def}}{=} \frac{1}{\Theta^{2(p-1)/d_q}}$$

Here, Θ is a constant. Initially set to 10,000, later experiments demonstrated that higher values of Θ—such as 500,000 (used in Llama 2 and 3 series of models) or 1,000,000 (in Qwen 2 and 2.5 series)—enable support for larger context sizes (hundreds of thousands of tokens).

The full rotated embedding $\operatorname{RoPE}(\mathbf{q}_t)$ is constructed by concatenating all the rotated pairs:

$$\operatorname{RoPE}(\mathbf{q}_t) \stackrel{\text{def}}{=} \operatorname{concat}\Big[\operatorname{RoPE}\big(\mathbf{q}_t(1)\big),\ \operatorname{RoPE}\big(\mathbf{q}_t(2)\big), \dots,\ \operatorname{RoPE}\Big(\mathbf{q}_t\big(d_q/2\big)\Big)\Big]$$

Note how the rotation frequency θ_p decreases quickly for each subsequent pair because of the exponential term in the denominator. This enables RoPE to capture fine-grained local position information in the early dimensions, where rotations are more frequent, and coarse-grained global position information in the later dimensions, where rotations slow down. This combination creates richer positional encoding, allowing the model to differentiate token positions in a sequence more effectively than using a single rotation frequency across all dimensions.

To illustrate the process, consider a 6-dimensional query vector at position t and $\Theta = 10{,}000$:

$$\mathbf{q}_t = \left[q_t^{(1)}, q_t^{(2)}, q_t^{(3)}, q_t^{(4)}, q_t^{(5)}, q_t^{(6)}\right]^\top \stackrel{\text{def}}{=} [0.8, 0.6, 0.7, 0.3, 0.5, 0.4]^\top$$

First, we split it into three pairs ($d_q/2 = 3$):

$$\mathbf{q}_t(1) = \left[q_t^{(1)}, q_t^{(2)}\right] = [0.8, 0.6]^\top$$
$$\mathbf{q}_t(2) = \left[q_t^{(3)}, q_t^{(4)}\right] = [0.7, 0.3]^\top$$
$$\mathbf{q}_t(3) = \left[q_t^{(5)}, q_t^{(6)}\right] = [0.5, 0.4]^\top$$

Each pair p undergoes a rotation by angle $\theta_p t$, where:

$$\theta_p = \frac{1}{10000^{2(p-1)/d_q}}$$

Let the position t be 100. First, we calculate the rotation angles for each pair (in radians):

$$\begin{aligned}
\theta_1 &= \frac{1}{10000^{2(1-1)/6}} = \frac{1}{10000^{0/6}} = 1.0000, &\text{therefore: } \theta_1 t &= 100.00\\
\theta_2 &= \frac{1}{10000^{2(2-1)/6}} = \frac{1}{10000^{2/6}} \approx 0.0464, &\text{therefore: } \theta_2 t &= 4.64\\
\theta_3 &= \frac{1}{10000^{2(3-1)/6}} = \frac{1}{10000^{4/6}} \approx 0.0022, &\text{therefore: } \theta_3 t &= 0.22
\end{aligned}$$

The rotated pair 1 is:

$$\begin{aligned}
\text{RoPE}\big(\mathbf{q}_{100}(1)\big) &= \begin{bmatrix}\cos(100) & -\sin(100)\\ \sin(100) & \cos(100)\end{bmatrix}\begin{bmatrix}0.8\\0.6\end{bmatrix} \approx \begin{bmatrix}0.86 & 0.51\\ -0.51 & 0.86\end{bmatrix}\begin{bmatrix}0.8\\0.6\end{bmatrix}\\
&= [0.99, 0.11]^\top
\end{aligned}$$

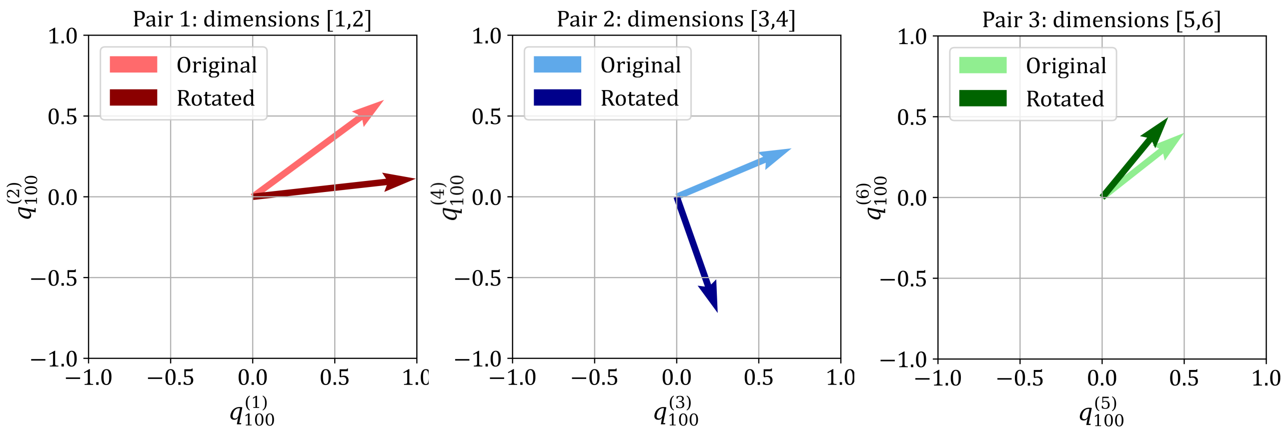
The rotated pair 2 is:

$$\text{RoPE}(\mathbf{q}_{100}(2)) = \begin{bmatrix}\cos(4.64) & -\sin(4.64)\\ \sin(4.64) & \cos(4.64)\end{bmatrix}\begin{bmatrix}0.7\\0.3\end{bmatrix} \approx \begin{bmatrix}-0.07 & 1.00\\ -1.00 & -0.07\end{bmatrix}\begin{bmatrix}0.7\\0.3\end{bmatrix}$$
$$= [0.25, -0.72]^\top$$

The rotated pair 3 is:

$$\text{RoPE}(\mathbf{q}_{100}(3)) = \begin{bmatrix}\cos(0.22) & -\sin(0.22)\\ \sin(0.22) & \cos(0.22)\end{bmatrix}\begin{bmatrix}0.5\\0.4\end{bmatrix} \approx \begin{bmatrix}0.98 & -0.21\\ 0.21 & 0.98\end{bmatrix}\begin{bmatrix}0.5\\0.4\end{bmatrix}$$
$$= [0.40, 0.50]^\top$$

These is what the original and rotated pairs look like when plotted:



The final RoPE-encoded vector is the concatenation of these pairs:

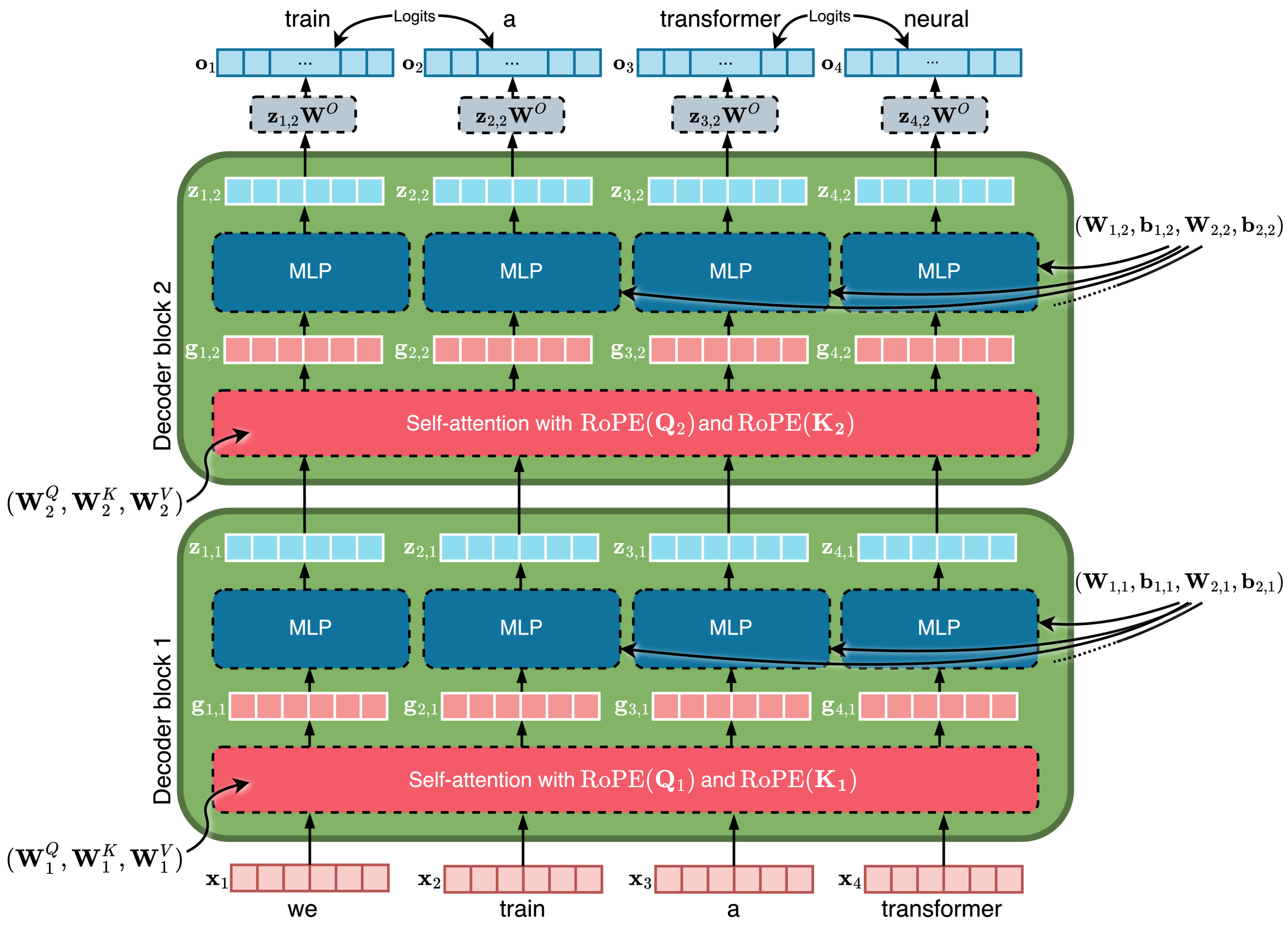
$$\text{RoPE}(\mathbf{q}_{100}) \approx [0.99, 0.11, 0.25, -0.72, 0.40, 0.50]^\top$$

The math for RoPE($\mathbf{k}_t$) is the same as for RoPE($\mathbf{q}_t$). In each decoder block, RoPE is applied to each row of the query ($\mathbf{Q}$) and key ($\mathbf{K}$) matrices within the self-attention mechanism.

> Value vectors only provide the information that is selected and combined after the attention weights are determined. Since the positional relationships are already captured in the query-key alignment, value vectors don't need their own rotary embeddings. In other words, the value vectors simply "deliver" the content once the positional-aware attention has identified where to look.

Recall that $\mathbf{Q}$ and $\mathbf{K}$ are generated by multiplying the decoder block inputs by weight matrices $\mathbf{W}^Q$ and $\mathbf{W}^K$, as illustrated in Figure 4.1. RoPE is applied immediately after obtaining $\mathbf{Q}$ and $\mathbf{K}$, and before the attention scores are calculated.

RoPE is applied across all decoder blocks, ensuring positional information flows consistently throughout the network's depth. The illustration below shows its implementation in two sequential decoder blocks.


train
Logits
a
transformer
Logits
neural
$\mathbf{o}_1$
$\mathbf{o}_2$
$\mathbf{o}_3$
$\mathbf{o}_4$
$\mathbf{z}_{1,2}\mathbf{W}^O$
$\mathbf{z}_{2,2}\mathbf{W}^O$
$\mathbf{z}_{3,2}\mathbf{W}^O$
$\mathbf{z}_{4,2}\mathbf{W}^O$
Decoder block 2
$\mathbf{z}_{1,2}$ $\mathbf{z}_{2,2}$ $\mathbf{z}_{3,2}$ $\mathbf{z}_{4,2}$
MLP
$(\mathbf{W}_{1,2}, \mathbf{b}_{1,2}, \mathbf{W}_{2,2}, \mathbf{b}_{2,2})$
$\mathbf{g}_{1,2}$ $\mathbf{g}_{2,2}$ $\mathbf{g}_{3,2}$ $\mathbf{g}_{4,2}$
Self-attention with $\mathrm{RoPE}(\mathbf{Q}_2)$ and $\mathrm{RoPE}(\mathbf{K}_2)$
$(\mathbf{W}_2^Q, \mathbf{W}_2^K, \mathbf{W}_2^V)$
Decoder block 1
$\mathbf{z}_{1,1}$ $\mathbf{z}_{2,1}$ $\mathbf{z}_{3,1}$ $\mathbf{z}_{4,1}$
MLP
$(\mathbf{W}_{1,1}, \mathbf{b}_{1,1}, \mathbf{W}_{2,1}, \mathbf{b}_{2,1})$
$\mathbf{g}_{1,1}$ $\mathbf{g}_{2,1}$ $\mathbf{g}_{3,1}$ $\mathbf{g}_{4,1}$
Self-attention with $\mathrm{RoPE}(\mathbf{Q}_1)$ and $\mathrm{RoPE}(\mathbf{K}_1)$
$(\mathbf{W}_1^Q, \mathbf{W}_1^K, \mathbf{W}_1^V)$
$\mathbf{x}_1$ $\mathbf{x}_2$ $\mathbf{x}_3$ $\mathbf{x}_4$
we
train
a
transformer

In this graph, the outputs of the second decoder block are used to compute logits for each position. This is achieved by multiplying the outputs of the final decoder block by a matrix of shape (embedding dimensionality, vocabulary size) shared across all positions. We'll explore this part in more detail when we implement the decoder model in Python.

The self-attention mechanism we've described would work as is. However, transformers typically employ an enhanced version called **multi-head attention**. This allows the model to focus on multiple aspects of information simultaneously. For example, one attention head might capture syntactic relationships, another might emphasize semantic similarities, and a third could detect long-range dependencies between tokens.

4.5. Multi-Head Attention

Once you understand self-attention, understanding multi-head attention is relatively straightforward. For each **head** h, from 1 to H, there is a separate triplet of attention matrices:

$$\left\{\left(\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V\right)\right\}_{h\in 1,\dots,H}$$

Each triplet is applied to the input vectors $\mathbf{x}_1, \dots, \mathbf{x}_4$, producing H matrices $\mathbf{G}_h$. For each head, this gives four vectors $\mathbf{g}_{h,1}, \dots, \mathbf{g}_{h,4}$, as shown in Figure 4.2 for three heads ($H = 3$). As you can see, the multi-head self-attention mechanism processes an input sequence through multiple self-attention "heads." For instance, with 3 heads, each head calculates self-attention scores for the input tokens independently. RoPE is applied separately in each head.

All input tokens $\mathbf{x}_1, \dots, \mathbf{x}_4$ are processed by all three heads, producing output matrices $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$. Each matrix $\mathbf{G}_h$ has as many rows as there are input tokens, meaning each head generates an embedding for every token. The embedding dimensionality of each $\mathbf{G}_h$ is reduced to one-third of the total embedding dimensionality. As a result, each head outputs lower-dimensional embeddings compared to the original embedding size.

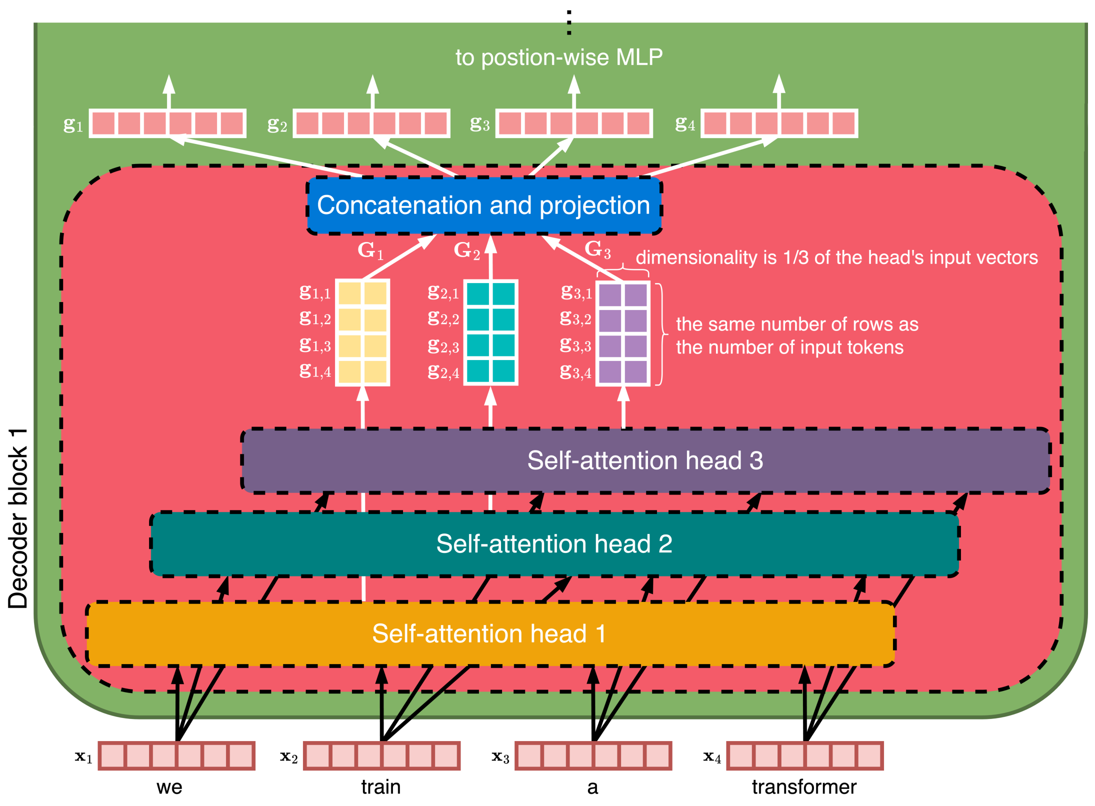
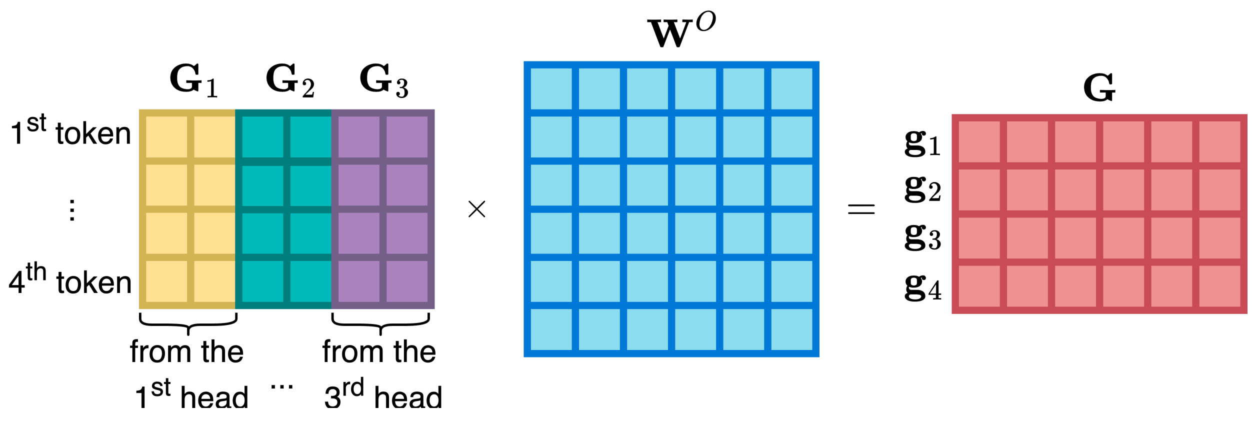

to postion-wise MLP
$\mathbf{g}_1$
$\mathbf{g}_2$
$\mathbf{g}_3$
$\mathbf{g}_4$
Concatenation and projection
$\mathbf{G}_1$
$\mathbf{G}_2$
$\mathbf{G}_3$
dimensionality is 1/3 of the head's input vectors
$\mathbf{g}_{1,1}$ $\mathbf{g}_{1,2}$ $\mathbf{g}_{1,3}$ $\mathbf{g}_{1,4}$
$\mathbf{g}_{2,1}$ $\mathbf{g}_{2,2}$ $\mathbf{g}_{2,3}$ $\mathbf{g}_{2,4}$
$\mathbf{g}_{3,1}$ $\mathbf{g}_{3,2}$ $\mathbf{g}_{3,3}$ $\mathbf{g}_{3,4}$
the same number of rows as the number of input tokens
Self-attention head 3
Self-attention head 2
Self-attention head 1
Decoder block 1
$\mathbf{x}_1$ we
$\mathbf{x}_2$ train
$\mathbf{x}_3$ a
$\mathbf{x}_4$ transformer

Figure 4.2: 3-head self-attention.

The outputs from the three heads are concatenated along the embedding dimension in the **concatenation and projection layer**, creating a single matrix that integrates information from all heads. This matrix is then transformed by the **projection matrix $\mathbf{W}^O$**, resulting in the final output matrix $\mathbf{G}$. This output is passed to the position-wise MLP:



Concatenating the matrices $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$ restores the original embedding dimensionality (e.g., 6 in this case). However, applying the trainable parameter matrix $\mathbf{W}^O$ enables the model to combine the heads' information more effectively than mere concatenation.

> Modern large language models often use up to 128 heads.

At this stage, the reader understands the Transformer model architecture at a high level. Two key technical details remain to explore: layer normalization and residual connections, both essential components that enable the Transformer's effectiveness. Let's begin with residual connections.

4.6. Residual Connection

Residual connections (or **skip connections**) are essential to the Transformer architecture. They solve the vanishing gradient problem in deep neural networks, enabling the training of much deeper models.

A network containing more than two layers is called a **deep neural network**. Training them is called **deep learning**. Before **ReLU** and residual connections, the **vanishing gradient problem** severely limited network depth. Remember that during gradient descent, partial derivatives update all parameters by taking small steps in the opposite direction of the gradient. In deeper networks, these updates become very small in earlier layers (those closer to the input), effectively halting parameter adjustment. Residual connections strengthen these updates by creating pathways for the gradient to "bypass" certain layers, hence the term skip connections.

To better understand the vanishing gradient problem, let's analyze a 3-layer neural network expressed as a **composite function**:

$$f(x) = f_3\left(f_2(f_1(x))\right),$$

where f_1 represents the first layer, f_2 represents the second layer, and f_3 represents the third (output) layer. Let these functions be defined as follows:

$$\begin{aligned} z = f_1(x) &\overset{\text{def}}{=} w_1 x + b_1 \\ r = f_2(z) &\overset{\text{def}}{=} w_2 z + b_2 \\ y = f_3(r) &\overset{\text{def}}{=} w_3 r + b_3 \end{aligned}$$

Here, w_l and b_l are scalar weights and biases for each layer $l \in \{1,2,3\}$.

Let's define the loss function L in terms of the network output $f(x)$ and the true label y as $L(f(x), y)$. The gradient of the loss L with respect to w_1, denoted as $\frac{\partial L}{\partial w_1}$, is given by:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot \frac{\partial f_3}{\partial f_2} \cdot \frac{\partial f_2}{\partial f_1} \cdot \frac{\partial f_1}{\partial w_1},$$

where:

$$\frac{\partial f_3}{\partial f_2} = w_3, \quad \frac{\partial f_2}{\partial f_1} = w_2, \quad \frac{\partial f_1}{\partial w_1} = x$$

So, we can write:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f_3} \cdot w_3 \cdot w_2 \cdot x$$

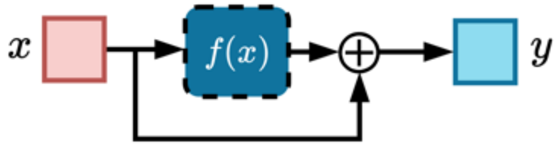
The vanishing gradient problem occurs when weights like w_2 and w_3 are small (less than 1). When multiplied together, they produce even smaller values, causing the gradient for earlier weights such as w_1 to approach zero. This issue becomes particularly severe in networks with many layers.

Take large language models as an example. These networks often include 32 or more decoder blocks. To simplify, assume all blocks are fully connected layers. If the average weight value is around 0.5, the gradient for the input layer parameters becomes $0.5^{32} \approx 0.0000000002$. This is extremely small. After multiplying by the learning rate, updates to the early layers become negligible. As a result, the network stops learning effectively.

Residual connections offer a solution to the vanishing gradient problem by creating shortcuts in the gradient computation path. The basic idea is simple: instead of passing only the output of a layer to the next one, the layer's input is added to its output. Mathematically, this is written as:

$$y = f(x) + x,$$

where x is the input, $f(x)$ is the layer's computed function, and y is the output. This addition forms the residual connection. Graphically, it is shown in the picture on the right. In this illustration, the input x is processed both through the layer (represented as $f(x)$) and added directly to the layer's output.



Now let's introduce residual connections into our 3-layer network. We'll see how this changes gradient computation and mitigates the vanishing gradient issue. Starting with the original network $f(x) = f_3\big(f_2(f_1(x))\big)$, let's add residual connections to layers 2 and 3:

$$\begin{aligned} z &\leftarrow f_1(x) \overset{\text{def}}{=} w_1 x + b_1 \\ r &\leftarrow f_2(z) \overset{\text{def}}{=} w_2 z + b_2 + z \\ y &\leftarrow f_3(r) \overset{\text{def}}{=} w_3 r + b_3 + r \end{aligned}$$

Our composite function becomes:

$$f(x) = w_3[w_2(w_1x + b_1) + b_2 + w_1x + b_1] + b_3 + w_2(w_1x + b_1) + b_2 + w_1x + b_1$$

Now, let's calculate the gradient of the loss L with respect to w_1:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_1}$$

Expanding $\frac{\partial f}{\partial w_1}$:

$$\begin{aligned} \frac{\partial f}{\partial w_1} &= \frac{\partial}{\partial w_1}\begin{bmatrix} \big(w_3\big(w_2(w_1x + b_1) + b_2 + (w_1x + b_1)\big) + b_3\big) + \\ \big(w_2(w_1x + b_1) + b_2 + (w_1x + b_1)\big) \end{bmatrix} \\ &= (w_3w_2 + w_3 + w_2 + 1) \cdot x \end{aligned}$$

Therefore, the full gradient is:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot (w_3w_2 + w_3 + w_2 + 1) \cdot x$$

Comparing this to our original gradient without residual connections:

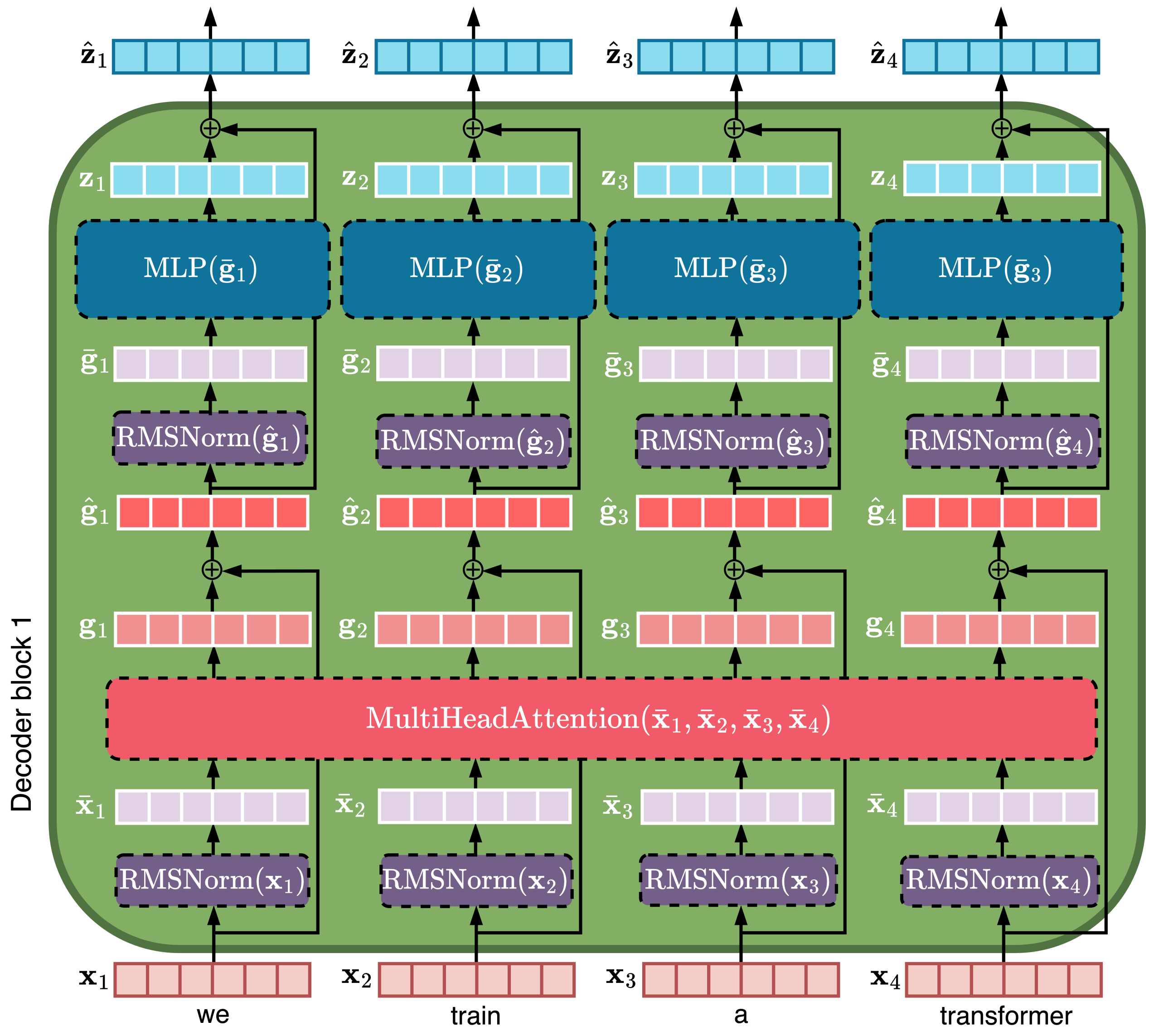
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot w_3 \cdot w_2 \cdot x$$

We observe that residual connections introduce three additional terms: w_3, w_2, and 1. This guarantees that the gradient will not vanish completely, even when w_2 and w_3 are small, due to the added constant term 1.

For example, if $w_2 = w_3 = 0.5$ as in the previous case:

- **Without residual connections**: $0.5 \cdot 0.5 = 0.25$
- **With residual connections**: $0.5 \cdot 0.5 + 0.5 + 0.5 + 1 = 2.25$

The illustration below depicts an encoding block with residual connections:



As shown, each decoder block includes two residual connections. The layers are now named like Python objects, which we will implement shortly. Additionally, two RMSNorm layers have been added. Let's discuss their purpose.

4.7. Root Mean Square Normalization

The RMSNorm layer applies **root mean square normalization** to the input vector. This operation takes place just before the vector enters the self-attention layer and the position-wise MLP. Let's illustrate this with a three-dimensional vector.

Suppose we have a vector $\mathbf{x} = [x^{(1)}, x^{(2)}, x^{(3)}]^\top$. To apply RMS normalization, we first calculate the **root mean square** (**RMS**) of the vector:

$$\text{RMS}(\mathbf{x}) = \sqrt{\frac{1}{3}\sum_{i=1}^{3}(x^{(i)})^2} = \sqrt{\frac{1}{3}[(x^{(1)})^2 + (x^{(2)})^2 + (x^{(3)})^2]}$$

Then, we normalize the vector by dividing each component by the RMS value to obtain $\tilde{\mathbf{x}}$:

$$\tilde{\mathbf{x}} = \frac{\mathbf{x}}{\text{RMS}(\mathbf{x})} = \left[\frac{x^{(1)}}{\text{RMS}(\mathbf{x})}, \frac{x^{(2)}}{\text{RMS}(\mathbf{x})}, \frac{x^{(3)}}{\text{RMS}(\mathbf{x})}\right]^\top$$

Finally, we apply the scale factor γ to each dimension of $\tilde{\mathbf{x}}$:

$$\bar{\mathbf{x}} = \text{RMSNorm}(\mathbf{x}) \overset{\text{def}}{=} \boldsymbol{\gamma} \odot \tilde{\mathbf{x}} = \left[\gamma^{(1)}\tilde{x}^{(1)}, \gamma^{(2)}\tilde{x}^{(2)}, \gamma^{(3)}\tilde{x}^{(3)}\right]^\top,$$

where $\odot$ denotes the **element-wise product**. The vector $\boldsymbol{\gamma}$ is a trainable parameter, and each RMSNorm layer has its own independent $\boldsymbol{\gamma}$.

The primary purpose of RMSNorm is to stabilize training by keeping the scale of the input to each layer consistent. This improves numerical stability, helping to prevent gradient updates that are excessively large or small.

Now that we've covered the key components of the Transformer architecture, let's summarize how a decoder block processes its input:

1. The input embeddings $\mathbf{x}_t$ first go through RMS normalization.
2. The normalized embeddings $\bar{\mathbf{x}}_t$ are processed by the multi-head self-attention mechanism, with RoPE applied to key and query vectors.
3. The self-attention output $\mathbf{g}_t$ is added to the original input $\mathbf{x}_t$ (residual connection).
4. This sum, $\hat{\mathbf{g}}_t$, undergoes RMS normalization again.
5. The normalized sum $\bar{\mathbf{g}}_t$ is passed through the multilayer perceptron.
6. The perceptron output $\mathbf{z}_t$ is added to the pre-RMS-normalization vector $\hat{\mathbf{g}}_t$ (another residual connection).
7. The result, $\hat{\mathbf{z}}_t$, is the output of the decoder block, serving as input for the next block (or the final output layer if it's the last block).

This sequence is repeated for each decoder block in the Transformer.

4.8. Key-Value Caching

During **training**, the decoder can process all positions in parallel because at each block it computes the query, key, and value matrices, $\mathbf{Q} = \mathbf{X}\mathbf{W}^Q$, $\mathbf{K} = \mathbf{X}\mathbf{W}^K$, and $\mathbf{V} = \mathbf{X}\mathbf{W}^V$, for the entire sequence $\mathbf{X}$. However, during an autoregressive (left-to-right) **inference**, tokens must be generated one at a time. Normally, each time we generate a new token, we would have to:

1. Calculate the key, query, and value vectors for the new token.
2. Recalculate the key and value matrices for all previous tokens.
3. Merge these with the new token's key and value vectors to compute self-attention for the new token.

Key-value caching skips step 2 by saving the key and value matrices from earlier tokens, avoiding repeated calculations. Since $\mathbf{W}^K$ and $\mathbf{W}^V$ are fixed after training, the key and value vectors of earlier tokens stay constant during inference. These vectors can be stored ("cached") after being computed once. For every new token:

- Its key and value vectors are computed using $\mathbf{W}^K$ and $\mathbf{W}^V$.
- These vectors are appended to the cached key-value pairs for self-attention.

Query vectors, however, are not cached because they depend on the current token being processed. Every time a new token is added, its query vector must be computed on-the-fly to attend to all cached keys and values.

This approach eliminates reprocessing the rest of the sequence, cutting computation significantly for long sequences. In each decoder block, cached keys and values are stored per attention head with shapes $(L \times d_h)$ for both matrices, where L grows by one with each new token, and d_h is the dimensionality of the query, key, and value vectors for that head. For a model with H attention heads, the combined key and value caches in each decoder block have shapes $(H \times L \times d_h)$.

> RoPE applies position-dependent rotations to vectors, but this doesn't interfere with caching. When a new token arrives, it simply takes the next available position index (if the sequence has L tokens, the new one becomes position $L + 1$), while previously processed tokens retain their original positions from 1 through L. This means the cached keys and values, already rotated according to their respective positions,

> remain unchanged. The rotation is only applied to the new token at position $L + 1$.

Now that we understand how the Transformer operates, we're ready to start coding.

4.9. Transformer in Python

Let's begin implementing the decoder in Python by defining the `AttentionHead` class:

```python
class AttentionHead(nn.Module):
    def __init__(self, emb_dim, d_h):
        super().__init__()
        self.W_Q = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_K = nn.Parameter(torch.empty(emb_dim, d_h))
        self.W_V = nn.Parameter(torch.empty(emb_dim, d_h))
        self.d_h = d_h

    def forward(self, x, mask):
        Q = x @ self.W_Q ❶
        K = x @ self.W_K
        V = x @ self.W_V ❷

        Q, K = rope(Q), rope(K) ❸

        scores = Q @ K.transpose(-2, -1) / math.sqrt(self.d_h
) ❹
        masked_scores = scores.masked_fill(mask == 0, float("
-inf")) ❺
        attention_weights = torch.softmax(masked_scores, dim=
-1) ❻
        return attention_weights @ V ❼
```

This class implements a single attention head in the multi-head attention mechanism. In the constructor, we initialize three trainable weight matrices: the query matrix `W_Q`, the key matrix `W_K`, and the value matrix `W_V`. Each of these is a `Parameter` tensor of shape `(emb_dim, d_h)`, where `emb_dim` is the input embedding dimension and `d_h` is the dimensionality of the query, key, and value vectors for this attention head.

In the `forward` method:

- Lines ❶ and ❷ compute the query, key, and value matrices by multiplying the input vector `x` with the respective weight matrices. Given that `x` has shape (`batch_size`, `seq_len`, `emb_dim`), `Q`, `K`, and `V` each have shape (`batch_size`, `seq_len`, `d_h`).

- Line ❸ applies the rotary positional encoding to `Q` and `K`. After the query and key vectors are rotated, line ❹ computes the attention scores. Here's a breakdown:
    - `K.transpose(-2, -1)` swaps the last two dimensions of `K`. If `K` has shape (`batch_size`, `seq_len`, `d_h`), transposing it results in (`batch_size`, `d_h`, `seq_len`). This prepares `K` for matrix multiplication with `Q`.
    - `Q @ K.transpose(-2, -1)` performs batch matrix multiplication, resulting in a tensor of attention scores of shape (`batch_size`, `seq_len`, `seq_len`).
    - As mentioned in Section 4.2, we divide by `sqrt(d_h)` for numerical stability.

> When the matrix multiplication operator `@` is applied to tensors with more than two dimensions, PyTorch uses **broadcasting**. This technique handles dimensions that aren't directly compatible with the `@` operator, which is normally defined only for two-dimensional tensors (matrices). In this case, PyTorch treats the first dimension as the batch dimension, performing the matrix multiplication separately for each example in the batch. This process is known as **batch matrix multiplication**.

- Line ❺ applies the causal mask. The `mask` tensor has the shape (`seq_len`, `seq_len`) and contains 0s and 1s. The `masked_fill` function replaces all cells in the input matrix where `mask == 0` with negative infinity. This prevents attention to future tokens. Since the `mask` lacks the batch dimension while `scores` includes it, PyTorch uses broadcasting to apply the `mask` to the `scores` of each sequence in the batch.

- Line ❻ applies softmax to the `scores` along the last dimension, turning them into attention weights. Then, line ❼ computes the output by multiplying these attention weights with `V`. The resulting output has the shape (`batch_size`, `seq_len`, `d_h`).

Given the attention head class, we can now define the `MultiHeadAttention` class:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        d_h = emb_dim // num_heads ❶
        self.heads = nn.ModuleList([
            AttentionHead(emb_dim, d_h)
            for _ in range(num_heads)
        ]) ❷
        self.W_O = nn.Parameter(torch.empty(emb_dim, emb_dim)
) ❸

    def forward(self, x, mask):
        head_outputs = [head(x, mask) for head in self.heads]
❹

        x = torch.cat(head_outputs, dim=-1) ❺
        return x @ self.W_O ❻
```

In the constructor:

- Line ❶ calculates `d_h`, the dimensionality of each attention head, by dividing the model's embedding dimensionality `emb_dim` by the number of heads.
- Line ❷ creates a `ModuleList` containing `num_heads` instances of `AttentionHead`. Each head takes the input dimensionality `emb_dim` and outputs a vector of size `d_h`.
- Line ❸ initializes `W_O`, a learnable **projection matrix** with shape (`emb_dim`, `emb_dim`) to combine the outputs from all attention heads.

In the `forward` method:

- Line ❹ applies each attention head to the input `x` of shape (`batch_size`, `seq_len`, `emb_dim`). Each head's output has shape (`batch_size`, `seq_len`, `d_h`).
- Line ❺ concatenates all heads' outputs along the last dimension. The resulting `x` has shape (`batch_size`, `seq_len`, `emb_dim`) since `num_heads * d_h = emb_dim`.
- Line ❻ multiplies the concatenated output by the projection matrix `W_O`. The output has the same shape as input.

Now that we have multi-head attention, the last piece needed for the decoder block is the position-wise multilayer perceptron. Let's define it:

```
class MLP(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.W_1 = nn.Parameter(torch.empty(emb_dim, emb_dim
* 4))
        self.B_1 = nn.Parameter(torch.empty(emb_dim * 4))
        self.W_2 = nn.Parameter(torch.empty(emb_dim * 4, emb_
dim))
        self.B_2 = nn.Parameter(torch.empty(emb_dim))

    def forward(self, x):
        x = x @ self.W_1 + self.B_1 ❶
        x = torch.relu(x) ❷
        x = x @ self.W_2 + self.B_2 ❸
        return x
```

In the constructor, we initialize learnable weights and biases.

In the `forward` method:

- Line ❶ multiplies the input `x` by the weight matrix `W_1` and adds the bias vector `B_1`. The input has shape (`batch_size`, `seq_len`, `emb_dim`), so the result has shape (`batch_size`, `seq_len`, `emb_dim * 4`).
- Line ❷ applies the **ReLU** activation function element-wise, adding non-linearity.

- Line ❸ multiplies the result by the second weight matrix `W_2` and adds the bias vector `B_2`, reducing the dimensionality back to (`batch_size`, `seq_len`, `emb_dim`).

The first linear transformation expands to 4 times the embedding dimensionality (`emb_dim * 4`) to provide the network with greater capacity for learning complex patterns and relationships between variables. The 4x factor balances expressiveness and efficiency. After expanding the dimensionality, it's compressed back to the original embedding dimensionality (`emb_dim`). This ensures compatibility with residual connections, which require matching dimensionalities. Empirical results support this expand-and-compress approach as an effective trade-off between computational cost and performance.

With all components defined, we're ready to set up the complete decoder block:

```python
class DecoderBlock(nn.Module):
    def __init__(self, emb_dim, num_heads):
        super().__init__()
        self.norm1 = RMSNorm(emb_dim)
        self.attn = MultiHeadAttention(emb_dim, num_heads)
        self.norm2 = RMSNorm(emb_dim)
        self.mlp = MLP(emb_dim)

    def forward(self, x, mask):
        attn_out = self.attn(self.norm1(x), mask) ❶
        x = x + attn_out ❷
        mlp_out = self.mlp(self.norm2(x)) ❸
        x = x + mlp_out ❹
        return x
```

The `DecoderBlock` class represents a single decoder block in a Transformer model. In the constructor, we set up the necessary layers: two `RMSNorm` layers, a `MultiHeadAttention` instance (configured with the embedding dimensionality and number of heads), and an `MLP` layer.

In the `forward` method:

- Line ❶ applies `RMSNorm` to the input `x`, which has shape (`batch_size`, `seq_len`, `emb_dim`). The output of `RMSNorm` keeps this shape. This normalized tensor is then passed to the multi-head attention layer, which outputs a tensor of the same shape.

- Line ❷ adds a **residual connection** by combining the attention output `attn_out` with the original input `x`. The shape doesn't change.
- Line ❸ applies the second `RMSNorm` to the result from the residual connection, retaining the same shape. This normalized tensor is then passed through the MLP, which outputs another tensor with shape (`batch_size`, `seq_len`, `emb_dim`).
- Line ❹ adds a second residual connection, combining `mlp_out` with its unnormalized input. The decoder block's final output shape is (`batch_size`, `seq_len`, `emb_dim`), ready for the next decoder block or the final output layer.

With the decoder block defined, we can now build the decoder transformer language model by stacking multiple decoder blocks sequentially:

```
class DecoderLanguageModel(nn.Module):
    def __init__(
        self, vocab_size, emb_dim,
        num_heads, num_blocks, pad_idx
    ):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size, emb_dim,
            padding_idx=pad_idx
        ) ❶
        self.layers = nn.ModuleList([
            DecoderBlock(emb_dim, num_heads) for _ in range(n
um_blocks)
        ]) ❷
        self.output = nn.Parameter(torch.rand(emb_dim, vocab_
size)) ❸

    def forward(self, x):
        x = self.embedding(x) ❹
        _, seq_len, _ = x.shape
        mask = torch.tril(torch.ones(seq_len, seq_len, device
=x.device)) ❺
```

```
        for layer in self.layers: ❻
            x = layer(x, mask)
        return x @ self.output ❼
```

In the constructor of the `DecoderLanguageModel` class:

- Line ❶ creates an embedding layer that converts input token indices to dense vectors. The `padding_idx` specifies the ID of the padding token, ensuring that padding tokens are mapped to zero vectors.
- Line ❷ creates a `ModuleList` with `num_blocks` `DecoderBlock` instances, forming the stack of decoder layers.
- Line ❸ defines a matrix to project the last decoder block's output to logits over the vocabulary, enabling next token prediction.

In the `forward` method:

- Line ❹ converts the input token indices to embeddings. The input tensor `x` has shape (`batch_size`, `seq_len`); the output has shape (`batch_size`, `seq_len`, `emb_dim`).
- Line ❺ creates the **causal mask**.
- Line ❻ applies each decoder block to the input tensor `x` with shape (`batch_size`, `seq_len`, `emb_dim`), producing an output tensor of the same shape. Each block refines the sequence and passes it to the next until the final block.
- Line ❼ projects the output of the final decoder block to vocabulary-sized logits by multiplying it with the `self.output` matrix, which has shape (`emb_dim`, `vocab_size`). After this batched matrix multiplication, the final output has shape (`batch_size`, `seq_len`, `vocab_size`), providing scores for each token in the vocabulary at each position in the input sequence. This output can then be used to generate the model's predictions as we will discuss in the next chapter.

The training loop for `DecoderLanguageModel` is the same as for the RNN (Section 3.6), so it is not repeated here for brevity. Implementations of `RMSNorm` and `RoPE` are also skipped. Training data is prepared just like for the RNN: the target sequence is offset by one position relative to the input sequence, as described in Section 3.7. The complete code for training the decoder language model is available in the thelmbook.com/nb/4.1 notebook.

In the notebook, I used these hyperparameter values: `emb_dim = 128`, `num_heads = 8`, `num_blocks = 2`, `batch_size = 128`, `learning_rate = 0.001`, `num_epochs = 1`, and `context_size = 30`. With these settings, the model achieved a perplexity of 55.19, improving on the RNN's 72.23. This is a good result given the comparable number of trainable parameters (8,621,963 for the Transformer vs. 8,292,619 for the RNN). The real strengths of transformers, however, become apparent at larger scales of model size, context length, and training data. Reproducing experiments at such scales in this book is, of course, impractical.

Let's look at some continuations of the prompt "The President" generated by the decoder model at later training steps:

```
The President has been in the process of a new deal to make a
decision on the issue .

The President 's office said the government had `` no intenti
on of making any mistakes '' .

The President of the United States has been a key figure for
the first time in the past ## years .
```

The "#" characters in the training data represent individual digits. For example, "##" likely represents the number of years.

***

If you've made it this far, well done! You now understand the mechanics of language models. But understanding the mechanics alone won't let you fully appreciate what modern language models are capable of. To truly understand, you need to work with one.

In the next chapter, we'll explore large language models (LLMs). We'll discuss why they're called *large* and what's so special about the size. Then, we'll cover how to finetune an existing LLM for practical tasks like question answering and document classification, as well as how to use LLMs to address a variety of real-world problems.

Chapter 5. Large Language Model

Large language models have transformed NLP through their remarkable capabilities in text generation, translation, and question-answering. But how can a model trained solely to predict the next word achieve these results? The answer lies in two factors: scale and supervised finetuning.

5.1. Why Larger Is Better

LLMs are built with a large number of parameters, large context windows, and trained on large corpora backed by substantial computational resources. This scale enables them to learn complex language patterns and even memorize information.

Creating a **chat LM**, capable of handling dialogue and following complex instructions, involves two stages. The first stage is **pretraining** on a massive dataset, often containing trillions of tokens. In this phase, the model learns to predict the next token based on context—similar to what we did with the RNN and decoder models, but at a vastly larger scale.

With more parameters and extended context windows, the model aims to "understand" the context as deeply as possible to improve the next token prediction and minimize the **cross-entropy** loss. For example, consider this context:

```
The CRISPR-Cas9 technique has revolutionized genetic engineer
ing by enabling precise modifications to DNA sequences. The p
rocess uses a guide RNA to direct the Cas9 enzyme to a specif
ic location in the genome. Once positioned, Cas9 acts like mo
lecular scissors, cutting the DNA strand. This cut activates
the cell's natural repair mechanisms, which scientists can ex
ploit to
```

To accurately predict the next token, the model must know:

1) about CRISPR-Cas9 and its components, such as guide RNA and Cas9 enzyme,
2) how CRISPR-Cas9 works—locating specific DNA sequences and cutting DNA,
3) about cellular repair mechanisms, and
4) how these mechanisms enable gene editing.

A well-trained LLM might suggest continuations like "insert new genetic material" or "delete unwanted genes." Choosing "insert" or "delete" over vague terms like "change" or "fix" requires encoding the context into embedding vectors that reflect a deeper understanding of the gene-editing process, rather than relying on surface-level patterns as count-based models do.

It's intuitive to think that if words and paragraphs can be represented by dense embedding vectors, then entire documents or complex explanations could theoretically be represented this way too. However, before LLMs were discovered, NLP researchers believed embeddings could only represent basic concepts like "animal," "building," "economy," "technology," "verb," or "noun." This belief is evident in the conclusion of one of the most influential papers of the 2010s, which detailed the training of a state-of-the-art language model at that time:

> *"As with all text generated by language models, the sample does not make sense beyond the level of short phrases. The realism could perhaps be improved with a larger network and/or more data. However, it seems futile to expect meaningful language from a machine that has never been exposed to the sensory world to which language refers."* (Alex Graves, "Generating Sequences With RNNs," 2014)

GPT-3 showed some ability to continue relatively complex patterns. But only with GPT-3.5—able to handle multi-stage dialogue and follow elaborate instructions—it became clear that something unexpected happens when a language model surpasses a certain parameter scale and is pretrained on a sufficiently large corpus.

Scale is fundamental to building a capable LLM. Let's look at the core features that make LLMs "large" and how these features contribute to their capabilities.

5.1.1. Large Parameter Count

One of the most striking features of LLMs is the sheer number of parameters they contain. While our decoder model has around 8 million parameters, state-of-the-art LLMs can reach hundreds of billions or even trillions of parameters.

In a transformer model, the number of parameters is largely determined by the embedding dimensionality (`emb_dim`) and the number of decoder blocks (`num_blocks`). As these values increase, the parameter count grows quadratically with embedding dimensionality in the self-attention and MLP layers, and linearly with the number of decoder blocks. Doubling the embedding dimensionality roughly quadruples the number of parameters in the attention and MLP components of each decoder block.

> **Open-weight models** are models with publicly accessible trained parameters. These can be downloaded and used for tasks like text generation or finetuned for specific applications. However, while the weights are open, the model's license governs its permitted uses, including whether commercial use is allowed. Licenses like Apache 2.0 and MIT permit unrestricted commercial use, but you should always review the license to confirm your intended use aligns with the creators' terms.

The table below shows key features of several open-weight LLMs compared to our tiny model:

	`num_blocks`	`emb_dim`	`num_heads`	`vocab_size`
Our model	2	128	8	32,011
Llama 3.1 8B	32	4,096	32	128,000
Gemma 2 9B	42	3,584	16	256,128
Gemma 2 27B	46	4,608	32	256,128
Llama 3.1 70B	80	8,192	64	128,000
Llama 3.1 405B	126	16,384	128	128,000

By convention, the number before "B" in the name of an open-weight model indicates its total number of parameters in billions.

> If you were to store each parameter of a 70B model as a 32-bit float number, it would require about 280GB of RAM—more storage than the Apollo 11 guidance computer had by a factor of over 30 million times.

This massive number of parameters allows LLMs to learn and represent a vast amount of information about grammar, semantics, world knowledge, and exhibit reasoning capabilities.

5.1.2. Large Context Size

Another crucial aspect of LLMs is their ability to process and maintain much larger contexts than earlier models. While our decoder model used a context of only 30 tokens, modern LLMs can handle contexts of thousands—and sometimes even millions—of tokens.

> GPT-3's 2,048-token context could accommodate roughly 4 pages of text. In contrast, Llama 3.1's 128,000-token context is large enough to fit the entire text of *"Harry Potter and the Sorcerer's Stone"* with room to spare.

The key challenge with processing long texts in transformer models lies in the self-attention mechanism's computational complexity. For a sequence of length n, self-attention requires computing attention scores between every pair of tokens, resulting in quadratic $O(n^2)$ time and space complexity. This means that doubling the input length quadruples both the memory requirements and computational cost. This quadratic scaling becomes particularly problematic for long documents—for instance, a 10,000-token input would require computing and storing 100 million attention scores for each attention layer.

The increased context size is made possible through architectural improvements and optimizations in attention computation. Techniques like **grouped-query attention** and **FlashAttention** (which are beyond the scope of this book) enable efficient memory management, allowing LLMs to handle much larger contexts without excessive computational costs.

LLMs typically undergo pretraining on shorter contexts around 4K-8K tokens, as the attention mechanism's quadratic complexity makes training on long sequences computationally intensive. Additionally, most training data naturally consists of shorter sequences.

Long-context capabilities emerge through **long-context pretraining**, a specialized stage following initial training. This process involves:

1. **Incremental training for longer contexts**: The model's context window gradually expands from 4,000-8,000 tokens to 128,000-256,000 tokens through a series of incremental stages. Each stage increases the context length and continues training until the model meets two key criteria: restoring its performance on short-context tasks while

successfully handling longer-context challenges like "needle in a haystack" evaluations.

> A **needle in a haystack** test evaluates a model's ability to identify and utilize relevant information buried within a very long context, typically by placing a crucial piece of information early in the sequence and asking a question that requires retrieving that specific detail from among thousands of tokens of unrelated text.

2. **Efficient scaling for self-attention**: To handle the computational demands of self-attention's quadratic scaling with sequence length, the approach implements **context parallelism**. This method splits input sequences into manageable chunks and uses an all-gather mechanism for memory-efficient processing.

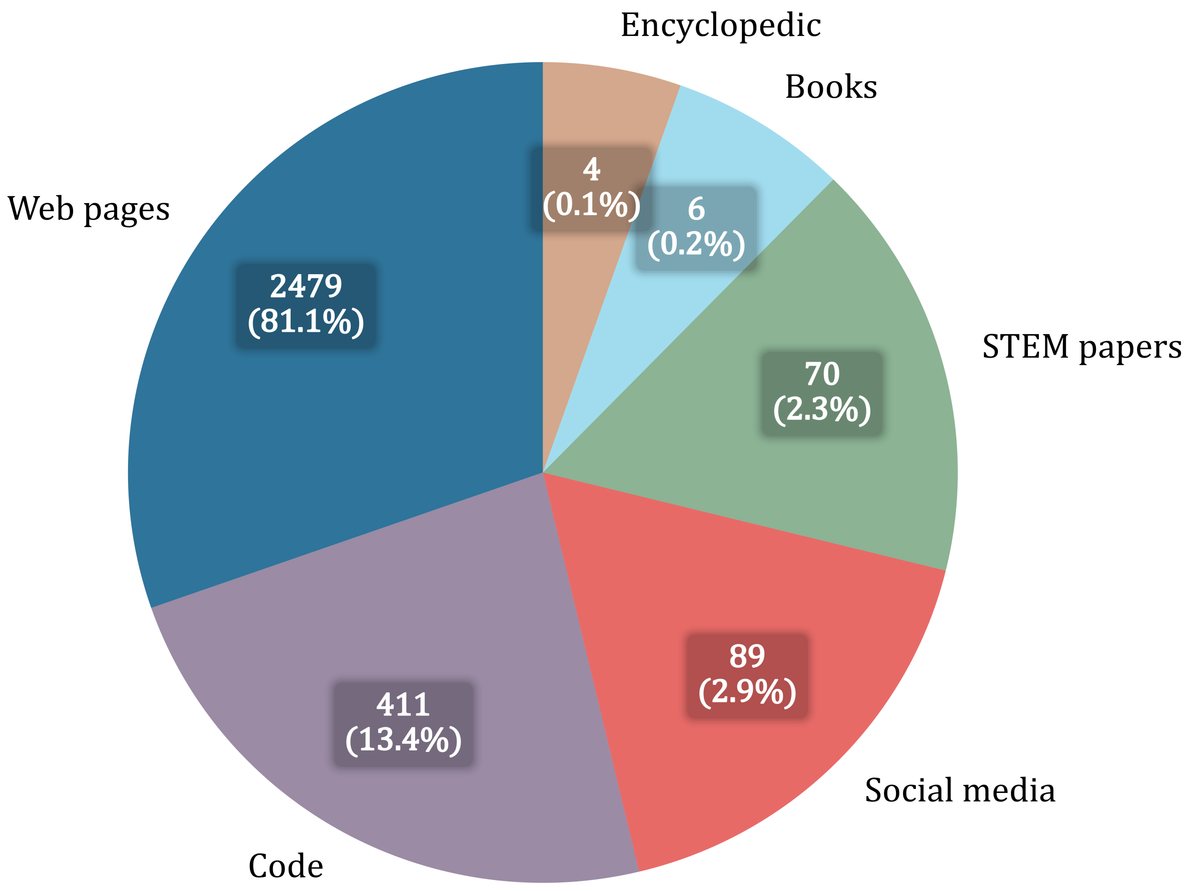
> **All-gather** is a collective communication operation in distributed computing where each GPU shares its local data with all other GPUs, aggregating the data so that every GPU ends up with a complete, concatenated dataset.

5.1.3. Large Training Dataset

The third factor behind LLMs' capabilities is the size of the corpus used for training. While our decoder was trained on a small corpus of news sentences with about 25 million tokens, modern LLMs use datasets with trillions of tokens. These datasets often include:

1) books and literature from different genres and eras,
2) web pages and online articles on diverse topics,
3) academic papers and scientific studies,
4) social media posts and discussions, and
5) code repositories and technical documents.

The diversity and scale of these datasets allow LLMs to learn a broad vocabulary, understand multiple languages, acquire knowledge on a wide array of topics—from history and science to current events and pop culture—adapt to various writing styles and formats, and acquire basic reasoning and problem-solving skills.



The illustration above depicts the composition of LLM training datasets, using the open **Dolma** dataset as an example. Segments represent different document types, with sizes scaled logarithmically to prevent web pages—the largest category—from overwhelming the visualization. Each segment shows both token count (in billions) and percentage of the corpus. While Dolma's 3 trillion tokens are substantial, they fall short of more recent datasets like Qwen 2.5's 18 trillion tokens, a number likely to grow in future iterations.

> It would take approximately 51,000 years for a human to read the entire Dolma dataset, reading 8 hours every day at 250 words per minute.

Since neural language models train on such vast corpora, they typically process the data just once. This **single-epoch training** approach prevents **overfitting** while reducing computational demands. Processing these enormous datasets multiple times would be extremely time-consuming and may not yield significant additional benefits.

5.1.4. Large Amount of Compute

If you tried to process 3 trillion tokens of the Dolma dataset on a single modern GPU, it would take over 100 years—which helps explain why major language models require massive computing clusters. Training an LLM demands significant computing power, often measured in **FLOPs** (**floating-point operations**) or GPU-hours. For context, while training our decoder model might take a few hours on a single GPU, modern LLMs can require thousands of GPUs running for months.

The computational demands grow with three main factors:

1) the number of parameters in the model,
2) the size of the training corpus, and
3) the context length used during training.

For example, training the Llama 3.1 series of models consumed approximately 40 million GPU-hours—equivalent to running a single GPU continuously for almost 4600 years. Llama 3.1's training process uses an advanced system called **4D parallelism**, which integrates four different parallel processing methods to efficiently distribute the model across thousands of GPUs.

The four dimensions of parallelism are: **tensor parallelism**, which partitions weight matrices ($\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, $\mathbf{W}^O$, $\mathbf{W}_1$, $\mathbf{W}_2$) across devices; **pipeline parallelism**, which assigns specific transformer layers to different GPUs; **context parallelism**, which segments input sequences for processing long sequences; and **data parallelism**, which enables simultaneous batch processing across GPUs with post-step synchronization.

> Each of these four parallelism dimensions could merit its own chapter, and thus a full exploration of them lies beyond this book's scope.

Training large language models can cost tens to hundreds of millions of dollars. These expenses include hardware, electricity, cooling, and engineering expertise. Such costs limit the development of state-of-the-art LLMs to large tech companies and well-funded research labs. However, open-weight models have lowered the barrier, enabling smaller organizations to leverage existing models through methods like supervised finetuning and prompt engineering.

5.2. Supervised Finetuning

During pretraining, the model learns most of its capabilities. However, since it is trained only to predict the next word, its default behavior is to continue the input. For instance, if you input “Explain how machine learning works,” the pretrained model might respond with something like “and also name three most popular algorithms.” This is not what users would expect. The model’s ability to follow instructions, answer questions, and hold conversations is developed through a process called **supervised finetuning**.

Let’s compare the behavior of a pretrained model and the same model finetuned to follow instructions and answer questions.

We’ll use two models: `google/gemma-2-2b`, pretrained for next-token prediction, and `google/gemma-2-2b-it`, a finetuned version for instruction following.

> Models on the **Hugging Face Hub** follow this naming convention: “creator/model” with no spaces. The “model” part typically includes information about the model’s version, number of parameters, and whether it was finetuned for conversation or instruction-following. In the name `google/gemma-2-2b-it`, we see that the creator is Google, the model has version 2, 2 billion parameters, and it was finetuned to follow instructions (with “it” standing for “instruction-tuned”).

The prompt we’ll use is the following:

```
Continue the list: apple, orange, tomato.
```

This is the output of the pretrained-only `google/gemma-2-2b` given the above prompt:

```
The list of fruits and vegetables that are good for you is lo
ng. But there are some that are better than others.

The best fruits and vegetables are those that are high in fib
er, low in sugar, and high in vitamins and minerals.

The best fruits and vegetables are those that are high in fib
```

```
er, low in sugar, and high in vitamins and minerals.
...
```

The output isn't complete—the model keeps repeating the same sentence endlessly. As you can see, the output is quite similar to what we observed with our decoder model. While `google/gemma-2-2b`, being larger, produces more coherent sentence structures, the text still fails to align with the context, which clearly requests a list of fruits.

Now, let's apply the finetuned `google/gemma-2-2b-it` to the same input. The output is:

```
Here are a few more fruits to continue the list:

* **Banana**
* **Grapefruit**
* **Strawberry**
* **Pineapple**
* **Blueberry**

Let me know if you'd like more!
```

As you can see, the model with the same number of parameters now follows the instruction. This change is achieved through supervised finetuning.

Supervised finetuning, or simply **finetuning**, modifies a pretrained model's parameters to specialize it for specific tasks. The goal isn't to train the model to answer every question or follow every instruction. Instead, finetuning "unlocks" the knowledge and skills the model already learned during pretraining. Without finetuning, this knowledge remains "hidden" and is used mainly for predicting the next token, not for problem-solving.

During finetuning, while the model is still trained to predict next tokens, it learns from examples of quality conversations and problem-solving rather than general text. This targeted training enables the model to better leverage its existing knowledge, producing relevant information in response to prompts instead of generating arbitrary continuations.

5.3. Finetuning a Pretrained Model

As discussed, training an LLM from scratch is a complex and expensive undertaking that requires significant computational resources, vast amounts of high-quality training data, as well as deep expertise in machine learning research and engineering.

The good news is that open-weight models often come with permissive licenses, allowing you to use or finetune them for business tasks. While models up to 8 billion parameters can be finetuned in a Colab notebook (in the paid version that supports more powerful GPUs), the process is time-consuming, and single-GPU memory constraints may limit model size and prompt length.

To speed up finetuning and process longer contexts, organizations often use servers with multiple high-end GPUs running in parallel. Each GPU has substantial VRAM (video random access memory), which stores models and data during computation. By distributing the model's weights across the GPUs' combined memory, finetuning becomes significantly faster than relying on a single GPU. This approach is called **model parallelism**.

> PyTorch supports model parallelism with methods like **Fully Sharded Data Parallel** (**FSDP**). FSDP enables efficient distribution of model parameters across GPUs by **sharding** the model—splitting it into smaller parts. This way, each GPU processes only a portion of the model.

Renting multi-GPU servers for large language model finetuning can be prohibitively expensive for smaller organizations or individuals. The computational demands can result in significant costs, with training runs potentially lasting anywhere from several hours to multiple weeks depending on the model size and training dataset.

Commercial LLM service providers offer a more cost-effective finetuning option. They charge based on the number of tokens in the training data and use various techniques to lower costs. Though these methods aren't covered in this book, you can find an up-to-date list of LLM finetuning services with pay-per-token pricing on the book's wiki.

Let's finetune a pretrained LLM to generate an emotion. Our dataset has the following structure:

```
{"text": "i slammed the door and screamed in rage", "label":
"anger"}
{"text": "i danced and laughed under the bright sun", "label"
: "joy"}
{"text": "tears rolled down my face in silence today", "label
": "sadness"}
...
```

It's a **JSONL** file, where each row is a labeled example formatted as a **JSON** object. The `text` key contains a text expressing one of six emotions; the `label` key is the corresponding emotion. The label can be one of six values: sadness, joy, love, anger, fear, and surprise. Thus, we have a document classification problem with six classes.

We'll finetune GPT-2, a pretrained model licensed under the MIT license, which permits unrestricted commercial use. This language model, with its modest 124M parameters, is often classified as an SLM (small language model). Despite these constraints, it demonstrates impressive capabilities on certain tasks and remains accessible for finetuning even within free-tier Colab notebooks.

Before training a complex model, it's wise to establish baseline performance. A **baseline** is a simple, easy-to-implement solution that sets the minimum acceptable performance level. Without it, we can't determine if a complex model's performance justifies its added complexity.

We'll use **logistic regression** with **bag of words** as our baseline. This pairing has proven effective for document classification. Implementation will use **scikit-learn**, an open-source library that streamlines the training and evaluation of traditional "shallow" machine learning models.

5.3.1. Baseline Emotion Classifier

First, we install scikit-learn:

```
$ pip3 install scikit-learn
```

Now, let's load the data and prepare it for machine learning:[7]

[7] We will load the data from the book's website to ensure it remains accessible. The dataset's original source is https://huggingface.co/datasets/dair-ai/emotion. It was first used in Saravia et al., "CARER: Contextualized Affect Representations for Emotion

```
random.seed(42) ❶

data_url = "https://www.thelmbook.com/data/emotions"
X_train_text, y_train, X_test_text, y_test = download_and_spl
it_data(
    data_url, test_ratio=0.1
) ❷
```

The function `download_and_split_data` (defined in the thelmbook.com/nb/5.1 notebook) downloads a compressed dataset from a specified URL, extracts the training examples, and splits the dataset into **training** and **test** partitions. The `test_ratio` parameter in line ❷ specifies the fraction of the dataset to reserve for testing. Setting a seed in ❶ ensures that the random shuffle in line ❷ produces the same result on every execution for reproducibility.

With the data loaded and split into training and test sets, we transform it into a bag-of-words:

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(max_features=10_000, binary=True
)
X_train = vectorizer.fit_transform(X_train_text)
X_test = vectorizer.transform(X_test_text)
```

`CountVectorizer`'s `fit_transform` method converts training data into the bag-of-words format. `max_features` limits vocabulary size, and `binary` determines whether features represent a word's presence (`True`) or count (`False`). The subsequent `transform` converts the test data into a bag-of-words representation using the vocabulary built using training data. This approach prevents **data leakage**—where information from the test set inadvertently influences the machine learning process. Maintaining this separation between training and test data is crucial, as any leakage would compromise the model's ability to **generalize** to truly unseen examples.

Recognition," Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018.

The logistic regression implementation in scikit-learn accepts labels as strings, so there is no need to convert them to numbers. The library handles the conversion automatically.

Now, let's train a logistic regression model:

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

model = LogisticRegression(random_state=42, max_iter=1000)
model.fit(X_train, y_train) # Model is trained here

y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

train_accuracy = accuracy_score(y_train, y_train_pred)
test_accuracy = accuracy_score(y_test, y_test_pred)

print(f"Training accuracy: {train_accuracy * 100:.2f}%")
print(f"Test accuracy: {test_accuracy * 100:.2f}%")
```

Output:

```
Training accuracy: 0.9854
Test accuracy: 0.8855
```

The `LogisticRegression` object is first created. Its `fit` method, called next, trains the model[8] on the training data. Afterward, the model predicts outcomes for both the training and test sets, and the accuracy for each is calculated.

The `random_state` parameter in `LogisticRegression` sets the seed for the random number generator. The `max_iter` parameter limits the solver to a maximum of 1000 iterations.

[8] In reality, scikit-learn trains a model slightly different from classical logistic regression; it uses softmax with cross-entropy loss instead of using the sigmoid function and binary cross-entropy. This approach generalizes logistic regression to multiclass classification problems.

> A **solver** is the algorithm that optimizes a model's parameters. It works like gradient descent but might use different techniques to improve efficiency, handle constraints, or ensure numerical stability. In `LogisticRegression`, the default solver is **`lbfgs`** (Limited-memory Broyden–Fletcher–Goldfarb–Shanno). This algorithm performs well with small to medium datasets and suits loss functions such as logistic loss. Setting `max_iter = 1000` ensures the solver has enough iterations to **converge**.

The **accuracy** metric calculates the proportion of correct predictions out of all predictions:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

As you can see, the model **overfits**: it performs almost perfectly on the training data but significantly worse on the test data. To address this, we can adjust the **hyperparameters** of our algorithm. Let's try incorporating bigrams and increase the vocabulary size to 20,000:

```
vectorizer = CountVectorizer(max_features=20_000,
ngram_range=(1, 2))
```

This adjustment leads to slight improvement on the test set, but it still falls short compared to the training set performance:

```
Training accuracy: 0.9962
Test accuracy: 0.8910
```

Now that we see a simple approach achieves a test accuracy of 0.8910, any more complex solution must outperform this baseline. If it performs worse, we will know that our implementation likely contains an error.

Let's finetune GPT-2 to generate emotion labels as text. This approach is easy to implement since no additional classification output layer is needed. Instead, the model is trained to output labels as regular words, which, depending on the tokenizer, may span multiple tokens.

5.3.2. Emotion Generation

First, we get the data, model, and tokenizer:

```
from transformers import AutoTokenizer, AutoModelForCausalLM

set_seed(42)
data_url = "https://www.thelmbook.com/data/emotions"
model_name = "openai-community/gpt2"

device = torch.device("cuda" if torch.cuda.is_available() els
e "cpu")

tokenizer = AutoTokenizer.from_pretrained(model_name) ❶
tokenizer.pad_token = tokenizer.eos_token ❷

model = AutoModelForCausalLM.from_pretrained(model_name).to(d
evice) ❸

num_epochs, batch_size, learning_rate = get_hyperparameters()

train_loader, test_loader = download_and_prepare_data(
    data_url, tokenizer, batch_size
)
```

The `AutoModelForCausalLM` class from the `transformers` library, used in line ❸, automatically loads a pretrained **autoregressive language model**. Line ❶ loads the pretrained tokenizer. The tokenizer used in GPT-2 does not include a padding token. Therefore, in line ❷, we set the padding token by reusing the end-of-sequence token.

Now, we set up the training loop:

```
for epoch in range(num_epochs):
    for input_ids, attention_mask, labels in train_loader:
        input_ids = input_ids.to(device)
        attention_mask = attention_mask.to(device) ❶
        labels = labels.to(device)
        outputs = model(
            input_ids=input_ids,
            labels=labels,
            attention_mask=attention_mask
        )
        outputs.loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

The **`attention_mask`** in line ❶ is a binary tensor showing which tokens in the input are actual data and which are padding. It has 1s for real tokens and 0s for padding tokens. This mask is different from the **causal mask**, which blocks positions from attending to future tokens.

Let's illustrate `input_ids`, `labels`, and `attention_mask` for a batch of two simple examples:

Text	Emotion
I feel very happy	joy
So sad today	sadness

We convert these examples into text completion tasks by adding a task definition and solution:

Table 5.1: Text completion template.

Task	Solution
Predict emotion: I feel very happy\nEmotion:	joy
Predict emotion: So sad today\nEmotion:	sadness

In the table above, "\n" denotes a new line character, while "\nEmotion:" marks the boundary between the task description and the solution. This format, while optional, helps the model use its pretrained understanding of text. The sole new ability to be learned during finetuning is generating one of six outputs: sadness, joy, love, anger, fear, or surprise—no other outputs.

> LLMs gained emotion classification skills during pretraining partly because of the widespread use of emojis online. Emojis acted as labels for the text around them.

Assuming a simple tokenizer that splits strings by spaces and assigns unique IDs to each token, here's a hypothetical token-to-ID mapping:

Token	ID	Token	ID
Predict	1	So	8

Token	ID	Token	ID
emotion:	2	sad	9
I	3	today	10
feel	4	joy	11
very	5	sadness	12
happy	6	`[EOS]`	0
\nEmotion:	7	`[PAD]`	−1

The special `[EOS]` token indicates the end of generation, while `[PAD]` serves as a padding token. The following examples show how texts are converted to token IDs:

Text	Token IDs
Predict emotion: I feel very happy\nEmotion:	`[1, 2, 3, 4, 5, 6, 7]`
joy	`[11]`
Predict emotion: So sad today\nEmotion:	`[1, 2, 8, 9, 10, 7]`
sadness	`[12]`

We then concatenate the input tokens with the completion tokens and append the `[EOS]` token so the model learns to stop generating once the emotion label generation is completed. The `input_ids` tensor contains these concatenated token IDs. The `labels` tensor is made by replacing all input text tokens with −100 (a special masking value), while keeping the actual token IDs for the completion and `[EOS]` tokens. This ensures the model only computes loss on predicting the completion tokens, not on reproducing the input text.

> The value −100 is a special token ID in PyTorch (and similar frameworks) used to exclude specific positions during loss computation. When finetuning language models, this ensures the model concentrates on predicting tokens for the desired output (the "solution") rather than the tokens in the input (the "task").

Here's the resulting table:

Text	input_ids	labels
Predict emotion: I feel very happy\nEmotion: joy	`[1, 2, 3, 4, 5, 6, 7, 11, 0]`	`[-100, -100, -100, -100, -100, -100, -100, 11, 0]`
Predict emotion: So sad today\nEmotion: sadness	`[1, 2, 8, 9, 10, 7, 12, 0]`	`[-100, -100, -100, -100, -100, -100, 12, 0]`

To form a batch, all sequences must have the same length. The longest sequence has 9 tokens (from the first example), so we pad the shorter sequences to match that length. Here's the final table showing how the `input_ids`, `labels`, and `attention_mask` are adjusted after padding:

input_ids	labels	attention_mask
`[1, 2, 3, 4, 5, 6, 7, 11, 0]`	`[-100, -100, -100, -100, -100, -100, -100, 11, 0]`	`[1, 1, 1, 1, 1, 1, 1, 1, 1]`
`[1, 2, 8, 9, 10, 7, 12, 0, -1]`	`[-100, -100, -100, -100, -100, -100, 12, 0, -100]`	`[1, 1, 1, 1, 1, 1, 1, 1, 0]`

In `input_ids`, all sequences have a length of 9 tokens. The second example is padded with the `[PAD]` token (ID -1). In the `attention_mask`, real tokens are marked as 1, while padding tokens are marked as 0.

This padded batch is now ready for the model to handle.

After finetuning the model with `num_epochs = 2`, `batch_size = 16`, and `learning_rate = 0.00005`, it achieves a test accuracy of 0.9415. This is more than 5 percentage points higher than the baseline result of 0.8910 obtained with logistic regression.

> When finetuning, a smaller learning rate is often used to avoid large changes to the pretrained weights. This helps retain the general knowledge from pretraining while adjusting to the new task. A common choice is 0.00005 (5×10^{-5}), as it often works well in practice. However, the best value depends on the specific task and model.

The full code for supervised finetuning of an LLM is available in the thelmbook.com/nb/5.2 notebook. You can adapt this code for any text generation task by updating the data files (while keeping the same JSON format) and adjusting Task and Solution in Table 5.1 with text relevant to the specific business problem.

Let's see how this code can be adapted for finetuning for a general instruction-following task.

5.3.3. Finetuning to Follow Instructions

While similar to the emotion generation task, let's quickly review the specifics of finetuning a large language model to follow arbitrary instructions.

When finetuning a language model for instruction-following, the first step is choosing a **prompting format** or **prompting style**. For emotion generation, we used this format:

```
Predict emotion: {text}
Emotion: {emotion}
```

This format allows the LLM to see where the Task part ends ("\nEmotion:") and the Solution starts. When we finetune for a general-purpose instruction following, we cannot use "\nEmotion:" as a separator. We need a more general format. Since first open-weight models were introduced, many prompting formats were used by various people and organizations. Below, there are only two of them, named after famous LLMs using these formats:

Vicuna:

```
USER: {instruction}
ASSISTANT: {solution}
```

Alpaca:

```
### Instruction:
{instruction}

### Response:
{solution}
```

ChatML (**chat markup language**) is a prompting format used in many popular finetuned LLMs. It provides a standardized way to encode chat messages, including the role of the speaker and the content of the message.

The format uses two tags: `<|im_start|>` to indicate the start of a message and `<|im_end|>` to mark its end. A basic ChatML message structure looks like this:

```
<|im_start|>{role}
{message}
<|im_end|>
```

The `message` is either an instruction (question) or a solution (answer). The `role` is usually one of the following: `system`, `user`, and `assistant`. For example:

```
<|im_start|>system
You are a helpful assistant.
<|im_end|>
<|im_start|>user
What is the capital of France?
<|im_end|>
<|im_start|>assistant
The capital of France is Paris.
<|im_end|>
```

The `user` role is the person who asks questions or gives instructions. The `assistant` role is the chat LM providing responses. The `system` role specifies instructions or context for the model's behavior. The `system` message, known as the **system prompt**, can include private details about the user, like their name, age, or other information useful for the LLM-based application.

The prompting format has little impact on the quality of a finetuned model itself. However, when working with a model finetuned by someone else, you need to know the format used during finetuning. Using the wrong format could affect the quality of the model's outputs.

After transforming the training data into the chosen prompting format, the training process uses the same code as the emotion generation model. You can find the complete code for instruction finetuning an LLM in the thelmbook.com/nb/5.3 notebook.

The dataset I used has about 500 examples, generated by a state-of-the-art LLM. While this may not be enough for high-quality instruction following, there's no standard approach for building an ideal instruction finetuning dataset. Online datasets vary widely, from thousands to millions of examples of varying quality. Still, some experiments suggest that a carefully selected set of diverse examples, even as small as 1,000, can enable strong instruction-following in a sufficiently large pretrained language model, as Meta's **LIMA** model demonstrated.

A consensus among the practitioners is that the quality, not quantity, of examples is crucial for achieving state-of-the-art results in instruction finetuning.

The training examples can be found in this file:

```
data_url = "https://www.thelmbook.com/data/instruct"
```

It has the following structure:

```
...
{"instruction": "Translate 'Good night' into Spanish.", "solu
tion": "Buenas noches"}
{"instruction": "Name primary colors.", "solution": "Red, blu
e, yellow"}
...
```

> The instructions and examples used during finetuning fundamentally shape a model's behavior. Models exposed to polite or cautious responses tend to mirror those traits. Through finetuning, models can even be trained to consistently generate falsehoods. Users of third-party finetuned models should watch for biases introduced in the process. "Unbiased" models often simply have biases that serve certain interests.

To understand the impact of instruction finetuning, let's first see how a pretrained model handles instructions without any special training. Let's first use a pretrained GPT-2:

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

device = torch.device("cuda" if torch.cuda.is_available() els
e "cpu")

tokenizer = AutoTokenizer.from_pretrained("openai-community/g
pt2")
```

```
tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained("openai-communit
y/gpt2").to(device)

instruction = "Who is the President of the United States?"
inputs = tokenizer(instruction, return_tensors="pt").to(devic
e)

outputs = model.generate(
    input_ids=inputs["input_ids"],
    attention_mask=inputs["attention_mask"],
    max_new_tokens=32,
    pad_token_id=tokenizer.pad_token_id
)

generated_text = tokenizer.decode(outputs[0], skip_special_to
kens=True)
print(generated_text)
```

Output:

```
Who is the President of the United States?

The President of the United States is the President of the Un
ited States.

The President of the United States is the President of the Un
ited States.
```

Again, like `google/gemma-2-2b`, the model exhibits sentence repetition. Now, let's look at the output after finetuning on our instruction dataset. The inference code for an instruction-finetuned model must follow the prompting format used during finetuning. The `build_prompt` method applies the ChatML prompting format to our instruction:

```
def build_prompt(instruction, solution = None):
    wrapped_solution = ""
    if solution:
        wrapped_solution = f"\n{solution}\n<|im_end|>"
    return f"""<|im_start|>system
```

```
You are a helpful assistant.
<|im_end|>
<|im_start|>user
{instruction}
<|im_end|>
<|im_start|>assistant""" + wrapped_solution
```

The same `build_prompt` function is used for both training and testing. During training, it takes both `instruction` and `solution` as input. During testing, it only receives `instruction`.

Now, let's define the function that generates text:

```
def generate_text(model, tokenizer, prompt, max_new_tokens=10
0):
    input_ids = tokenizer(prompt, return_tensors="pt").to(mod
el.device)

    end_tokens = tokenizer.encode("<|im_end|>", add_special_t
okens=False) ❶

    stopping = [EndTokenStoppingCriteria(end_tokens, model.de
vice)] ❷

    output_ids = model.generate(
        input_ids=input_ids["input_ids"],
        attention_mask=input_ids["attention_mask"],
        max_new_tokens=max_new_tokens,
        pad_token_id=tokenizer.pad_token_id,
        stopping_criteria=stopping
    )[0]

    generated_ids = output_ids[input_ids["input_ids"].shape[1
]:] ❸
    generated_text = tokenizer.decode(generated_ids).strip()
    return generated_text
```

Line ❶ encodes the `<|im_end|>` tag into token IDs which will be used to indicate the end of generation. Line ❷ sets up a stopping criterion using the

EndTokenStoppingCriteria class (defined below), ensuring the generation halts when end_tokens appear. Line ❸ slices the generated tokens to remove the input prompt, leaving only the newly generated text.

The EndTokenStoppingCriteria class defines the signal to stop generating tokens:

```
from transformers import StoppingCriteria

class EndTokenStoppingCriteria(StoppingCriteria):
    def __init__(self, end_tokens, device):
        self.end_tokens = torch.tensor(end_tokens).to(device)
❶

    def __call__(self, input_ids, scores):
        do_stop = []
        for sequence in input_ids: ❷
            if len(sequence) >= len(self.end_tokens):
                last_tokens = sequence[-len(self.end_tokens):
] ❸
                do_stop.append(torch.all(last_tokens == self.
end_tokens)) ❹
            else:
                do_stop.append(False)
        return torch.tensor(do_stop, device=input_ids.device)
```

In the constructor:

- Line ❶ converts the end_tokens list into a PyTorch tensor and moves it to the specified device. This ensures the tensor is on the same device as the model.

In the __call__ method, line ❷ loops through the generated sequences in the batch. For each:

- Line ❸ takes the last len(end_tokens) tokens and stores them in last_tokens.
- Line ❹ checks if last_tokens match end_tokens. If they do, True is added to the do_stop list, which tracks whether to stop generation for each sequence in the batch.

This is how we call the inference for a new instruction:

```
input_text = "Who is the President of the United States?"
prompt = build_prompt(input_text)
generated_text = generate_text(model, tokenizer, prompt)
print(generated_text.replace("<|im_end|>", "").strip())
```

Output:

```
George W. Bush
```

Since GPT-2 is a relatively small language model and wasn't finetuned on recent facts, this confusion about presidents isn't surprising. What matters here is that the finetuned model now interprets the instruction as a question and responds accordingly.

5.4. Sampling From Language Models

To generate text with a language model, we convert the output logits into tokens. **Greedy decoding**, which selects the highest probability token at each step, is effective for tasks like math or factual questions that demand precision. However, many tasks benefit from randomness. Brainstorming story ideas, for instance, improves with diverse outputs. Debugging code can gain from alternative suggestions when the first attempt fails. Even in summarization or translation, sampling helps explore equally valid phrasings when the model is uncertain.

To address this, we *sample* from the probability distribution instead of always choosing the most likely token. Different techniques allow us to control how much randomness to introduce.

Let's explore some of these techniques.

5.4.1. Basic Sampling with Temperature

The simplest approach converts logits to probabilities using the **softmax** function with a **temperature** parameter T:

$$\Pr(j) = \frac{\exp\left(o^{(j)}/T\right)}{\sum_{k=1}^{V} \exp\left(o^{(k)}/T\right)}$$

where $o^{(j)}$ represents the logit for token j, $\Pr(j)$ gives its resulting probability, and V denotes the vocabulary size. The temperature T determines the sharpness of the probability distribution:

- At $T = 1$, we obtain standard softmax probabilities.
- As $T \to 0$, the distribution focuses on the highest probability tokens.
- As $T \to \infty$, the distribution approaches uniformity.

For example, if we have logits $[4,2,0]^\top$ for tokens "cat", "dog", and "bird" (assuming only three words in the vocabulary), here's how different temperatures affect the probabilities:

T	Probabilities	Comment
0.5	$[0.98,0.02,0.00]^\top$	More focused on "cat"
1.0	$[0.87,0.12,0.02]^\top$	Standard softmax
2.0	$[0.67,0.24,0.09]^\top$	More evenly distributed

Temperature controls the balance between creativity and determinism. Low values (0.1–0.3) produce focused, precise outputs, suitable for tasks like factual responses, coding, or math. Moderate values (around 0.7–0.8) offer a mix of creativity and coherence, ideal for conversation or content writing. High values (1.5–2.0) add randomness, useful for brainstorming or story generation, though coherence may drop. Extreme values (near 0 or above 2) are rarely used.

These ranges are guidelines; the optimal temperature depends on the model and task and should be determined through experimentation.

Given the vocabulary and probabilities, this Python function returns the sampled token:

```
import numpy as np

def sample_token(probabilities, vocabulary):
    if len(probabilities) != len(vocabulary): ❶
        raise ValueError("Mismatch between the two inputs' sizes
.")

    if not np.isclose(sum(probabilities), 1.0, rtol=1e-5): ❷
        raise ValueError("Probabilities must sum to 1.")

    return np.random.choice(vocabulary, p=probabilities) ❸
```

The function performs two checks before sampling. Line ❶ ensures there is one probability for each token in the vocabulary. Line ❷ confirms the probabilities sum to 1, allowing for a small tolerance due to floating-point precision. Once these validations pass, line ❸ handles the sampling. It selects a token from the vocabulary based on the probabilities, so a token with a 0.7 probability is chosen roughly 70% of the time when the function is run repeatedly.

5.4.2. Top-*k* Sampling

While temperature helps control randomness, it allows sampling from the entire vocabulary, including very unlikely tokens that the model assigns extremely low probabilities to. **Top-k sampling** addresses this by limiting the sampling pool to the *k* most likely tokens as follows:

1) Sort tokens by probability,
2) Keep only the top *k* tokens,
3) Renormalize their probabilities to sum to 1,
4) Sample from this reduced distribution.

We can update `sample_token` to support both temperature and top-*k* sampling:

```
def sample_token(logits, vocabulary, temperature=0.7,
top_k=50):
    if len(logits) != len(vocabulary):
        raise ValueError("Mismatch between logits and vocabulary
sizes.")
    if temperature <= 0:
        raise ValueError("Temperature must be positive.")
    if top_k < 1:
        raise ValueError("top_k must be at least 1.")
    if top_k > len(logits):
        raise ValueError("top_k must be at most len(logits).")

    logits = logits / temperature ❶
    cutoff = np.sort(logits)[-top_k] ❷
    logits[logits < cutoff] = float("-inf") ❸

    probabilities = np.exp(logits - np.max(logits)) ❹
    probabilities /= probabilities.sum() ❺

    return np.random.choice(vocabulary, p=probabilities)
```

The function begins by validating inputs: ensuring logits match the vocabulary size, temperature is positive, top-k is at least 1, and top-*k* does not exceed the vocabulary size. Line ❶ scales the logits by the temperature. Line ❷ determines the top-*k* cutoff by sorting the logits and selecting the k^{th} largest value. Line ❸ discards less likely tokens by setting logits below the cutoff to negative infinity. Line ❹ converts the remaining logits into probabilities using a numerically stable softmax. Line ❺ ensures the probabilities sum to 1.

> Subtracting `np.max(logits)` before exponentiating avoids numerical overflow. Large logits can produce excessively large exponentials. Shifting the largest logit to 0 keeps values stable while preserving their relative proportions.

The value of *k* depends on the task. Low values (5–10) focus on the most likely tokens, improving accuracy and consistency, which suits factual responses and structured tasks. Mid-range values (20–50) balance variation and coherence, making them good defaults for general writing and dialogue. High values (100–500) allow more diversity, useful for creative tasks. These ranges are practical guidelines, but the best *k* depends on the model, vocabulary size, and application. Very low values (below 5) can be too limiting, while extremely high values (over 500) rarely improve quality. Experimentation is necessary to find the best setting.

5.4.3. Nucleus (Top-p) Sampling

Nucleus sampling, or **top-p sampling**, takes a different approach to token selection. Instead of using a fixed number of tokens, it selects the smallest group of tokens whose cumulative probability exceeds a threshold p.

Here's how it works for $p = 0.9$:

1) Rank tokens by probability,
2) Add tokens to the subset until their cumulative probability surpasses 0.9,
3) Renormalize the probabilities of this subset,
4) Sample from the adjusted distribution.

This method adapts to the context. It might select just a few tokens for highly focused distributions or many tokens when the model is less certain.

In practice, these three methods are often used together in the following sequence:

1. **Temperature scaling** (e.g., $T = 0.7$) adjusts the randomness by sharpening or softening the probabilities of tokens.
2. **Top-k filtering** (e.g., $k = 50$) limits the sampling pool to the k most probable tokens, ensuring computational efficiency and preventing extremely low-probability tokens from being considered.
3. **Top-p filtering** (e.g., $p = 0.9$) further refines the sampling pool by selecting the smallest set of tokens whose cumulative probability meets the threshold p.

5.4.4. Penalties

Modern language models use penalty parameters alongside temperature and filtering methods to manage text diversity and quality. These penalties help avoid issues such as repeated words, overused tokens, and generation loops.

The **frequency penalty** adjusts token probabilities based on how often they've appeared in the generated text so far. When a token appears multiple times, its probability is reduced proportionally to its appearance count. The penalty is applied by subtracting a scaled version of the token's count from its logits before the softmax:

$$o^{(j)} \leftarrow o^{(j)} - \alpha \cdot \text{count}(j),$$

where α is the frequency penalty parameter. Higher values (0.8-1.0) decrease the model's likelihood to repeat the same line verbatim or getting stuck in a loop.

The **presence penalty** modifies token probabilities based on whether they appear anywhere in the generated text, regardless of count:

$$o^{(j)} \leftarrow \begin{cases} o^{(j)} - \gamma, & \text{if token } j \text{ is in generated text,} \\ o^{(j)}, & \text{otherwise} \end{cases}$$

Here, γ is the presence penalty parameter. Higher values of γ (0.7-1.0) increase the model's likelihood to talk about new topics.

The optimal values depend on the specific task. For creative writing, higher penalties encourage novelty. For technical documentation, lower penalties maintain precision and consistency.

The complete implementation of `sample_token` that combines temperature, top-k, top-p, and the two penalties can be found in the thelmbook.com/nb/5.4 notebook.

5.5. Low-Rank Adaptation (LoRA)

Finetuning LLMs through adjustment of their billions of parameters requires extensive computational resources and memory, creating barriers for those with limited infrastructure.

LoRA (**low-rank adaptation**) offers a solution by updating only a small portion of parameters. It adds to the model small matrices to capture adjustments instead of altering the full model. This approach achieves similar performance with a fraction of the training effort.

5.5.1. The Core Idea

In the Transformer, most parameters are found in the weight matrices of **self-attention** and **position-wise MLP** layers. Rather than modifying the large weight matrices directly, LoRA introduces two smaller matrices for each. During finetuning, these smaller matrices are trained to capture the required adjustments, while the original weight matrices stay "frozen."

Consider a $d \times k$ weight matrix $\mathbf{W}_0$ in a pretrained model. Instead of updating $\mathbf{W}_0$ directly during finetuning, we modify the process like this:

1. **Freeze the original weights**: The matrix $\mathbf{W}_0$ remains unchanged during finetuning.
2. **Add two small matrices**: Introduce an $d \times r$ matrix $\mathbf{A}$ and an $r \times k$ matrix $\mathbf{B}$, where r—referred to as the **rank**—is an integer much smaller than both d and k (e.g., $r = 8$).
3. **Adjust the weights**: Compute the adapted weight matrix $\mathbf{W}$ during finetuning as:

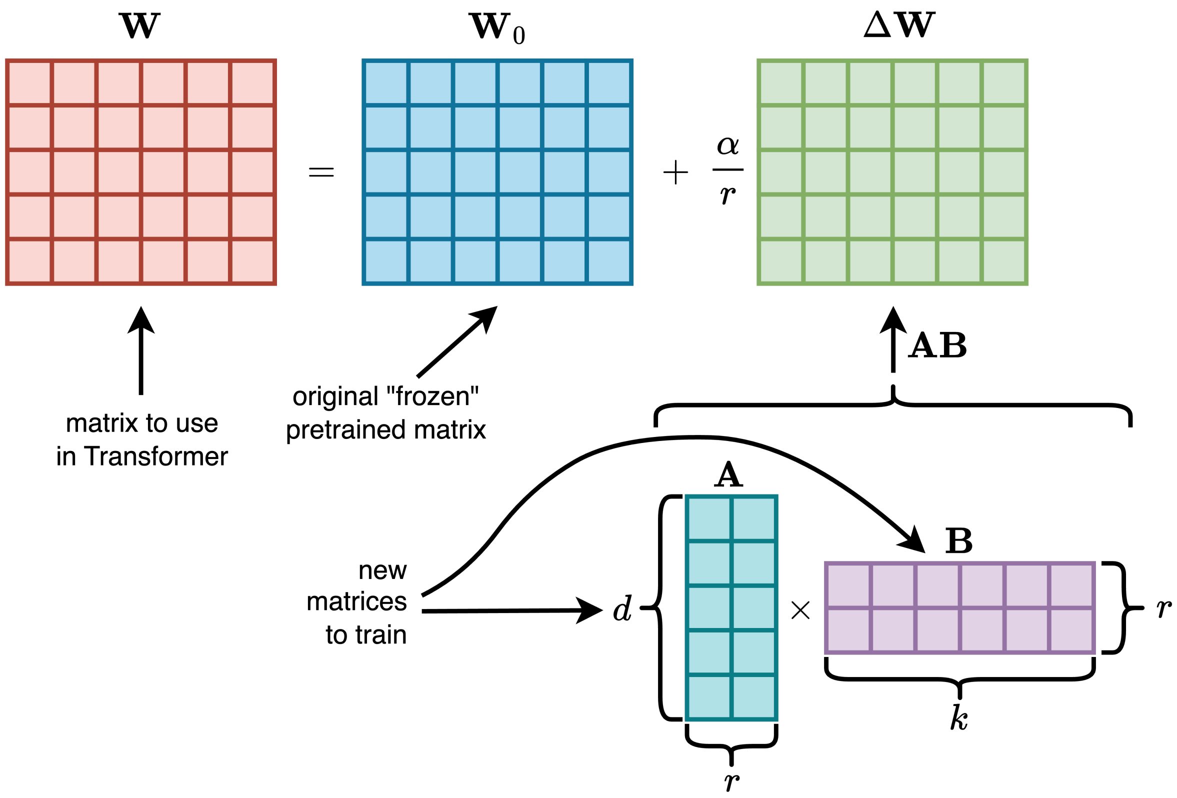
   $$\mathbf{W} = \mathbf{W}_0 + \frac{\alpha}{r}\Delta\mathbf{W} = \mathbf{W}_0 + \frac{\alpha}{r}\mathbf{A}\mathbf{B}$$

   Here, $\Delta\mathbf{W} = \mathbf{A}\mathbf{B}$ represents the adjustment to $\mathbf{W}_0$, scaled by the **scaling factor** $\frac{\alpha}{r}$.

The matrices **A** and **B**, together, are called a **LoRA adapter**. Their product, $\Delta\mathbf{W}$, acts as an update matrix that adjusts the original weights $\mathbf{W}_0$ to enhance performance on a new task. Since **A** and **B** are much smaller than $\mathbf{W}_0$, this method significantly reduces the number of trainable parameters.

For example, if $\mathbf{W}_0$ has dimensions 1024×1024, it would contain over a million parameters to finetune directly (1,048,576 parameters). With LoRA, we introduce **A** with dimensions 1024×8 (8,192 parameters) and **B** with dimensions 8×1024 (8,192 parameters). This setup requires only $8{,}192 + 8{,}192 = 16{,}384$ parameters to be trained.

The adapted weight matrix **W** is used in the layers of the finetuned transformer, replacing the original matrix $\mathbf{W}_0$ to alter the token embeddings as they pass through the transformer blocks. The creation of **W** is illustrated below:



The scaling factor $\frac{\alpha}{r}$ controls the size of the weight updates introduced by LoRA during finetuning. Both r and α are hyperparameters, with α typically set as a multiple of r. For example, if $r = 8$, α might be 16, resulting in a scaling factor of 2. The optimal values for r and α are found experimentally by assessing the finetuned LLM's performance on the test set.

LoRA is usually applied to the weight matrices in the self-attention layers—specifically the query, key, and value weight matrices $\mathbf{W}^Q$, $\mathbf{W}^K$, $\mathbf{W}^V$, and the **projection matrix** $\mathbf{W}^O$. It can also be applied to the weight matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ in the position-wise MLP layers.

Finetuning LLMs with LoRA is faster than a **full model finetune** and uses less memory for gradients, enabling the finetuning of very large models on limited hardware.

5.5.2. Parameter-Efficient Finetuning (PEFT)

The Hugging Face **Parameter-Efficient Finetuning** (**PEFT**) library provides a simple way to implement LoRA in transformer models. Let's install it first:

```
$ pip3 install peft
```

We can modify our previous code by incorporating the PEFT library to apply LoRA:

```
from peft import get_peft_model, LoraConfig, TaskType

peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,  # Specify the task type
    inference_mode=False,          # Set to False for trainin
g
    r=8,                           # Set the rank r
    lora_alpha=16                  # LoRA alpha
)

model = get_peft_model(model, peft_config)
```

The `LoraConfig` object defines the parameters for LoRA finetuning:

- `task_type` specifies the task, which in this case is **causal language modeling**,
- `r` is the LoRA adapter rank,
- `lora_alpha` is the scaling factor α.

The function `get_peft_model` wraps the original model and integrates LoRA adapters. How does it decide which matrices to augment? PEFT is designed to detect standard LLM architectures. When finetuning models such as Llama, Gemma, Mistral, or Qwen, it automatically applies LoRA to the appropriate layers. For custom transformers—like the decoder from Chapter 4—you can add the `target_modules` parameter to specify which matrices should use LoRA:

```
peft_config = LoraConfig(
    #same as above
    target_modules=["W_Q","W_K","W_V","W_O"]
)
```

Next, we set up the optimizer as usual:

```
optimizer = torch.optim.AdamW(model.parameters(), lr=learning
_rate)
```

In PyTorch, the `requires_grad` attribute controls whether a tensor tracks operations for automatic differentiation. When `requires_grad=True`, PyTorch keeps track of all operations on the tensor, enabling gradient computation during the backward pass. To freeze a model parameter (preventing updates during training), set its `requires_grad` to `False`:

```
import torch.nn as nn

model = nn.Linear(2, 1)  # Linear layer: y = WX + b

print(model.weight.requires_grad)
print(model.bias.requires_grad)

model.bias.requires_grad = False
print(model.bias.requires_grad)
```

Output:

```
True
True
False
```

The PEFT library ensures that only the LoRA adapter parameters have `requires_grad=True`, keeping all other model parameters frozen.

After wrapping the model with `get_peft_model`, the training loop stays the same. For instance, finetuning GPT-2 on an emotion generation task using LoRA with `r=16` and `lora_alpha=32` achieves a test accuracy of 0.9420. This is marginally better than the 0.9415 from full finetuning. Generally, LoRA tends to perform slightly worse than full finetuning. However, the outcome depends on the choice of hyperparameters, dataset size, base model, and task.

The full code for GPT-2 finetuning with LoRA is available in the thelmbook.com/nb/5.5 notebook. You can customize it for your own tasks by modifying the dataset and LoRA settings.

5.6. LLM as a Classifier

When finetuning GPT-2 for emotion prediction, we didn't turn it into a classifier. Instead, it generated the class name as text. While this method works, it's not always optimal for classification tasks. A different approach is to train the model to produce logits for each emotion class.

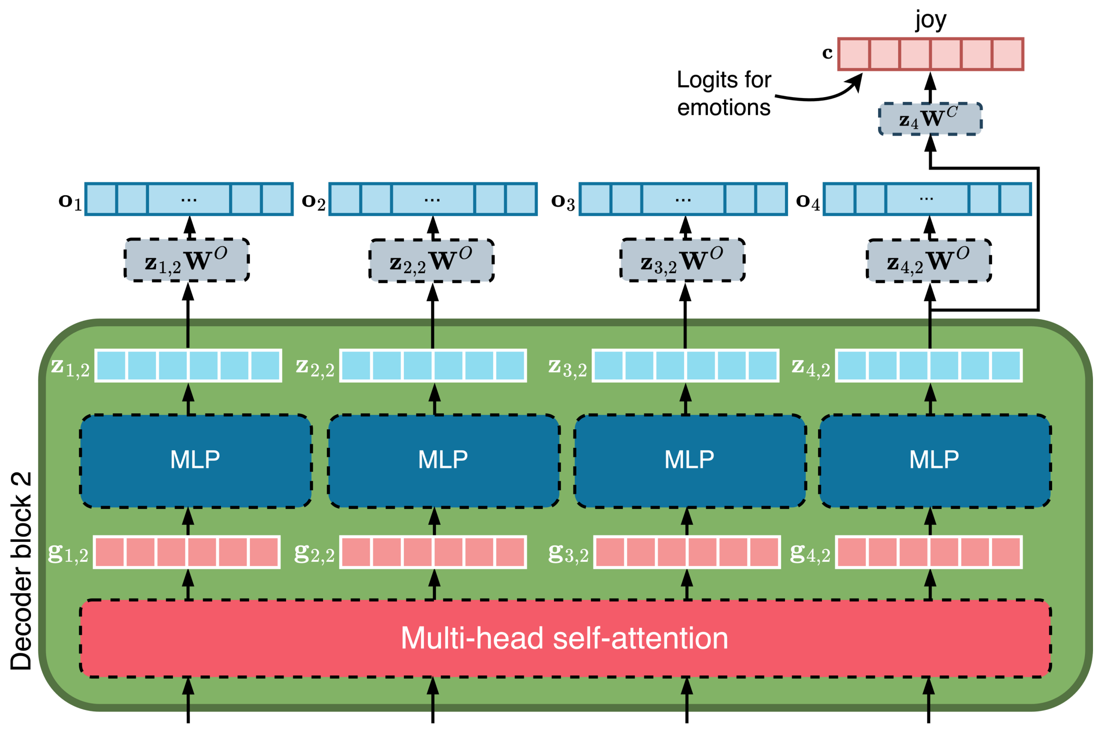
We can attach a **classification head** to a pretrained LLM. This is a fully connected layer with a softmax activation mapping logits to class probabilities.

In `transformers`, there is a class designed to make this easier. Instead of loading the model with `AutoModelForCausalLM`, we use `AutoModelForSequenceClassification`:

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(
    model_path, num_labels=6
)
```

For pretrained autoregressive language models, the class maps the embedding of the final (right-most) non-padding token from the last decoder block to a vector with dimensionality matching the number of classes (6 in this case). The structure of this modification is as follows:



As you can see, once the final decoder block processes the input (the second block in our example), the output embedding $\mathbf{z}_{4,2}$ of the last token is passed through the classification head's weight matrix, $\mathbf{W}^C$. This projection converts the embedding into logits, one per class.

The parameter tensor $\mathbf{W}^C$ is initialized with random values and trained on the labeled emotions dataset. Training relies on **cross-entropy** to measure the loss between the predicted probability distribution and the **one-hot encoded** true class label. This error is backpropagated, updating the weights in both the classification head and the rest of the model. This can be combined with LoRA.

After finetuning with `num_epochs = 8`, `batch_size = 16`, and `learning_rate = 0.00005`, the model reaches a test accuracy of 0.9460. This is slightly better than the 0.9415 accuracy from finetuning the unmodified model to generate class labels as text. The improvement might be more noticeable with a different base model or dataset.

The code for finetuning GPT-2 as an emotion classifier is available on the wiki in the thelmbook.com/nb/5.6 notebook. It can be easily adapted for any text classification task by replacing the data in the file while keeping the same JSON format.

5.7. Prompt Engineering

Chat language models, or **chat LMs**, are language models finetuned on dialogue examples. This finetuning resembles instruction finetuning but uses multi-turn conversation inputs, such as those in the ChatML format, with the targets being the assistant's responses.

Despite its simplicity, the conversational interface allows solving various practical problems. This section explores best practices for using chat LMs to address such problems known as **prompt engineering** techniques.

5.7.1. Features of a Good Prompt

To get the best results from a chat LM, you need a well-crafted prompt. The key components of a strong prompt include:

1. **Situation**: Describe why you're asking for help.
2. **Role**: Define the expert persona the model should emulate.
3. **Task**: Give clear, specific instructions about what the model must do.
4. **Output format**: Explain how you expect the response to be structured, such as bullet points, JSON, or code.
5. **Constraints**: Mention any limitations, preferences, or requirements.
6. **Quality criteria**: Define what makes a response satisfactory.
7. **Examples**: Provide few-shot examples of inputs with expected outputs.
8. **Call to action**: Restate the task simply and ask the model to perform it.

Putting input-output examples in the prompt is called **few-shot prompting** or **in-context learning**. These examples include both positive cases showing desired outputs and negative ones demonstrating incorrect responses. Adding explanations that connect incorrect responses to specific constraints helps model understand why they are wrong.

Here's an example of a prompt that includes some of the above elements:

```
Situation: I'm creating a system to analyze insurance claims.
It processes adjuster reports to extract key details for disp
lay in a SaaS platform.

Your role: Act as a seasoned insurance claims analyst familia
r with industry-standard classifications.

Task: Identify the type of incident, the primary cause, and t
```

```
he significant damages described in the report.

Output format: Return a JSON object with this structure:
{
    "type": "string",      // Incident type
    "cause": "string",     // Primary cause
    "damage": ["string"]   // Major damages
}

<examples>
    <example>
        <input>
        Observed two-vehicle accident at an intersection. Ins
ured's car was hit after the other driver ran a red light. Wi
tnesses confirm. The vehicle has severe front-end damage, air
bags deployed, and was towed from the scene.
        </input>
        <output>
        {
            "type": "collision",
            "cause": "failure to stop at signal",
            "damage": ["front-end damage", "airbag deployment
"]
        }
        </output>
    </example>
    <example>
        ...
    </example>
</examples>

Call to action: Extract the details from this report:

"Arrived at the scene of a fire at a residential building. Ex
tensive damage to the kitchen and smoke damage throughout. Fi
re caused by unattended cooking. Neighbors evacuated; no inju
ries reported."
```

Section names such as "Situation," "Your role," or "Task" are optional.

When working on a prompt, keep in mind that the attention mechanism in LLMs has limitations. It might concentrate on certain parts of a prompt while overlooking others. A good prompt strikes a balance between detail and brevity. Excessive detail can overwhelm the model, while insufficient detail risks leaving gaps that the model may fill with incorrect assumptions.

> I used XML tags for few-shot examples because they clearly define example boundaries and are familiar to LLMs from pretraining on structured data. Furthermore, chat LM models are often finetuned using conversational examples with XML structures. Using XML isn't mandatory though, but could be helpful.

5.7.2. Followup Actions

The first solution from a model is often imperfect. User analysis and follow-up are key to getting the most out of a chat LM. Common follow-up actions include:

1. Asking the LLM whether its solution contains errors or can be simplified without breaking the constraints.
2. Copying the solution and starting a new conversation from scratch with the same LLM. In this new conversation, the user can ask the model to validate the solution as if it were "provided by an expert," without revealing it was generated by the same model.
3. Using a different LLM to review or enhance the solution.
4. For code outputs, running the code in the execution environment, analyzing the results, and giving feedback to the model. If code fails, the full error message and stack traceback can be shared with the model.

When working with the same chat LM for follow-ups, especially in tasks like coding or handling complex structured outputs, it's generally a good idea to start fresh after three-five exchanges. This recommendation comes from two key observations:

1. Chat LMs are typically finetuned using examples of short conversations. Creating long, high-quality conversations for finetuning is both difficult and costly, so the training data often lacks examples of long

interactions focused on problem solving. As a result, the model performs better with shorter exchanges.

2. Long contexts can cause errors to accumulate. In the self-attention mechanism, the softmax is applied over many positions to compute weights for combining value vectors. As the context length increases, inaccuracies build up, and the model's "focus" may shift to irrelevant details or earlier mistakes.

When starting fresh, it's important to update the initial prompt with key details from earlier follow-ups. This helps the model avoid repeating previous mistakes. By consolidating the relevant information into a clear, concise starting point, you ensure the model has the context it needs without relying on the long and noisy history of the prior conversation.

5.7.3. Code Generation

One valuable use of chat LMs is generating code. The user describes the desired code, and the model tries to generate it. As we know, modern LLMs are pretrained on vast collections of open-source code across many programming languages. This pretraining allows them to learn syntax and many standard or widely used libraries. Seeing the same algorithms implemented in different languages also enables LLMs to form shared internal representations (like synonyms in **word2vec**), making them generally indifferent to the programming language when reading or creating code.

Moreover, much of this code includes comments and annotations, which help the model understand the code's purpose—what it is designed to achieve. Sources like StackOverflow and similar forums add further value by providing examples of problems paired with their solutions. The exposure to such data gave LLMs an ability to respond with relevant code. Supervised finetuning improved their skill in interpreting user requests and turning them into code.

As a result, LLMs can generate code in nearly any language. For high-quality results, users must specify in detail what code should do. For example, providing a detailed docstring like this:

```
Write Python code that implements a method with the following
specifications:

def find_target_sum(numbers: list[int], target: int) -> tuple
```

```
:
    """Find pairs of indices in a list whose values sum to a 
target.

    Args:
        numbers: List of integers to search through. Can be e
mpty.
        target: Integer sum to find.

    Returns:
        Tuple of two distinct indices whose values sum to tar
get,
        or None if no solution exists.

    Examples:
        >>> find_target_sum([2, 7, 11, 15], 9)
        (0, 1)
        >>> find_target_sum([3, 3], 6)
        (0, 1)
        >>> find_target_sum([1], 5)
        None
        >>> find_target_sum([], 0)
        None

    Requirements:
        - Time complexity: O(n)
        - Space complexity: O(n)
        - Each index can only be used once
        - If multiple solutions exist, return any valid solut
ion
        - All numbers and target can be any valid integer
        - Return None if no solution exists
    """
```

Providing a highly detailed docstring can sometimes feel as time-consuming as coding the function itself. A less detailed description might seem more practical, but this increases the likelihood of the generated code not fully meeting user needs. In such cases, users can review the output and refine their instructions with additional requests or constraints.

> By the way, the book's official website, thelmbook.com, was created entirely through collaboration with an LLM. While it wasn't generated perfectly on the first try, through iterative feedback, multiple conversation restarts, and switching between different chat LLMs when needed, I refined every element you see—from the graphics to the animations—until they met my vision.

Language models can generate functions, classes, or even entire applications. However, the chance of success decreases as the level of abstraction increases. If the problem resembles model's training data, the model performs well with minimal input. However, for novel or unique business or engineering problems, detailed instructions are crucial for good results.

> If you decide to use a brief prompt to save time, ask the model to pose clarifying questions. You can also request it to describe the code it plans to generate first. This allows you to adjust or add details to the instructions before code is created.

5.7.4. Documentation Synchronization

A common challenge in software development is keeping documentation synchronized with code changes. As codebases evolve, documentation often becomes outdated, leading to confusion and reduced maintainability. LLMs offer an automated solution to this problem through integration with **version control systems**.

The process involves creating a documentation synchronization pipeline that leverages the LLM's ability to understand both code and natural language. When developers stage changes for commit, the pipeline:

1. Uses an LLM to analyze the staged differences and identify affected documentation files in the project's documentation directory. The model examines code changes and determines which documentation files might need updates.
2. Both the existing documentation content and staged code changes are then passed to another LLM call. This second step generates updated documentation that reflects the code modifications while maintaining the existing documentation's style and structure.

3. Places the updated documentation in the staging area alongside code changes. This allows developers to review both code and documentation updates together before committing, ensuring accuracy and maintaining a single source of truth.

This approach treats documentation as a first-class citizen in the development process, ensuring it evolves alongside the code.

> While LLMs can help maintain documentation alignment, they should not operate autonomously. Human review remains crucial to verify the accuracy of generated documentation updates and ensure they align with the team's communication standards.

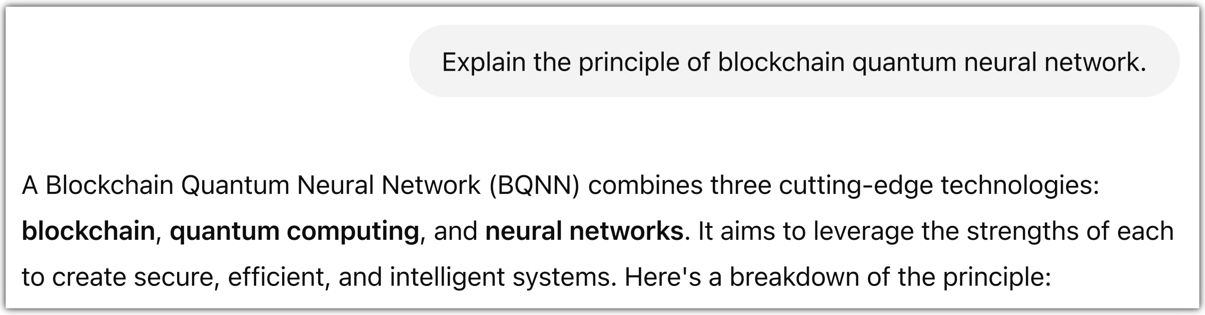
This pipeline is especially useful for keeping API documentation, architectural descriptions, and implementation guides up to date. However, like other LLM-based systems, it must include safeguards against hallucinations. We discuss this next.

5.8. Hallucinations

A major challenge with LLMs is their tendency to produce content that seems plausible but is factually incorrect. These inaccuracies, called **hallucinations**, create problems for using LLMs in production systems where reliability and accuracy are required.

5.8.1. Reasons for Hallucinations

Hallucinations in LLMs are by design. As we know, these models are optimized to predict the next token that fits the context, not to ensure factual accuracy. During pretraining, they learn to generate coherent text by following language patterns. However, no training dataset can cover every fact. When the model faces knowledge gaps, it tries to fill them with plausible content based on patterns it recognizes. This often results in fabricated details. Here's an example of hallucinated information from a widely used chat LM:


Explain the principle of blockchain quantum neural network.

A Blockchain Quantum Neural Network (BQNN) combines three cutting-edge technologies: **blockchain**, **quantum computing**, and **neural networks**. It aims to leverage the strengths of each to create secure, efficient, and intelligent systems. Here's a breakdown of the principle:

> As you can imagine, "Blockchain Quantum Neural Network (BQNN)" is not a real concept. The LLM's two-page explanation, including detailed descriptions of how it works, is entirely fabricated.

Low quality of training data also contributes to hallucinations. During pretraining on large volumes of internet text, models are exposed to both accurate and inaccurate information. They learn these inaccuracies but lack the ability to differentiate between truth and falsehood.

Finally, LLMs generate text one token at a time. This approach means that errors in earlier tokens can cascade, leading to increasingly incoherent outputs.

5.8.2. Preventing Hallucinations

Hallucinations cannot be completely avoided, but they can be minimized. A practical way to reduce hallucinations is by grounding the model's responses in verified information. This is done by including relevant factual context directly in the prompt. For instance, rather than posing an open-ended question, we can provide specific documents or data for the model to reference and instruct the model to only answer based on the provided documents.

This method, called **retrieval-augmented generation** (RAG), anchors the model's output to verifiable facts. The model still generates text but does so—most of the time—within the limits of the provided context, which significantly reduces hallucinations.

Here's how RAG works: a user submits a query, and the system searches a knowledge base—like a document repository or database—for relevant information. It uses keyword matching and embedding-based search, where the query is converted into an embedding vector. Documents with similar embeddings are retrieved using **cosine similarity**. To handle long documents, they are split into smaller chunks before embedding.

The retrieved content is added to the prompt alongside the user's question. This approach merges the strengths of traditional information retrieval with the language generation capabilities of LLMs. For example, if a user asks about a company's latest quarterly results, the RAG system would first retrieve the most recent financial reports and use them to produce the response, avoiding reliance on potentially outdated training data.

Another way to reduce hallucinations is by finetuning the model on reliable, domain-specific knowledge using unlabeled documents. For instance, a question-answering system for law firms could be finetuned on legal documents, case law, and statutes to improve accuracy within the legal domain. This approach is often referred to as **domain-specific pretraining**.

For critical applications, implementing a multi-step verification workflow can provide additional protection against hallucinations. This might involve using multiple models with different architectures or training data to cross-validate responses and having domain experts review generated content before it's used in production.

However, it's important to recognize that hallucinations cannot be completely eliminated with current LLM technology. While we can implement various safeguards and detection mechanisms, the most robust approach is to design systems that account for this limitation.

For instance, in a customer service application, an LLM could draft responses, but human review would be necessary before sending messages containing specific product details or policy information. Similarly, in a code generation system, the model might generate code, but automated tests and human review should always occur before deployment.

> The potential for hallucinations was notably demonstrated when Air Canada's customer service chatbot provided incorrect information about bereavement travel rates to a passenger. The chatbot falsely claimed that customers could book full-price tickets and later apply for reduced fares, contradicting the airline's actual policy. When the passenger tried to claim the fare reduction, Air Canada's denial led to a small claims court case, resulting in an $812 CAD (near $565 USD) compensation order. This case highlights the tangible business

> consequences of AI inaccuracies, including financial losses, customer frustration, and reputational damage.

Success with LLMs lies in recognizing that hallucinations are an inherent limitation of the technology. However, this issue can be managed through thoughtful system design, safeguards, and a clear understanding of when and where these models should be applied.

5.9. LLMs, Copyright, and Ethics

The widespread deployment of LLMs has introduced novel challenges in copyright law, particularly regarding training data usage and the legal status of AI-generated content. These issues affect both the companies developing LLMs and the businesses building applications with them.

5.9.1. Training Data

The first major copyright consideration involves training data. LLMs are trained on large text datasets that include copyrighted material such as books, articles, and software code. While some claim that this might qualify as fair use,[9] this has not been tested in court. The issue is further complicated by the models' capacity to output protected content. This legal uncertainty has already sparked high-profile lawsuits from authors and publishers against AI companies, posing risks for businesses using LLM applications.

> Meta's decision to withhold its multimodal Llama model from the European Union in July 2024 exemplifies the growing tension between AI development and regulatory compliance. Citing concerns over the region's "unpredictable" regulatory environment, particularly regarding the use of copyrighted and personal data for training, Meta joined other tech giants like Apple in limiting AI deployments in European

[9] Fair use is a U.S. legal doctrine. Other regions handle copyright exceptions differently. The EU relies on "fair dealing" and specific statutory exceptions, Japan has distinct copyright limitations, and other countries apply unique rules for permitted uses. This variation complicates global LLM deployment, as training data allowed under U.S. fair use might violate copyright laws elsewhere.

> markets. This restriction highlights the challenges companies face in balancing innovation with regional regulations.

When selecting models for commercial use, companies should review the training documentation and license terms. Models trained primarily on **public domain** or properly licensed materials involve lower legal risks. However, the massive datasets required for effective LLMs make it nearly impossible to avoid copyrighted material entirely. Businesses need to understand these risks and factor them into their development strategies.

Beyond legal issues, training LLMs on copyrighted material raises ethical concerns. Even when legally permissible, using copyrighted works without consent may appear exploitative, especially if the model outputs compete with the creators' work. Transparency about training data sources and proactive engagement with creators can help address these concerns. Ethical practices should also involve compensating creators whose contributions significantly improve the model, fostering a more equitable system.

5.9.2. Generated Content

The copyright status of content generated by LLMs presents challenges that traditional copyright law cannot easily resolve. Copyright law is built around the assumption of human authorship, leaving it unclear whether AI-generated works qualify for protection or who the rightful owner might be. Another issue is that LLMs can sometimes reproduce portions of their training data verbatim, including copyrighted material. This ability to generate exact reproductions—beyond learning abstract patterns—raises serious legal questions.

Some businesses address these challenges by using LLMs as assistive tools rather than independent creators. For example, a marketing team might use an LLM to draft text, leaving human writers to edit and finalize it. This approach maintains clearer copyright ownership while leveraging AI's efficiency. Similarly, software developers use LLMs to generate code snippets, which they review and integrate into larger systems. By 2024, this practice had grown significantly—at Google, over 25% of all code was generated by LLMs and then refined by developers.

To minimize copyright risks in LLM applications, companies often implement technical safeguards.

One method involves comparing model outputs against a database of copyrighted materials to detect verbatim copies. For example, a company may maintain a repository of copyrighted texts and employ similarity detection methods—such as **cosine similarity** or **edit distance**—to flag outputs that surpass a defined similarity threshold.

However, these methods are not foolproof. Paraphrased content can make the output formally distinct while remaining substantively similar, which automated systems may fail to detect. To handle this, businesses often supplement these tools with human review to ensure compliance.

5.9.3. Open-Weight Models

The copyright status of model weights poses legal questions separate from those concerning training data or generated outputs. Model weights encode patterns learned during training and could be viewed as derivative works of the training data. This leads to the question: does sharing weights amount to indirectly redistributing the original copyrighted materials, even in their transformed form? Some argue that weights are an abstract transformation and constitute new intellectual property. Others contend that if weights can reproduce fragments of the training data, they inherently include copyrighted content and should be treated similarly under copyright law.

This debate carries serious implications for open-source AI development. If model weights are classified as derivative works, sharing and distributing models trained on copyrighted data could become legally restricted, even if the training process qualifies as fair use. As a result, some organizations have shifted to training models solely on public domain or explicitly licensed content. However, this strategy often limits the effectiveness of the models, as the smaller, restricted datasets typically lead to reduced performance.

As laws around LLMs evolve, businesses must stay flexible. They may need to adjust workflows as courts define legal boundaries or revise policies as AI-specific legislation appears. Consulting intellectual property lawyers with AI expertise can help manage these risks.

5.9.4. Broader Ethical Considerations

Beyond copyright concerns, LLMs raise significant ethical challenges that affect society at large. One fundamental issue is **explainability**. While LLMs can articulate reasoning for their outputs and provide detailed explanations when asked, this verbal explanation capability differs from true algorithmic transparency. The model's explanations are post-hoc rationalizations—generated text that sounds plausible but may not reflect the actual computational process that produced the original output. This creates a unique challenge where the model appears transparent while its underlying decision-making process remains opaque. This limitation becomes particularly significant in high-stakes applications like healthcare or legal services.

The question of **bias** presents another challenge. LLMs trained on internet data inevitably absorb societal biases present in their training data. These models can perpetuate or amplify discriminatory patterns in areas such as gender, race, age, and cultural background. For instance, they might generate different responses to equivalent prompts that only differ in demographic details, or produce content that reinforces stereotypes.

Organizations deploying LLMs must implement structured evaluation protocols, including automated bias detection across demographic groups and audits using standardized test sets. This should include deploying concrete safeguards like toxic language filters, mandatory human review for high-stakes decisions, and clear user notifications about AI involvement.

Chapter 6. Further Reading

You've learned the core concepts of language modeling throughout this book. There are many advanced topics to explore on your own, and this final chapter provides pointers for further study. I've chosen topics that represent important current developments in the field, from architectural innovations to security considerations.

6.1. Mixture of Experts

Mixture of experts (**MoE**) is an architectural pattern designed to increase model capacity without a proportional rise in cost. Instead of a single position-wise **MLP** processing all tokens in a decoder block, MoE uses multiple specialized sub-networks called **experts**. A **router network** (or **gate network**) decides which tokens are processed by which experts.

The core idea is activating only a subset of experts for each token. This **sparse** activation reduces active computations while enabling larger overall parameter counts. **Sparse MoE layers** replace traditional MLP layers, using techniques like **top-k routing** and **load balancing** to efficiently assign tokens to experts.

This concept gained attention with the **Switch Transformer** and has been applied in models such as **Mixtral 8x7B**, which has 47B total parameters but only activates about 13B during inference.

6.2. Model Merging

Model merging combines multiple pretrained models to make use of their complementary strengths. Techniques include **model soups**, **SLERP** (spherical interpolation that maintains parameter norms), and **task vector algorithms** such as **TIES-Merging** and **DARE**.

These methods generally rely on some architectural similarity or compatibility between models. The **passthrough** method stands out by concatenating layers from different LLMs. This approach can create models with unconventional parameter counts (e.g., 13B by merging two 7B models). Such models are often called **frankenmerges**.

mergekit is a popular open-source tool for merging and combining language models that implements many of these techniques. It provides a flexible configuration system for experimenting with different merging strategies and architectures.

6.3. Model Compression

Model compression addresses deploying LLMs in resource-limited environments by reducing size and computation needs without greatly sacrificing performance. Neural networks are often **over-parameterized**, containing redundant units that can be optimized.

Key methods include **post-training quantization**, which lowers parameter precision (e.g., 32-bit floats to 8-bit integers), **quantization-aware training**, training models at lower precision, such as **QLoRA** (quantized low-rank adaptation), **unstructured pruning**, removing individual weights by importance, **structured pruning**, removing components like layers or attention heads, and **knowledge distillation**, where a smaller "student" model learns from a larger "teacher" model.

6.4. Preference-Based Alignment

Preference-based alignment methods help align LLMs with user values and intent, so they produce helpful and safe outputs. A widely used approach is **reinforcement learning from human feedback** (**RLHF**), where humans rank model responses, a **reward model** is trained on these rankings, and then the LLM is finetuned to optimize for higher reward.

Another approach is **constitutional AI** (CAI), which uses a set of guiding principles or a "constitution" that the model refers to when producing its output; the model can **self-critique** and revise its responses based on these principles. Both strategies address the problem that LLMs, when trained on vast internet text, may generate harmful or **misaligned** responses, but they differ in how they incorporate human oversight and explicit guidelines.

6.5. Advanced Reasoning

Advanced reasoning techniques enable large language models to handle complex tasks by (1) training them to generate an explicit **chain of thought** (**CoT**) for step-by-step reasoning and (2) equipping them with **function calling** capabilities to invoke external APIs or tools, thereby addressing limitations of simple prompt-response patterns. Chain-of-thought reasoning can significantly improve performance on tasks such as multi-step mathematics and logical inference, while function calling allows offloading specialized computations to external frameworks.

Additionally, **tree of thought** (*ToT*) extends CoT by exploring multiple reasoning paths in a tree-like structure. **Self-consistency** further refines reasoning by aggregating multiple CoT outputs for the most consistent answer. **ReAct** (**reasoning+act**) integrates reasoning with action-taking, allowing models to interact with environments dynamically. **Program-aided language models** (**PAL**) leverage interpreters (e.g., Python) to execute code for precise calculations.

6.6. Language Model Security

Jailbreak attacks and **prompt injection** are major security vulnerabilities in LLMs. Jailbreaks bypass the model's safety controls by crafting specific inputs that trick the model into producing restricted content, often using techniques like roleplaying as a different character or setting up hypothetical scenarios. For example, an attacker might prompt the model to act as a pirate to obtain instructions on illegal activities.

In contrast, prompt injection attacks manipulate how LLM applications combine **system prompts** with user input, allowing attackers to alter the application's behavior. For instance, an attacker could insert commands that make the application execute unauthorized actions. While jailbreaks primarily risk exposing harmful or restricted content, prompt injection presents more severe security implications for applications with privileged access, such as those that read emails or execute system commands.

6.7. Vision Language Model

Vision language models (**VLMs**) integrate an LLM with a **vision encoder** to handle both text and images. Unlike traditional models that process modalities in isolation, VLMs excel at **multimodal reasoning**, enabling them to perform a variety of vision tasks by following natural language instructions without task-specific retraining. The architecture includes three main components: a **CLIP**-based (contrastive language-image pretraining) **vision encoder** trained on millions of image-text pairs to understand visual content, a **cross-attention** mechanism that allows the VLM to integrate and reason about visual and textual information, and the language model itself that generates and interprets text. VLMs are developed through multiple training stages, starting with pretraining to align the visual and language components, followed by supervised finetuning to improve their ability to understand and respond to user prompts.

6.8. Preventing Overfitting

Techniques for preventing **overfitting** are essential for achieving model **generalization**, ensuring that models perform well not just on training data but also on new, unseen examples. The primary defense against overfitting is **regularization**, which includes methods like **L1** and **L2**. These techniques add specific penalty terms—such as the sum of absolute or squared weights—to the loss function, limiting the size of model parameters and encouraging simpler models.

Dropout is a regularization method for neural networks. It works by randomly deactivating some units during each training step. This encourages the network to develop multiple independent pathways, reducing reliance on specific features. **Early stopping** prevents overfitting by monitoring validation performance. Training stops when validation accuracy stops improving or starts to decline, avoiding the memorization of random noise happening at later epochs.

A **validation set** is similar to the **test set** in that it is used to evaluate the model's performance on unseen data; however, the key difference is that the validation set is used during the training process to tune hyperparameters and make decisions such as early stopping, while the test set is reserved for final evaluation to measure the model's performance after training is complete.

6.9. Concluding Remarks

You've come a long way in understanding language models, from the basic building blocks of machine learning to the inner workings of transformers and the practical aspects of working with large language models. You now have a solid technical foundation that lets you not only understand how these models work but also implement and adapt them for your own purposes.

New architectures, training methods, and applications of language models are emerging. You now have the tools to read research papers, follow technical discussions, and evaluate new developments critically. Whether you aim to train models or build systems using them, you have the core concepts to proceed confidently.

I encourage you to stay curious and hands-on—implement the concepts you've learned, experiment with different approaches, and keep up with the latest developments. Consider starting with some of the advanced topics covered in this chapter, but remember that the fundamentals you've learned here will serve as your compass in navigating future innovations.

A good way of keeping up with the latest developments is to subscribe to the book's newsletter.

The book ends here. Remember to check the companion wiki from time to time for updates on developments in various language modeling areas. Please don't forget that the book is shared under the *read first, buy later* principle. So, if you're reading this as a PDF and don't recall paying for it, you are probably the right person to purchase the book.

6.10. More From the Author

If you're still reading, it likely means you enjoyed the book and are wondering what else you can read from this author. I have two more books that will definitely enhance your understanding of machine learning and build on the knowledge and intuition you've gained about language models:

- **The Hundred-Page Machine Learning Book** offers a concise yet thorough overview of core machine learning concepts, ranging from fundamental statistics to advanced algorithms. It's an excellent companion to the language modeling material covered here.
- **Machine Learning Engineering** covers the practical aspects of designing, deploying, and maintaining ML systems at scale. If you're looking to move beyond experimentation and create robust, real-world machine learning applications, this book will guide you through every stage of the machine learning engineering lifecycle.

Index

"This book cleared up a lot of conceptual confusion for me about how Machine Learning actually works—it is a gem of clarity. The worked examples and notebook applications gave me a solid starting point for exploration. Even if you are not planning a career in machine learning applications, this is a solid foundation for thinking about the capabilities of these unique new tools."

Vint Cerf, *Internet Pioneer*