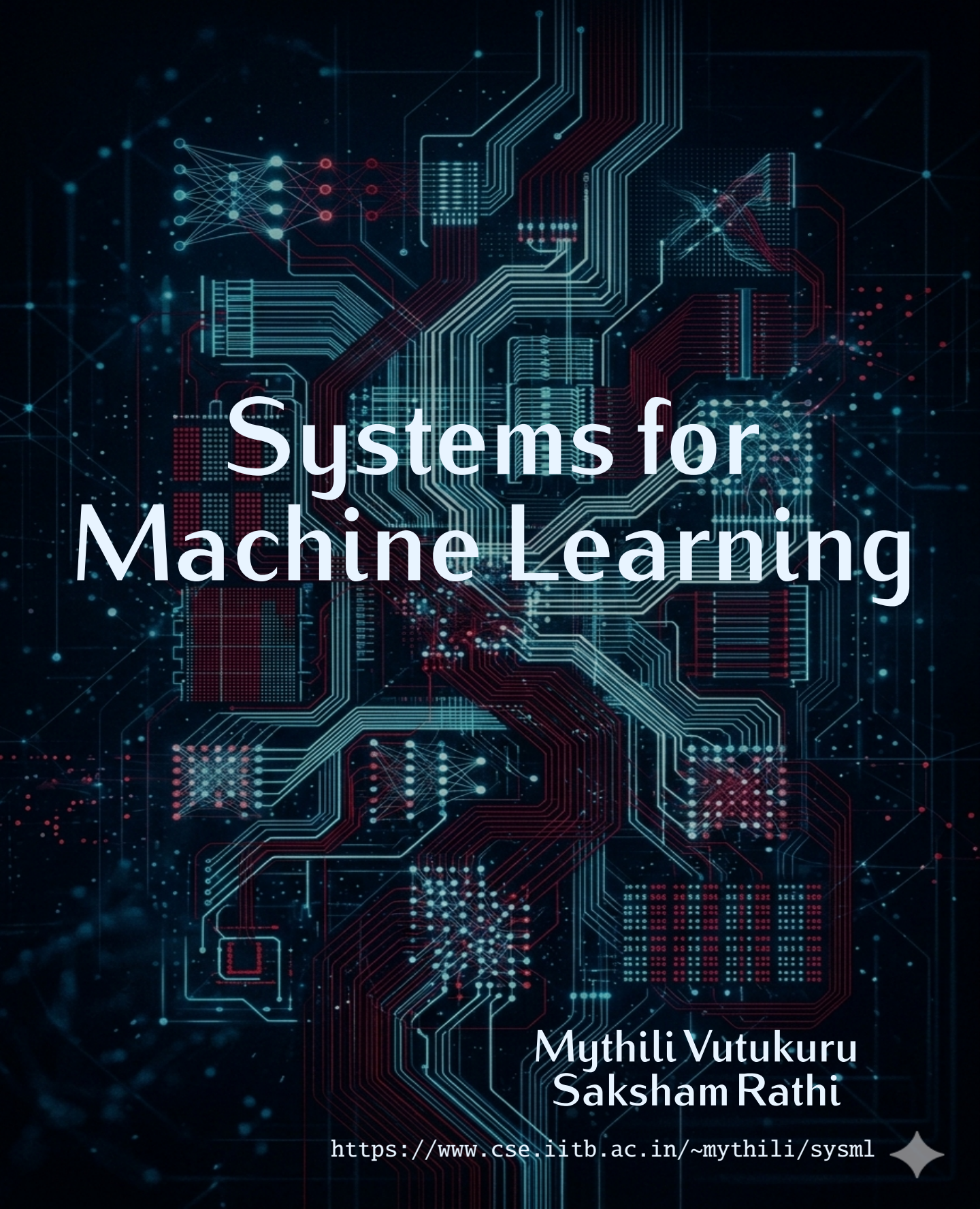

Systems for Machine Learning
Mythili Vutukuru
Saksham Rathi
https://www.cse.iitb.ac.in/~mythili/sysml

DRAFT CHAPTER

October 2026

1

Introduction

1.1 Why Study Systems for Machine Learning? **3**
1.1.1 History of AI and ML 4
1.1.2 What is at stake? 7
1.2 Software and Hardware Systems **9**
1.2.1 GPUs 9
1.2.2 Software Stacks 14
1.3 Book Outline **17**
1.4 How to Use the Book? **20**
1.4.1 Reading the Chapters 20
1.4.2 Practice Problems 22
1.4.3 Programming Assignments 23
1.4.4 Slides and the Companion Website 24

If you learned to code any time in the last fifteen years, chances are Stack Overflow saved you at least once. Stuck on a cryptic error message at 2 A.M. in the morning? Paste it into a search engine, land on Stack Overflow, and there, almost like magic, someone else in the world had already hit the exact same wall and someone else had already solved it. For a long time, this was how programming worked. Millions of questions, millions of answers, and a whole generation of engineers who grew up believing that any bug could be defeated with the right search query.

Then something strange happened. Around 2020, question volume on Stack Overflow started to drift downward (Figure 1.1)[1]. In November 2022, a chatbot called ChatGPT was released to the public, and the slow fade turned into a landslide. Within a couple of years, monthly question counts had fallen by well over half, and some measurements put the decline above ninety percent

[1] Stack Overflow may have lost the programmers, but it still had the image we needed!

from the platform's peak. A website that had defined an entire era of software development, had in the space of two or three years, been pushed to the margins of how programmers actually get help. This example clearly illustrates the impact that artificial intelligence and machine learning (AI/ML) systems have had on computing, and on people's lives in general, in the recent past.

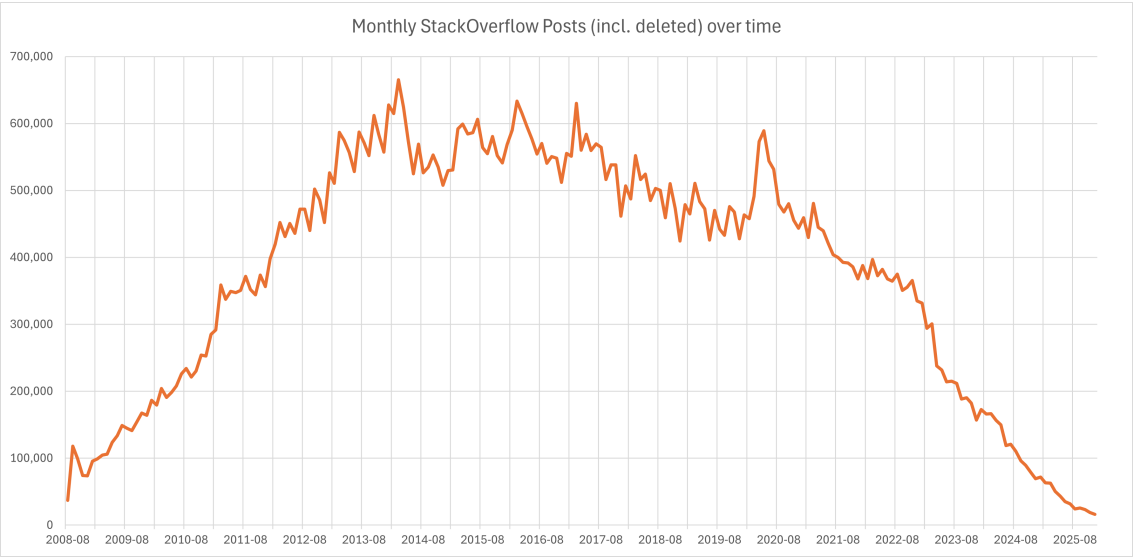


Figure 1.1: Monthly Stack Overflow posts over time; Image credit: [einpoklum, 2026]

Why does a story about a question-and-answer website belong in the opening pages of a book about machine learning systems? Because it is the perfect illustration of what this book is really about. The popularity of ChatGPT is not just due to the brilliant machine learning algorithms that underpin modern large language models. A big part of its success is due to an enormous, carefully engineered pile of infrastructure sitting underneath it: data collected and curated from across the internet, a model trained on that data using thousands of processors, and a serving system capable of answering queries from millions of people at once, in real time. This book is about that underlying machinery — the part of the story that rarely makes headlines, but is the actual reason the headlines happen at all. This book is about the hardware and software infrastructure stack that supports modern machine learning systems, which make all this magic happen at a global scale.

We begin this chapter by motivating the reader on why one should study the "systems" aspects of machine learning, by highlighting the role played by the systems infrastructure in the recent widespread adoption of AI/ML. We then examine an overview of all the topics covered in this book, and how one can navigate the book to get the most of out of it.

1.1 Why Study Systems for Machine Learning?

Every serious machine learning course you will ever take, and every impressive AI demo you will ever see, secretly rests on the same three pillars: the model, the data, and the infrastructure that ties them together (Figure 1.2). The model is the part that most people think of when they hear the words "machine learning". It is a mathematical construction, often built from layers of simple arithmetic operations, that learns patterns from data instead of being explicitly programmed with rules. This is the world of machine learning courses, of loss functions, gradients and optimizations, and it is a rich and beautiful field on its own. Data, the second pillar, is the fuel. No amount of mathematical cleverness helps a model that has nothing meaningful to learn from. Collecting, storing, cleaning, and organizing enormous volumes of data is its own discipline, usually called data engineering, and it comes with its own tools.

The infrastructure is the third pillar that this book cares about most. It is the software and hardware that actually runs the training and the inference: the clusters of thousands of machines, specialized hardware chips and accelerators, networks that move data and gradients around fast enough for training to finish in days instead of decades, and serving systems that let a model answer a billion queries without falling over. Models and datasets get most of the attention, but infrastructure is the quiet third leg of the stool, and without it, the other two simply cannot stand.

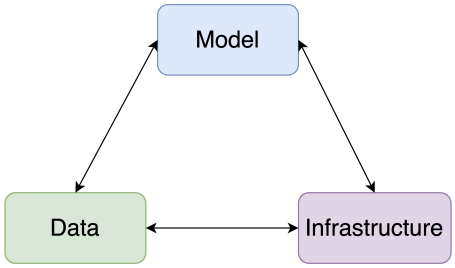


Figure 1.2: The three pillars of machine learning

It helps to picture machine learning not as a single event, where a model appears fully formed, but as a continuous loop that keeps turning long after the first version ships. It starts with data collection, where a continuous stream of raw material is pulled in from wherever it lives. That raw data gets ingested and prepared for downstream use, then analyzed and curated so that only the right slice of it is selected. Selected data gets labeled, then validated to make sure it is actually usable, and finally prepared, split, and versioned so it is ready to feed a model. From there, the loop moves into model training, where learning algorithms chew through the prepared data to produce candidate models. Those models are evaluated against key performance indicators, validated to check they are safe and effective enough to ship, and, if they pass, deployed into a real, live system. Once deployed, the system's online performance feeds straight back into data collection, because the mistakes a

model makes in the real world are often the most valuable training examples of all (Figure 1.3). Data needs identified along the way flow back into curation, and fixes flow back into the earliest stages of collection. Every single stage in this loop, from ingestion to deployment, depends on infrastructure to run at any meaningful scale.

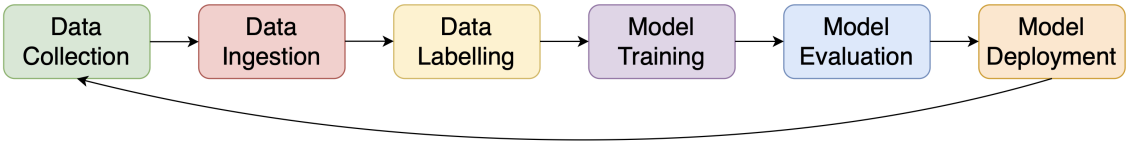


Figure 1.3: The machine learning lifecycle

As we will see in this section, progress in artificial intelligence has never depended only on smarter statistical models or better-curated datasets. It has always depended just as much on the infrastructure available to run those models on. Time and again, the biggest leaps forward in machine learning have come not from a clever new equation but from systems finally catching up to the scale that the models and the data demanded. This relationship also runs in the opposite direction. Systems constraints, like how quickly data can move between memory and a processor, have directly shaped which algorithms succeed and which do not. Specialized hardware such as tensor processing units, and algorithms deliberately redesigned to exploit that hardware, exist because raw computational limits forced researchers to get creative. And there is a more urgent reason to care about this than intellectual curiosity: training and serving today's largest models consumes staggering amounts of power and physical resources. As the scale of AI keeps growing, optimizing machine learning systems may well be essential for our collective survival, in the very literal sense of how much energy and raw material the world can spare for it.

1.1.1 History of AI and ML

Artificial intelligence did not arrive suddenly. It crept forward in fits and starts across seven decades. The earliest systems, built in the 1950s and 1960s, were rule-based expert systems with hand-coded logic that could, for example, encode the rules of a board game well enough to play it. Later came statistical learning systems, the kind of technology behind early spam filters, which learned simple patterns from labeled examples instead of following hand-written rules. From there the field moved into what we now call classical machine learning: decision trees, support vector machines, and small neural networks, all of which could learn moderately complex patterns, but this type of learning hit its limitations soon enough.

Deep learning, the branch built on large neural networks trained on large datasets, is the newest chapter, and by far the loudest one. According to Google's own long-running timeline of the field, early artificial intelligence stirred excitement in the middle of the twentieth century, machine learning quietly began to flourish through applications like spam detection in the 1990s and 2000s, and then

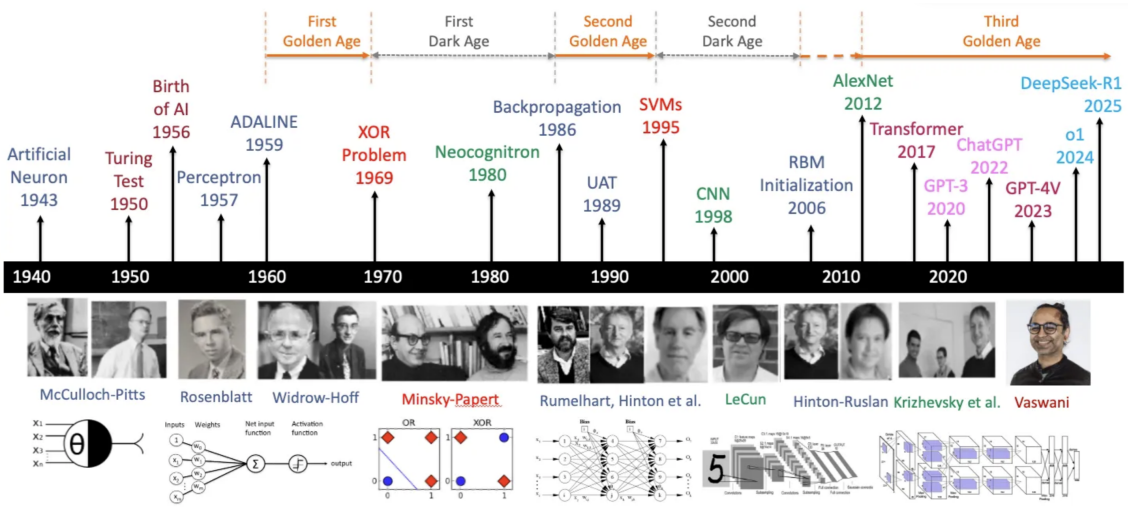


Figure 1.4: History of AI and ML; Image credit: [Po, 2024]

deep learning breakthroughs drove a boom starting in the 2010s. Figure 1.4 shows three very different eras, each waiting for the right combination of ideas, data, and hardware to finally click into place.

The Breakthrough: AlexNet

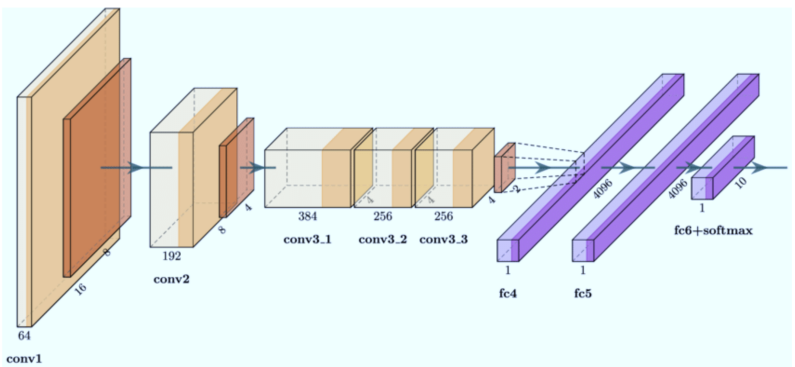


Figure 1.5: AlexNet; Image credit: [H, n.d.]

If you had to point to a single moment when deep learning started reshaping the entire technology industry, most researchers would point to 2012, and to a system called AlexNet [Krizhevsky et al., 2012]. AlexNet was entered into the ImageNet image classification challenge that year by Alex Krizhevsky, working with his PhD advisor Geoffrey Hinton and collaborator Ilya Sutskever. What

made AlexNet remarkable was not just that it won, but how decisively it won, and how it got there. Instead of training on ordinary processors, the team trained their network on graphics processing units (GPUs) — hardware originally built for rendering video game frames — and discovered that this hardware could push through the arithmetic of neural network training much faster than a conventional processor could. The result was an eight-layer convolutional neural network, and its success is often described as the moment "deep" learning earned its name. A network with this many layers, trained this quickly, on this much data, had simply never been demonstrated before.

Here is the part of the story that tends to surprise people. The core mathematical ideas behind AlexNet were not new in 2012. Perceptrons, neural networks, and the backpropagation algorithm used to train them had all been around for decades, some of them since the 1950s and 1960s. Researchers had proposed these ideas long before anyone knew how to make them work at scale, and for most of the 1990s and 2000s, neural networks were a relatively quiet corner of machine learning, respected but not dominant.

So what changed? Three things, arriving at almost exactly the same time. Large datasets became available, thanks to the internet, the spread of smartphones, and the rise of cloud computing, all of which made it possible to collect and store the sheer volume of labelled examples that deep networks need to learn anything useful. Huge computing power became available, most importantly in the form of parallel processing capability sitting inside GPUs. While GPUs were initially developed for graphical processing and rendering tasks, they were repurposed to run general-purpose ML workloads because the hardware design happened to be extremely good at the kind of matrix arithmetic neural networks rely on. Finally, a series of smaller algorithmic breakthroughs made training and inference more stable and more efficient than they had been in earlier decades. None of these three ingredients was sufficient on its own. It took all three arriving together, in the 2010s, for the stars to finally align.

Today: ChatGPT, Everyone's Best Friend

While AlexNet heralded the era of deep learning in the world of computing, generative AI based tools like ChatGPT have truly pushed AI/ML into the forefront of the daily lives of the general populace. Modern large language models (LLMs) and other genAI models are generally based on the transformer architecture, which follows the same fundamental recipe of neural networks invented decades earlier — layers of weighted connections, trained with variants of the backpropagation algorithm, with weights adjusted using an optimizer. What has changed, however, is the scale, by several orders of magnitude, in every dimension, from parameters to training data to the compute requirements. Table 1.1 lays out the comparison, and it is worth pausing to appreciate just how far things moved in less than a decade after AlexNet.

With newer, larger models being developed at breathtaking pace, it is then not surprising that the underlying hardware and software systems infrastructure has also had to scale commensurately, to ensure the feasibility of training and deploying these large models. More interestingly, the models themselves have adapted to run more efficiently over the underlying hardware, using hardware-

Feature	AlexNet (2012)	GPT-3 (2020)
Model type	Convolutional neural network for image classification	Generative pre-trained transformer for natural language tasks
Parameters	Approximately 60 million	175 billion
Training data size	The ImageNet database, roughly 1.2 million labelled images	Roughly 45 terabytes of raw text, filtered and compressed to about 570 gigabytes
Training hardware	Two Nvidia GTX 580 GPUs, running for about six days	A distributed computing environment using thousands of GPUs running for several weeks

Table 1.1: From AlexNet to GPT-3

aware performance optimization techniques. For example, the "attention" layer in the transformer architecture, which is responsible for incorporating context from previous words into current word, now has a very efficient implementation (called "FlashAttention"), whose key idea is to tailor the attention computation to the compute and memory architecture of modern GPUs. This synergy between the ML models and the underlying systems infrastructure has delivered the widespread adoption and AI/ML that we see today.

1.1.2 What is at stake?

If you are still unconvinced on the significance of the "systems" aspects of AI/ML, let us give you more reasons. As models have grown from millions of parameters to hundreds of billions, and now into the trillions, the cost of training and deploying these models has grown alongisde them. Training a single frontier language model can cost more than the media budget of a Hollywood blockbuster, and the electricity bill alone can rival that of a small city's daily consumption. Data centers built specifically for AI workloads are now among the largest consumers of electricity and water in the regions where they are built, and their footprint is growing faster than most national power grids were designed to handle. In fact, some new data centres are being built next to dedicated power plants, because the surrounding grid simply cannot supply enough electricity on its own.

None of this is an argument against building large models. It is an argument for building them well. A poorly engineered training pipeline might waste half its compute on idle GPUs waiting for data, and a poorly engineered serving system might waste enormous energy on inefficient inference. Every improvement in systems efficiency, therefore, is not just a cost saving; it is directly a reduction in the physical resources the entire endeavour consumes. But if you get the systems aspects wrong, a training run that should take three weeks stretches into three months, burning through more

resources than it should.

Now, let us move on to more mundane reasons beyond saving the planet! Two companies can deploy nearly identical LLMs today, but the one with the better inference infrastructure will answer queries faster, cheaper, and at far greater scale, and thereby become more popular. This has created enormous demand for engineers who sit at the intersection of machine learning and traditional computer systems. Understanding how to train and serve models efficiently at scale is no longer a niche specialization. In fact, it has become one of the most sought-after skill sets in the entire technology industry, precisely because so few machine learning engineers are also trained to think through the systems aspects, including hardware accelerator architecture, software systems design, programming abstractions and compilation worklows, GPU networking, and so on. This book aims to bridge this knowledge gap, and fill this need in the industry today.

1.2 Software and Hardware Systems

Every model that has ever been trained, and every model that has ever answered a query, has ultimately run as electrical signals moving through physical silicon. It is easy to forget this while writing a line of Python that says `model.fit(data)`, but underneath that single line sits an enormous stack of hardware and software, each layer built to hide the complexity of the layer below it. This section walks that stack from the bottom up: first the hardware that does the actual arithmetic, then the software layers that let a human write ordinary-looking code and have it run correctly across thousands of chips at once. Along the way we will meet a lot of jargon: HBM, FLOPs, systolic arrays, kernels, compilers, and more. None of it is as scary as it sounds, and by the end of this section you should be able to read a GPU spec sheet the way a car enthusiast reads its features.

1.2.1 GPUs

A central processing unit, or CPU, is built to be a generalist. It is designed to run a huge variety of tasks, one or a handful at a time, as quickly and flexibly as possible, and it is very good at this job. A modern CPU might have somewhere between four and sixty-four cores, each one a powerful, independent brain capable of handling complex branching logic, unpredictable memory access, and rapid switching between different tasks.

Neural network training and inference, however, do not need a handful of powerful generalist brains. They need an enormous number of simple, repetitive arithmetic operations, mostly matrix multiplications and additions, performed on huge blocks of data at the same time. Multiplying a batch of images by a weight matrix does not require clever branching logic. It requires doing the exact same simple multiplication tens of thousands of times, independently, as quickly as possible. This is precisely the kind of workload a graphics processing unit, or GPU, was built for.

GPUs were not originally built with machine learning in mind. They were built to render video game graphics, which requires computing the position of millions of independent pixels, many times per second. To handle this, GPU manufacturers packed their chips with thousands of small, simple cores instead of a handful of powerful ones, trading flexibility for raw parallel throughput. It turns out that training a neural network looks, mathematically, quite similar to rendering a scene: both are dominated by large, independent, parallel arithmetic operations. When researchers in the early 2010s started running neural network training on GPUs instead of CPUs, exactly as we saw with AlexNet in the previous section, they found the same chips could push through neural network arithmetic dramatically faster than any CPU of the time.

Anatomy of a Modern GPU

A modern GPU built for machine learning, such as an NVIDIA A100, H100, or B200, is organized very differently from a CPU. Instead of a small number of large cores, it contains thousands of small arithmetic cores grouped into clusters, alongside a smaller number of specialized tensor cores, which are circuits purpose-built to perform the matrix multiply-and-accumulate operations that dominate

deep learning (Figure 1.6). A single tensor core can multiply two small matrices together and add the result to a third matrix in a single hardware operation, called a fused multiply-add, that is common in machine learning workloads.

Just as important as the arithmetic cores is the memory system feeding them. GPUs used for machine learning are equipped with high-bandwidth memory, or HBM, a type of memory stacked in physical layers directly next to the compute cores to minimize the distance data has to travel. A huge amount of systems engineering in machine learning, as we will see in later chapters, is really about moving data efficiently from memory to compute units of a GPU. A kernel is memory-bound if it spends more time waiting on data than actually computing, and compute-bound if the arithmetic itself is the bottleneck. Squeezing more performance out of a GPU is very often about turning memory-bound operations into compute-bound ones.

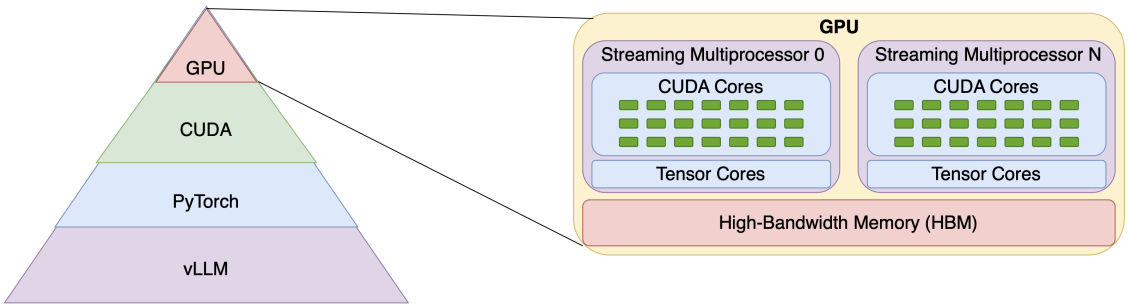


Figure 1.6: Software and hardware systems

Table 1.2 puts some real numbers behind this story. A typical server CPU offers a few hundred gigabytes per second of memory bandwidth and a handful of teraFLOPs of general-purpose arithmetic, where a FLOPs is short for "flaoting point operations per second". In contrast, a single modern GPU offers memory bandwidth measured in terabytes per second and arithmetic throughput measured in thousands of teraFLOPs, or petaFLOPs, when using the low-precision formats described below. These are about two orders of magnitude higher than what CPUs offer, and explain why the entire field of deep learning runs on GPUs rather than CPUs.

The GPU Generations

It is worth pausing to appreciate just how quickly this hardware has evolved, because the pace itself is part of the story of modern AI. NVIDIA's data-center GPUs have gone through a rapid sequence of architectural generations, each with its own code name, and each roughly doubling or tripling the effective throughput available for training large models. The Volta generation, headlined by the V100, was the first to introduce dedicated tensor cores and is often credited with kicking off the modern era of large-scale deep learning. The Ampere generation followed with the A100, which became the chip behind the first wave of large language models, including the original GPT-3. Then came

	Server CPU	V100	A100	H100	B200
Memory bandwidth	~0.3–0.4 TB/s	0.9 TB/s	2.0 TB/s	3.35 TB/s	8.0 TB/s
FP16/BF16 (TFLOPS)	~2–5	125	312	~990	~2,250
FP8 (TFLOPS)	—	—	—	~1,980	~4,500
Memory type	DDR5	HBM2	HBM2e	HBM3	HBM3e

Table 1.2: Peak specifications of a representative server CPU and four generations of NVIDIA datacenter GPUs

Hopper, headlined by the H100 and later the higher-memory H200, named after computing pioneer Grace Hopper, which introduced a new lower-precision numerical format called FP8 specifically to accelerate transformer workloads. Most recently, the Blackwell generation, headlined by the B100 and B200 chips, pushed the frontier even further, introducing an FP4 format and enormous multi-GPU modules such as the GB200 NVL72, discussed below, designed to be wired together into a single giant GPU.

This naming pattern is not an accident: NVIDIA names many of its architectures after scientists, and tracking the sequence, Volta, Ampere, Hopper, Blackwell, is a handy way to remember the rough chronological order in which the last decade of AI hardware progress happened. The next generation in this sequence, Rubin, named after astronomer Vera Rubin, was announced for release in the second half of 2026. Early specifications point to a further large jump: each Rubin GPU is expected to ship with 288 gigabytes of a new memory type called HBM4, offering around 22 terabytes per second of memory bandwidth, nearly three times that of a B200, alongside a sixth generation of the NVLink interconnect. NVIDIA has already signaled that the generation after Rubin, code-named Feynman, is planned for 2028, which gives a sense of just how short the cycle between major hardware jumps has become.

Numerical Formats

One especially important axis along which GPUs have evolved is numerical precision, i.e., how many bits are used to represent each number a model works with. Early deep learning largely used 32-bit floating point numbers, abbreviated FP32, the same format used throughout most of scientific computing. Researchers soon discovered that neural networks tolerate much lower precision than most numerical code, because a small amount of rounding error in a weight update rarely changes the final outcome by much. This observation launched a steady march toward smaller number formats: 16-bit floating point (FP16) and its more dynamic-range-friendly cousin BF16 (bfloat16) became standard for training. More recently, 8-bit (FP8) and even 4-bit (FP4) formats have been introduced for both training and inference on the newest hardware. As Table 1.2 shows, dropping from FP16 to FP8 roughly doubles peak throughput on the same chip, and Blackwell's FP4 mode doubles it again, to around 9,000 TFLOPS on a single B200. Each halving of precision roughly

doubles the arithmetic throughput and halves the memory footprint of a model, which is why so much recent hardware design effort has gone into supporting ever-smaller formats without destroying model quality. This general strategy, called quantization when applied after training and mixed-precision training when applied during it, is one of the most important and recurring ideas in modern machine learning systems.

Beyond NVIDIA

NVIDIA GPUs currently dominate machine learning, holding about eighty percent of the AI accelerator market, but they are far from the only game in town, and it is worth knowing the landscape of alternatives, collectively often called AI accelerators, summarized in Table 1.3. Google has, since 2015, designed its own custom chips called Tensor Processing Units, or TPUs, which are application-specific integrated circuits, meaning chips built from the ground up for a single purpose rather than adapted from a general-purpose design. TPUs are built around a hardware design called a systolic array, a grid of arithmetic units that pass data directly to their neighbors in a synchronized rhythm, which is extremely efficient for the large matrix multiplications that dominate deep learning, though less flexible than a GPU for other kinds of workloads. TPUs are typically deployed in large interconnected groups called pods, connected by Google's own custom interconnect, and Google has used successive TPU generations internally to train models like Gemini. The latest widely available generation, Trillium (also called TPU v6e), delivers roughly 918 TFLOPS of BF16 performance per chip, nearly five times its predecessor, while remaining significantly more power-efficient per unit of throughput than a comparable GPU.

Other companies have entered this race with their own custom silicon. Amazon has designed its own accelerators, Trainium for training and Inferentia for inference, to reduce its reliance on NVIDIA hardware in AWS data centers; the current Trainium2 generation is used to build ultra-servers claimed to deliver over 80 petaFLOPS of aggregate throughput. AMD produces the MI300X as a direct competitor to NVIDIA's H100 and H200, notable for shipping with 192 gigabytes of HBM3, more memory capacity than an H100, which makes it attractive for serving very large models where memory capacity, not raw compute, is the bottleneck. Cerebras has taken an unusually extreme approach, building wafer-scale chips: rather than cutting a silicon wafer into many small chips the usual way, Cerebras uses an entire wafer as a single enormous chip. Its third-generation Wafer-Scale Engine packs roughly 900,000 AI cores and 44 gigabytes of on-chip memory onto one piece of silicon, connected internally by a fabric that moves data at around 21 petabytes per second. Groq, meanwhile, has built chips called LPUs, or language processing units, specifically optimized for extremely fast inference on already-trained language models rather than for training new ones. Rather than relying on HBM at all, a Groq chip keeps everything in a few hundred megabytes of extremely fast on-chip SRAM, trading memory capacity for latency so that Groq can serve some models at several hundred tokens per second per user, well beyond what a single GPU can achieve.

This proliferation of specialized hardware reflects a broader trend. As machine learning workloads become large and predictable enough to justify the enormous cost of designing custom silicon, more

Accelerator	Maker	Chip type	Distinguishing idea
TPU (Trillium, v6e)	Google	Custom ASIC	Systolic array; deployed in large interconnected pods
Trainium2 / Inferentia2	Amazon	Custom ASIC	In-house training/inference chips built to cut AWS's reliance on NVIDIA
MI300X	AMD	GPU	192 GB HBM3 on a single chip, more capacity than an H100, aimed at memory-hungry inference
Wafer-Scale Engine 3	Cerebras	Wafer-scale ASIC	An entire silicon wafer used as one chip; ~900,000 cores, no off-chip HBM
LPU	Groq	Custom ASIC	All-SRAM design (no HBM) built purely for very low-latency inference

Table 1.3: A snapshot of AI accelerators beyond the NVIDIA GPU line

and more companies have decided it is worth building their own chips rather than renting someone else's. Even so, NVIDIA's advantage is not just about raw chip specifications; as we will see shortly, an enormous software ecosystem has been built specifically around NVIDIA hardware, and matching specifications is a very different challenge from matching a decade of accumulated software tooling.

Connecting GPUs Together

A single GPU, no matter how powerful, is not enough to train a modern large model. Training large models today requires hundreds or thousands of GPUs working together, which means those GPUs need to talk to each other constantly, exchanging gradients and activations as training proceeds. Within a single server, GPUs are typically connected by a high-speed interconnect such as NVIDIA's NVLink, paired with a switch called NVSwitch that lets every GPU in a server talk to every other GPU at full speed simultaneously, rather than being bottlenecked by shared cables. Across servers, GPUs are connected using specialized networking hardware such as InfiniBand, designed to move data between machines with very low delay and very high throughput.

These interconnects have themselves gone through generational leaps that mirror the GPU generations they accompany. The first generation of NVLink, introduced alongside the Pascal architecture in 2016, offered around 160 gigabytes per second of bandwidth per GPU. By Volta this had roughly doubled to 300 gigabytes per second, and it kept climbing with each subsequent generation: 600 gigabytes per second with Ampere, 900 gigabytes per second with Hopper, and 1.8 terabytes per second with Blackwell. The upcoming Rubin generation is expected to double this again, to 3.6 terabytes per second per GPU. This matters enormously, because a training run split across thousands of GPUs spends a significant fraction of its time not computing at all, but simply waiting for gradients to finish being exchanged across this network; a slow interconnect can silently waste an enormous

fraction of an otherwise state-of-the-art cluster's compute budget. NVIDIA's most extreme expression of this idea is the GB200 NVL72, a single rack that physically wires seventy-two Blackwell GPUs together through NVSwitch so that they behave, from the software's perspective, like one enormous GPU with a combined memory pool.

1.2.2 Software Stacks

A GPU sitting on a shelf cannot train a model any more than a stack of bricks can build a house on its own; both need a set of instructions and tools to turn raw material into something useful. Machine learning software is built as a stack of layers, each one hiding the complexity of the layer beneath it, so that a researcher can write a few lines of familiar-looking Python and have it eventually turn into millions of individual hardware instructions running correctly across thousands of GPUs. This stack is also a genuinely competitive and fast-moving part of the field in its own right: entire companies are built around making one layer of it faster, and the choice of software stack can matter almost as much as the choice of hardware. We now describe an example software stack that is popular today (Figure 1.6), before surveying some of the major alternatives at each layer.

Low-Level GPU Programming: CUDA

At the very bottom of the software stack sits CUDA [Nickolls et al., 2008], a parallel computing platform and programming model created by NVIDIA specifically to let users easily program their GPUs. Before CUDA existed, getting a GPU to do general-purpose computation, rather than just rendering graphics, required tricking it into doing math it was never designed for. CUDA changed that by giving developers a proper programming language, an extension of C++, along with a set of libraries for writing code that runs directly on the thousands of small cores inside a GPU. Very few machine learning practitioners write raw CUDA code themselves. Instead, they rely on libraries built on top of CUDA, most importantly cuDNN, a library of highly optimized implementations of frequently used deep learning operations, such as convolutions and matrix multiplications. CUDA's dominance is also, notably, a major reason NVIDIA has remained so far ahead of its hardware competitors: an enormous amount of the world's machine learning software has been written and tuned specifically using CUDA over the last decade, creating a software moat that is arguably as important as any single chip design.

High-level programming: PyTorch, TensorFlow

Sitting above CUDA is a high-level machine learning programming framework, and the most widely used one today is PyTorch [Paszke et al., 2019]. PyTorch lets a researcher define a neural network, describe how data should flow through it, and train it, all using ordinary-looking Python code, without ever thinking about tensor cores, memory bandwidth, or CUDA kernels directly. When a researcher writes an operation like a matrix multiplication in PyTorch, the framework translates that operation into the appropriate CUDA calls, dispatches it to the GPU, and hands the result back to the user after the GPU finishes its execution. PyTorch also handles automatic differentiation, which

involves keeping track of every operation performed on a tensor so that gradients needed to update a model's weights during training can be computed automatically rather than by hand. This single feature is arguably what made modern deep learning research practical at all; nobody wants to derive gradients for a hundred-layer network using pen-and-paper!

PyTorch is not the only framework of this kind. Google's JAX has become a serious competitor, especially in research settings and on TPU hardware, built around a slightly different philosophy: instead of PyTorch's imperative, run-as-you-go style, JAX encourages writing pure, mathematically clean functions and then transforming them automatically, for example turning a function that operates on a single example into one that operates on an entire batch, using a transformation called `vmap`. TensorFlow, an older framework also from Google, dominated the field before PyTorch became more popular with researchers in the late 2010s, and remains widely used in some production deployment settings today.

Compilers: Triton, XLA

Writing raw CUDA is powerful but painfully low-level, so an entire layer of the software stack has grown up around the idea of a machine learning compiler: a tool that takes higher-level code describing a computation and automatically translates, and heavily optimizes, it into fast low-level GPU instructions. OpenAI's Triton is a popular example, letting researchers write GPU kernels, which are small, self-contained pieces of code that run directly on the GPU, using a Python-like language. The Triton compiler then automatically handles the details of memory layout and scheduling that would otherwise require expert-level CUDA knowledge. Google's XLA, short for Accelerated Linear Algebra, plays a similar role but works at the level of an entire computation graph, aggressively fusing many small operations into fewer, larger ones to reduce the number of times data has to move in and out of memory, directly attacking the memory-bound bottleneck discussed earlier. More recently, PyTorch itself has grown a built-in compiler stack, centered on tools called TorchDynamo and TorchInductor and exposed to users through the single function call `torch.compile`, which can automatically speed up ordinary PyTorch code without the user writing any custom kernels at all. Underneath many of these tools sits MLIR, a compiler infrastructure project that provides a shared set of building blocks so that different machine learning compilers do not each have to reinvent the same optimizations from scratch.

The idea of kernel fusion, the central trick most of these compilers rely on, is easiest to see with a small example. Consider a simple sequence of operations that adds two tensors, multiplies the result by a scalar, and applies an activation function. Written naively, this is three separate GPU operations, and each one reads its input from HBM and writes its output back to HBM, so the data makes three full round trips to memory even though the computation itself is almost trivial. A compiler that performs kernel fusion instead recognizes that all three operations can be combined into a single kernel: the data is read from memory once, all three computations are performed on it while it sits in the GPU's tiny, extremely fast on-chip memory, and only the final result is written back. Because these operations are memory-bound rather than compute-bound, cutting the memory

traffic by a factor of three can produce a speedup close to that same factor, even though the total amount of arithmetic performed has not changed at all. This is the single biggest reason compiled PyTorch code, run through `torch.compile`, can often be reported as running thirty to fifty percent faster than the same code run naively, without the researcher ever optimizing a line of CUDA.

Inference engines: vLLM, SGLang

Training a model and serving a model are different problems with different bottlenecks, and this distinction becomes especially important once a model is deployed to answer real user queries. A naive way to serve a large language model is to simply run it through PyTorch, one request at a time. This works, but it wastes an enormous amount of GPU capacity, because generating text happens one token at a time, and a GPU sitting idle between tokens for a single user is a GPU not serving anyone else. vLLM [Kwon et al., 2023] is an open-source inference engine built specifically to solve this problem. It packs many different users' requests into GPU memory far more efficiently, using a technique called PagedAttention that manages GPU memory the way an operating system manages RAM, in fixed-size pages, to avoid wasting space. vLLM also uses continuous batching, where new requests are slotted in and completed requests are removed on the fly, rather than waiting for an entire batch of requests to finish together before starting the next one. The practical effect is that vLLM can serve significantly more queries per GPU than a naive implementation can, which is why it has become one of the most widely adopted serving engines for open-source large language models today, alongside competitors like NVIDIA's TensorRT-LLM and SGLang.

1.3 Book Outline

This book is organized as a journey through the machine learning stack, starting with the models themselves, moving down into the hardware that runs them, and then climbing back up through the layers of software and systems that make training and serving these models possible at scale. Below is a short preview of what each chapter covers.

Chapter 1: Introduction

The book opens with the story of how tools like ChatGPT reshaped the way people write code and find answers, and asks a bigger question: what actually makes such a system possible? The answer, we argue, rests on three pillars underneath every machine learning system: the model, the data, and the infrastructure. While models and data play an important role, the rest of the book is devoted primarily to the infrastructure pillar, the part that rarely gets discussed, but enables everything else to work at scale.

Chapter 2: Overview of Deep Learning Concepts

Before we can talk about the systems that run deep learning models, we need a working understanding of the models themselves, and specifically, why they are so demanding on the underlying infrastructure. This chapter walks through the major families of neural network architectures: simple fully connected networks, convolutional networks built for images, recurrent networks built for sequences, and Transformers, the architecture behind virtually every modern large language model today. For each one, we look at what problem it was designed to solve and where its particular hunger for compute, memory, or bandwidth comes from. This sets up the rest of the book, since every optimization and every piece of infrastructure we study later exists because of the resource demands introduced in this chapter.

Chapter 3: Programming AI Hardware

Having seen why deep learning models are so resource-hungry, we turn to the hardware built to satisfy that hunger. This chapter explains why ordinary processors are a poor fit for deep learning and how graphics processing units (GPUs), and other specialized chips, designed around massive parallelism and fast memory access, are better suited to run ML workloads. We look at how a GPU is organized internally, how thousands of lightweight threads run at once, and how multiple GPUs are connected together to work as a single larger machine. We then introduce CUDA, the programming framework used to actually write code for GPUs, and build an understanding of how a program written by a human eventually turns into thousands of coordinated threads running on real hardware.

Chapter 4: Hardware-Aware Performance Optimizations

With a working knowledge of the hardware in hand, this chapter asks a practical question: how do we actually get a GPU to run a model efficiently? We start by learning a simple but powerful way of

reasoning about performance, which tells us whether a given operation is limited by raw computation speed or by how quickly data can be moved. From there, we survey a toolbox of hardware-aware performance optimization techniques used throughout the industry to speed up real workloads, from batching and reusing data already sitting close to the compute units, to tiling large computations so they fit into fast memory. We also study FlashAttention, a widely used technique that applies these ideas to speed up the attention mechanism of Transformers. The chapter closes by looking at ways of changing the model itself to make it lighter and cheaper to run, including shrinking it and compressing precision using techniques like pruning and quantization.

Chapter 5: Machine Learning Programming Frameworks

Having looked at hardware and low-level optimizations largely in isolation, we next look at high-level programming frameworks like PyTorch and TensorFlow that enable users to write code at a higher level of abstraction. We look at how these frameworks represent a model as a computation graph, and how this graph is used to automatically compute gradients during training, without the user having to work out any calculus by hand. We also introduce ML compilation, the process of transforming this high-level graph into low-level code optimized for a specific piece of hardware. Together, these ideas explain how a few lines of readable Python code end up running as highly efficient, hardware-specific instructions.

Chapter 6: Distributed Training

A single GPU, no matter how well optimized, eventually runs out of room for today's largest models, so this chapter looks at how the work of training is spread across many devices at once. We introduce two foundational strategies: data parallelism, where every device holds a full copy of the model and works through a different slice of the training data, and model parallelism, where the model itself is split apart so that no single device needs to hold it all. The chapter also touches on several more specialized strategies developed for today's largest models, and closes with real-world case studies showing how these techniques come together in production systems used to train large language models.

Chapter 7: Network Optimizations

Distributed training only works if the GPUs involved can actually communicate with each other efficiently, so this chapter looks at the networking layer underneath the distributed system of the previous chapter. We look at why ML training traffic is particularly demanding, arriving in short, bursty spikes rather than as a smooth steady stream, and how the physical layout, or topology of a cluster's network is designed to handle this pattern. We take a closer look at the communication libraries such as NVIDIA's NCCL, along with the low-level networking technology such as RDMA, that let GPUs on different machines exchange data without routing through the CPU.

Chapter 8: LLM Inference Optimizations

While the previous two chapters focus on training large models, this chapter turns to the equally important problem of serving an already-trained model to real users. Inference is a different kind of challenge from training: it is real-time, and a serving system must respond quickly to individual requests while still keeping the underlying hardware busy and efficient across many requests at once, two goals that often pull in opposite directions. We begin by revisiting how a model generates text one token at a time, and the two distinct phases involved: an initial phase that processes the user's prompt, and a second phase that generates output tokens one by one. We then study the key techniques that make inference efficient in practice, including caching previously computed results, batching and scheduling multiple requests together, managing memory carefully, offloading work when a device runs low on resources, and splitting inference itself across multiple devices.

1.4 How to Use the Book?

A systems book is a different kind of reading experience from a math textbook or a novel, and it helps to know what you are getting into before you start. This section is a short guide to the moving parts of this book: the different kinds of content you will encounter in each chapter, the resources that live outside the book, and a few honest suggestions to instructors and students on how to get the most out of all these resources.

Everything described in this section, the chapters themselves, the accompanying slides, and the programming assignments, are all hosted together on the companion website for this book, at https://www.cse.iitb.ac.in/~mythili/sysml/. Each chapter of the book is linked there as a standalone PDF alongside a matching slide deck and, where applicable, a link to the corresponding programming assignments in the GitHub repository. Treat the website as the gateway to all of this book's resources.

1.4.1 Reading the Chapters

The chapters of this book are not meant to be taught or read as one long, uninterrupted chain. The first three chapters form a spine, Chapter 2 builds the deep learning background needed to understand why models are so demanding in the first place, and Chapter 3 builds on that background to explain the hardware and programming model (CUDA), used to actually meet those demands. Everything else in the book depends on these two chapters, so they should be read in order, at the start.

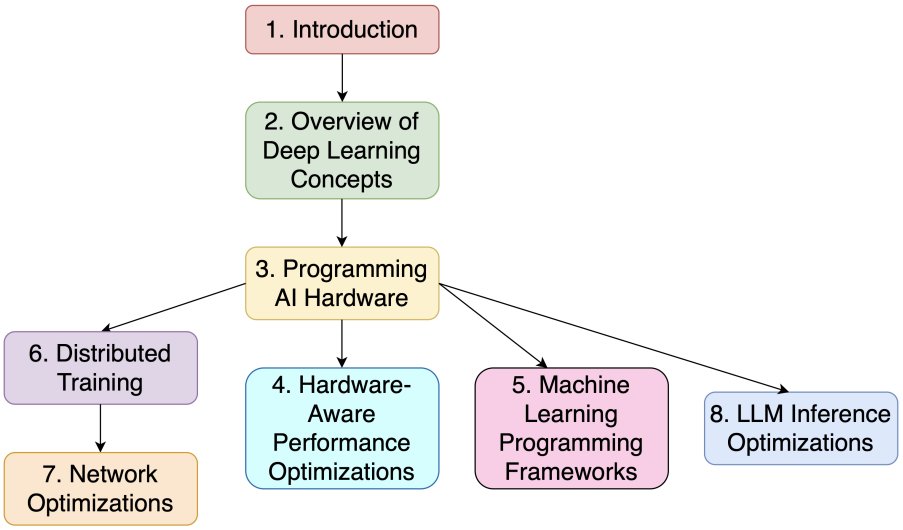


Figure 1.7: Chapter organization

From Chapter 3 onward, the book branches, as shown in Figure 1.7. Chapter 4 goes deeper

into hardware-aware optimizations that squeeze more performance out of the GPU programming model introduced in Chapter 3. Chapter 5 steps back up a level of abstraction to look at high-level programming frameworks such as PyTorch that most practitioners actually use. Chapter 6 moves in yet another direction, addressing what happens when a single GPU is no longer enough and training must be spread across many devices, and this in turn leads into Chapter 7, which looks at the networking layer that distributed training depends on. Chapter 8, on LLM inference, draws directly on the hardware and programming foundations of Chapter 3 as well, but asks a different question of them, one about serving already-trained models rather than training new ones.

What this means in practice is that once you have read Chapters 1 through 3, you do not need to go through the rest of the book in numerical order. A reader mainly interested in serving systems could move from Chapter 3 straight to Chapter 8; a reader mainly interested in training large models could move from Chapter 3 into Chapters 6 and 7 and treat Chapters 4 and 5 as optional ones to return to later. So, feel free to let your own interests, or the syllabus of the course you are using this book for, decide the order in which you navigate the chapters.

We have also tried to be honest about the pace of the field. Systems for machine learning is one of the fastest-moving corners of computer science, and some of the specific numbers, models, and tools mentioned in this book will be out of date within a couple of years of publication. What should not go out of date are the underlying principles, e.g., why a particular optimization exists, what trade-off it is making, and what problem it was designed to solve. As you read, try to keep these two layers separate in your head: the names of specific frameworks or GPU chipsets will change, but the reasons they were built the way they were will not. This is also why, throughout the book, we try to motivate a technique by the problem it was built to solve before describing the technique itself. Systems knowledge resists being memorized and rewards being rebuilt, so the trade-off a technique makes should stay with you longer than the specific trick it uses. It is also worth reading this book with a healthy sense of scale. Systems for machine learning is full of numbers, e.g., the size of a model, the bandwidth of an interconnect, the latency of a memory access, and it helps to remember that the specific values matter less than the ability to compare them meaningfully.

Another useful habit while reading this book is to look for recurring patterns of thought that keep cropping up at several places. For example, several times in this book, we ask the question of which resource a given operation is actually limited by: is it waiting on computation, on memory, or on communication? This single question, asked repeatedly, leads to several of the ideas in this book. Look out for such fundamental ways of thinking and use them to build your intuition about machine learning systems as well. In the end, we believe that a lot of good ideas in computer systems come from common sense honed on technical knowledge and experience in a particular domain, which is machine learning in this case.

Within a chapter, keep an eye out for a few types of content. Algorithm and code boxes appear whenever an idea is best expressed as a short, precise sequence of steps, and these are worth reading slowly, line by line, rather than just skimming through. Examples are shown below. Figures in a systems book often carry information that would take a full page of text to express, so it is worth

pausing at each one and making sure you can explain it back in your own words before moving on. Footnotes[2] are used at times, to highlight or describe interesting concepts of the book.

Code 1.1: Sample Code Box

```
print("Code snippets are boring!")
```

Algorithm 1.1: Sample Algorithm Box

Require: A functioning brain, $\mathcal{B}$.
1: Read the book. If necessary, read it again.
2: Load relevant concepts into $\mathcal{B}$ from the textbook.
3: Compute Understanding = $f(\text{concepts}, \text{coffee})$.
4: If Understanding < threshold, go back to Step 1.
5: **return** Understanding

We would like to share a few unsolicited, but hopefully helpful, tips as you read the book. Work through the book in short, deliberate chunks, a section or two at a time, rather than pushing through a whole chapter in one sitting. Before you start a section, spend thirty seconds previewing it, to give your brain a scaffold to hang the details onto as you go. For example, before reading the Flash Attention section in full, skimming its headings alone (i.e., Standard Attention, The Softmax Obstacle, The FlashAttention Algorithm) already tells you the shape of the argument: an obstacle is identified and then resolved. Once you finish a full chapter, it is also worth pausing to build a short outline or concept map, and to connect the ideas in a chapter with those in other chapters you have read. Finally, the references at the end of the chapter provide good additional reading to instructors and students who wish to dive deeper.

1.4.2 Practice Problems

Most of the sections end with a set of practice problems. These range from short conceptual questions that check whether you have understood a concept, to longer problems that ask you to work through a calculation, such as estimating the memory footprint of a model or the communication cost of a distributed training strategy. We expect the practice problems to be useful for instructors as ready-made in-class exercises, and for students as practice questions for exams.

We recommend attempting these problems with pen and paper before looking at any solutions, since the act of working through the numbers yourself is where most of the actual learning in a systems course happens. Systems concepts have a way of feeling obvious once you have read the

[2] You will realize that most of the times, they are used for cracking lame humour!

explanation, and it is very easy to convince yourself you understood something even when you have not. Practice problems are the check against that.

A few habits will save you time and catch mistakes when you actually sit down to solve one of these problems. Before computing anything, restate the problem in your own words and pin down exactly what is given and what is asked. Wherever possible, solve the problem symbolically first, keeping quantities like batch size, bandwidth, and model dimensions as variables rather than plugging in numbers immediately. This makes it easier to verify the correctness of the answer, as the units should work out by construction. Finally, once you do have a number, ask whether it is plausible at all. Comparing your answer to a number you already trust catches far more errors than re-checking your calculations does.

If you get stuck on a practice problem, the first thing to do is go back to the relevant section and reread it with the specific problem in mind. Most of these problems are designed to be answerable directly from the chapter's content. Solutions to all the problems are made available underneath the problems, but please look over the solutions only after you have made a honest attempt yourself.

1.4.3 Programming Assignments

Alongside the conceptual material, this book comes with a series of programming assignments that let you build the systems you are reading about, rather than only reading about them. These live in a companion GitHub repository at `https://github.com/mythilivutukuru/SysMLBook`, and are linked from the end of corresponding sections, so you can work through an assignment right after finishing the section. The same assignments are also linked chapter-by-chapter from the companion website, alongside the chapter PDF and slides. The programming assignments are self-contained enough to be assigned as exercises by instructors without any additional instructions.

Early assignments in the book are in Python, and have you implementing pieces of a multi-layer perceptron or a sliding-window attention mechanism. From Chapter 3 onward, a large fraction of the assignments are written in CUDA, starting from a simple vector addition kernel, and building up through matrix multiplication, shared memory tiling, and attention kernels, mirroring the order in which these ideas are introduced in the text. Later assignments move to advanced topics like distributed training paradigms or LLM inference engine architecture.

Each assignment folder in the repository is self-contained and follows the same structure: a `README.md` file with the problem statement, background, and hints; a starter file, typically named `student.py`, `student.cu`, or `student.cpp`, with `TODO` markers showing exactly where your code needs to go; and, for select assignments, a complete `solution` file you can check your work against once you are done. The problem descriptions and implementation hints live directly in these files and in the accompanying `README`. We recommend cloning the repository once, at the start of the course, and pulling it periodically, since new assignments continue to be added over time.

We also strongly encourage you to write the code for these assignments yourself, without lean-

ing on AI coding assistants to generate the implementation.[3] The entire point of tiling a matrix multiplication is the intuition you build while wrestling with the details like tensor dimensions, or debugging tricky race conditions, and that intuition does not transfer from reading a correct solution someone else produced. To get started with an assignment, clone the repository, move into the folder for the chapter and topic you are working on, and read that folder's `README.md` before writing any code. From there, fill in the `TODO` sections in the starter file, and follow the evaluation instructions in the `README` to check whether your implementation is correct.

1.4.4 Slides and the Companion Website

Each chapter of the book is paired with a slide deck, linked from the companion website alongside the chapter PDF, and these two forms of the same material are meant to be used together rather than as substitutes for one another. The slides distil a chapter down to its key figures, definitions, and formulas, and are primarily meant to be used in a lecture by an instructor, or by a student to review the concepts learnt from reading the book. The most effective way to use the slides is along with the reading of the textbook, rather than instead of it.

Programming Assignment 1.1: Kaggle Environment Setup

In this assignment, you will set up and familiarize yourself with the Kaggle Notebooks environment, which you will use for programming assignments throughout the textbook. You need to complete the following steps: create a Kaggle account and verify your phone number (required to unlock GPU/TPU access), create a new notebook and explore the available session options (accelerator, language, persistence, and internet settings), and run basic Python and shell commands to confirm the environment is working correctly. You will also enable a GPU accelerator, verify it is active using `nvidia-smi`, and confirm that PyTorch can detect and use the GPU via `torch.cuda.is_available()`. Finally, you will explore the Kaggle container's directory structure and practice writing, compiling, and running a simple C program. A detailed walkthrough of each step is available at:

https://github.com/mythilivutukuru/SysMLBook/tree/main/kaggle_setup_guide

Programming Assignment 1.2: PyTorch Fundamentals

In this assignment, you will get hands-on practice with the core PyTorch functions you will rely on throughout the rest of the textbook's programming assignments: creating tensors, inspecting and converting their properties, indexing and reshaping them, combining and splitting them, performing elementwise and matrix math, computing reductions, using broadcasting and comparisons, applying in-place operations, computing gradients with autograd, managing devices (CPU vs. GPU), and ensuring reproducibility

[3] We are fully aware of the irony of a machine-learning systems textbook telling you to put down the AI tools.

with manual seeds. You will complete this assignment on Kaggle Notebooks, where `torch`, `numpy`, and `pytest` are preinstalled. Open `student.py` and implement each function according to its docstring, using the accompanying `# TODO` hint to identify which `torch` call is expected; later sections build on the tensor basics introduced in earlier ones. You will verify your implementation by running the provided test suite with `pytest`.

https://github.com/mythilivutukuru/SysMLBook/tree/main/pytorch_fundamentals

References

einpoklum (2026). *How does the continued decline in posts since May '25 influence our interpretation of the state of SO?* Meta Stack Overflow. URL: https://meta.stackoverflow.com/questions/437921/how-does-the-continued-decline-in-posts-since-may-25-influence-our-interpretati.

H, Yasser (n.d.). *AlexNet*. Kaggle. Kaggle notebook. URL: https://www.kaggle.com/code/yasserh/alexnet.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet classification with deep convolutional neural networks". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica (2023). "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. New York, NY, USA. URL: https://doi.org/10.1145/3600006.3613165.

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron (2008). "Scalable parallel programming with CUDA". In: *ACM SIGGRAPH 2008 Classes*. URL: https://doi.org/10.1145/1401132.1401152.

Paszke, Adam, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala (2019). "PyTorch: an imperative style, high-performance deep learning library". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*.

Po, LM (2024). *A Brief History of AI with Deep Learning*. Medium. URL: https://medium.com/@lmpo/a-brief-history-of-ai-with-deep-learning-26f7948bc87b.