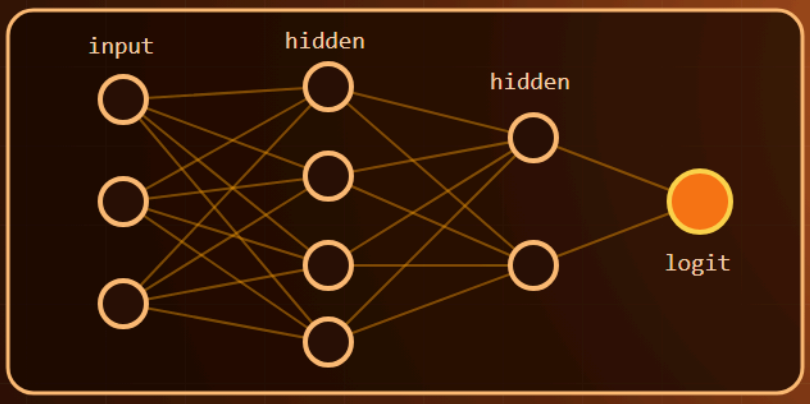
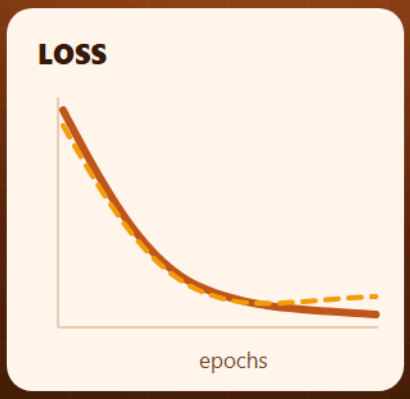





```python
for xb, yb in train_loader:
  optimizer.zero_grad()
  logits = model(xb)
  loss = loss_fn(logits, yb)
  loss.backward()
  optimizer.step()
```




```
(64, 1, 28, 28)   → conv →   (64, 32, 7, 7)   → flatten →   (64, 1568)
```

A HANDS-ON GUIDE

Deep Learning with PyTorch

Tensors, training loops, CNNs, sequence and transformer models, and deploying models

Autograd · nn.Module · DataLoaders · CNNs · Transfer learning
LSTMs · Attention · Transformers · Export · Serving

TENSORS · VISION · SEQUENCES · TRANSFORMERS

THENEXBOOKS

Deep Learning with PyTorch

Tensors, Training Loops, CNNs, Sequence and Transformer Models, and Deploying Models
A hands-on guide — every example runs on an ordinary CPU

First Edition

THENEXBOOKS

Deep Learning with PyTorch
Tensors, Training Loops, CNNs, Sequence and Transformer Models, and Deploying Models

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. PyTorch, torchvision, and the wider Python ecosystem change quickly; function names, defaults, warnings, and installation commands can differ between versions. Training results such as losses and accuracies vary across hardware, operating systems, and library versions even with fixed random seeds. Always consult the official documentation for the versions you use.

PyTorch is a trademark of The Linux Foundation. Python is a trademark of the Python Software Foundation. NVIDIA and CUDA are trademarks of NVIDIA Corporation. Apple and Apple Silicon are trademarks of Apple Inc. Windows is a trademark of the Microsoft group of companies. Other product and company names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the PyTorch Foundation, The Linux Foundation, or any other company mentioned.

Larkspur Goods, the retailer used as the running example in this book, is fictional, as are its stores, staff, products, customer reviews, sales figures, and product images. All datasets except the handwritten digits are synthetic and generated by a script printed in Appendix B; the digit images are the small teaching dataset bundled with scikit-learn. No result in this book describes real customers or real sales.

Preface

Deep learning can feel like two different subjects. One is a world of impressive demos — models that recognize images, write text, and forecast the future. The other is a pile of code that refuses to train: shapes that do not match, a loss that stays flat, an accuracy that looks too good to be true. Most people who start learning deep learning get stuck somewhere between the two, because the demos hide the mechanics and the mechanics are rarely explained in order.

This book closes that gap. We start with the smallest building block — a tensor — and work upward one idea at a time: gradients, a hand-written gradient descent loop, neural network modules, data loaders, a complete training loop, convolutional networks for images, recurrent networks for text and time series, attention, transformers, and finally the work that turns a trained model into something people can use: checkpoints, export, quantization, and a prediction service. Every concept arrives at the moment it solves a real problem, and every piece of code is small enough to read in one sitting.

All examples were run while writing this book, on an ordinary CPU, and the outputs you see are real. No example needs a GPU, a cloud account, or a large download: the datasets are small, generated by a short script, or bundled with scikit-learn, so each training run finishes in seconds to about a minute. We explain GPUs carefully and write device-independent code, so the same scripts run faster on a CUDA or Apple Silicon machine when you have one.

Who This Book Is For

- **Python developers** who want to understand deep learning from the inside instead of copying notebooks they cannot debug.
- **Data analysts and data scientists** who know classic machine learning and want to add neural networks for images, text, and sequences.
- **Students and self-learners** who want a clear, practical path through tensors, autograd, CNNs, RNNs, and transformers, with math kept short and always explained in words.
- **Engineers** who need to take a PyTorch model into production: saving, exporting, optimizing, and serving it.

You should be comfortable with basic Python: functions, classes, lists and dictionaries, and running a script from a terminal. Some NumPy experience helps but is not required. You do not need prior machine learning courses or advanced mathematics; when we need an idea such as a derivative or a train/test split, we explain it.

What You Will Learn

- How tensors work — shapes, broadcasting, memory, devices — and how autograd computes gradients for you.

- How to build models with `nn.Module`, feed them with `Dataset` and `DataLoader`, and write a training loop you fully understand.
- How to make training work: initialization, normalization, regularization, learning-rate schedules, and a debugging checklist.
- How convolutional networks see images, how data augmentation makes them robust, how residual networks work, and how transfer learning reuses what a model has learned.
- How to turn text into tensors, classify reviews with embeddings and LSTMs, and forecast store demand with sequence models — always compared against honest baselines.
- How attention and transformers work, built from scratch and then with PyTorch's modules, including a tiny language model that generates text.
- How to make results reproducible, save checkpoints, track experiments, profile and speed up code, export models with `torch.export`, quantize them, and serve predictions through an API.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces deep learning, installs PyTorch, and covers tensors, autograd, and gradient descent written by hand. **Part II — Training Neural Networks** (Chapters 6–10) builds models with `nn.Module`, loads data, writes the training loop, trains a first network on handwritten digits, and shows how to make training work reliably. **Part III — Computer Vision with CNNs** (Chapters 11–15) covers convolutions, a CNN for product thumbnails, data augmentation, modern architectures, and transfer learning. **Part IV — Sequences and Text** (Chapters 16–19) covers tokenization and embeddings, recurrent networks for review sentiment, demand forecasting, and attention. **Part V — Transformer Models** (Chapters 20–23) builds the transformer piece by piece, trains a transformer classifier and a tiny text generator, and explains pretraining and fine-tuning. **Part VI — Deploying Models** (Chapters 24–28) covers reproducibility and checkpoints, performance, export and quantization, serving predictions, and a complete end-to-end capstone. The appendices provide a PyTorch cheat sheet, templates and reference code, a production checklist, and a glossary.

Each chapter ends with **Try It Yourself** exercises and a **Key Takeaways** box for quick review.

The Running Example: Larkspur Goods

Throughout the book we join the small data team at **Larkspur Goods**, a fictional home-and-lifestyle retailer with three stores — Riverside, Harbor, and Uptown — and an online shop. Mara leads the team and asks the business questions, Dev is the machine learning engineer who has to run everything in production, and Ines is a junior analyst learning deep learning alongside you. Together they build three systems: a **thumbnail classifier** that tags catalog images as mugs, bottles, books, plates, t-shirts, or lamps; a **review sentiment model** that flags unhappy customers for the care team; and a **demand forecaster** that predicts the next seven days of sales for each store. As a warm-up, they teach a network to read handwritten digits from paper stock-count sheets.

The datasets are deliberately small and fully reproducible. One script, printed in Appendix B, generates the product thumbnails, 400 customer reviews, and two years of daily store sales; the digits come with scikit-learn. Because every reader has exactly the same data, you can compare your results with ours — and because the data is small, you can train every model on a laptop.

Versions and Conventions

Examples were tested with **Python 3.12**, **PyTorch 2.14**, and **torchvision 0.29** on a CPU-only Windows machine, together with recent versions of NumPy, pandas, scikit-learn, and matplotlib. Almost everything also works with other recent PyTorch 2.x releases, on macOS and Linux, and on GPUs. Where a feature is new, deprecated, or behaves differently between versions, a Note says so. Code that needs something we did not run while writing — a GPU, an internet download, or a library that was not installed — is clearly labeled "not run in this book's tests".

Training involves randomness, and floating-point results depend on hardware and library versions. We fix random seeds so that results are repeatable on one machine, but your numbers may differ slightly from ours. When you see "on our machine we got 0.9617", expect something close to it rather than the identical digits.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause wrong results, wasted time, or security problems.

Code, file names, and commands appear in `monospace`. Each script has a caption with its file name, such as `ch12_cnn.py`; terminal commands appear under a **Terminal** caption and program output under an **Output** caption. Long lines are wrapped to fit the page. The book's shared helper modules, `larkspur.py` and `models.py`, grow chapter by chapter; their final versions are printed in Appendix B.

Contents

Preface

PART I — FOUNDATIONS

1 What Deep Learning Is and Where PyTorch Fits

2 Setting Up PyTorch: Installation, Devices, and Your First Tensors

3 Tensors in Depth: Shapes, Indexing, Broadcasting, and Memory

4 Autograd: How PyTorch Computes Gradients

5 Gradient Descent by Hand: Fitting Larkspur's Demand Trend

PART II — TRAINING NEURAL NETWORKS

6 Building Models with nn.Module

7 Datasets and DataLoaders: Feeding Data to Models

8 The Training Loop: Loss Functions, Optimizers, and Evaluation

9 Your First Neural Network: Reading Count-Sheet Digits

10 Making Training Work: Initialization, Normalization, Regularization, and Schedules

PART III — COMPUTER VISION WITH CNNS

11 Convolutions from First Principles

12 Building a CNN: Classifying Product Thumbnails

13 Data Augmentation with torchvision Transforms

14 Modern CNN Architectures: Batch Norm, Residual Connections, and torchvision Models

15 Transfer Learning and Fine-Tuning

PART IV — SEQUENCES AND TEXT

16 Text as Tensors: Tokenization, Vocabularies, and Embeddings

17 Recurrent Networks: RNN, LSTM, and GRU for Review Sentiment

18 Forecasting Store Demand with Sequence Models

19 Attention from Scratch

PART V — TRANSFORMER MODELS

20 Inside the Transformer: Multi-Head Attention, Positions, and Encoder Blocks

21 Building a Transformer Classifier for Reviews

22 Generating Text: A Tiny Decoder-Only Language Model

23 Pretraining, Fine-Tuning, and the Pretrained-Model Ecosystem

PART VI — DEPLOYING MODELS

24 Reproducibility, Checkpoints, and Experiment Tracking

25 Performance: GPUs, Mixed Precision, torch.compile, and Profiling

26 Exporting and Optimizing Models: torch.export, ONNX, and Quantization

27 Serving Predictions: Batch Scoring and a Prediction API

28 Capstone: Larkspur's Deep Learning System End to End — and What to Learn Next

APPENDICES

A PyTorch Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What deep learning is and where PyTorch fits, setting up a working environment, tensors from the ground up, how autograd computes gradients, and gradient descent written by hand to fit Larkspur's demand trend.

1 · What Deep Learning Is

2 · Setting Up PyTorch

3 · Tensors in Depth

4 · Autograd

5 · Gradient Descent by Hand

CHAPTER 1 · PART I — FOUNDATIONS

What Deep Learning Is and Where PyTorch Fits

Before we install anything, we build a clear mental picture: what a neural network actually is, how it learns from examples, when deep learning is the right tool and when it is not, and why PyTorch is a comfortable place to learn it. We also meet Larkspur Goods, the fictional retailer whose problems we solve throughout the book.

A Program That Writes Part of Itself

Most programs you have written follow rules that you typed in. If a customer's basket is over a certain amount, apply free shipping. If a date falls on a weekend, skip it. You decide the logic, the computer follows it, and when the program is wrong you read the code and fix the rule.

Some problems resist this approach. Try writing rules that decide whether a small grayscale picture shows a mug or a lamp. You might start with "a mug has a handle on one side", then discover that some mugs are photographed from the front and the handle is hidden, that some lamps have a curved arm that looks like a handle, and that a slightly darker photo breaks every brightness threshold you chose. The same thing happens with text. "The vase is lovely" is positive, "the vase is not lovely" is negative, and "delivery was slow, but the vase is lovely" is mostly positive. Each rule you add creates two new exceptions.

Machine learning flips the arrangement. Instead of writing the rule, we write a flexible function with many adjustable numbers in it, show it thousands of examples together with the answers we want, and let an automatic procedure adjust those numbers until the function's answers match ours on the examples. The program we write is the *learning procedure*; the rule itself is discovered from data and lives in the adjusted numbers.

Deep learning is the part of machine learning in which that flexible function is a **neural network**: a stack of simple layers, each of which transforms its input a little, with the output of one layer feeding the next. "Deep" simply means there are several such layers rather than one. Each layer is easy to understand on its own. The power comes from stacking them and from tuning all of their numbers together.

Layers of learned transformations

Here is the idea in its most concrete form. A grayscale thumbnail of 28 by 28 pixels is a grid of 784 brightness values. A first layer might combine those 784 values into 128 new numbers, where each new number is a weighted sum of all the pixels plus an offset, passed through a simple bend (a

function that turns negative values into zero, for example). A second layer combines the 128 numbers into 64. A final layer combines the 64 numbers into six scores, one per product class. The class with the highest score is the prediction.

Nothing in that description mentions handles, shades, or shapes. The weights start as random numbers, so the first predictions are nonsense. Training nudges every weight in the direction that makes the scores for the correct class a little higher and the others a little lower, and repeats that nudge many thousands of times. After training, if we inspect the first layer, we often find that some of its weighted sums respond to edges or bright blobs in certain positions. Nobody programmed those detectors; they emerged because they were useful. This is what people mean when they say deep networks *learn features*.

Mathematically, a neural network is a **function approximator**. With enough layers and units it can represent a very wide range of input-to-output relationships, which is both the promise and the danger: a function flexible enough to fit the truth can also fit the noise. Much of this book is about steering that flexibility toward patterns that hold up on new data.

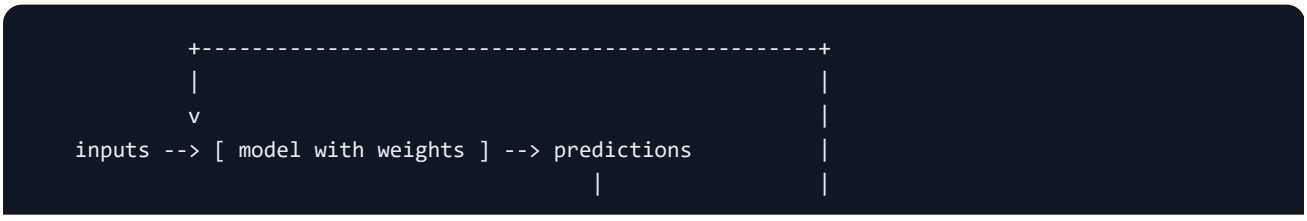
How deep learning relates to machine learning and AI

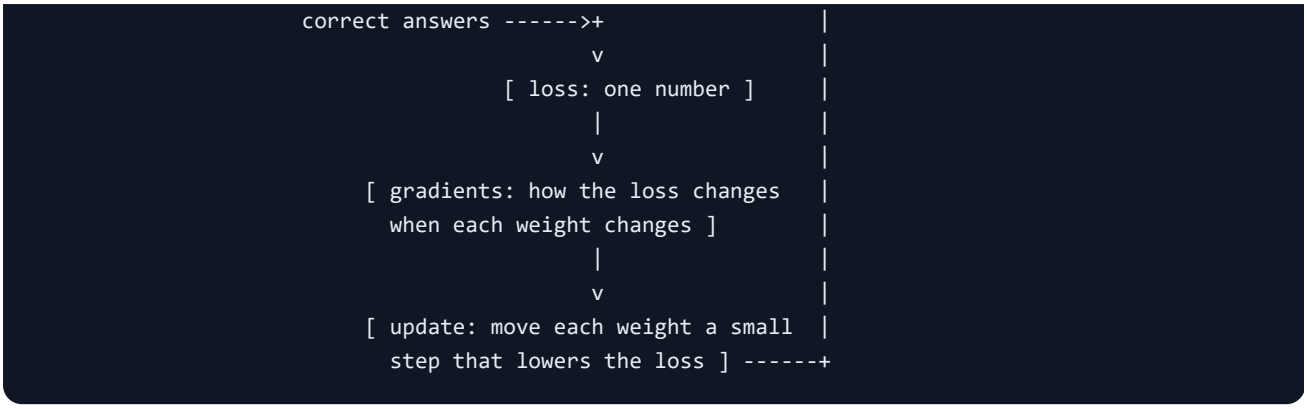
Artificial intelligence is the broad goal of making computers do tasks that we associate with intelligence, by any method, including hand-written rules and search. **Machine learning** is the subset of methods that learn from data rather than from rules typed in by a programmer. Linear regression, decision trees, gradient-boosted trees, and nearest-neighbor methods are all machine learning. **Deep learning** is the subset of machine learning that uses neural networks with several layers, trained with gradients. The large language models behind modern chat assistants are deep learning models, but so is a three-layer network that reads handwritten digits on a laptop in half a second. We will build the second kind many times and understand the first kind by the end of Part V.

> **NOTE**
>
> Throughout the book we explain basic machine-learning ideas such as training and test sets, overfitting, and accuracy the first time we use them. You do not need a prior machine-learning course; you do need to be comfortable writing Python functions and classes and running scripts from a terminal.

The Training Idea in One Picture

Every model in this book, from a straight line in Chapter 5 to a small language model in Chapter 22, is trained by the same cycle.





Let us walk through the boxes in plain words.

1. **Model and prediction.** We pass a batch of inputs, such as 64 thumbnails, through the network and get predictions, such as six scores per thumbnail. This step is called the **forward pass**.
2. **Loss.** We compare the predictions with the correct answers and condense the comparison into a single number called the **loss**. A small loss means the predictions are close to the answers. For a forecast, the loss might be the average squared difference between predicted and actual units sold. For a classifier, it is a number that is small when the model gives a high score to the right class.
3. **Gradients.** For every weight in the model, we compute how much the loss would change if that weight were nudged slightly up. This collection of sensitivities is the **gradient**. Computing it by hand for a model with a hundred thousand weights would be hopeless, and this is exactly where PyTorch's automatic differentiation, called **autograd**, does the work for us. This step is called the **backward pass**.
4. **Update.** We move each weight a small step in the direction that lowers the loss. The size of the step is controlled by the **learning rate**. The object that applies the update is the **optimizer**.

Then we take the next batch and repeat. One pass over the whole training set is an **epoch**; each individual update is a **step**. A typical small model in this book trains for a few dozen epochs, which means thousands of steps, and finishes in seconds on an ordinary CPU.

Imagine standing on a foggy hillside, wanting to reach the valley. You cannot see it, but you can feel which way the ground slopes, step downhill, and feel again. The loss is the height, the weights are your position, and the gradient is the slope. The procedure is called **gradient descent**, and Chapter 5 implements it by hand before we let PyTorch's optimizers do it for us.

Training, validation, and test data

A model that only memorizes its examples is useless; we care about how it performs on new inputs. So we always hold some labeled data back. The **training set** is used to adjust the weights. The **test set** is kept aside and used only to measure the final result. Often we also carve a **validation set** out of the training data to make decisions during development, such as when to stop training or which learning rate to use, without peeking at the test set. When a model scores much better on training data than on held-out data, it is **overfitting**: it has learned details of the specific examples rather than the general pattern. We will watch for that sign in every chapter.

When Deep Learning Shines, and When It Does Not

Deep learning is a powerful tool, not a universal one. Knowing when to reach for it saves time and money.

It tends to shine when the input is **raw and high-dimensional** and the useful features are hard to describe by hand. Images, audio, natural-language text, and long sequences of measurements fall into this group. A photo has thousands of pixels, none of which means much alone, and the useful information lies in patterns across many of them. Networks with the right structure, such as convolutional layers for images (Part III) or attention layers for text (Part V), are very good at discovering those patterns, especially when there are many labeled examples or a large amount of unlabeled data to pretrain on.

It is often *not* the best choice in these situations:

- **Small tabular data.** If your data is a spreadsheet with a few thousand rows and a few dozen meaningful columns, such as customer age, region, and order count, gradient-boosted trees and other classical methods are frequently as accurate or more accurate, train faster, and are easier to explain. Try them first.
- **Problems a simple rule solves.** If "flag any review with a one-star rating" already does what the business needs, a neural network adds cost without benefit.
- **Too little labeled data and no related data to borrow from.** A flexible model trained on a hundred examples usually memorizes them. Chapter 15 shows how transfer learning can help, and also how far it falls short with only 60 images.
- **Decisions that must be fully explained.** A network's reasoning is spread across thousands of weights; a simpler model may be the responsible choice.

A good habit, which we follow in every project in this book, is to build a simple **baseline** first, such as "predict the same weekday last week" for demand, and only keep a deep model if it clearly beats that baseline on held-out data. You will see that sometimes it does by a wide margin, and sometimes only barely. We report both honestly.

> **TIP**
> Before training any network, write down in one sentence what decision its output will support and who acts on it. "Flag likely negative reviews so the customer-care team reads them first" is a good sentence. It tells you what errors cost, which metric matters, and whether a simple rule might already be enough.

Why PyTorch

PyTorch is an open-source library for tensor computation and deep learning in Python. There are other good frameworks, and the concepts in this book carry over to them. We chose PyTorch for teaching for a few concrete reasons.

It feels like ordinary Python

A PyTorch model is a Python class. Its `forward` method is ordinary Python code that you can step through in a debugger or add `print` statements to, and a dataset is a class with `__len__` and `__getitem__` methods.

Eager execution

When you write `c = a @ b` in PyTorch, the multiplication runs immediately and `c` holds the result. This style is called **eager execution**. You can inspect every intermediate value, and errors point at the failing line. When you need speed or deployment, `torch.compile` and `torch.export` (Chapters 25 and 26) capture the model into a graph after the fact.

Autograd

PyTorch records the operations you perform on tensors that need gradients and can replay them backward to compute all gradients with one call, `loss.backward()`. Because the record is built as the code runs, a model can use loops and conditions freely; the graph is simply whatever actually happened on this run. Chapter 4 explores autograd in detail.

An ecosystem around the core

We use the companion package **torchvision** for image transforms and standard vision architectures. The wider community publishes many pretrained models and libraries built on PyTorch; we describe them in Chapter 23 but every example runs with just PyTorch, torchvision, NumPy, pandas, scikit-learn, and matplotlib.

> **NOTE**
>
> The examples in this book were tested with PyTorch 2.x on a CPU; specifically, PyTorch 2.14 with torchvision 0.29 and Python 3.12. Most of the code runs unchanged on other recent versions. Where an API has changed recently, such as the safer default for `torch.load`, we point it out where it matters.

CPUs, GPUs, and What You Need

Training a neural network is mostly arithmetic on large arrays of numbers: multiplying matrices, adding vectors, applying a simple function to every element. A **CPU**, the general-purpose processor in every computer, has a modest number of powerful cores and is good at a wide variety of tasks. A **GPU**, originally designed to draw graphics, has a very large number of simpler cores that excel at doing the same operation on many numbers at once. For big models and big datasets, that parallelism makes training dramatically faster, which is why large-scale deep learning runs on GPUs and similar accelerators.

PyTorch represents where a tensor lives as a **device**. The common ones are `cpu`, `cuda` (NVIDIA GPUs), and `mps` (the GPU in Apple Silicon Macs). The same model code can run on any of them if you move the model and data to the chosen device. In Chapter 2 we write a small helper, `get_device()`, that picks a GPU when one is available and falls back to the CPU otherwise, and we use it everywhere.

You do **not** need a GPU for this book. Every example was designed to run on an ordinary CPU in well under a minute, most of them in a few seconds. The datasets are deliberately small and the models are deliberately compact so that you can run, change, and rerun everything on a laptop. The ideas and habits are the same at every scale. GPU-specific snippets, which appear mainly in Chapter 25, are clearly labeled as not run in this book's tests.

> **WARNING**
>
> Training numbers such as losses and accuracies can differ slightly from machine to machine, even with the same code and the same random seed, because different CPUs, thread counts, GPUs, and library versions perform floating-point arithmetic in slightly different orders. Whenever we show a training result we say "on our machine we got" and you should expect something close, not identical. Shapes, parameter counts, and dataset sizes, on the other hand, are exact and should match yours precisely.

Meet Larkspur Goods

Abstract examples are easy to forget, so the whole book follows one fictional company. **Larkspur Goods** is a home-and-lifestyle retailer with three physical stores, **S01 Riverside**, **S02 Harbor**, and **S03 Uptown**, and an online shop selling kitchenware, bedding, lighting, decor, and outdoor goods. Like many mid-sized retailers, it has more data than it uses and a small team trying to change that.

Three people on that team will appear from time to time to ask the questions real projects raise:

- **Mara** leads the data team. She asks the business questions: what will this save, how often will it be wrong, and who will act on it?
- **Dev** is the machine-learning engineer. He owns the training scripts, the saved models, and the prediction service that other systems call.
- **Ines** is a junior analyst learning deep learning alongside you. She asks the questions you are probably thinking, such as why the loss went up or why the shapes do not match.

All three are fictional, as is the company, and every dataset in the book except one is generated by a short script you will run in Chapter 2. That makes the results reproducible and lets us discuss them honestly: our data is cleaner and simpler than real business data, and we say so whenever it matters.

The warm-up: count-sheet digits

Larkspur's stores still do some stock counts on paper. Staff write quantities by hand on count sheets, and someone types them in later. A model that reads handwritten digits could pre-fill those numbers and ask a person to check only the uncertain ones. For this warm-up we use a small public teaching dataset of handwritten digits that ships with scikit-learn: 1,797 grayscale images of 8 by 8 pixels, each labeled 0 to 9. It is a stand-in for Larkspur's sheets, not data from them, and it is perfect for learning because it is tiny and needs no download. Part II builds a network for it that trains in under a second on our CPU.

Project 1: the thumbnail classifier

The online catalog has thousands of product thumbnails, and someone has to tag each with its product type so that search and filters work. The team wants a model that auto-tags thumbnails into six classes: **mug, bottle, book, plate, tshirt, and lamp**. Our version uses small synthetic 28 by 28 grayscale drawings: 3,000 for training and 600 for testing. Part III builds convolutional networks for them, studies data augmentation, and then faces a realistic twist: a new photo studio changes the look of the images and only 60 new examples are labeled. Real product photos are much harder than our simple drawings; we will be explicit about what does and does not carry over.

Project 2: review sentiment

Customers leave short reviews on the online shop. The customer-care team wants negative reviews flagged so they can respond quickly. Our dataset has 400 short fictional reviews, each labeled positive or negative. Many have a twist such as "The picture frame is not worth the price, but my family loves the picture frame", where the part after "but" decides the sentiment. A few labels even disagree with the text, as happens with real ratings. Part IV turns text into tensors and compares a model that ignores word order with recurrent networks that read in sequence, and Part V builds a small transformer for the same task.

Project 3: the demand forecaster

Store managers order stock a week ahead. The team wants a model that predicts the next 7 days of unit sales for each store from its recent history. Our dataset covers two years of daily sales for the three stores, with weekly patterns, a holiday season, promotions, and a gentle upward trend. Chapter 5 fits a trend line to it by hand; Chapter 18 builds a sequence model and compares it with simple baselines.

A trained model helps nobody until other software can use it, so Part VI turns the three models into a small system with saved checkpoints, exported models, batch scoring, and a prediction API.

The Plan of the Book

The book is organized in six parts that build on each other. You can skim parts you already know, but the code in later parts reuses helpers created earlier, so it is easiest to read in order the first time.

Part	Chapters	What you build
I. Foundations	1–5	Setup, tensors, autograd, gradient descent by hand on demand data
II. Training Neural Networks	6–10	Modules, data loaders, the training loop, the digit reader, training techniques
III. Computer Vision with CNNs	11–15	Convolutions, the thumbnail classifier, augmentation, residual networks, transfer learning
IV. Sequences and Text	16–19	Tokenization and embeddings, LSTM sentiment, demand forecasting, attention
V. Transformer Models	20–23	Transformer blocks, a review transformer, a tiny language model, pretraining ideas
VI. Deploying Models	24–28	Checkpoints and logs, performance, export, a prediction API, the capstone

Part I is deliberately low-level: many bugs in deep learning code are shape bugs, so Chapter 3 builds fluency with tensors, Chapter 4 opens up autograd, and Chapter 5 runs the entire training cycle with nothing hidden.

How to Read the Code

Every example in this book was run, and the output you see under an **Output** caption was pasted from that run, never typed by hand. Code blocks are labeled with the file they belong to. The reader's project folder is called `larkspur-dl/`, and it grows chapter by chapter:

- `make_data.py` generates the datasets into `data/`. You create and run it in Chapter 2.
- `larkspur.py` collects shared helpers, such as seeding, device selection, data loading, and the training loop. It starts in Chapter 7 and grows as we go.
- `models.py` holds the model classes. It starts in Chapter 9.
- Each chapter has its own script named `chNN_topic.py`, such as `ch02_first_tensors.py` or `ch12_cnn.py`, which you run from the project folder.

Appendix B contains the final versions of the shared files, the data generator, a reusable training template, and a table of every reference result in the book, so you can always compare your numbers with ours.

A first look at PyTorch code

Installation is the subject of the next chapter, but it helps to see what PyTorch code looks like before we set it up. The snippet below creates a tensor that holds one example with three numbers, builds a single linear layer that maps three inputs to two outputs, and applies it. Do not worry about running it yet.

ch01_peek.py

```
import torch
from torch import nn

torch.manual_seed(42)
x = torch.tensor([[0.2, 0.7, 0.1]])   # one example with 3 numbers
layer = nn.Linear(3, 2)               # 3 inputs -> 2 outputs
y = layer(x)
print(x.shape, "->", y.shape)
print(y)
print(layer.weight.shape, layer.bias.shape)
```

Output

```
torch.Size([1, 3]) -> torch.Size([1, 2])
tensor([[0.1291, 0.3682]], grad_fn=<AddmmBackward0>)
torch.Size([2, 3]) torch.Size([2])
```

Several ideas from this chapter are already visible in these few lines. A **tensor** is PyTorch's array type, similar to a NumPy array. Its `shape` of `[1, 3]` means one row (one example) with three columns (three features). The layer holds a weight matrix of shape `[2, 3]` and a bias vector of shape `[2]`; these eight numbers, six weights and two biases, are the adjustable parameters that training will tune. Calling the layer computes a weighted sum of the inputs plus the bias for each of the two outputs. Because the layer starts with random weights, the two output values are arbitrary; we fixed the random seed with `torch.manual_seed(42)` so that the numbers are repeatable.

The odd-looking `grad_fn=<AddmmBackward0>` is autograd at work. PyTorch noted that `y` was produced by a matrix multiply-and-add operation involving parameters that need gradients, and it kept a record so that it can later compute how a loss built from `y` depends on each weight. Chapter 4 explains this record in detail. For now, notice that we did not have to ask for it: tracking gradients is built in.

Conventions used in the code

We keep the code style consistent so that you can focus on the ideas. Imports follow a fixed order: the Python standard library, then third-party packages such as `torch`, then the project's own modules imported as `import larkspur as lk` and `from models import ...`. We use the same variable names everywhere: `model`, `optimizer`, `loss_fn`, `train_loader` and `test_loader`, `xb` and `yb` for a batch of inputs and labels, `logits` for raw model scores, `probs` for probabilities, and `device` for the chosen device. When you meet these names in any chapter, they mean the same thing.

Weights start random and data is shuffled, so we seed every random number generator before building each model; you will see `set_seed(42)` in almost every training script from Chapter 7 onward. Seeding makes your own runs repeatable, which is what you need for debugging and fair comparisons.

The most effective way to use this book is to run each script, compare your output with ours, then change something and predict the result before rerunning. Intuition for the many knobs of deep learning comes mainly from turning them.

> **TIP**
> Keep a plain-text lab notebook, even a single file, where you record each experiment: what you changed, the final numbers, and one sentence about what you learned. It takes seconds per run and saves hours when you later wonder which settings produced your best model. Chapter 24 turns this habit into automatic CSV and JSON logs.

What Deep Learning Will Not Do for You

A model reflects the data it was trained on. If past demand was depressed by a stock shortage, a forecaster learns that shortage as if it were a pattern; if labels are inconsistent, the model learns the inconsistency. No architecture fixes data problems. Test accuracy is also an estimate, not a guarantee: when the world changes, as when Larkspur's new photo studio changes the look of thumbnails in Chapter 15, performance can drop sharply without any change in the code.

Finally, a model is part of a decision, and the decision belongs to people. A sentiment model that wrongly flags a happy customer costs a few minutes of a support agent's time; one that misses an angry customer may cost the customer. Mara's question, "who acts on this, and what happens when it is wrong?", is the right one to ask before a single line of training code is written.

Try It Yourself

1. Pick three tasks from your own work or daily life. For each, decide whether hand-written rules, a classical machine-learning model, or deep learning is the most sensible first approach, and write one sentence explaining why.
2. For each of Larkspur's three projects, write the one-sentence decision statement suggested in the first Tip: what the output supports and who acts on it. Then name one kind of mistake the model could make and what it would cost.
3. Using the training-cycle diagram, explain in your own words to an imaginary colleague what a loss, a gradient, and a learning rate are. Try not to use any formulas.
4. Look at the `ch01_peek.py` output again. How many adjustable numbers does `nn.Linear(3, 2)` have? How many would `nn.Linear(784, 128)` have? Check your answer later, after installing PyTorch in Chapter 2, by summing `p.numel()` over `layer.parameters()`.
5. Suggest a simple baseline for each of the three projects that does not use any learning at all. Keep your list; Chapters 8, 12, 16, and 18 compare every model with a baseline, and Chapter 18 uses exactly this kind of no-learning baseline for demand.

Key Takeaways

- Machine learning replaces hand-written rules with a flexible function whose numbers are adjusted from examples; deep learning uses a stack of neural-network layers as that function.
- Every model is trained by the same cycle: forward pass, loss, gradients by backpropagation, and an optimizer step, repeated over many batches and epochs.
- Always keep held-out data; a large gap between training and held-out performance signals overfitting.
- Deep learning shines on images, text, audio, and sequences with plenty of data; on small tabular data or problems a rule can solve, simpler methods are often better, so build a baseline first.
- PyTorch is Pythonic and eager, has autograd built in, and offers tools such as `torch.compile` and `torch.export` when you need speed or deployment.
- GPUs speed up large-scale training, but every example in this book runs on an ordinary CPU; device-agnostic code lets the same script use a GPU when one is available.
- Larkspur Goods gives us a digit reader warm-up and three projects: a thumbnail classifier, a review sentiment model, and a store demand forecaster.
- Training numbers vary slightly across machines; shapes, parameter counts, and dataset sizes do not.