WIRED
TO LEARN

LAUNCH SEQUENCE

A series of three books

Book 1 · Ignition
Wired to Learn
Neural networks from first principles to intelligent applications

Book 2 and **Book 3**
To follow. Each will build on the one before it, carrying the reader from the foundations laid here to advanced, professional-level practice.

launchsequencebooks.com

LAUNCH SEQUENCE · BOOK 1

WIRED TO LEARN

Build, Train, Optimize and Deploy Neural Networks with Python, NumPy and PyTorch

From Your First Neuron to Intelligent Applications

Adi Maaita

Wired to Learn
Build, Train, Optimize and Deploy Neural Networks with Python, NumPy and PyTorch
From Your First Neuron to Intelligent Applications

Launch Sequence · Book 1 · Ignition. *Launch Sequence* is a series of three books. This is the first; the two that follow will build on it toward advanced, professional-level practice.

Copyright © 2026 Adi Maaita. All rights reserved.

No part of this book may be reproduced, stored or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in reviews.

Every program in this book is included because it runs. The companion code (folder `code/`) reproduces every listing, and the scripts in `scripts/` regenerate every plotted figure. The author has made every effort to ensure accuracy but assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Companion code: `github.com/launchsequencebooks/wired-to-learn`
Errata and updates: `launchsequencebooks.com`

Using the code. The companion code is released under the MIT License: you may use, modify and redistribute the listings, including in commercial software, provided the license notice is kept. The text, figures and the Chapter 35 passage corpus are copyright of the author and are not covered by that license; short excerpts may be quoted in reviews and teaching with attribution.

Python is a trademark of the Python Software Foundation. PyTorch and ONNX are trademarks of the Linux Foundation. NumPy is a trademark of NumFOCUS. Docker is a trademark of Docker, Inc. All other trademarks are the property of their respective owners.

First edition 2026.
Launch Sequence Books, Amman, Jordan.

CHAPTER 1

Meet the Neural Network

In this chapter you will:

- see how *learning from examples* differs from ordinary programming;
- meet the biological neuron and the artificial neuron it inspired;
- place neural networks inside the bigger picture of AI and machine learning;
- walk through eighty years of neural-network history in one page;
- watch a network learn, and run your first network in about 30 lines of Python.

1.1 A machine that learns

Write a program that decides whether a photo contains a cat. Go on, try.

You might start with rules: *if there are two pointy ears and whiskers and fur…* Within minutes the rules collapse. What is an "ear" in terms of pixels? What about a cat seen from behind, a cartoon cat, a cat in the dark, or a fox that looks a lot like a cat? Every rule you add creates three new exceptions.

Neural networks solve problems like this by turning programming upside down. Instead of writing the rules ourselves, we show the computer thousands of *examples* (photos labeled "cat" or "not cat") and let it discover the rules on its own. Figure 1.1 shows the difference.

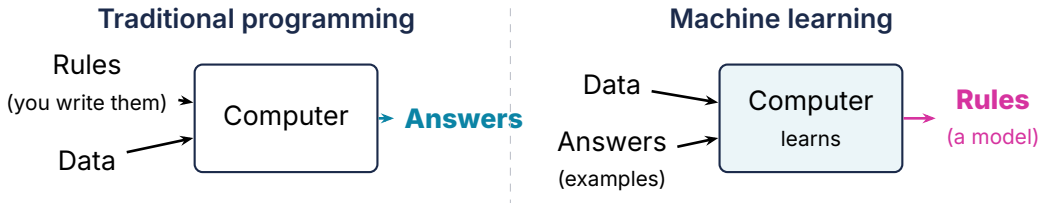


Figure 1.1. Traditional programs turn rules and data into answers. Machine learning turns data and answers into rules. A neural network is one powerful way to represent those learned rules.

The "rules" a neural network learns are not if-statements. They are thousands, millions or billions of numbers called **weights**. Each weight controls how strongly one part of the network influences another. Learning means finding good values for all those weights. Once trained, the network is just a function: numbers go in (pixels, words, sensor readings) and numbers come out (probabilities, prices, translated words).

> **INTUITION** Think of a neural network as a huge mixing desk in a recording studio, with a slider for every weight. Training is an assistant who listens to the output, compares it with the song you wanted, and nudges every slider a little to bring the output closer. Repeat this a million times and the desk plays your song, even from a recording it has never heard before.

1.2 Inspired by the brain

Your brain contains roughly 86 billion nerve cells called **neurons**. A neuron collects electrical signals from other neurons through branches called *dendrites*. If the combined input is strong enough, the neuron "fires" and sends its own signal down a long fiber, the *axon*, to thousands of other neurons. The connection points, *synapses*, can be strong or weak, and they change with experience. As far as neuroscientists can tell, that change is a large part of how we learn.

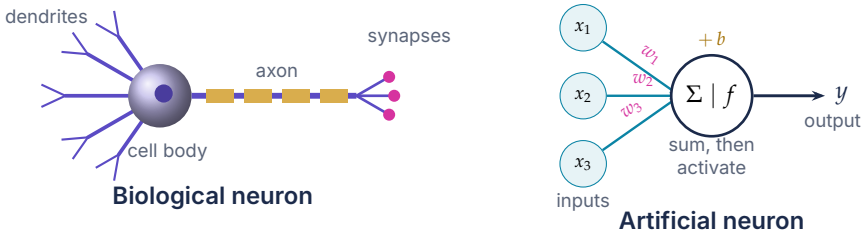


Figure 1.2. The biological neuron (left) inspired the artificial neuron (right). Inputs x_i play the role of incoming signals, weights w_i the strength of synapses, the bias b how easily the neuron fires, and the activation function f the decision to fire.

The **artificial neuron** keeps only the essence. It receives numbers $x_1, x_2, \ldots$, multiplies each by a weight $w_1, w_2, \ldots$, adds the products and a *bias* b, and passes the total through an **activation function** f that decides how strongly to "fire":

$$y = f\big(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b\big). \tag{1.1}$$

That is the entire unit. Connect many of them in layers, with the outputs of one layer becoming the inputs of the next, and you have a **neural network** (Figure 1.3).

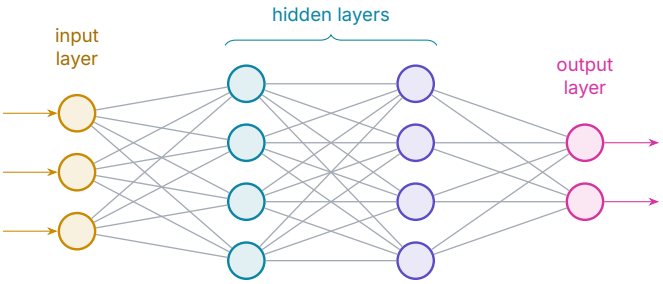


Figure 1.3. A small fully connected network. Every line is a weight. This one has $3\times4 + 4\times4 + 4\times2 = 36$ weights plus $4 + 4 + 2 = 10$ biases (one per neuron after the input layer). Modern language models have hundreds of billions of weights.

> **PITFALL** Artificial neurons are inspired by biology, not copies of it. Real neurons communicate with timed spikes, use dozens of chemical messengers, and rewire physically. Treat the brain as a source of inspiration, not as a specification: deep learning progressed fastest when it followed mathematics and experiments rather than anatomy.

1.3 Where neural networks fit

Artificial intelligence (AI) is the broad goal of making machines do things that seem to require intelligence. **Machine learning** (ML) is the part of AI where systems improve from data instead of being explicitly programmed. **Deep learning** is machine learning with neural networks that have many layers (hence "deep"). Figure 1.4 shows how they nest.

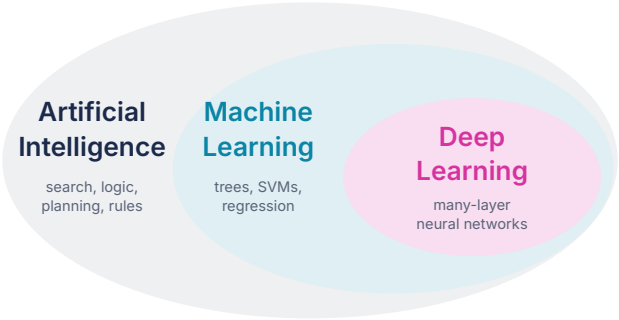


Figure 1.4. Deep learning is a subset of machine learning, which is a subset of artificial intelligence.

Why has deep learning come to dominate? Classic machine-learning methods need a human to design the *features*, deciding that "number of edges" or "average color" are useful descriptions of an image. Deep networks learn their own features, layer by layer: the first layers of an image network learn to detect edges, the next layers combine edges into textures and shapes, and later layers combine shapes into eyes, wheels or faces. Given enough data and computing power, learned features beat hand-designed ones almost every time.

1.4 Eighty years in one page

Neural networks have a dramatic history of excitement, disappointment and comeback. Figure 1.5 picks out the milestones you will meet again in this book.

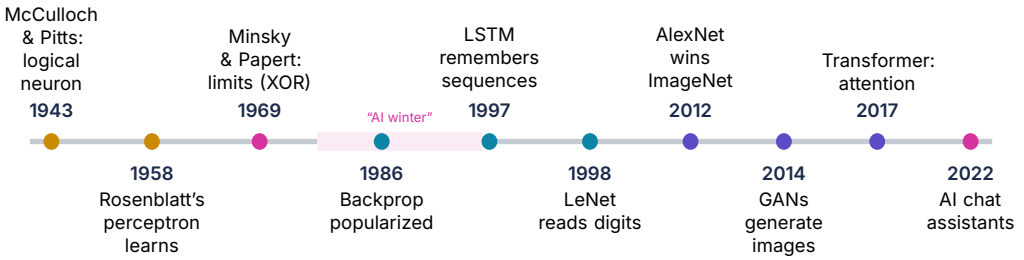


Figure 1.5. Milestones in neural-network history. Many "new" ideas are decades old; what changed after 2010 was abundant data, fast GPUs, and a handful of crucial tricks.

In 1958 Frank Rosenblatt's *perceptron* learned to classify simple patterns, and the press predicted thinking machines within years. In 1969 Marvin Minsky and Seymour Papert proved that a single perceptron cannot even compute the "exclusive or" (XOR) of two inputs (you will reproduce their result in Chapter 3). Funding dried up. The fix, networks with hidden layers trained by *backpropagation*, was popularized in 1986, but computers were too slow and datasets too small for it to shine.

Then three things arrived together: the internet produced enormous labeled datasets, graphics cards (GPUs) built for video games turned out to be perfect for neural-network arithmetic, and researchers found tricks such as ReLU activations and dropout that made deep networks trainable. In 2012 a network called AlexNet cut the error rate of the ImageNet image-recognition competition from 26% to 15%, and the deep-learning era began. In 2017 the *Transformer* architecture appeared; it now powers language, speech and image models alike.

1.5 What can neural networks do?

Given enough examples, almost any problem that maps an input to an output is a candidate for a neural network. Table 1.1 previews the families of networks you will build and the applications they power.

1.6 Watching a network learn

Before we look at any code, let us watch learning happen. Figure 1.6 shows two interleaved spirals of points, one spiral per class. A small network with two hidden layers is trained to tell them apart. (The figure is made by `scripts/figs_part1.py` using scikit-learn's ready-made network, since we have not built our own yet; in Chapter 8 you will redraw it with TinyNet.) The shaded background shows what the network would predict for every location, and the dark line is its **decision boundary**.

Table 1.1. Common tasks, the network families suited to them, and where you build each one.

Task	Input → output	Network family	Ch.
Spam detection, credit risk	features → class	fully connected (multilayer perceptron)	6–9
Photo tagging, X-ray screening	image → label	convolutional (CNN)	15–16
Demand or traffic forecasting	time series → future values	recurrent (LSTM/GRU)	17
Translation, chat, summarization	text → text	Transformer	18
Fraud and intrusion detection	normal data → anomaly score	autoencoder	19
Synthetic images, data augmentation	noise → image	GAN, diffusion	20
Molecules, social and IT networks	graph → property	graph neural network	21
Game playing, robotics	state → action	deep reinforcement learning	24

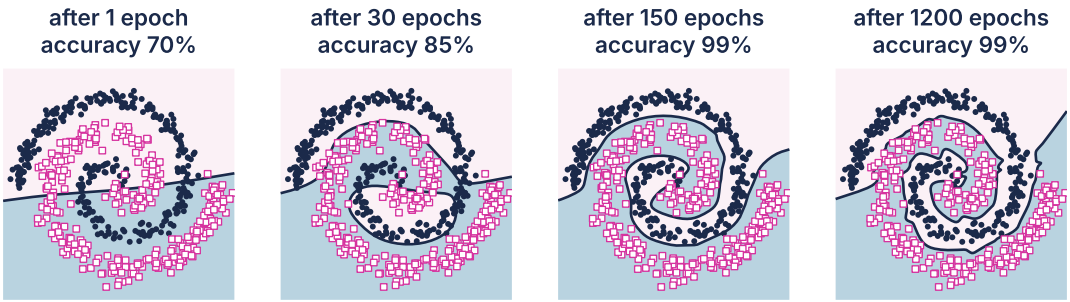


Figure 1.6. A network learning the two-spirals problem. After one epoch (one pass through the data) it can only draw a nearly straight line. With training, the boundary bends and twists until it follows the spirals almost perfectly. No human told it what a spiral is. Made by `scripts/figs_part1.py`.

At first the network's boundary is almost a straight line, and it gets about 70% of the points right just by luck and geometry. As training continues, the boundary develops curves, then a full spiral. Nobody programmed the shape; it emerged from thousands of tiny weight adjustments, each one reducing the error a little. By the end of Part II you will understand every one of those adjustments and be able to write the code that makes them.

1.7 Your first neural network

Here is a complete neural network in NumPy that learns XOR, the function Minsky and Papert proved a single perceptron cannot learn. It has 2 inputs, a hidden layer of 4 neurons, and 1 output. Do not worry about the details yet. Read it the way you would look at a map of a city you are about to explore: notice the main roads.

Listing 1.1. Your first neural network learns XOR. code/ch01/first_network.py

```
# Your first neural network: it learns XOR in about 30 lines of NumPy.
# Don't worry if some lines look mysterious - by Chapter 7 you will
# be able to write every one of them yourself.
import numpy as np

rng = np.random.default_rng(42)

# The four XOR examples: output is 1 when exactly one input is 1
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=float)
y = np.array([[0], [1], [1], [0]], dtype=float)

# A tiny network: 2 inputs -> 4 hidden neurons -> 1 output
W1 = rng.normal(0, 1, (2, 4)); b1 = np.zeros(4)
W2 = rng.normal(0, 1, (4, 1)); b2 = np.zeros(1)
sigmoid = lambda z: 1 / (1 + np.exp(-z))

for step in range(5001):
    # forward pass: make a prediction (@ is matrix multiplication, Chapter 2)
    h = np.tanh(X @ W1 + b1)
    p = sigmoid(h @ W2 + b2)
    # backward pass: how should each weight change to reduce the error?
    dz2 = (p - y) / len(X)
    dW2, db2 = h.T @ dz2, dz2.sum(0)   # .T flips rows and columns (Chapter 2)
    dz1 = (dz2 @ W2.T) * (1 - h**2)
    dW1, db1 = X.T @ dz1, dz1.sum(0)
    # learn: nudge every weight a little bit downhill
    for param, grad in ((W1, dW1), (b1, db1), (W2, dW2), (b2, db2)):
        param -= 1.0 * grad
    if step % 1000 == 0:
        loss = -np.mean(y * np.log(p) + (1 - y) * np.log(1 - p))
        print(f"step {step:4d}  loss {loss:.4f}")

print("predictions:", p.round(3).ravel())
```

Running it prints:

```
step    0  loss 0.8299
step 1000  loss 0.0024
step 2000  loss 0.0011
step 3000  loss 0.0007
step 4000  loss 0.0005
step 5000  loss 0.0004
predictions: [0.    0.999 1.    0.001]
```

The four predictions match the XOR targets 0, 1, 1, 0. The main roads of the program are the same in every neural network you will ever write:

1. **Forward pass** (lines 19–20): push the inputs through the layers to get a prediction.
2. **Loss** (line 30): measure how wrong the prediction is with a single number. This program computes it only for printing, once every 1,000 steps; the backward pass on line 22 starts directly from $p - y$, which, as Chapter 5 shows, is the loss's own derivative.
3. **Backward pass** (lines 22–25): work out how each weight contributed to the error. This is *backpropagation* (Chapter 7).
4. **Update** (lines 27–28): move every weight a small step in the direction that reduces the loss. This is *gradient descent* (Chapter 5).

PITFALL in-place updates Line 28 says `param -= 1.0 * grad`, not `param = param - 1.0 * grad`. The first form changes the numbers inside the array that `W1` (or `b1`, `W2`, `b2`) points to. The second creates a new array and binds it to the loop variable `param` only; the real weights never move, the loss never falls, and Python raises no error. When you modify this program, keep the `-=`. Chapter 8 builds a library around this exact behavior.

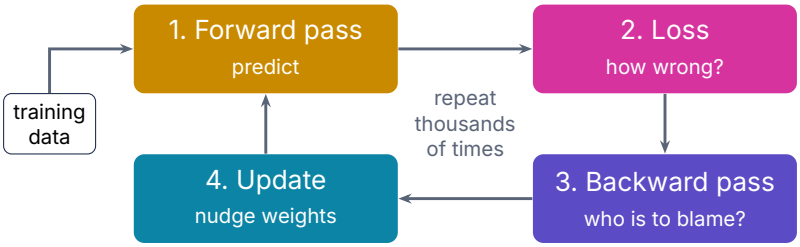


Figure 1.7. The training loop: four steps that every neural network repeats, from a 4-neuron toy to a giant language model.

TRY IT Run `first_network.py`, then change the number of hidden neurons from 4 to 1 (change the 4s in lines 13–14). Can a network with one hidden neuron learn XOR? Now try 2 hidden neurons and different random seeds. You have just met two themes of this whole book: *capacity*, how much a network can represent, and the luck of initialization, which decides whether training finds that capacity.

Lines 22–25 of Listing 1.1 used four pieces of mathematics without naming them: a transpose, a matrix product, a derivative and the chain rule. Chapter 2 gives you exactly those tools, with a picture for each, and nothing more.

Summary

KEY TAKEAWAYS

- Machine learning finds rules from examples; neural networks store those rules as weights.
- An artificial neuron computes a weighted sum plus a bias and passes it through an activation function. Networks are layers of such neurons.
- Deep learning (many-layer networks) learns its own features, which is why it dominates vision, language and speech.
- Every network trains with the same loop: forward pass, loss, backward pass, update.

Exercises

1.1 [concept] In your own words, explain the difference between traditional programming and machine learning. Give one problem better solved by each.

1.2 [concept] Count the weights and biases in a fully connected network with layer sizes $784 \to 256 \to 128 \to 10$. (This is the digit reader you will build in Chapter 9; check your answer against Table 6.1.)

1.3 [concept] For each application, name the input, the output, and the network family from Table 1.1: (a) predicting a hospital's electricity demand for tomorrow; (b) flagging unusual logins on a corporate network; (c) reading license plates.

1.4 [code] In `first_network.py`, record the loss every 100 steps in a list and plot it with Matplotlib. Describe the shape of the curve.

1.5 [code] Change the learning rate in line 28 from `1.0` to `0.1` and to `10.0`. What happens to training in each case? Keep your notes; Chapter 5 explains what you saw.