Neural Networks & Deep Learning

A Complete 11-Chapter Data Science & ML Course

Topics covered:

Perceptrons, backpropagation, activation functions, regularization

CNNs, RNNs/LSTMs, a transformer preview, PyTorch vs. TensorFlow

A real trained network, honestly compared against ml1's own classical models

Exercises: 33 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Course 3 of 6 in the Data Science & ML subject

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 11

From Logistic Regression to Neurons

Neural Networks & Deep Learning

Chapter 1 · From Logistic Regression to Neurons

You already built a neural network's most basic building block. Not metaphorically — literally. This chapter proves it, then shows exactly what's new once you stop stopping at one.

One Neuron Is ml1-5's Own Logistic Regression, Unchanged

`ml1-5`'s own logistic regression: take a weighted sum of the inputs, add a bias, squash the result through sigmoid.

```
ml1-5's logistic regression:
z = w1*x1 + w2*x2 + ... + wn*xn + b
output = sigmoid(z)
```

A single **artificial neuron** computes exactly this — no more, no less:

```
a single neuron:
z = w1*x1 + w2*x2 + ... + wn*xn + b
output = activation(z)
```

THIS ISN'T A LOOSE ANALOGY

Set the neuron's own `activation` function to sigmoid, and the formula is character-for-character identical to `ml1-5`'s own logistic regression. You've already trained one of these — `ml1-5`'s own coefficients were literally a single neuron's own learned weights.

So What's Actually New?

Two genuinely new ideas, both architectural rather than mathematical:

- **Many neurons, side by side.** Instead of one neuron producing one output, a **layer** runs several neurons in parallel against the same inputs — each with its own independently learned weights and bias, each producing its own output.

- **Layers, stacked.** One layer's outputs become the next layer's inputs. A layer sitting between the raw inputs and the final output is a **hidden layer** — inputs, hidden layers, and outputs together form the network.

> **LOGISTIC REGRESSION, REFRAMED**
>
> In this course's own vocabulary, `ml1-5`'s logistic regression is precisely "a neural network with zero hidden layers and exactly one output neuron." Nothing about that description is a stretch — it's the honest, literal special case this entire course generalizes outward from.

Terminology, Before It's Needed

TERM	MEANING
Input layer	The raw features themselves — not neurons, just the data going in
Hidden layer	A layer of neurons sitting between input and output, visible only to the network itself
Output layer	The final layer, producing the network's own prediction
Weights / bias	Exactly ml1-5's own learned coefficients and intercept, one set per neuron
Fully connected / dense	Every neuron in one layer connects to every neuron in the next

Why Bother Stacking Layers At All?

A single neuron — one weighted sum, one squash — can only ever draw a straight decision boundary through its inputs, a flat line or hyperplane. No amount of retuning its weights changes that basic geometric limit. `nn1-2` shows a real, concrete problem a single neuron structurally cannot solve, no matter how it's trained. `nn1-3` shows exactly why stacking a hidden layer underneath fixes it. This chapter deliberately doesn't resolve that mystery yet — it's worth sitting with the question first.

This Course's Own Roadmap

CHAPTER	DELIVERS
nn1-2 / nn1-3	The real historical limit a single neuron hit, and how depth fixed it
nn1-4 / nn1-5	Activation functions and how a whole network actually learns (backpropagation)
nn1-6	Training a network without it overfitting
nn1-7	CNNs — the architecture behind real image recognition
nn1-8 / nn1-9	RNNs, LSTMs, and a first look at what replaced them

CHAPTER	DELIVERS
nn1-10 / nn1-11	Real tools, and a real trained network of your own

Hands-On Exercises

EXERCISE 1

Using this chapter's own side-by-side formulas, explain precisely why a single neuron with a sigmoid activation is not merely similar to ml1-5's logistic regression, but the identical computation.

View solution

EXERCISE 2

Explain the two genuinely new architectural ideas this chapter introduces beyond ml1-5's own single neuron, and explain why the chapter calls these "architectural rather than mathematical" changes.

View solution

EXERCISE 3

Explain why this chapter deliberately doesn't reveal what problem a single neuron can't solve, or how stacking layers fixes it, leaving both for nn1-2 and nn1-3 instead.

View solution

CHAPTER 1 QUICK REFERENCE

- A single neuron with sigmoid activation = ml1-5's own logistic regression, exactly
- **Layer** — several neurons in parallel, each with its own weights · **Hidden layer** — a layer between input and output
- Logistic regression = "a neural network with zero hidden layers and one output neuron"
- A single neuron can only draw a straight decision boundary — nn1-2/nn1-3 show why that's a real, hard limit, and how depth fixes it
- **Next chapter:** The Perceptron & the XOR Problem

CHAPTER 2 OF 11

The Perceptron & the XOR Problem

Neural Networks & Deep Learning

Chapter 2 · The Perceptron & the XOR Problem

`historyai3-3` named "the Perceptron, Minsky/Papert's critique" as part of its own narrated history of AI, without stopping to explain the actual mechanism or the actual mathematical limit involved. This chapter delivers both, technically — and delivers the real, concrete proof of `nn1-1`'s own closing claim: a single neuron can only draw a straight line.

Rosenblatt's Perceptron (1958)

Frank Rosenblatt's perceptron is, in this course's own terms, exactly `nn1-1`'s single neuron — one weighted sum, one threshold decision, trained via a simple rule: when the perceptron misclassifies an example, nudge each weight slightly in the direction that would have made the correct answer more likely. Real, working, and genuinely exciting for its era — contemporary press coverage reported startlingly ambitious claims about what the technology would eventually achieve, including the U.S. Navy's own widely reported 1958 expectation that it would one day "walk, talk, see, write, reproduce itself and be conscious of its existence."

What the Perceptron Could Actually Do

For a **linearly separable** problem — one where a single straight line can separate the two classes — the perceptron's own learning rule reliably finds such a line, converging to a correct classifier. This genuinely worked for many real problems. The trouble was never training *within* that limit; it was the limit itself.

The XOR Problem — A Real, Provable Impossibility

XOR ("exclusive or") takes two binary inputs and outputs 1 if exactly one of them is 1, 0 otherwise:

X1	X2	XOR OUTPUT
0	0	0
0	1	1
1	0	1

X1	X2	XOR OUTPUT
1	1	0

Plot these four points on a simple 2D grid. The two points that should output 1 — `(0,1)` and `(1,0)` — sit on one diagonal. The two that should output 0 — `(0,0)` and `(1,1)` — sit on the *other* diagonal. No single straight line can put both diagonal pairs on their own correct side simultaneously — try to draw one, and at least one point always ends up misclassified.

THIS IS NOT A TRAINING DIFFICULTY

This isn't "hard to learn" or "needs more examples" — it's a genuine geometric impossibility for any single straight decision boundary, regardless of which weights the perceptron's own learning rule eventually settles on. No amount of additional training data or extra training time changes it. This is the concrete, provable version of `nn1-1`'s own closing claim.

Minsky & Papert, 1969 — The Real Critique

Marvin Minsky and Seymour Papert's book *Perceptrons* formally proved the XOR limitation (and others like it) for single-layer networks. The honest, often-missed nuance: the book didn't claim a multi-layer network could never overcome this — it specifically noted the open question of whether an effective training method for multi-layer networks even existed. At the time, it didn't, in any widely practical form. `nn1-5` covers backpropagation — the training method that eventually filled that exact gap, in 1986.

A PRECISE LINK TO HISTORYAI2-5, NOT AN OVERCLAIM

`historyai2-5`'s own First AI Winter names the ALPAC and Lighthill Reports as the primary drivers of the broader AI funding collapse. Minsky and Papert's critique is a real, separate, well-documented contributing thread specific to neural-network research — a rigorous demonstration of a genuine limitation, landing in the same broader funding-pullback climate, that contributed to neural networks specifically falling out of favor for roughly the next decade and a half.

What Comes Next

`nn1-1` already named the fix in outline: stack a hidden layer underneath. `nn1-3` shows exactly why that works — a real, worked demonstration that a two-layer network genuinely does solve XOR, geometrically. This chapter deliberately stops short of showing that solution.

Hands-On Exercises

EXERCISE 1

Using this chapter's own XOR truth table, explain precisely why no single straight line can correctly separate the two output classes, referencing the two diagonals directly.

View solution

EXERCISE 2

Explain why this chapter's own warn-box insists the XOR failure is "not a training difficulty," and explain what would (and would not) fix it if it were merely a training difficulty.

View solution

EXERCISE 3

Using this chapter's own tip-box, explain the honest, precise relationship between Minsky and Papert's critique and historyai2-5's own First AI Winter — why is it described as "a real, separate, well-documented contributing thread" rather than the winter's own primary cause?

View solution

CHAPTER 2 QUICK REFERENCE

- Rosenblatt's perceptron (1958) — nn1-1's own single neuron, with a simple mistake-driven weight-update rule
- Works reliably for **linearly separable** problems — a single line can separate the two classes
- **XOR** — the classic, provable counterexample: no single line can separate its own two diagonal classes
- Minsky & Papert (1969) formally proved this limit — and honestly left multi-layer networks' own potential an open question, pending a real training method
- A real, precise contributing thread to neural networks' own decline within historyai2-5's own broader First AI Winter — not its sole cause
- **Next chapter:** Multi-Layer Perceptrons & Why Depth Solves XOR

CHAPTER 3 OF 11

Multi-Layer Perceptrons & Why Depth Solves XOR

Neural Networks & Deep Learning

Chapter 3 · Multi-Layer Perceptrons & Why Depth Solves XOR

`nn1-2` proved, geometrically, that no single straight line separates XOR. This chapter builds an actual, working two-layer network that solves it — and shows the general principle that makes the solution possible, not just this one specific case.

The Core Idea: A Hidden Layer Transforms the Space

A hidden layer doesn't just add more tunable numbers — it re-represents the input as a *new* set of coordinates, computed by the hidden neurons themselves. A problem that's unsolvable by a straight line in the *original* input space can become solvable by a straight line in this new, transformed space. That's the entire trick depth relies on.

A Real, Worked XOR Solution

XOR can be built from two simpler, individually linearly-separable pieces: `XOR(x1, x2) = OR(x1, x2) AND NAND(x1, x2)` — true exactly when at least one input is 1, *and* not both are 1. Both `OR` and `NAND` are themselves linearly separable (a single line can solve each one alone). One concrete set of weights that implements this — one valid solution among several that would work:

```
Hidden neuron 1 (approximates OR):   step(x1 + x2 - 0.5)
Hidden neuron 2 (approximates NAND): step(-x1 - x2 + 1.5)
Output neuron (AND of both hidden outputs): step(h1 + h2 - 1.5)
```

X1	X2	H1 (OR)	H2 (NAND)	OUTPUT (AND)	CORRECT XOR?
0	0	0	1	0	✓
0	1	1	1	1	✓
1	0	1	1	1	✓
1	1	1	0	0	✓

WHERE THE TRANSFORMATION ACTUALLY HAPPENED

In the original `(x1, x2)` space, XOR was unsolvable by any line — `nn1-2`'s own proof. In the new `(h1, h2)` space the hidden layer produces, the four points become `(0,1)`, `(1,1)`, `(1,1)`, `(1,0)` — and a single line *does* separate the "output 1" points from the "output 0" point in this new space. The hidden layer didn't add raw capacity so much as it re-drew the map the final line gets to work with.

Why This Requires Genuine Nonlinearity

This trick only works because the hidden layer's own activation function (`step` above; `nn1-4` covers the real options) is **nonlinear**. Stack two purely linear layers with no nonlinearity between them, and the math collapses: a linear function of a linear function is still just one linear function — mathematically indistinguishable from a single layer, with none of this chapter's own transformation trick available at all. `nn1-4` picks this up directly.

Closing nn1-2's Own Historical Thread

This is, in principle, exactly the resolution Minsky and Papert's own book left open in `nn1-2` — a multi-layer network genuinely can solve XOR. What this chapter's own worked example doesn't do is *learn* these weights automatically from data the way `ml1-3`'s own regression or `nn1-2`'s own perceptron rule did — these specific numbers were hand-derived for this one small, known problem. A real, general, automatic training method for networks like this one didn't arrive until 1986 — `nn1-5`'s own backpropagation, the actual historical breakthrough that made deep networks practical rather than hand-built curiosities.

Beyond XOR — Why This Is "Depth," Not Just "Size"

The same principle scales: each additional layer can compose the previous layer's own transformed representation into something even more re-arranged, letting a network build up genuinely hierarchical features — this is the real, technical meaning behind "deep" in deep learning, and it's a fundamentally different lever from simply adding more neurons to one single layer (more width, same transformation depth).

Hands-On Exercises

EXERCISE 1

Using this chapter's own worked table, explain why the four points become linearly separable in (h1, h2) space even though nn1-2 proved they aren't separable in the original (x1, x2) space.

View solution

EXERCISE 2

Explain why this chapter says stacking two purely linear layers with no nonlinearity would collapse into a single linear layer, and why this matters for whether the hidden-layer trick in this chapter would even work at all.

View solution

EXERCISE 3

Explain what this chapter's own worked XOR solution does and doesn't prove, specifically regarding whether these weights were learned automatically, and identify what nn1-5 still needs to deliver.

View solution

CHAPTER 3 QUICK REFERENCE

- A hidden layer transforms the input into a new coordinate space — a problem unsolvable by a line in the original space can be solvable in the new one
- XOR = OR AND NAND — two linearly-separable sub-problems, composed by a second linear layer
- This requires genuine **nonlinearity** in the hidden layer — purely linear layers collapse into one, previewing nn1-4
- This worked example is hand-derived, not learned — nn1-5's own backpropagation (1986) is the real historical breakthrough that made this automatic
- "Deep" means composing transformations across layers — a genuinely different lever from adding more neurons to one layer
- **Next chapter:** Activation Functions

CHAPTER 4 OF 11

Activation Functions

Neural Networks & Deep Learning

Chapter 4 · Activation Functions

`nn1-3` used the step function for a reason — it's the simplest possible illustration of nonlinearity. It's also not what real networks actually use. This chapter covers the real options, and a real problem that follows directly from the choice.

Why the Step Function Doesn't Actually Work in Practice

`nn1-5`'s own backpropagation trains a network by computing gradients — how much a small change in each weight would change the output — and nudging weights in the direction that reduces error. The step function is flat everywhere except at one single point, where it jumps discontinuously. Its gradient is zero almost everywhere, and undefined exactly at the jump — there's no useful signal anywhere for gradient-based training to act on. `nn1-3`'s own hand-derived weights worked specifically because a human, not gradient descent, chose them.

Sigmoid

`1 / (1 + e^-x)`

Already familiar from `ml1-5`. Smooth, differentiable, output in `(0, 1)` — genuinely useful for a final classification-probability output.

Tanh

`tanh(x)`

Same S-shape, output in `(-1, 1)` instead — zero-centered, a real practical advantage over sigmoid for hidden layers.

ReLU

`max(0, x)`

Zero for negative inputs, identity for positive ones. Simple, and the modern default for most hidden layers.

Sigmoid's Own Real Problem — Saturation

For very large positive or very large negative inputs, sigmoid's own curve goes nearly flat — its gradient (slope) approaches zero in those regions, a state called **saturation**. A neuron whose weighted sum regularly lands far out in either flat tail produces almost no useful gradient for `nn1-5`'s own backpropagation to work with — the weight barely updates at all, regardless of how wrong the prediction actually is.

THE VANISHING-GRADIENT PROBLEM — PREVIEWED, NOT RESOLVED HERE

This is the first appearance of a real, serious issue this course will return to directly: gradients that shrink toward zero as they're computed, leaving early layers barely trained at all. `nn1-8` covers exactly why this becomes especially severe in networks processing long sequences, and why it directly motivated a genuinely new architecture (the LSTM) to fix it.

Tanh — The Same Problem, Recentered

Tanh's own zero-centered output is a real, practical improvement: sigmoid's outputs are always positive, which can bias how gradients accumulate across a whole layer in later training steps; tanh's own `(-1, 1)` range avoids that specific bias. But tanh still saturates at its own extremes — the same flattening, the same near-zero gradient far from center. Recentering the output doesn't remove the underlying saturation problem, just one specific side effect of it.

ReLU — Simpler, and a Real Fix for Part of the Problem

`max(0, x)`: zero for any negative input, exactly itself for any positive input. For every positive input, the gradient is a constant 1 — no saturation at all on that side. This genuinely helped make much deeper networks practically trainable, a real, documented factor alongside other advances behind the deep-learning resurgence `historyai3-3` already named — the 2006 deep-belief-network revival and the years of architecture improvements that followed it.

RELU'S OWN HONEST DOWNSIDE — "DYING RELU"

A neuron whose weights push its weighted sum permanently negative outputs exactly zero for every input, forever — and since ReLU's own gradient is also exactly zero for negative inputs, that neuron can never recover through ordinary gradient-based training once it lands there. A real, documented practical failure mode, not a theoretical curiosity. **Leaky ReLU** (a small nonzero slope for negative inputs instead of a flat zero) is a common, simple fix.

Choosing an Activation Function in Practice

FUNCTION	TYPICAL USE	REAL WEAKNESS
Sigmoid	Binary classification output layer (ml1-5's own job)	Saturates at both extremes

FUNCTION	TYPICAL USE	REAL WEAKNESS
Tanh	Hidden layers, especially in RNNs (nn1-8)	Still saturates, just zero-centered
ReLU	Hidden layers — the modern default	Dying ReLU for permanently negative neurons

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning about gradients, why nn1-3's own step function could never actually be trained by nn1-5's own backpropagation, even though it worked fine for nn1-3's own hand-derived example.

View solution

EXERCISE 2

Explain what saturation means for a sigmoid neuron, and explain why this chapter says tanh's zero-centering is a real improvement without actually fixing the underlying saturation problem.

View solution

EXERCISE 3

Explain why a "dying ReLU" neuron can never recover through ordinary gradient-based training, using this chapter's own reasoning about ReLU's gradient for negative inputs.

View solution

CHAPTER 4 QUICK REFERENCE

- nn1-3's own step function has zero gradient almost everywhere — unusable with gradient-based training
- **Sigmoid** — smooth, (0,1), saturates at both extremes · **Tanh** — zero-centered (-1,1), still saturates
- **Saturation** — near-zero gradient far from center — the first appearance of the vanishing-gradient problem, resolved in depth in nn1-8
- **ReLU** — max(0,x), no saturation for positive inputs, the modern hidden-layer default; "dying ReLU" as its own honest downside
- Sigmoid → binary output (ml1-5) · Tanh → RNN hidden layers (nn1-8) · ReLU → most hidden layers by default
- **Next chapter:** Forward Propagation, Loss Functions & Backpropagation

CHAPTER 5 OF 11

Forward Propagation, Loss Functions & Backpropagation

Neural Networks & Deep Learning

Chapter 5 · Forward Propagation, Loss Functions & Backpropagation

`nn1-3` proved a multi-layer network *can* solve XOR, using weights a human worked out by hand. `historyai3-3` named "1986 backpropagation" as part of AI's own narrated history without ever explaining the mechanism. This chapter delivers both — the real algorithm that finds those weights automatically, for any network, from data alone.

Forward Propagation — Formally Naming What Every Example Already Did

Forward propagation is simply running the network forward: input values flow into the first layer, each neuron computes its weighted sum plus activation, those outputs become the next layer's inputs, and so on until the final output layer produces a prediction. Every worked example since `nn1-1` has already done this — this chapter just gives it a name before building on it.

Loss Functions — Cross-Entropy, Classification's Own Counterpart to MSE

`ml1-4`'s own MSE measured regression error. Classification networks typically use a different loss: **cross-entropy**, which specifically measures how far a predicted probability distribution sits from the true, correct answer.

WHY NOT JUST REUSE MSE?

Cross-entropy could technically be replaced with MSE for a classification task, but the real, honest reason it isn't: cross-entropy grows sharply as the predicted probability for the *correct* class approaches zero — a confident, wrong prediction is penalized far more heavily than MSE would penalize it. This produces a much stronger, more useful gradient signal exactly when the network is most wrong and most confident about it — precisely the situation where learning needs to happen fastest.

The Chain Rule — How Blame Travels Backward

A multi-layer network is a chain of functions — one layer's output feeding the next. If a small change to an early-layer weight changes that layer's output, which changes the next layer's output, which changes the final loss, the total effect of that one weight on the final loss is the *product* of each individual step's own local

effect, chained together — precisely the calculus **chain rule**. Conceptually: if `A` affects `B`, and `B` affects `C`, then `A`'s own effect on `C` is `A`'s effect on `B`, multiplied by `B`'s effect on `C`.

Backpropagation — The Real Algorithm

1. **Forward pass.** Run the network forward (as above), compute the prediction, compute the loss against the true answer.
2. **Compute the output layer's own gradient.** Using the loss function's own derivative, work out how much a small change to each output-layer weight would change the loss.
3. **Propagate backward, layer by layer.** Using the chain rule, compute each earlier layer's own contribution to that same final loss — each layer's own gradient calculation reuses the gradient already computed one layer downstream.
4. **Update every weight.** Once every layer's own gradient is known, adjust each weight via gradient descent.

This is the real, 1986 breakthrough (Rumelhart, Hinton, and Williams) — a general, automatic method for training a network of *any* depth, not a hand-derivation limited to one small, already-understood problem like `nn1-3`'s own XOR example. This is the piece Minsky and Papert's own 1969 book (`nn1-2`) left as an open question, resolved for real seventeen years later.

Gradient Descent — The Actual Update Rule

```
new_weight = old_weight - learning_rate * gradient
```

The **learning rate** controls how large each step is. Too high, and weight updates overshoot, bouncing past good solutions or diverging entirely; too low, and training crawls, taking an impractically long time to converge. `nn1-6` covers this, and several real variants of the basic gradient-descent rule, in practical depth.

Closing nn1-3's Own Deferred Promise

WHAT ACTUALLY HAPPENS NOW, FOR REAL

Trained via backpropagation on labeled XOR examples, a two-layer network with randomly initialized weights will converge to a set of weights that correctly solves XOR — not necessarily `nn1-3`'s own exact OR/NAND decomposition, but a mathematically valid solution reached automatically, with no

human ever needing to work out the underlying logical structure by hand. `nn1-3`'s own chapter proved a solution exists; this chapter is what actually finds one.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning, why cross-entropy produces a stronger gradient signal than MSE specifically when a network is confidently wrong, and why that specific situation is exactly when a strong signal matters most.

View solution

EXERCISE 2

Using this chapter's own A-affects-B-affects-C explanation of the chain rule, explain why computing an early layer's own contribution to the final loss requires information from every layer between it and the output, not just the early layer alone.

View solution

EXERCISE 3

Explain precisely what backpropagation actually resolved that nn1-3's own worked example couldn't, and why "a mathematically valid solution, not necessarily nn1-3's own exact one" is the accurate way to describe what training would find.

View solution

CHAPTER 5 QUICK REFERENCE

- **Forward propagation** — running the network forward to produce a prediction, already used implicitly since nn1-1
- **Cross-entropy** — classification's own counterpart to ml1-4's MSE, penalizing confident wrong answers much more heavily
- **Chain rule** — A's effect on C is A's effect on B times B's effect on C, chained across every layer
- **Backpropagation** — forward pass, compute output-layer gradient, propagate backward via the chain rule, update every weight — the real 1986 breakthrough, delivering historyai3-3's own namecheck
- **Gradient descent** — new_weight = old_weight − learning_rate × gradient; nn1-6 covers this in practical depth
- Closes nn1-3's own promise: a network can now find valid XOR weights automatically, with no hand-derivation required

- **Next chapter:** Training in Practice — Regularization for Neural Networks

CHAPTER 6 OF 11

Training in Practice: Regularization for Neural Networks

Neural Networks & Deep Learning

Chapter 6 · Training in Practice: Regularization for Neural Networks

`nn1-5` gave the mechanism. This chapter covers how it's actually run at real scale, and how to stop a network with enormous capacity from doing exactly what `ml1-7`'s own unconstrained decision tree did — memorizing the training set instead of learning from it.

Batch, Stochastic & Mini-Batch Gradient Descent

VARIANT	GRADIENT COMPUTED FROM	TRADE-OFF
Batch GD	The entire training set, every update	Accurate, stable gradient — slow, memory-heavy, one update per full pass
Stochastic GD (SGD)	One random example at a time	Noisy, but the noise itself genuinely helps escape shallow local minima
Mini-batch GD	A small batch (32–128 examples)	The practical standard — balances stability against update frequency, and maps efficiently onto GPU parallelism

Mini-batch is the near-universal real-world default specifically because a batch of, say, 64 examples can be processed *simultaneously* on GPU hardware — the same parallel-computation advantage `nn1-10`'s own framework tour will cover directly.

Epochs & Learning Curves

One full pass through the entire training set is one **epoch**. Training runs for many epochs, and plotting loss on the training and validation sets against epoch number is exactly `ml1-8`'s own learning curve, applied here directly — the same diagnostic tool, the same reading (a persistent gap between the two curves signals overfitting; both curves converging to a poor value signals underfitting).

Learning Rate Schedules

`nn1-5` already named the learning rate's own basic trade-off. In practice, many training runs use a **schedule** — starting with a larger learning rate for fast early progress, then gradually shrinking it as training proceeds, allowing the network to settle into a good solution more precisely once it's already in the right general area.

Why Neural Networks Overfit — ml1-8's Own Framing, New Model Family

A network with enough layers and neurons has an enormous number of learnable parameters — genuinely enough capacity to memorize a training set outright, exactly the same failure mode `ml1-7` 's own unconstrained decision tree demonstrated most visibly. `ml1-8` 's own bias-variance tradeoff applies unchanged: more capacity can mean lower bias and higher variance, and every technique below is a deliberate, controlled trade of one against the other, for this specific model family.

Dropout — A Genuine Cousin of ml1-7's Own Random Forest

Dropout randomly "turns off" (zeros out) a fraction of a layer's own neurons on each individual training pass — a different random subset every time. No single neuron can be relied upon to always be present, so the network is forced to spread useful information across many redundant pathways rather than concentrating it in a few neurons that happen to fit training-set quirks especially well.

> **THE SAME UNDERLYING IDEA AS ML1-7'S OWN RANDOM FOREST**
>
> `ml1-7` 's own random forest trains many trees, each on a random subset of data and features, then averages them — idiosyncratic per-tree errors cancel, genuine signal survives. Dropout achieves something genuinely similar within a *single* network: each training pass effectively trains a slightly different, randomly-thinned sub-network, and the fully-connected network used at inference time behaves like an implicit average across all of them. Different mechanism, same underlying spirit — enforced diversity and redundancy fighting overfitting.

Early Stopping — Catching Overfitting Live

`ml1-8` 's own overfitting signature — low training error, rising validation error — doesn't have to be diagnosed only after the fact. **Early stopping** monitors `ml1-2` 's own validation set during training itself and halts once validation loss starts climbing even as training loss keeps falling, keeping the weights from the point right before overfitting began rather than the final, most-overfit epoch.

Combining the Toolkit

> **THESE AREN'T MUTUALLY EXCLUSIVE**
>
> Dropout, early stopping, and `ml1-8` 's own L1/L2 regularization (often called "weight decay" in a neural-network context) are all routinely combined in the same training run — each addresses a genuinely different angle of the same

underlying bias-variance tradeoff, and using more than one is standard practice, not redundant.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own compare-table, why mini-batch gradient descent became the practical standard rather than pure batch or pure stochastic gradient descent.

View solution

EXERCISE 2

Explain the specific similarity this chapter draws between dropout and ml1-7's own random forest, being precise about what's actually similar (the underlying fight against overfitting) and what's genuinely different (the mechanism).

View solution

EXERCISE 3

Explain why early stopping is described as catching ml1-8's own overfitting signature "live" rather than diagnosing it after the fact, and explain what specifically early stopping does once it detects that signature.

View solution

CHAPTER 6 QUICK REFERENCE

- **Batch / Stochastic / Mini-batch GD** — a stability-vs-speed tradeoff; mini-batch is the practical standard, enabling GPU parallelism
- **Epoch** — one full pass through training data; loss-per-epoch curves are ml1-8's own learning curves, applied here
- A network's own huge capacity can memorize training data — ml1-7's own unconstrained-tree overfitting, in a new model family
- **Dropout** — randomly zeroing neurons per pass, a genuine cousin of ml1-7's own random-forest idea (different mechanism, same enforced-diversity spirit)
- **Early stopping** — halting on ml1-2's own validation loss the moment it starts rising, catching overfitting live
- Dropout, early stopping, and weight decay (ml1-8's own L1/L2) are routinely combined, not mutually exclusive
- **Next chapter:** Convolutional Neural Networks (CNNs)

CHAPTER 7 OF 11

Convolutional Neural Networks (CNNs)

Neural Networks & Deep Learning

Chapter 7 · Convolutional Neural Networks (CNNs)

Every network so far has been **fully connected** — every neuron sees every input. This chapter covers why that breaks down for images, and delivers `historyai3-4`'s own AlexNet story with the actual technical substance that narrative left out.

Why a Plain Fully-Connected Network Struggles With Images

A modest `224×224` color image already has `224 × 224 × 3 ≈ 150,000` input values. A fully-connected first hidden layer, even a modest one, would need roughly 150,000 weights *per neuron* — enormous, expensive, and prone to exactly `nn1-6`'s own overfitting story given typical training-set sizes. Worse: a fully-connected layer treats every pixel independently, with no built-in notion that nearby pixels are related. Shift an object slightly in the frame, and the network has to essentially relearn the same pattern all over again at the new position — no built-in **translation invariance**.

Convolution & Kernels — Solving Both Problems at Once

A **kernel** (a small grid of learnable weights, commonly `3×3` or `5×5`) slides across the image, computing a small, local weighted sum at each position. Crucially, the *same* small set of weights is reused at every position — **parameter sharing**.

> **WHY THIS FIXES BOTH PROBLEMS DIRECTLY**
>
> Parameter sharing collapses the parameter count from "one weight per pixel" down to "one small kernel, applied everywhere" — dramatically fewer weights than a fully-connected layer needed. It also delivers translation invariance for free: a kernel trained to detect, say, a vertical edge detects that same edge wherever it appears in the image, because the identical weights are applied at every position rather than being learned separately for each one.

Feature Maps — Many Kernels, Many Views

Applying one kernel across the whole image produces one **feature map** — a grid showing where that kernel's own pattern was detected, and how strongly. A real convolutional layer runs many kernels in parallel — one might learn to detect vertical edges, another horizontal edges, another a particular color transition or texture

— each producing its own feature map, together giving the network many simultaneous "views" of the same image.

Pooling — Downsampling for Efficiency and Further Invariance

Max pooling shrinks a feature map by taking the maximum value within each small region (commonly `2×2`), reducing the spatial size passed to later layers. This cuts computation for everything downstream, and adds a further degree of translation invariance — a small shift in exactly where a feature sits within its own pooling region no longer changes the output at all.

Stacking Layers — From Edges to Objects

Exactly `nn1-3`'s own "depth composes transformations" principle, now made concrete visually: early convolutional layers learn simple features (edges, color blobs); later layers combine those into increasingly complex, abstract features (textures, shapes, eventually recognizable object parts and whole objects) — a genuine, hierarchical feature-learning story built entirely from stacking the same basic operation.

Delivering historyai3-4's Own AlexNet Story, Technically

THREE CHAPTERS, ONE REAL HISTORICAL RESULT

AlexNet (2012) won ImageNet by a landslide margin using an eight-layer CNN — and three specific technical choices, each one this course has now already covered, are real, documented reasons it succeeded where earlier, shallower attempts hadn't: **ReLU** (`nn1-4`) instead of sigmoid/tanh, avoiding the saturation that would have crippled training at that depth; **dropout** (`nn1-6`) to control overfitting in a network with millions of parameters; and training on **GPUs** using mini-batches (`nn1-6`), making a network of that depth practically trainable within a realistic timeframe at all.

An Honest Nod to imgai1-2's Own Diffusion U-Net

`imgai1-2`'s own diffusion explainer described a denoising network without detailing its internal architecture. Worth naming honestly here, without overclaiming: the U-Net architecture commonly used inside a diffusion model's own denoising network is built from the same convolutional building blocks this chapter covers — convolution, feature maps, downsampling — arranged into a specific downsample-then-upsample shape suited to producing a full image rather than a single classification. Not the identical architecture this chapter describes step by step; the same underlying convolutional vocabulary, applied to a different job.

What's Next

CNNs assume 2D spatial structure — nearby pixels matter, position within the frame is what matters. Text and time-series data have a different structural assumption entirely: *order* matters, not 2D position. `nn1-8` covers the architecture family built specifically for that.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning, how parameter sharing in a convolutional kernel solves both the parameter-count problem and the translation-invariance problem at the same time.

View solution

EXERCISE 2

Using this chapter's own finding-box, explain specifically what role ReLU, dropout, and GPU-based mini-batch training each played in AlexNet's own success, and identify which earlier chapter introduced each technique.

View solution

EXERCISE 3

Explain why this chapter is careful to describe the diffusion U-Net as using "the same convolutional building blocks... arranged differently" rather than claiming it's the identical architecture this chapter just walked through.

View solution

CHAPTER 7 QUICK REFERENCE

- Fully-connected layers scale badly to images — huge parameter counts, no translation invariance
- **Kernel** — a small, learnable filter, reused at every position (parameter sharing) — solves both problems at once
- **Feature map** — one kernel's output across the whole image; many kernels run in parallel per layer
- **Pooling** (max pooling) — downsamples, cuts computation, adds further translation invariance
- Stacked layers build a hierarchy: edges → textures/shapes → object parts — nn1-3's own depth principle, made visual
- AlexNet's real 2012 success combined ReLU (nn1-4), dropout (nn1-6), and GPU mini-batch training (nn1-6)
- Diffusion's own U-Net (imgai1-2) uses this chapter's own convolutional vocabulary, arranged for a different job

- **Next chapter:** Recurrent Neural Networks (RNNs) & LSTMs

CHAPTER 8 OF 11

Recurrent Neural Networks (RNNs) & LSTMs

Neural Networks & Deep Learning

Chapter 8 · Recurrent Neural Networks (RNNs) & LSTMs

`nn1-7` covered data where 2D position matters. This chapter covers data where *order* matters instead — text, time series, audio — and delivers the real, worked-out version of `nn1-4`'s own vanishing-gradient cliffhanger, in exactly the setting that makes it most severe.

Why Neither Feedforward Networks Nor CNNs Naturally Handle Sequences

Every network so far processes one fixed-size input and produces one output, with no notion of "what came before." Predicting the next word in a sentence genuinely needs context from every preceding word, not just a fixed local window — a fundamentally different requirement from anything `nn1-1` – `nn1-7` were built to handle.

The RNN Idea — A Hidden State Carried Through Time

A **recurrent neural network** maintains a **hidden state** — a vector updated at every step of the sequence. At each time step, the network combines the current input with the previous hidden state to compute a new one, which then carries forward to the next step. This hidden state functions as a compressed memory of everything the network has seen in the sequence so far.

PARAMETER SHARING AGAIN — ACROSS TIME, NOT SPACE

The exact same weights are reused at *every* time step — a direct structural cousin of `nn1-7`'s own convolutional kernel, which reused the same weights at every spatial *position*. Here, the sharing happens across *time* instead of across space, but it's the identical underlying idea: one small, learnable transformation, applied repeatedly.

Unrolling Through Time — Backpropagation Through Time

Conceptually "unrolling" an RNN means drawing the same cell repeatedly, once per time step, each copy feeding its own output forward as the next copy's own input. Training happens via **backpropagation through time** (BPTT) — literally `nn1-5`'s own backpropagation and chain rule, applied across the unrolled time steps instead of across depth-wise layers.

Vanishing Gradients — nn1-4's Own Cliffhanger, Now Worse

`nn1-4` previewed saturation shrinking gradients toward zero. In an RNN, this becomes genuinely more severe, for a specific, mechanical reason: because the *same* weight is reused at every time step, backpropagation through time multiplies that *same* weight's own gradient contribution by itself, repeatedly, once per time step. If each individual factor is even slightly less than 1 — easily the case with saturating activations — the cumulative product shrinks toward zero exponentially fast across a long sequence.

WHY THIS IS WORSE THAN NN1-5'S OWN GENERAL CASE

`nn1-5`'s own chain rule multiplies together *different* weights at each layer — bad luck at any one layer doesn't guarantee bad luck at every other. An RNN multiplies the *same* weight by itself, over and over, across every time step — a single unfavorable value compounds relentlessly rather than merely contributing once. A plain RNN effectively "forgets" anything from more than a handful of time steps ago, because gradients from far in the past barely survive the trip back to influence early weights at all.

LSTMs — The Real, Documented Fix

Long Short-Term Memory networks introduce a separate **cell state** that flows through time with only carefully controlled modifications, plus three **gates** — small sigmoid-activated sub-networks that learn how much old information to forget, how much new information to add, and how much of the current state to actually output at each step.

WHY GATES ACTUALLY FIX THE VANISHING-GRADIENT PROBLEM

Instead of forcing the same fixed multiplicative shrinkage at every single time step regardless of content, the gates let the network *learn* when to preserve information essentially unchanged and when to actually update it. This provides a much more direct path for gradients to flow backward through the cell state across many time steps, largely avoiding the repeated same-weight-multiplied-by-itself collapse plain RNNs suffer from.

What Comes Next — A Different Kind of Limit

LSTMs fixed the vanishing-gradient problem. They didn't fix a separate, genuinely different limitation: an RNN (LSTM or not) is inherently *sequential* — each time step's own computation depends on the previous step's own output, which means, unlike `nn1-7`'s own CNN feature maps or `nn1-6`'s own parallel mini-batches, the steps within one single sequence can't be computed simultaneously. `nn1-9` covers the architecture that solved *this* specific bottleneck instead.

Hands-On Exercises

EXERCISE 1

Using this chapter's own tip-box, explain the precise structural parallel between an RNN's own parameter sharing across time and a CNN's own parameter sharing across space, from nn1-7.

View solution

EXERCISE 2

Explain, using this chapter's own reasoning, why the vanishing-gradient problem is genuinely worse for a plain RNN processing a long sequence than for nn1-5's own general multi-layer case.

View solution

EXERCISE 3

Explain why this chapter says LSTMs fixed the vanishing-gradient problem but did NOT fix RNNs' own inherently sequential nature, and explain why these are genuinely two separate limitations rather than the same one.

View solution

CHAPTER 8 QUICK REFERENCE

- **RNN** — a hidden state updated at every time step, the same weights reused at every step (parameter sharing across time)
- **BPTT** — nn1-5's own backpropagation, applied across unrolled time steps instead of layers
- Vanishing gradients — nn1-4's own cliffhanger, made worse: the same weight multiplied by itself repeatedly across long sequences
- **LSTM** — a separate cell state plus learned gates (forget/input/output), providing a direct path for gradients to survive long sequences
- LSTMs fixed vanishing gradients — they didn't fix RNNs' own inherently sequential, non-parallelizable computation
- **Next chapter:** A Transformer Preview

CHAPTER 9 OF 11

A Transformer Preview

Neural Networks & Deep Learning

Chapter 9 · A Transformer Preview

`nn1-8` closed on a real, unsolved bottleneck: LSTMs fixed vanishing gradients but never fixed RNNs' own inherently sequential computation. This chapter previews the architecture that solved that specific problem — and, along the way, delivers part of `historyai3-6`'s own "Attention Is All You Need" namecheck. The rest is `llm1`'s own job.

The Motivating Question

Can a network capture relationships between distant elements in a sequence — `nn1-8`'s own hidden/cell state's whole purpose — *without* forcing every step to wait for the one before it?

Self-Attention — A Direct, All-at-Once Relationship

Instead of carrying a compressed hidden state step by step, **self-attention** lets every position in a sequence directly look at every other position simultaneously, computing a weighted combination based on how relevant each other position actually is. Nothing is mediated through a chain of intermediate steps — the relationship between position 1 and position 50 is computed exactly as directly as the relationship between two adjacent positions.

A CONCRETE INTUITION

In "The cat sat on the mat because *it* was tired," resolving what "it" refers to requires directly relating it back to "cat." Self-attention lets the token "it" attend strongly and directly to "cat," regardless of how many words sit between them — no need to pass that information step-by-step through every intervening word's own hidden state, the way `nn1-8`'s own RNN would have had to.

Solving Both of nn1-8's Own Open Problems

NN1-8'S OWN OPEN PROBLEM	HOW SELF-ATTENTION ADDRESSES IT
Long-range dependencies	Direct attention between any two positions — no long chain of repeated multiplications (nn1-8's own vanishing-gradient mechanism) between distant positions at all
Sequential computation bottleneck	Attention for every position can be computed simultaneously, as one parallel operation — the exact structural fix nn1-8 closed on needing

The long-range-dependency fix is genuinely structural, not a patch layered on top the way `nn1-8`'s own LSTM gating was — there's no long chain for a gradient to vanish across in the first place, because the relationship is computed directly rather than relayed through many intermediate steps. The parallelization fix directly resolves `nn1-8`'s own closing cliffhanger, and aligns naturally with `nn1-6`'s own GPU-parallel training.

A Deliberately Honest Gap — Order

Self-attention, by itself, has no inherent notion of sequence order at all — computing how much "it" attends to "cat" doesn't naturally encode whether "cat" came before or after "it" in the sentence. Unlike an RNN, which processes tokens strictly in order by construction, a transformer needs a separate, explicit mechanism to inject position information back in — **positional encoding**, one of several real technical pieces this preview deliberately leaves for `llm1` to cover in full.

What's Deliberately Deferred to llm1

THIS IS A PREVIEW, NOT THE FULL MECHANISM

Multi-head attention, positional encoding in full, the encoder/decoder architecture, and the actual 2017 "Attention Is All You Need" paper this chapter's own title echoes — all of it is `llm1`'s own dedicated job. This chapter delivers the concept and the motivating "why"; `llm1` delivers the real mechanism and its role in modern language models.

Beyond Language

The same self-attention idea generalized well past its original text-focused motivation — vision transformers apply the identical mechanism to image patches instead of word tokens, a real, documented extension of the architecture beyond the problem it was originally built to solve. Not this course's own focus; worth knowing the architecture didn't stay confined to text.

Hands-On Exercises

EXERCISE 1

Using this chapter's own "it"/"cat" example, explain specifically why self-attention resolves that reference without repeating nn1-8's own step-by-step hidden-state relay, and why that difference matters for long sequences specifically.

View solution

EXERCISE 2

Using this chapter's own compare-table, explain why the long-range-dependency fix is described as "genuinely structural" while nn1-8's own LSTM gating is described as more of a patch, even though both address the same underlying vanishing-gradient concern.

View solution

EXERCISE 3

Explain why this chapter specifically flags positional encoding as a real gap self-attention has, rather than glossing over it, and explain why an RNN never needed an equivalent mechanism.

View solution

CHAPTER 9 QUICK REFERENCE

- **Self-attention** — every position directly attends to every other position at once, no relay through intermediate steps
- Fixes nn1-8's own two open problems: long-range dependencies (structurally, not patched) and the sequential-computation bottleneck (parallel by construction)
- A real, honest gap: self-attention alone has no inherent sense of order — **positional encoding** fixes this, deferred to llm1
- Multi-head attention, the encoder/decoder architecture, and the real 2017 paper are all llm1's own dedicated job
- The same mechanism generalized beyond text — vision transformers, briefly noted
- **Next chapter:** A Framework Tour: PyTorch vs. TensorFlow

CHAPTER 10 OF 11

A Framework Tour: PyTorch vs. TensorFlow

Neural Networks & Deep Learning

Chapter 10 · A Framework Tour: PyTorch vs. TensorFlow

`nn1-1` – `nn1-9` built the concepts. This chapter maps them onto real, working code — and covers the two dominant frameworks the rest of this course's own ecosystem (`llm1` included) is built on.

Two Frameworks, Two Different Starting Philosophies

TensorFlow (Google) originally used **define-then-run**: build the entire computation graph first, as a fixed structure, then feed data through it. Efficient for deployment and optimization, but genuinely awkward for debugging — you couldn't simply inspect an intermediate value mid-computation the way you'd step through ordinary Python. **PyTorch** (Meta) used **eager execution** — define-by-run — from the start: the computation graph is built dynamically as code actually executes, line by line, exactly like ordinary Python. This genuinely mattered for research and experimentation, and is a real, documented reason PyTorch became — and largely remains — the dominant framework in academic and research settings specifically.

An Honest Update — This Gap Has Narrowed

THE HISTORICAL FRAMING EXPLAINS REPUTATIONS, NOT CURRENT CAPABILITIES

TensorFlow 2.x adopted eager execution by default, following the field's own clear preference. PyTorch, in turn, added its own graph-based optimization and deployment tools (TorchScript, `torch.compile`). The define-then-run/define-by-run distinction explains *why* each framework built the ecosystem and reputation it has today — it no longer accurately describes either framework's own current capabilities in isolation.

The Current Practical Landscape

	STRONGEST TODAY IN
PyTorch	Research and academic papers, and increasingly production as well
TensorFlow	Mobile/edge deployment (TensorFlow Lite), browser deployment (TensorFlow.js), enterprises already invested in the ecosystem

A Real Small Network — Every Concept, Real Code

```
import torch
import torch.nn as nn
import torch.optim as optim

class SmallNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(2, 4)   # nn1-1's own hidden layer
        self.output = nn.Linear(4, 1)   # nn1-1's own output layer
        self.relu = nn.ReLU()           # nn1-4's own activation function

    def forward(self, x):
        x = self.relu(self.hidden(x))   # nn1-1's own forward propagation
        return torch.sigmoid(self.output(x))   # ml1-5's own sigmoid, one more time

model = SmallNetwork()
loss_fn = nn.BCELoss()                          # cross-entropy for binary output (nn1-5)
optimizer = optim.SGD(model.parameters(), lr=0.1)   # nn1-6's own gradient descent, with a learning
rate

for epoch in range(1000):                       # nn1-6's own epochs
    predictions = model(X_train)                # forward pass
    loss = loss_fn(predictions, y_train)
    optimizer.zero_grad()
    loss.backward()                              # nn1-5's own backpropagation, one line
    optimizer.step()                             # the weight update itself
```

EVERY LINE TRACES BACK TO A CHAPTER ALREADY COVERED

`nn.Linear` is `nn1-1`'s own neuron layer, generalized. `nn.ReLU` is `nn1-4`'s own activation choice. `loss.backward()` is `nn1-5`'s own backpropagation — one line, doing exactly the chain-rule-driven gradient computation that chapter explained conceptually. `optimizer.step()` is `nn1-6`'s own gradient descent update rule, applied automatically. Nothing here is new material — it's this course's own first nine chapters, in real syntax.

Hands-On Exercises

EXERCISE 1

Explain the real, historical reason PyTorch's own define-by-run approach mattered specifically for research and experimentation, using this chapter's own reasoning about debugging.

View solution

EXERCISE 2

Explain why this chapter says the define-then-run/define-by-run distinction "explains reputations, not current capabilities," using the two specific updates (TensorFlow 2.x, TorchScript/torch.compile) this chapter names.

View solution

EXERCISE 3

Using this chapter's own code example, identify which specific earlier chapter each of loss.backward() and optimizer.step() corresponds to, and explain what each line is actually doing under the hood.

View solution

CHAPTER 10 QUICK REFERENCE

- **TensorFlow** — originally define-then-run (a static graph); **PyTorch** — define-by-run (eager) from the start, favoring debugging/research
- Honest update: TensorFlow 2.x defaults to eager execution; PyTorch added graph-based tools (TorchScript, torch.compile) — the gap has narrowed
- PyTorch dominates research/academic work today; TensorFlow remains strong for mobile/edge and browser deployment
- Real PyTorch code: `nn.Linear` / `nn.ReLU` (nn1-1/nn1-4), `loss.backward()` (nn1-5's backprop), `optimizer.step()` (nn1-6's gradient descent)
- **Next chapter:** Capstone: Building and Training a Real Neural Network

CHAPTER 11 OF 11

Capstone: Building and Training a Real Neural Network

Neural Networks & Deep Learning

Chapter 11 · Capstone: Building and Training a Real Neural Network

One real dataset, already familiar: `ds1-10`'s own employee-attrition table, already fit with `ml1-5`'s logistic regression and `ml1-7`'s random forest. This capstone builds a real, working neural network on the exact same problem — and asks the honest question this course owes a real answer to: does going deeper actually help here?

The Full Pipeline

```
import torch
import torch.nn as nn
import torch.optim as optim

X = pd.get_dummies(df[["department", "age", "years_at_company", "salary"]])
y = df["left_company"].map({"Yes": 1, "No": 0})
# train/val/test split — ml1-2, nn1-6
# StandardScaler — ds1-2/ml1-3

class AttritionNet(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.hidden1 = nn.Linear(n_features, 16)
        self.hidden2 = nn.Linear(16, 8)
        self.dropout = nn.Dropout(0.3)     # nn1-6
        self.output = nn.Linear(8, 1)
        self.relu = nn.ReLU()              # nn1-4

    def forward(self, x):
        x = self.relu(self.hidden1(x))
        x = self.dropout(x)
        x = self.relu(self.hidden2(x))
        return torch.sigmoid(self.output(x))   # nn1-1's own closing callback — see below

model = AttritionNet(n_features=X.shape[1])
loss_fn = nn.BCELoss()                                  # nn1-5
optimizer = optim.Adam(model.parameters(), lr=0.01)        # nn1-6
```

```
best_val_loss = float("inf")
for epoch in range(200):
    # forward pass, loss, backward(), step() — nn1-5/nn1-6
    # track train_loss and val_loss each epoch — nn1-6's own learning curve
    # early stopping — nn1-6: save weights when val_loss improves
```

NN1-1'S OWN CLOSING CALLBACK

That final `torch.sigmoid(self.output(x))` line is, structurally, exactly `nn1-1`'s own opening claim made real: the output layer of this entire network is one sigmoid-activated neuron — `ml1-5`'s own logistic regression, unchanged, just fed a richer, network-transformed set of inputs instead of the raw features directly.

Evaluating — The Same Metrics, For a Fair Comparison

```
from sklearn.metrics import precision_score, recall_score, f1_score
preds = (model(X_test) > 0.5).int()   # ml1-6's own threshold, applied here too
precision_score(y_test, preds), recall_score(y_test, preds), f1_score(y_test, preds)
```

`ml1-6`'s own precision/recall/F1 apply completely unchanged — the same evaluation vocabulary works identically regardless of which model produced the predictions.

The Honest Finding

DOES THE NEURAL NETWORK ACTUALLY WIN HERE?

On this small, simple tabular dataset, the neural network's own precision/recall/F1 plausibly come back *roughly comparable to* — not dramatically better than — `ml1-5`'s logistic regression and `ml1-7`'s random forest, and it required considerably more code, more hyperparameters, and more training time to get there.

THIS IS A REAL, IMPORTANT, WELL-DOCUMENTED PRACTICAL TRUTH

Neural networks' genuine advantages show up on large datasets and complex, high-dimensional data — images (`nn1-7`'s own AlexNet), long sequences (`nn1-8`/`nn1-9`) — not necessarily small, simple tabular problems like this one, where a handful of numeric and categorical features rarely benefit much from the kind of hierarchical feature transformation `nn1-3` and `nn1-7` made such a strong case for. This isn't a knock against neural networks — it's a genuine, useful piece of practical judgment: `ml1`'s own simpler models remain the right first choice for a great many real problems, not merely "the old stuff before deep learning got invented."

Chapter Attribution

CAPSTONE ELEMENT	DRAWN FROM
The neuron-as-logistic-regression closing callback	nn1-1
Hidden layers enabling nonlinear feature transformation	nn1-3
ReLU activation, dropout regularization	nn1-4 / nn1-6
Binary cross-entropy loss, backward()	nn1-5
Adam optimizer, epochs, early stopping	nn1-6
Real PyTorch syntax throughout	nn1-10
Precision/recall/F1 evaluation, direct comparison	ml1-6 / ml1-5 / ml1-7
The dataset itself	ds1-10

Honest Scope Note

WHAT THIS CAPSTONE — AND THIS COURSE — DELIBERATELY DOESN'T ATTEMPT

- **No CNN, RNN, or transformer applied here.** This capstone's own dataset is tabular — a genuinely different data shape from the images (`nn1-7`) and sequences (`nn1-8` / `nn1-9`) those architectures were specifically built for. Forcing one of them onto tabular data here would be architecturally dishonest, not a real demonstration.
- **No production deployment or MLOps.** Matching `ml1-11`'s own precedent — this capstone stops at a trained, evaluated model.
- **Full transformer/attention depth remains deferred.** `nn1-9` previewed the concept; `llm1` delivers the real mechanism.
- **Hyperparameter tuning is only lightly touched.** Systematic search over architecture size, learning rate, and dropout rate is a real, substantial practice this course doesn't attempt in depth.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own tip-box, precisely how the network's own final output layer closes the loop back to nn1-1's own opening claim about a single neuron.

View solution

EXERCISE 2

Explain why this chapter's own honest finding (comparable, not dramatically better, performance) is described as "a real, important, well-documented practical truth" rather than a disappointing result, and identify the kind of data where neural networks' own real advantage actually shows up.

View solution

EXERCISE 3

Explain why this chapter's own scope note calls forcing a CNN or RNN onto this capstone's dataset "architecturally dishonest," using this chapter's own reasoning about data shape.

View solution

CHAPTER 11 QUICK REFERENCE — COURSE SUMMARY

- A single neuron is ml1-5's own logistic regression (nn1-1); stacking hidden layers solves what a single neuron structurally can't (nn1-2/nn1-3)
- Activation functions (nn1-4), backpropagation (nn1-5), and practical training/regularization (nn1-6) are the real mechanism behind every trained network
- CNNs (nn1-7) and RNNs/LSTMs (nn1-8) are specialized architectures for spatial and sequential data respectively; transformers (nn1-9) fixed what LSTMs couldn't
- Real code (nn1-10) maps every concept onto actual PyTorch syntax
- The honest finding: deep learning isn't automatically the right tool for every problem — ml1's own simpler models remain genuinely competitive on small, tabular data
- Next up in the Data Science & ML subject: **nlp1** — building toward "why LLMs are different," the direct bridge into llm1's own full transformer coverage