# PHASE 2
# Deep Learning Foundations

Neurons, backpropagation, optimization, regularization, and PyTorch — from first principles

**Phase goal**

Understand neural networks from the ground up — forward pass, backpropagation, optimization — *before* the framework does it for you. By the end you have hand-derived backprop and trained the same network in raw NumPy and PyTorch.

**Estimated time:** 6–8 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

A self-paced, depth-first path from mathematical foundations to production deep learning.

# Contents

# 1 From linear models to neural networks

A neural network is a stack of *learned* feature transformations followed by a linear model. Each layer applies an affine map then a nonlinearity; composing many of them lets the network represent functions that no single linear model can. Everything you learned in Phases 0–1 carries over: dot products, gradients, cross-entropy, regularization, the bias–variance tradeoff. The new ingredient is **depth**, and the new skill is computing gradients through many composed functions: backpropagation.

**Key idea**

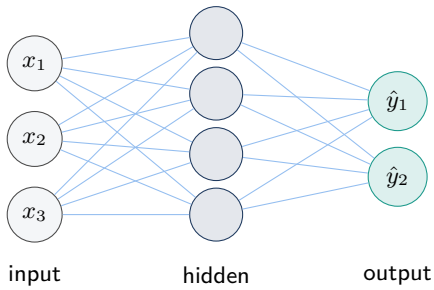
A neural network is just a big differentiable function $f_{\boldsymbol{\theta}}(\mathbf{x})$ with millions of parameters $\boldsymbol{\theta}$. We pick a loss, compute $\nabla_{\boldsymbol{\theta}} L$ by the chain rule (backprop), and descend. That is the *entire* algorithm. Phases 3–6 only change the *architecture* of $f$; the training loop stays the same.

# 2 Neural network basics

## 2.1 The perceptron and the MLP

A single artificial neuron computes $a = \phi(\mathbf{w}^\top \mathbf{x} + b)$ — a dot product, a bias, and a nonlinearity $\phi$. A **multilayer perceptron (MLP)** stacks layers of these:

$$\mathbf{h}^{(1)} = \phi(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), \quad \mathbf{h}^{(2)} = \phi(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}), \quad \ldots, \quad \hat{\mathbf{y}} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}.$$

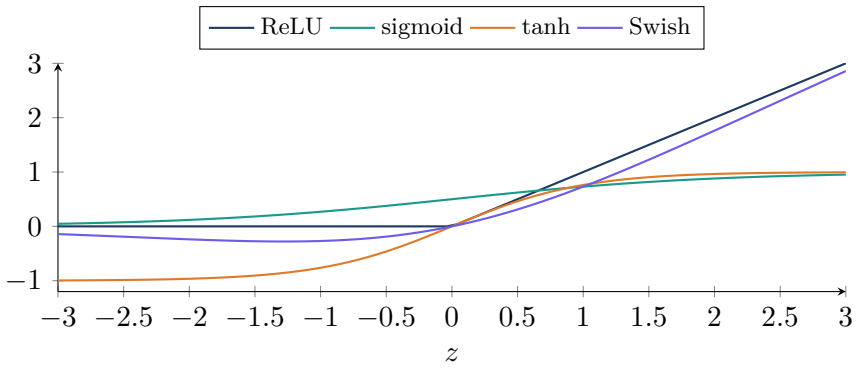



**Intuition**

Without a nonlinearity, stacking layers collapses: $\mathbf{W}^{(2)}(\mathbf{W}^{(1)}\mathbf{x}) = (\mathbf{W}^{(2)}\mathbf{W}^{(1)})\mathbf{x}$ is still linear. The nonlinearity is what gives depth its power. The **universal approximation theorem** says even one hidden layer (wide enough) can approximate any continuous function — but *depth* makes this efficient, reusing features hierarchically.

## 2.2 Activation functions

| Activation | Definition | Notes |
|---|---|---|
| Sigmoid | $\sigma(z) = \frac{1}{1+e^{-z}}$ | saturates, vanishing gradients; use only at output |
| Tanh | $\tanh(z)$ | zero-centered sigmoid; still saturates |
| ReLU | $\max(0, z)$ | default; cheap, no saturation for $z > 0$; "dying ReLU" |
| Leaky ReLU | $\max(\alpha z, z)$ | small slope $\alpha$ for $z < 0$ fixes dying ReLU |
| GELU | $z\,\Phi(z)$ | smooth; standard in Transformers |
| Swish/SiLU | $z\,\sigma(z)$ | smooth, often $\geq$ ReLU in deep nets |




### 2.3 Loss functions

Regression uses MSE; classification uses cross-entropy on top of a softmax. Both are negative log-likelihoods (Phase 0). For numerical stability, frameworks fuse softmax and cross-entropy into one op (`CrossEntropyLoss` takes raw logits, not probabilities).

## 3 Backpropagation — derive it once by hand

Backprop is the chain rule applied systematically over a computational graph, reusing intermediate results so the cost of all gradients equals a constant multiple of one forward pass.

**Definition**

For a layer $\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b}$, $\mathbf{h} = \phi(\mathbf{z})$, given the upstream gradient $\boldsymbol{\delta} = \partial L/\partial \mathbf{h}$, backprop computes:

$$\frac{\partial L}{\partial \mathbf{z}} = \boldsymbol{\delta} \odot \phi'(\mathbf{z}), \quad \frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{z}}\,\mathbf{a}^\top, \quad \frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{z}}, \quad \frac{\partial L}{\partial \mathbf{a}} = \mathbf{W}^\top \frac{\partial L}{\partial \mathbf{z}},$$

where $\odot$ is elementwise product. The last term is the gradient passed to the previous layer — the recursion.

**Two-layer network, end to end**

Network: $\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$, $\mathbf{a}_1 = \mathrm{ReLU}(\mathbf{z}_1)$, $\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2$, $\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{z}_2)$, loss $L = -\sum_k y_k \log \hat{y}_k$ (one-hot $\mathbf{y}$).

1. **Output gradient** (the famous clean result of softmax+CE): $\dfrac{\partial L}{\partial \mathbf{z}_2} = \hat{\mathbf{y}} - \mathbf{y}$.
2. **Layer 2:** $\dfrac{\partial L}{\partial \mathbf{W}_2} = (\hat{\mathbf{y}} - \mathbf{y})\,\mathbf{a}_1^\top$, $\dfrac{\partial L}{\partial \mathbf{b}_2} = \hat{\mathbf{y}} - \mathbf{y}$.
3. **Backprop to hidden:** $\dfrac{\partial L}{\partial \mathbf{a}_1} = \mathbf{W}_2^\top(\hat{\mathbf{y}} - \mathbf{y})$, then $\dfrac{\partial L}{\partial \mathbf{z}_1} = \dfrac{\partial L}{\partial \mathbf{a}_1} \odot \mathbb{1}[\mathbf{z}_1 > 0]$.
4. **Layer 1:** $\dfrac{\partial L}{\partial \mathbf{W}_1} = \dfrac{\partial L}{\partial \mathbf{z}_1}\,\mathbf{x}^\top$, $\dfrac{\partial L}{\partial \mathbf{b}_1} = \dfrac{\partial L}{\partial \mathbf{z}_1}$.

Do this derivation on paper at least once. The pattern — multiply by the local derivative, then push back through $\mathbf{W}^\top$ — is *all* of backprop.

**Intuition**

The cleanness of $\partial L/\partial \mathbf{z}_2 = \hat{\mathbf{y}} - \mathbf{y}$ is not a coincidence: it is why softmax pairs with cross-entropy and sigmoid pairs with binary cross-entropy. The gradient is simply "prediction minus target," which is also the gradient of linear regression's MSE — a deep unity across models.

### 3.1 Weight initialization

If weights are too large, activations explode; too small, they vanish. Principled schemes keep the variance of activations (and gradients) roughly constant across layers:

- **Xavier/Glorot** (for tanh/sigmoid): $\mathrm{Var}(w) = \frac{2}{n_{\text{in}}+n_{\text{out}}}$.
- **He** (for ReLU): $\mathrm{Var}(w) = \frac{2}{n_{\text{in}}}$ — accounts for ReLU zeroing half the inputs.

**Common pitfall**

Never initialize all weights to the same constant (e.g. zero): every neuron in a layer then computes the identical gradient and stays identical forever — the *symmetry-breaking* failure. Random init breaks symmetry. Getting initialization wrong is a classic reason a network "won't train" even though the code is correct.

## 4 Optimization

### 4.1 Gradient descent variants

**Batch** GD uses the whole dataset per step (accurate, slow, memory-heavy). **Stochastic** GD (SGD) uses one example (noisy, fast). **Mini-batch** (32–512 examples) is the practical sweet spot — it exploits vectorized hardware and the gradient noise actually helps generalization.

### 4.2 Momentum and adaptive methods

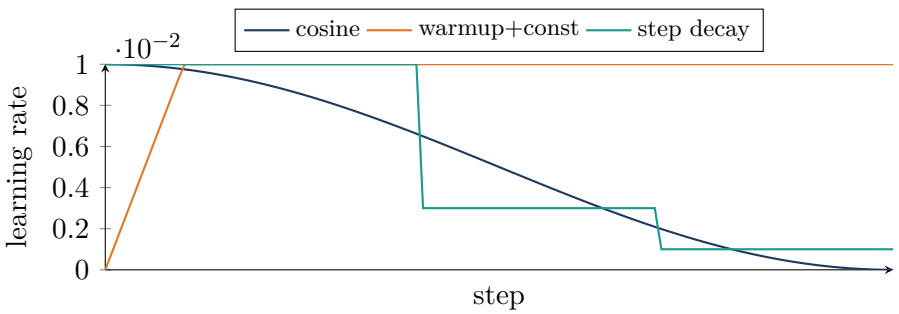
$$\begin{aligned}
\text{Momentum:}\quad & \mathbf{v}_t = \beta \mathbf{v}_{t-1} + \nabla L, \quad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \mathbf{v}_t \\
\text{RMSProp:}\quad & s_t = \rho s_{t-1} + (1-\rho)(\nabla L)^2, \quad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla L / \sqrt{s_t + \epsilon} \\
\text{Adam:}\quad & \text{momentum} + \text{RMSProp, with bias correction}
\end{aligned}$$

**Key idea**

**Adam** (and **AdamW**) is the default optimizer for deep learning. Momentum accelerates along consistent directions and damps oscillation (fixing the zig-zag you saw in Phase 0); RMSProp adapts the per-parameter step size. **AdamW** decouples weight decay from the gradient update and is the correct choice for Transformers. Start with Adam/AdamW; reach for tuned SGD+momentum when squeezing the last accuracy on vision models.

### 4.3 Learning-rate scheduling

The single most important hyperparameter is the learning rate. Common schedules: **step decay** (cut by 10× at milestones), **cosine annealing** (smooth decay to near zero), and **warmup** (ramp up over the first few hundred steps — essential for Transformers to avoid early divergence). A **learning-rate finder** (sweep LR and watch loss) quickly brackets a good value.



### 4.4 Vanishing/exploding gradients and clipping

In deep or recurrent nets, repeated multiplication by Jacobians can shrink gradients to zero (vanishing) or blow them up (exploding). Mitigations: ReLU-family activations, careful init, normalization layers, residual connections (Phase 3), and **gradient clipping** (cap the gradient norm) — standard practice when training RNNs and Transformers.

## 5 Regularization & generalization

Deep nets have enormous capacity, so controlling overfitting is central.

- **Dropout:** randomly zero a fraction of activations during training; an implicit ensemble that prevents co-adaptation. Disabled at inference.
- **Weight decay (L2):** penalize large weights; in AdamW this is decoupled from the adaptive scaling.
- **Early stopping:** halt when validation loss stops improving — cheap and effective.
- **Data augmentation:** expand the effective dataset with label-preserving transforms (crucial in vision, Phase 3).

### 5.1 Normalization layers

**Batch normalization** normalizes each feature across the mini-batch, stabilizing and accelerating training (and acting as a mild regularizer). **Layer normalization** normalizes across features within each example — batch-size independent and the standard in Transformers and RNNs.

**Common pitfall**

BatchNorm behaves differently in train vs. eval mode (it uses running statistics at inference). Forgetting `model.eval()` (and `torch.no_grad()`) at inference is a top source of mysterious metric drops. BatchNorm also misbehaves with very small batches — prefer GroupNorm/LayerNorm there.

## 6 Frameworks

### 6.1 Autograd and computational graphs

Frameworks record operations into a graph and apply backprop automatically. PyTorch builds the graph *dynamically* (define-by-run), so models are plain Python — easy to debug and to write custom architectures. Each tensor carries `requires_grad`; calling `.backward()` populates `.grad`.

### 6.2 PyTorch — the recommended primary framework

PyTorch is transparent, dominant in research, and the path of least resistance for custom architectures (everything in Phases 3–6). The canonical training loop is worth memorizing:

```
import torch, torch.nn as nn

model = nn.Sequential(nn.Linear(784, 256), nn.ReLU(),
                      nn.Linear(256, 10))            # logits
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)
loss_fn = nn.CrossEntropyLoss()                       # expects raw
    logits

for epoch in range(epochs):
```

```
    model.train()
    for xb, yb in train_loader:
        opt.zero_grad()                 # 1. clear old gradients
        logits = model(xb)              # 2. forward pass
        loss = loss_fn(logits, yb)      # 3. compute loss
        loss.backward()                 # 4. backprop (autograd)
        opt.step()                      # 5. update parameters
    model.eval()
    with torch.no_grad():
        ...                             # validation, no graph built
```

**Intuition**

Those five lines — zero, forward, loss, backward, step — are the heartbeat of *all* deep learning, from MNIST to GPT. Everything else (data loading, architectures, schedulers, mixed precision) wraps around this loop.

### 6.3 TensorFlow / Keras

Worth recognizing: Keras offers a concise high-level API, and TensorFlow has strong production/mobile tooling (TF Lite, TF Serving) — relevant if you ever need on-device inference. You do not need to learn it deeply now; concepts transfer directly from PyTorch.

## 7 Checkpoint project

**A neural net from scratch, then in PyTorch**

**Part A — raw NumPy, manual backprop (no autograd).** Implement a 2-layer MLP for MNIST:

- forward pass (linear → ReLU → linear → softmax), cross-entropy loss;
- manual backprop using the worked example above;
- mini-batch SGD with He initialization;
- a **gradient check**: compare your analytic gradients to numerical $\frac{L(\theta+\epsilon)-L(\theta-\epsilon)}{2\epsilon}$ — they must agree to ∼1e-6. This single test catches almost every backprop bug.

Target ≥95% test accuracy and a smoothly decreasing loss curve.

**Part B — reimplement in PyTorch.** The same architecture using `nn.Module` and `autograd`. Confirm you get comparable accuracy with far less code, then experiment: swap SGD→Adam, add dropout and weight decay, add a LR schedule, and plot how each affects the validation curve.

**Deliverables:** both implementations, the gradient-check output, overlaid loss/accuracy curves, and a short reflection on what autograd did for you and what each regularizer changed.

```
import numpy as np
def softmax(z):
    z = z - z.max(1, keepdims=True)          # numerical stability
    e = np.exp(z); return e / e.sum(1, keepdims=True)

def forward(X, W1, b1, W2, b2):
    Z1 = X @ W1 + b1; A1 = np.maximum(0, Z1)
    Z2 = A1 @ W2 + b2; P = softmax(Z2)
    return Z1, A1, Z2, P
```

```
def backward(X, Y, Z1, A1, P, W2):           # Y is one-hot
    n = X.shape[0]
    dZ2 = (P - Y) / n                        # softmax+CE gradient
    dW2 = A1.T @ dZ2;            db2 = dZ2.sum(0)
    dA1 = dZ2 @ W2.T
    dZ1 = dA1 * (Z1 > 0)                     # ReLU derivative
    dW1 = X.T @ dZ1;            db1 = dZ1.sum(0)
    return dW1, db1, dW2, db2
```

**Phase 2**

- **Andrej Karpathy** — *Neural Networks: Zero to Hero* (builds backprop and a GPT from scratch; exceptional).
- **Michael Nielsen** — *Neural Networks and Deep Learning* (free online; the clearest backprop derivation).
- **PyTorch** official tutorials (pytorch.org) — start with "Learn the Basics".
- **Deep Learning** (Goodfellow, Bengio, Courville) — Part II for the authoritative treatment.

## 8 Phase 2 self-check

- Derive backprop for a 2-layer net on paper and pass a numerical gradient check in code.
- Explain why nonlinearities and proper initialization are necessary.
- Contrast SGD, momentum, RMSProp, and Adam, and say when you would use each.
- Explain dropout, weight decay, batch vs. layer norm, and early stopping.
- Write the 5-step PyTorch training loop from memory and explain each line.