OVERVIEW
Machine Learning & Deep Learning

A depth-first curriculum, from math foundations to production

Phase goal

Turn "I can call `.fit()`" into "I understand, can implement, and can ship." This guide is the map for the eight phases that follow; each phase ships as its own self-contained PDF.

Estimated time: 9–14 months at 8–12 hrs/week | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 How to use this curriculum

This is a *depth-first* path written for someone who already programs well. The assumption is Python comfort and solid software-engineering instincts; the gap we are closing is the *conceptual and mathematical* one. Every phase therefore insists that you implement core ideas from scratch at least once before reaching for a library. You only truly understand backpropagation after you have computed a gradient by hand and watched your NumPy version agree with PyTorch's autograd to five decimal places.

Key idea

The fastest way to *forget* machine learning is to learn it as a sequence of API calls. The fastest way to *own* it is to alternate between three modes: (1) derive the math, (2) implement it from scratch, (3) then use the production library and understand exactly what it is doing for you. Each phase is built around that loop.

1.1 The three-pass method for every topic

1. **Intuition pass.** Watch a video (3Blue1Brown, StatQuest, Karpathy) or read a blog until you can explain the idea to a colleague in two sentences without equations.
2. **Derivation pass.** Work the math with pen and paper. Re-derive at least the central result (the normal equation, the backprop recursion, the attention formula).
3. **Implementation pass.** Code it from scratch in NumPy, verify against a reference library, *then* learn the library's idioms.

1.2 What "done" looks like

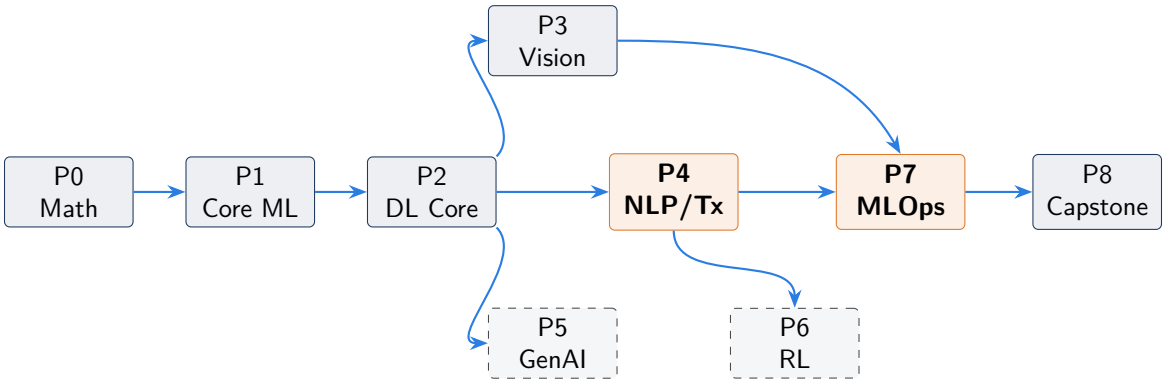
Each phase ends with a **checkpoint project**. Do not skip these. They are the difference between recognition ("I have seen ResNet") and recall-with-application ("I can fine-tune a ResNet, explain skip connections, and serve it behind an API"). Treat each checkpoint as a small portfolio piece: a clean repository, a short write-up, and a reproducible result.

2 The eight phases at a glance

#	Phase	What you walk away able to do	Time
0	Math & Tooling	Read ML papers' math; implement gradient descent from scratch	3–5 wk
1	Core ML	Implement and tune classical models; build leakage-free pipelines	6–8 wk
2	DL Foundations	Hand-derive backprop; train MLPs in raw NumPy and PyTorch	6–8 wk
3	Computer Vision	Fine-tune CNNs/ViTs; understand detection & segmentation	4–6 wk
4	NLP & Transformers	Build a Transformer and a RAG system from first principles	6–8 wk
5	Generative Models	Train VAEs/GANs/diffusion; understand modern image generation	4–5 wk
6	Reinforcement Learning	Implement DQN and policy gradients; understand RLHF	3–4 wk
7	MLOps & Deployment	Containerize, serve, monitor, and CI/CD an ML system	4–6 wk
8	Capstones	Ship 2–3 end-to-end projects in your own domains	ongoing

2.1 Dependency graph

The arrows show hard prerequisites. Phases 3, 5, and 6 branch off the deep-learning core and can be reordered or deferred; Phase 4 (NLP/Transformers) and Phase 7 (MLOps) carry the most current career value.



Intuition

Orange = highest current leverage (NLP/Transformers and MLOps). Dashed = optional/deferrable on a first pass (Generative models, RL). If your time is constrained, the

compressed path below skips the dashed nodes and circles back later.

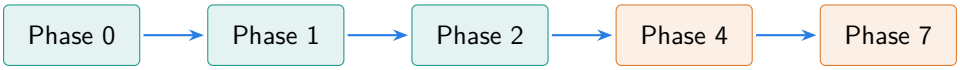
3 Two routes through the material

3.1 The complete route (recommended)

Phase 0 → 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8. This is the canonical order and builds the most robust mental model. Budget 9–14 months.

3.2 The compressed high-value route

Phase 0 → 1 → 2 → 4 → 7, then capstones. This skips computer vision, generative models, and RL on the first pass and concentrates on the two areas where most real-world and career value currently sits: *transformers/NLP/LLMs* and *deployment*. You can always return for Phases 3, 5, and 6. Budget 5–7 months.



The compressed route: foundations, then transformers, then ship.

4 Tooling you will set up once and reuse

- **Environment:** Python 3.11+, managed with `conda` or `uv` / `venv`. One environment per major phase keeps dependency conflicts contained.
- **Core stack:** `numpy`, `pandas`, `matplotlib`, `scikit-learn`, `jupyterlab`.
- **Deep learning:** `torch` (primary), `torchvision`, later `transformers`, `datasets`.
- **Compute:** a laptop is fine through Phase 2. From Phase 3 on, use a GPU — Google Colab (free/Pro), Kaggle Notebooks (free GPU), or a cloud instance.
- **Experiment hygiene:** Git from day one. Add Weights & Biases or MLflow by Phase 2 so every run is logged.

Common pitfall

Do not let environment setup become a multi-week yak-shave. Get a minimal working stack, start learning, and add tools as a phase demands them. "Notebook that runs" beats "perfect reproducible Docker-Compose monorepo" when you are on Phase 0.

5 A reference library to keep on hand

Books (keep within reach all year)

- **Hands-On Machine Learning** — Aurélien Géron. The best single practical book for Phases 1–3.
- **Mathematics for Machine Learning** — Deisenroth, Faisal, Ong. Free PDF; pairs with Phase 0.
- **Deep Learning** — Goodfellow, Bengio, Courville. The field's reference text; dense but authoritative.
- **Designing Machine Learning Systems** — Chip Huyen. The practitioner's MLOps

bible (Phase 7).
- **Reinforcement Learning: An Introduction** — Sutton & Barto. Free PDF; the canonical RL text.

Courses & video series

- Andrew Ng — *Machine Learning* & *Deep Learning* Specializations (Coursera).
- Andrej Karpathy — *Neural Networks: Zero to Hero* (YouTube). Exceptional, code-first.
- Stanford **CS231n** (vision) and **CS224n** (NLP) — free lecture videos.
- **fast.ai** — *Practical Deep Learning for Coders*: a top-down alternative path.
- **StatQuest** (Josh Starmer) and **3Blue1Brown** — the best intuition builders.

Communities & staying current

- **Kaggle** — competitions and, crucially, the winning-solution write-ups.
- **Papers With Code** — papers paired with reference implementations.
- **Hugging Face Papers** / **arXiv** — for staying current after the curriculum.

6 Domain tie-ins (make it compound)

Wherever possible, pull your own problem domains into the checkpoints — shipping/logistics data, bilingual (Chinese↔English) document processing, mobile/on-device inference, and API/backend integration. Learning compounds fastest when each new technique is immediately applied to a problem you already understand deeply. The capstone phase makes this explicit.

How to study with these PDFs

For each phase document: (1) skim the whole PDF once for the map; (2) work section by section, pausing at every *Worked example* to reproduce it yourself; (3) heed every *Common pitfall*; (4) do the checkpoint project before moving on; (5) keep a running "cheat-sheet" of formulas and gotchas in your own words. Teaching the material — even to a rubber duck — is the final test of understanding.

PHASE 0
Math & Tooling Foundations

Linear algebra, calculus, probability, statistics, and the scientific Python stack

Phase goal

Build the mathematical intuition that makes every later algorithm *click* instead of feeling like magic. By the end you can read the math in an ML paper, and you have implemented gradient descent from scratch.

Estimated time: 3–5 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 Why this phase matters

Almost everyone who "bounces off" machine learning does so for the same reason: they tried to learn the algorithms before they had the language the algorithms are written in. That language is three branches of mathematics — linear algebra, calculus, and probability/statistics — plus a small, fixed toolbox of software. You do not need to become a mathematician. You need *fluency*: the ability to look at $\nabla_{\mathbf{w}} \frac{1}{2}\|\mathbf{X}\mathbf{w}-\mathbf{y}\|^2 = \mathbf{X}^\top(\mathbf{X}\mathbf{w}-\mathbf{y})$ and read it as a sentence rather than a wall of symbols.

Key idea

Machine learning is, almost entirely, **optimizing a function of many variables**. Linear algebra is how we *represent* the data and parameters compactly; calculus is how we *improve* the parameters; probability is how we *reason about uncertainty* in the data and the model. Every later phase is a variation on this theme.

1.1 How to study Phase 0

Resist the urge to "complete" mathematics. Learn each concept to the depth where you can (a) explain it in words, (b) compute a small example by hand, and (c) reproduce it in NumPy. Use 3Blue1Brown for geometric intuition, then Mathematics for Machine Learning (free PDF) for rigor, then code. Spend roughly 40% linear algebra, 25% calculus, 25% probability/stats, 10% tooling.

2 Linear algebra

2.1 Vectors: the atoms

A vector $\mathbf{x} \in \mathbb{R}^n$ is an ordered list of n numbers. In ML it is almost always a *data point* (a row of features) or a set of *parameters*. Two complementary pictures matter:

- **Geometric:** an arrow from the origin to a point in n-dimensional space.
- **Data:** "this house = [1800 sq ft, 3 beds, 1995 built, …]".

Definition

The **dot product** (inner product) of $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ is $\mathbf{x}^\top\mathbf{y} = \sum_{i=1}^n x_i y_i = \|\mathbf{x}\|\,\|\mathbf{y}\|\cos\theta$, where θ is the angle between them. It measures *alignment*: positive when the vectors point similarly, zero when orthogonal, negative when opposed.

Intuition

The dot product is the single most important operation in ML. A neuron computes $\mathbf{w}^\top\mathbf{x}+b$ — a dot product of weights and inputs. "Similarity" in embeddings, recommender systems, and attention is a (normalized) dot product. When you see cosine similarity in a vector database, it is just $\frac{\mathbf{x}^\top\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}$.

2.1.1 Norms: measuring size

A norm measures the length of a vector. The three you must know:

$$\|\mathbf{x}\|_1 = \sum_i |x_i| \ \text{(L1, "Manhattan")}, \quad \|\mathbf{x}\|_2 = \sqrt{\textstyle\sum_i x_i^2} \ \text{(L2, "Euclidean")}, \quad \|\mathbf{x}\|_\infty = \max_i |x_i|\,.$$

For matrices, the **Frobenius norm** $\|\mathbf{A}\|_F = \sqrt{\sum_{i,j} A_{ij}^2}$ is the L2 norm of the flattened matrix. These reappear immediately in Phase 1 as regularizers: L2 (Ridge) shrinks weights smoothly, L1 (Lasso) drives some weights exactly to zero.

2.2 Matrices and matrix multiplication

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a grid of numbers, and is best understood as a **linear transformation** from $\mathbb{R}^n$ to $\mathbb{R}^m$. The product $\mathbf{Ax}$ is a new vector; $(\mathbf{AB})$ composes two transformations.

Definition

For $\mathbf{A} \in \mathbb{R}^{m \times k}$, $\mathbf{B} \in \mathbb{R}^{k \times n}$, the product $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times n}$ has entries $C_{ij} = \sum_{\ell=1}^{k} A_{i\ell} B_{\ell j}$. The inner dimension k must match. Matrix multiplication is associative $(\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C})$ and distributive, but *not* commutative ($\mathbf{AB} \neq \mathbf{BA}$ in general).

Intuition

Three ways to read $\mathbf{AB}$, each useful: (1) entry (i, j) = row i of $\mathbf{A}$ dotted with column j of $\mathbf{B}$; (2) each *column* of the output is $\mathbf{A}$ applied to the corresponding column of $\mathbf{B}$; (3) the output is a sum of outer products. A forward pass through a neural-network layer is exactly $\mathbf{XW}$ — a batch of inputs times a weight matrix. ML *is* batched matrix multiplication.

Matrix multiplication by hand

Let $\mathbf{A} = \left(\begin{smallmatrix} 1 & 2 \\ 0 & 1 \end{smallmatrix}\right)$, $\mathbf{B} = \left(\begin{smallmatrix} 3 & 1 \\ 2 & 4 \end{smallmatrix}\right)$. Then

$$\mathbf{AB} = \left(\begin{smallmatrix} 1\cdot3+2\cdot2 & 1\cdot1+2\cdot4 \\ 0\cdot3+1\cdot2 & 0\cdot1+1\cdot4 \end{smallmatrix}\right) = \left(\begin{smallmatrix} 7 & 9 \\ 2 & 4 \end{smallmatrix}\right).$$

Check non-commutativity: $\mathbf{BA} = \left(\begin{smallmatrix} 3 & 7 \\ 2 & 8 \end{smallmatrix}\right) \neq \mathbf{AB}$. The matrix $\mathbf{A}$ is a *shear*: it leaves the x-axis fixed and slides points right in proportion to their height.

2.2.1 Transpose, identity, inverse

The **transpose** $\mathbf{A}^\top$ swaps rows and columns: $(\mathbf{A}^\top)_{ij} = A_{ji}$, with $(\mathbf{AB})^\top = \mathbf{B}^\top \mathbf{A}^\top$. The **identity** $\mathbf{I}$ has ones on the diagonal and acts as the "do nothing" transform: $\mathbf{Ix} = \mathbf{x}$. The **inverse** $\mathbf{A}^{-1}$ (when it exists, for square $\mathbf{A}$) undoes the transform: $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$.

Common pitfall

In code you almost never compute an explicit inverse. To solve $\mathbf{Ax} = \mathbf{b}$, call `np.linalg.solve(A, b)`, *not* `np.linalg.inv(A) @ b`. The former is faster, more numerically stable, and avoids amplifying round-off error. Forming $\mathbf{A}^{-1}$ is a red flag in numerical code.

2.3 Rank, determinant, and what can go wrong

The **rank** of a matrix is the number of linearly independent columns — the dimension of the space its outputs actually span. A matrix is *full rank* when no column is a combination of the others. The **determinant** $\det(\mathbf{A})$ measures how the transform scales volume; $\det(\mathbf{A}) = 0$ means the transform collapses space into a lower dimension (and the matrix is singular / non-invertible).

Intuition

Rank deficiency is the geometry behind *multicollinearity* in regression: if two features are perfectly correlated, the data matrix is rank-deficient, $\mathbf{X}^\top\mathbf{X}$ is singular, and the normal equation has no unique solution. Regularization (adding $\lambda\mathbf{I}$) is partly a fix for this — it nudges the matrix back to full rank.

2.4 Eigenvalues, eigenvectors, and eigendecomposition

For a square matrix $\mathbf{A}$, a nonzero vector $\mathbf{v}$ is an **eigenvector** with **eigenvalue** λ if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}.$$

The transform $\mathbf{A}$ acts on $\mathbf{v}$ by pure scaling — no rotation. Eigenvectors are the "natural axes" of the transformation.

Definition

If $\mathbf{A} \in \mathbb{R}^{n\times n}$ has n independent eigenvectors, it admits an **eigendecomposition** $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$, where columns of $\mathbf{Q}$ are eigenvectors and $\mathbf{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$. For a *symmetric* matrix, $\mathbf{Q}$ is orthogonal ($\mathbf{Q}^{-1} = \mathbf{Q}^\top$), giving $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top$ — the spectral theorem.

Intuition

Covariance matrices are symmetric and positive semi-definite; their eigenvectors are the *principal axes* of the data cloud and their eigenvalues are the variance along each axis. That is exactly PCA (Phase 1). In optimization, the eigenvalues of the *Hessian* tell you the curvature of the loss surface — large spread of eigenvalues (high condition number) is precisely why plain gradient descent zig-zags and why we need momentum and adaptive methods (Phase 2).

2.5 Singular Value Decomposition (SVD)

SVD generalizes eigendecomposition to *any* matrix, square or not. Every $\mathbf{A} \in \mathbb{R}^{m\times n}$ factors as

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top,$$

where $\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{V} \in \mathbb{R}^{n\times n}$ are orthogonal, and $\mathbf{\Sigma}$ is diagonal with non-negative **singular values** $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$. Geometrically every linear map is a *rotation, then a scaling, then another rotation.*

Key idea

SVD is the Swiss-army knife of applied linear algebra. Keeping only the top-k singular values gives the **best rank-k approximation** of a matrix (Eckart–Young theorem). This single fact powers PCA, latent semantic analysis, recommender systems (low-rank matrix factorization), image compression, and the "intrinsic dimension" intuition behind LoRA fine-tuning in Phase 4.

Low-rank approximation = compression

A grayscale image is a matrix of pixel intensities. Compute its SVD, keep the largest k singular values, and reconstruct $\mathbf{A}_k = \mathbf{U}_{:,:k}\mathbf{\Sigma}_{:k,:k}\mathbf{V}^\top_{:,:k}$. With k far smaller than the image dimensions you recover a recognizable image using a fraction of the numbers — a vivid

demonstration that real data lives near a low-dimensional subspace.

```python
import numpy as np
U, s, Vt = np.linalg.svd(A, full_matrices=False)   # A is (m, n)
k = 30
A_k = (U[:, :k] * s[:k]) @ Vt[:k, :]               # rank-k reconstruction
energy = s[:k].sum() / s.sum()                      # fraction of "energy" kept
```

2.6 Vector spaces, basis, and projection

A **basis** is a minimal set of vectors whose linear combinations generate a space; coordinates are just the coefficients in some basis. **Projection** onto a subspace finds the closest point in that subspace — the geometric heart of least-squares regression. The projection of $\mathbf{y}$ onto the column space of $\mathbf{X}$ is $\hat{\mathbf{y}} = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$; the residual $\mathbf{y} - \hat{\mathbf{y}}$ is orthogonal to that space. That orthogonality *is* the normal equation you will derive in Phase 1.

Linear algebra

- **3Blue1Brown** — "Essence of Linear Algebra" (watch the whole series; it builds the geometric picture better than any text).
- **Mathematics for Machine Learning**, Ch. 2–4 (free PDF).
- **Gilbert Strang**, MIT 18.06 lectures — the classic deep dive; his treatment of the four fundamental subspaces and SVD is definitive.

3 Calculus

3.1 Derivatives and the chain rule

The derivative $f'(x)$ is the instantaneous rate of change — the slope of the tangent line. The **chain rule** composes rates of change:

$$\frac{d}{dx} f(g(x)) = f'(g(x))\, g'(x).$$

Key idea

The chain rule, applied mechanically across the layers of a network, *is* backpropagation (Phase 2). If you understand $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$ deeply, you already understand the core of deep learning; the rest is bookkeeping over many variables.

3.2 Partial derivatives, gradients, Jacobians, Hessians

For a function $f(\mathbf{x})$ of several variables, the **partial derivative** $\partial f/\partial x_i$ varies one input while holding the rest fixed. Collecting them gives the **gradient**:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)^\top \in \mathbb{R}^n.$$

Definition

The gradient **points in the direction of steepest ascent** of f, and its negative points toward steepest descent. Its magnitude is the rate of steepest change. This is the entire justification for gradient descent: to decrease a loss, step in the direction $-\nabla f$.

For a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the **Jacobian** $\mathbf{J} \in \mathbb{R}^{m \times n}$ collects all partials $J_{ij} = \partial f_i / \partial x_j$. The **Hessian** $\mathbf{H} \in \mathbb{R}^{n \times n}$ collects second partials $H_{ij} = \partial^2 f / \partial x_i \partial x_j$ and describes *curvature*.

Intuition

Gradient = slope (first order, "which way is downhill"). Hessian = curvature (second order, "how sharply the slope changes"). First-order methods (SGD, Adam) use only gradients because the Hessian is enormous for deep nets (n can be billions). But knowing the Hessian exists explains *why* learning rates matter, why some directions need momentum, and why ill-conditioned losses are hard.

3.3 Gradients you will use constantly

Memorize these matrix-calculus identities; they cover most of classical ML.

$$\nabla_{\mathbf{x}}(\mathbf{a}^\top \mathbf{x}) = \mathbf{a}, \qquad \nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x},$$
$$\nabla_{\mathbf{x}} \tfrac{1}{2}\|\mathbf{x}\|_2^2 = \mathbf{x}, \qquad \nabla_{\mathbf{w}} \tfrac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}).$$

Deriving the least-squares gradient

Let $L(\mathbf{w}) = \frac{1}{2}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 = \frac{1}{2}(\mathbf{X}\mathbf{w} - \mathbf{y})^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$. Expand:

$$L = \tfrac{1}{2}\big(\mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}\big).$$

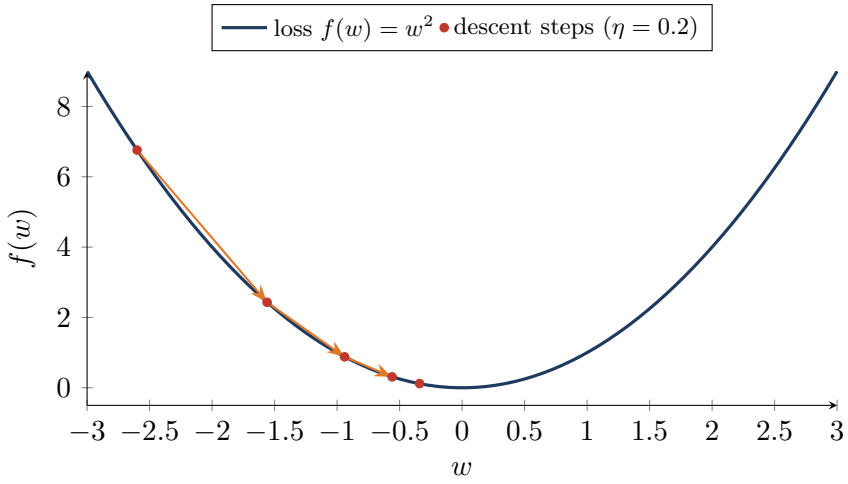
Differentiate term by term using the identities above:

$$\nabla_{\mathbf{w}} L = \mathbf{X}^\top \mathbf{X}\mathbf{w} - \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y}).$$

Setting $\nabla_{\mathbf{w}} L = \mathbf{0}$ yields the **normal equation** $\mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y}$, i.e. $\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$. You just derived linear regression's closed form — and the projection formula from the previous section.

3.4 Why gradient descent works

Gradient descent iterates $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla f(\mathbf{w}_t)$. The first-order Taylor expansion explains it: near $\mathbf{w}_t$, $f(\mathbf{w}_t + \Delta) \approx f(\mathbf{w}_t) + \nabla f(\mathbf{w}_t)^\top \Delta$. Choosing $\Delta = -\eta \nabla f$ makes the inner product as negative as possible (for small step), so f decreases. The step size η (learning rate) trades speed against the risk of overshooting — too large and you diverge, too small and you crawl.



Common pitfall

The negative gradient is the steepest descent direction only *locally*. On curved or ill-conditioned surfaces, naive steps zig-zag across valleys. This is the practical reason momentum, RMSProp, and Adam exist (Phase 2). Also: gradient descent finds a local minimum, not necessarily the global one — though for the convex losses of Phase 1 (linear/logistic regression), local = global.

Calculus

- **3Blue1Brown** — "Essence of Calculus".
- **Mathematics for Machine Learning**, Ch. 5 (Vector Calculus) — the matrix-calculus reference for ML.
- *The Matrix Cookbook* (free PDF) — a lookup table of gradient identities.

4 Probability & statistics

4.1 Random variables and distributions

A **random variable** maps outcomes to numbers. You must know these distributions and *where each shows up*:

Distribution	Models	Shows up in ML as
Bernoulli(p)	a single yes/no trial	binary classification target
Binomial(n, p)	# successes in n trials	counts, A/B test conversions
Categorical	one of K classes	softmax output, multiclass labels
Gaussian(μ, σ^2)	symmetric continuous noise	errors, weight init, latent spaces
Poisson(λ)	event counts per interval	arrivals, rare-event counts
Exponential(λ)	waiting time between events	survival, time-to-event

Definition

The **Gaussian (normal)** density is $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\big(-\frac{(x-\mu)^2}{2\sigma^2}\big)$. It is the default model for noise (justified by the Central Limit Theorem), the basis of the squared-error loss (negative log-likelihood of Gaussian noise *is* MSE), and the standard prior/latent in VAEs

and diffusion models (Phase 5).

4.2 Expectation, variance, covariance, correlation

$$\mathbb{E}[X] = \sum_x x\,p(x) \text{ (or } \int x\,p(x)\,dx), \quad \mathrm{Var}(X) = \mathbb{E}[(X-\mathbb{E}X)^2] = \mathbb{E}[X^2] - (\mathbb{E}X)^2.$$

For two variables, **covariance** $\mathrm{Cov}(X,Y) = \mathbb{E}[(X-\mathbb{E}X)(Y-\mathbb{E}Y)]$ measures co-movement, and **correlation** $\rho = \mathrm{Cov}(X,Y)/(\sigma_X\sigma_Y) \in [-1,1]$ is its scale-free version. The **covariance matrix** $\boldsymbol{\Sigma}$ of a random vector collects all pairwise covariances; it is symmetric PSD, and its eigen-structure is PCA.

Common pitfall

Correlation measures *linear* association only. Two variables can be strongly dependent yet have $\rho = 0$ (e.g. $Y = X^2$ with X symmetric about 0). And of course correlation is not causation — a lesson with real teeth once you start interpreting feature importances and "the model says X drives Y".

4.3 Conditional probability and Bayes' theorem

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}, \qquad \underbrace{\mathbb{P}(\theta \mid \text{data})}_{\text{posterior}} = \frac{\overbrace{\mathbb{P}(\text{data} \mid \theta)}^{\text{likelihood}}\ \overbrace{\mathbb{P}(\theta)}^{\text{prior}}}{\underbrace{\mathbb{P}(\text{data})}_{\text{evidence}}}.$$

Intuition

Bayes' theorem is how you *update beliefs with evidence*. It is the engine behind Naive Bayes classifiers, the Bayesian view of regularization (a prior on weights), Bayesian hyperparameter optimization (Phase 1), and the conceptual frame for nearly all probabilistic modeling. Internalize "posterior $\propto$ likelihood $\times$ prior."

Base rates: why a 99%-accurate test can mostly be wrong

A disease affects 1 in 1000 people. A test is 99% accurate (both sensitivity and specificity). You test positive — what is the chance you are sick? Let D = disease, $+$ = positive.

$$\mathbb{P}(D \mid +) = \frac{\mathbb{P}(+ \mid D)\mathbb{P}(D)}{\mathbb{P}(+ \mid D)\mathbb{P}(D) + \mathbb{P}(+ \mid \neg D)\mathbb{P}(\neg D)} = \frac{0.99 \cdot 0.001}{0.99 \cdot 0.001 + 0.01 \cdot 0.999} \approx 0.09.$$

Only **9%**. The rare base rate dominates. This is the same trap as evaluating a classifier on a heavily imbalanced dataset by accuracy alone (Phase 1) — and why precision/recall exist.

4.4 MLE and MAP estimation

Given data and a parametric model, **Maximum Likelihood Estimation (MLE)** chooses parameters that make the observed data most probable:

$$\hat{\boldsymbol{\theta}}_{\text{MLE}} = \arg\max_{\boldsymbol{\theta}} \prod_i p(x_i \mid \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \sum_i \log p(x_i \mid \boldsymbol{\theta}).$$

Maximum A Posteriori (MAP) adds a prior: $\hat{\boldsymbol{\theta}}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} \left[\sum_i \log p(x_i \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \right]$.

Key idea

Most loss functions are disguised negative log-likelihoods. Gaussian noise ⇒ MLE = least squares (MSE). Bernoulli/categorical labels ⇒ MLE = cross-entropy. A Gaussian prior on weights ⇒ MAP = L2 regularization (Ridge); a Laplace prior ⇒ L1 (Lasso). Seeing losses this way unifies half of Phases 1 and 2.

4.5 Inferential statistics: CLT, intervals, hypothesis tests

The **Central Limit Theorem** states that the mean of many independent samples is approximately Gaussian, regardless of the underlying distribution. This is why averages are well-behaved and why error bars work. A **confidence interval** quantifies uncertainty in an estimate; a ***p*-value** is the probability of seeing data at least as extreme as observed *if the null hypothesis were true*.

Common pitfall

A p-value is *not* the probability that the null hypothesis is true, and "$p < 0.05$" is not proof of importance. In ML this matters most when you compare models: a 0.2% accuracy bump on one test split may be noise. Use cross-validation and, where stakes are high, statistical tests on the per-fold results before declaring a winner (Phase 1).

Probability & statistics

- **StatQuest** (Josh Starmer) — the most intuitive stats/ML explainers on the internet.
- **Khan Academy** — Statistics & Probability (for systematic coverage and practice problems).
- **Mathematics for Machine Learning**, Ch. 6 (Probability & Distributions).

5 Tooling: the scientific Python stack

You already program well, so this is about idioms, not syntax. The goal is to think in *vectorized array operations*, not Python loops.

5.1 NumPy — the foundation

NumPy's `ndarray` is the substrate everything else builds on. The two ideas that separate beginners from fluent users are **vectorization** and **broadcasting**.

Key idea

Broadcasting lets arrays of different shapes combine without explicit loops by virtually stretching size-1 dimensions. Standardizing a data matrix is one line: `(X - X.mean(0)) / X.std(0)`. Internalizing broadcasting rules (align shapes from the right; dimensions must be equal or one of them 1) is the single highest-leverage NumPy skill.

```
import numpy as np
X = np.random.randn(1000, 5)          # 1000 samples, 5 features
mu, sigma = X.mean(axis=0), X.std(axis=0)   # shape (5,)
```

```
Xz = (X - mu) / sigma                    # broadcast (1000,5)-(5,) ->
    (1000,5)

# Vectorized vs. loop: compute pairwise squared distances to a centroid
c = X.mean(0)
d2 = ((X - c) ** 2).sum(axis=1)          # shape (1000,), no Python loop
```

Common pitfall

The most common silent bug in ML code is a **shape mismatch that broadcasts "successfully"** into the wrong thing (e.g. a `(n,)` vector subtracted from an `(n,1)` column produces an `(n,n)` matrix). Print `.shape` obsessively. A second classic: views vs. copies — slicing returns a view, so in-place edits can mutate the original array.

5.2 Pandas — tabular data

Pandas `DataFrame`s handle real-world messy tables: mixed types, missing values, joins, groupbys. Learn `read_csv`, indexing with `.loc` / `.iloc`, `groupby().agg()`, `merge`, and handling `NaN` (`fillna`, `dropna`). You will live in Pandas during the Phase 1 EDA and feature-engineering work.

5.3 Matplotlib & Seaborn — seeing the data

Plotting is not decoration; it is debugging. Always look at your data and your model's errors. Histograms reveal skew and outliers, scatter plots reveal relationships, residual plots reveal model misfit, and learning curves reveal over/underfitting (Phase 1). Seaborn sits on Matplotlib for fast statistical plots (`pairplot`, `heatmap` of a correlation matrix).

5.4 Jupyter / Colab, Git, and a little SQL

- **Jupyter / Colab:** interactive exploration. Colab gives free GPUs (vital from Phase 3). Beware hidden state from out-of-order cell execution — restart-and-run-all before trusting a notebook.
- **Git:** version every experiment. Commit notebooks and configs; keep large data and model weights out of the repo (use `.gitignore`, DVC, or a model registry later in Phase 7).
- **SQL:** you are likely already strong here. For ML it is the data-wrangling front door — `SELECT`, `JOIN`, `GROUP BY`, window functions for feature aggregation pull training sets straight from warehouses.

Intuition

Treat notebooks as a *lab notebook*, not production code. Explore in Jupyter, then refactor the keeper logic into importable `.py` modules with functions and tests. This discipline pays off enormously by Phase 7, when notebooks must become pipelines.

6 Checkpoint project

Gradient descent from scratch

Goal: implement gradient descent with no ML libraries and *see* it work, cementing the calculus–optimization link that underlies every later phase.

Part A — 1D warm-up. Minimize $f(w) = w^2$ (or a quartic with two minima to observe local minima). Implement the update $w \leftarrow w - \eta f'(w)$ by hand-deriving f'. Plot f and overlay the sequence of iterates as points connected by arrows (as in the figure above).

Part B — 2D bowl. Minimize $f(\mathbf{w}) = \frac{1}{2}\mathbf{w}^\top \mathbf{A}\mathbf{w}$ for a 2×2 symmetric positive-definite $\mathbf{A}$ (so the level sets are ellipses). Derive $\nabla f = \mathbf{A}\mathbf{w}$. Draw a contour plot and trace the descent path. Make $\mathbf{A}$ ill-conditioned (e.g. eigenvalues 1 and 20) and watch the path zig-zag — you have now *seen* why conditioning and momentum matter.

Part C — real loss, real data. Fit a line $y = w_0 + w_1 x$ to noisy synthetic data by minimizing MSE via gradient descent. Verify your solution matches the closed-form normal equation $\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$. Plot the loss vs. iteration (it should decrease smoothly) and the fitted line over the data.

Deliverables: a clean notebook or repo with (1) the descent-path figure, (2) the contour/zig-zag figure, (3) the loss curve, (4) a short paragraph explaining what the learning rate does and what you observed when it was too large.

Stretch goals: add momentum and compare convergence; sweep η and plot final loss vs. η; implement everything with explicit shapes asserted (`assert grad.shape == w.shape`).

```python
import numpy as np

def gradient_descent(grad_fn, w0, lr=0.1, steps=100):
    """Generic GD; grad_fn(w) returns the gradient at w."""
    w = np.array(w0, dtype=float)
    history = [w.copy()]
    for _ in range(steps):
        w = w - lr * grad_fn(w)
        history.append(w.copy())
    return w, np.array(history)

# Part C: linear regression via GD, then check vs. the normal equation.
rng = np.random.default_rng(0)
X = np.c_[np.ones(200), rng.uniform(-3, 3, 200)]   # design matrix (200,2)
w_true = np.array([1.5, -2.0])
y = X @ w_true + 0.5 * rng.standard_normal(200)

grad = lambda w: X.T @ (X @ w - y) / len(y)         # MSE gradient
w_gd, hist = gradient_descent(grad, [0.0, 0.0], lr=0.05, steps=500)
w_closed = np.linalg.solve(X.T @ X, X.T @ y)         # normal equation
print("GD     :", w_gd)
print("closed :", w_closed)        # should match to a few decimals
```

7 Phase 0 self-check

Before moving to Phase 1, you should be able to, without notes:

- Explain what $\mathbf{Ax}$ does geometrically and compute a small matrix product by hand.

- State what eigenvectors/eigenvalues and the SVD are, and name one ML use of each.
- Derive the gradient of the least-squares loss and the normal equation.
- Explain why $-\nabla f$ is the descent direction and what the learning rate controls.
- Recover MSE and cross-entropy as negative log-likelihoods, and L2/L1 as MAP priors.
- Apply Bayes' theorem to a base-rate problem and interpret the result.
- Vectorize a computation with NumPy broadcasting instead of a Python loop.

Intuition

If two or three of these feel shaky, that is normal — revisit just those, then proceed. You do *not* need mastery of all of mathematics; you need enough fluency that the notation in Phase 1 and 2 reads as meaning rather than noise. The remaining gaps will close through use.

PHASE 1
Core Machine Learning

Classical models, evaluation, feature engineering, and tuning — implemented, not just imported

Phase goal

Understand classical ML deeply enough to implement the key algorithms from scratch and to build a leakage-free, well-evaluated pipeline — not just to call `.fit()`.

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 Orientation

Classical ("tabular") machine learning is still where most production value is created outside of the LLM frontier: churn prediction, fraud, pricing, demand forecasting, ranking. It is also the best place to build the disciplines — evaluation, validation, leakage avoidance — that separate practitioners from people who overfit Kaggle leaderboards. Deep learning inherits all of these disciplines; learning them on simple models first is far cheaper.

Key idea

The model is rarely the hard part. **Framing the problem, building trustworthy evaluation, engineering features, and avoiding leakage** are what determine success. A logistic regression evaluated correctly beats a deep net evaluated incorrectly every single time.

2 ML fundamentals

2.1 The learning setups

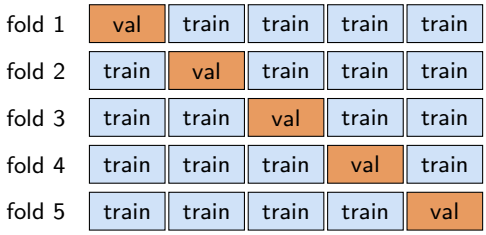
- **Supervised:** learn a mapping $f : \mathbf{x} \mapsto y$ from labeled pairs. Regression ($y \in \mathbb{R}$) or classification (y categorical).
- **Unsupervised:** find structure without labels — clustering, dimensionality reduction, density estimation.
- **Semi-supervised:** a little labeled data plus a lot of unlabeled data (common and economically important).
- **Reinforcement:** an agent learns from reward by interacting with an environment (Phase 6).

2.2 Train / validation / test, and cross-validation

You split data to estimate *generalization* — performance on data the model has never seen. The cardinal rule: the **test set is touched exactly once**, at the very end.

Definition

k-fold cross-validation partitions the training data into k folds, trains on $k-1$ and validates on the held-out fold, rotating k times, then averages. It uses data efficiently and gives a variance estimate of your metric. **Stratified** k-fold preserves class proportions in each fold — essential for imbalanced classification.



Common pitfall

For **time-series** data, random k-fold leaks the future into the past. Use forward-chaining (expanding-window) splits where validation always lies after training in time. For **grouped** data (multiple rows per customer), use `GroupKFold` so the same group never appears in both train and validation — otherwise you measure memorization, not generalization.

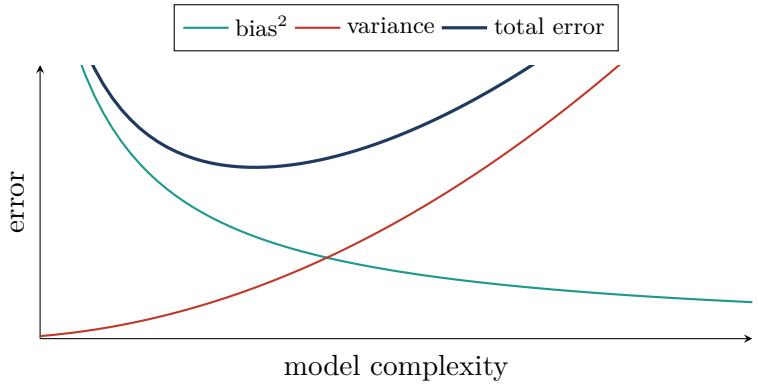
2.3 Bias–variance tradeoff

Expected test error decomposes (for squared loss) into three parts:

$$\mathbb{E}\big[(y-\hat{f}(\mathbf{x}))^2\big] = \underbrace{\big(\text{Bias}[\hat{f}(\mathbf{x})]\big)^2}_{\text{too simple}} + \underbrace{\text{Var}[\hat{f}(\mathbf{x})]}_{\text{too sensitive}} + \underbrace{\sigma^2}_{\text{irreducible}} .$$

Intuition

High bias = underfitting: the model is too simple to capture the signal (a line through curved data). **High variance = overfitting:** the model chases noise and changes wildly with the training sample. Regularization, more data, and simpler models reduce variance; richer models and better features reduce bias. The art is balancing them — and the validation curve (later) is how you *see* the balance.



2.4 Evaluation metrics — choose before you model

2.4.1 Regression

MSE $= \frac{1}{n}\sum(y_i-\hat{y}_i)^2$ (penalizes large errors quadratically); RMSE $= \sqrt{\text{MSE}}$ (same units as y); MAE $= \frac{1}{n}\sum|y_i-\hat{y}_i|$ (robust to outliers); $R^2 = 1 - \frac{\sum(y_i-\hat{y}_i)^2}{\sum(y_i-\bar{y})^2}$ (fraction of variance explained).

2.4.2 Classification

From the confusion matrix (TP, FP, TN, FN):

$$\text{precision} = \frac{TP}{TP+FP}, \quad \text{recall} = \frac{TP}{TP+FN}, \quad F_1 = \frac{2\cdot\text{prec}\cdot\text{rec}}{\text{prec}+\text{rec}}.$$

ROC-AUC measures ranking quality across all thresholds (probability a random positive outranks a random negative). **PR-AUC** is more informative under heavy class imbalance.

Common pitfall

Accuracy is a trap on imbalanced data. If 99% of transactions are legitimate, "predict legit always" scores 99% accuracy and catches zero fraud. Pick the metric from the *business cost*: recall when misses are expensive (disease, fraud), precision when false alarms are expensive (spam filters), F_1/PR-AUC for balance, and calibrated probabilities when you need to threshold by cost.

2.4.3 Ranking

For search/recommendation, **NDCG** (normalized discounted cumulative gain) rewards putting relevant items high; **MRR** (mean reciprocal rank) focuses on the position of the first relevant result. These matter the moment your API powers a search or recommendation feature.

3 Regression & classification

3.1 Linear regression — two solvers

The model is $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. With the bias folded into $\mathbf{w}$ by appending a 1 to $\mathbf{x}$:

- **Closed form (normal equation):** $\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ — exact, but $O(d^3)$ in the number of features and unstable if $\mathbf{X}^\top \mathbf{X}$ is ill-conditioned.
- **Gradient descent:** scales to huge n and d, and generalizes to models with no closed form. (You implemented this in Phase 0.)

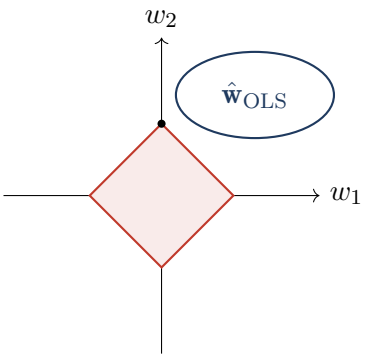
3.2 Regularization: Ridge, Lasso, ElasticNet

Add a penalty on weight magnitude to combat overfitting and multicollinearity:

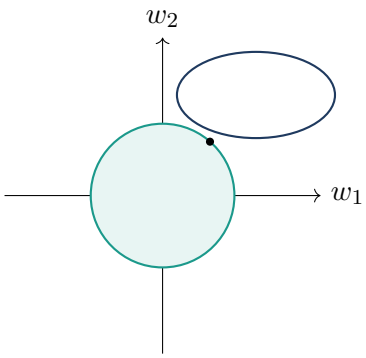
$$\text{Ridge: } \min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2, \qquad \text{Lasso: } \min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1 .$$

Key idea

Ridge (L2) shrinks all weights smoothly toward zero (and fixes the singular $\mathbf{X}^\top \mathbf{X}$ by adding $\lambda \mathbf{I}$). **Lasso (L1)** drives some weights *exactly* to zero, performing automatic feature selection — its diamond-shaped constraint region has corners on the axes. **ElasticNet** blends both. Recall from Phase 0: Ridge = Gaussian prior on weights (MAP), Lasso = Laplace prior.



Lasso (L1): corner ⇒ sparsity



Ridge (L2): smooth shrinkage

3.3 Logistic and softmax regression

For binary classification, logistic regression models $\mathbb{P}(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ with the sigmoid $\sigma(z) = 1/(1 + e^{-z})$. It is trained by minimizing the **cross-entropy** (negative log-likelihood of the Bernoulli):

$$L(\mathbf{w}) = -\sum_i \left[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \right], \quad \hat{p}_i = \sigma(\mathbf{w}^\top \mathbf{x}_i).$$

For K classes, **softmax regression** outputs $\hat{p}_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$ and minimizes categorical cross-entropy. This exact pairing (softmax + cross-entropy) is the output layer of nearly every classifier you will build in Phases 2–4.

Intuition

"Logistic regression" is a misnomer — it is a *classifier*. It is a single-layer neural network with a sigmoid activation. Master its loss and gradient now and the deep-learning version in Phase 2 is just the same idea stacked.

3.4 Naive Bayes

Apply Bayes' theorem with the ("naive") assumption that features are conditionally independent given the class: $\mathbb{P}(y \mid \mathbf{x}) \propto \mathbb{P}(y) \prod_j \mathbb{P}(x_j \mid y)$. Despite the unrealistic assumption it is fast, needs little data, and is a strong baseline for text classification (bag-of-words).

3.5 K-Nearest Neighbors

KNN predicts using the k closest training points (majority vote or average). It has *no training phase* (lazy learning) but expensive inference, and degrades in high dimensions (the curse of dimensionality). It is a useful conceptual anchor: it makes the role of distance metrics and feature scaling vivid.

4 Tree-based methods

4.1 Decision trees

A tree recursively splits the feature space to make regions as "pure" as possible. Split quality for classification uses **Gini impurity** $G = 1 - \sum_k p_k^2$ or **entropy** $H = -\sum_k p_k \log p_k$; the chosen split maximizes **information gain** (impurity reduction). Trees are interpretable and need no feature scaling, but a single deep tree overfits badly (high variance).

Gini of a split

A node has 40 positives and 10 negatives: $G = 1 - (0.8^2 + 0.2^2) = 0.32$. A candidate split yields children $\{30^+, 2^-\}$ and $\{10^+, 8^-\}$ with Gini $1 - (\frac{30}{32})^2 - (\frac{2}{32})^2 = 0.117$ and $1 - (\frac{10}{18})^2 - (\frac{8}{18})^2 = 0.494$. Weighted child impurity $= \frac{32}{50}(0.117) + \frac{18}{50}(0.494) = 0.253 < 0.32$, so the split *reduces* impurity and is worth making.

4.2 Random forests (bagging)

Bagging (bootstrap aggregating) trains many trees on bootstrap resamples and averages them, slashing variance. Random forests add a twist: at each split only a random subset of features is considered, decorrelating the trees so averaging helps more. Robust, low-tuning, excellent default for tabular data.

4.3 Gradient boosting

Boosting builds trees *sequentially*, each new tree fitting the *residual errors* of the ensemble so far — gradient descent in function space.

Key idea

Gradient-boosted trees (XGBoost, LightGBM, CatBoost) are the reigning champions of tabular ML and win the majority of Kaggle tabular competitions. Defaults to know: tune learning rate × number of trees together, use early stopping on a validation set, and watch `max_depth` / `num_leaves` to control overfitting. LightGBM is

fastest on large data; CatBoost handles categoricals natively.

	Random Forest	Gradient Boosting
Trees built	in parallel, independently	sequentially, on residuals
Reduces mainly	variance	bias (then variance)
Tuning sensitivity	low	higher (LR, depth, #trees)
Risk	rarely overfits	overfits if over-trained

4.4 Interpretability: feature importance & SHAP

Built-in importances (split gain) are quick but biased toward high-cardinality features. **SHAP** values, grounded in cooperative game theory, fairly attribute each prediction to its features and are consistent and locally accurate. SHAP summary and dependence plots are the modern standard for explaining tabular models to stakeholders — and for catching leakage (a single feature with implausibly dominant SHAP is a red flag).

5 Support Vector Machines

An SVM finds the hyperplane that **maximizes the margin** — the distance to the nearest points (support vectors). The soft-margin formulation allows some violations, controlled by C (large C = fewer violations, higher variance). The **kernel trick** replaces dot products $\mathbf{x}_i^\top \mathbf{x}_j$ with a kernel $k(\mathbf{x}_i, \mathbf{x}_j)$, implicitly mapping to a high-dimensional space without computing it — enabling nonlinear boundaries with linear, polynomial, or RBF kernels.

Intuition

SVMs were dominant before deep learning and remain strong on small/medium datasets with clear margins. The RBF kernel "places a bump" around each support vector. The big-picture lesson — maximize margin for robustness — echoes throughout ML (it is the intuition behind large-margin and contrastive losses later).

6 Unsupervised learning

6.1 Clustering

K-Means alternates assigning points to the nearest centroid and recomputing centroids; it minimizes within-cluster variance but assumes spherical, equal-size clusters and needs k chosen (elbow/silhouette methods). **Hierarchical** clustering builds a dendrogram (no preset k). **DBSCAN** finds density-connected clusters of arbitrary shape and labels outliers, without specifying k — but is sensitive to its density parameters.

6.2 Dimensionality reduction

PCA projects onto the top eigenvectors of the covariance matrix (the SVD of centered data) — a *linear* method that maximizes retained variance, great for compression and de-noising. **t-SNE** and **UMAP** are *nonlinear* methods for *visualization* of clusters in 2D/3D.

Common pitfall

t-SNE/UMAP are for *visualization*, not as preprocessing for a downstream model. Distances and cluster sizes in a t-SNE plot are not faithful, and results depend heavily on perplexity/neighbors. Never read "these clusters are far apart so they are very different" off a t-SNE plot. Use PCA when you need a stable, invertible, distance-preserving reduction.

6.3 Anomaly detection

Isolation Forest isolates outliers with random splits (anomalies need fewer splits); **One-Class SVM** learns a boundary around the normal data. Both are unsupervised and directly relevant to fraud/defect/shipment-anomaly use cases.

7 Feature engineering & data prep

In tabular ML this is where most of the winning happens.

- **Missing data:** understand *why* it is missing; impute (mean/median/model-based) and optionally add a "was-missing" indicator.
- **Outliers:** detect (IQR, z-score), then cap/winsorize or model robustly — do not blindly delete.
- **Categorical encoding:** one-hot for low cardinality; **target/mean encoding** for high cardinality (with cross-fold smoothing to avoid leakage); ordinal only when order is real.
- **Scaling:** standardize (z-score) or normalize ($[0,1]$). Required for KNN, SVM, PCA, and neural nets; irrelevant for trees.
- **Class imbalance:** class weighting, undersampling the majority, or **SMOTE** (synthetic minority oversampling). Apply resampling *inside* cross-validation only.

Data leakage — the #1 cause of "too good to be true" results

Leakage is any information from the test set (or the future) sneaking into training. Classic forms: scaling/imputing/encoding on the *full* dataset before splitting; target-encoding without cross-fold isolation; including a feature that is a proxy for the label (e.g. "account closed date" when predicting churn). **Fix:** fit every transform on the training fold only, and wrap the entire flow in a `Pipeline` so cross-validation re-fits transforms per fold.

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import cross_val_score

num = Pipeline([("imp", SimpleImputer(strategy="median")),
                ("sc", StandardScaler())])
cat = Pipeline([("imp", SimpleImputer(strategy="most_frequent")),
                ("oh", OneHotEncoder(handle_unknown="ignore"))])
pre = ColumnTransformer([("num", num, num_cols), ("cat", cat,
    cat_cols)])

model = Pipeline([("pre", pre), ("clf",
    HistGradientBoostingClassifier())])
# Transforms are re-fit on each training fold -> no leakage.
scores = cross_val_score(model, X, y, cv=5, scoring="roc_auc")
```

```
print(scores.mean(), scores.std())
```

8 Model selection & tuning

- **Grid search** exhaustively tries combinations (expensive, fine for few hyperparameters).
- **Random search** samples combinations — usually more efficient than grid for the same budget.
- **Bayesian optimization (Optuna)** models the objective and proposes promising points; best for expensive models and larger search spaces. Use pruning to kill bad trials early.

8.1 Learning vs. validation curves

Learning curves plot train/validation score vs. *training-set size*: a large persistent gap signals high variance (get more data / regularize); both curves low and converged signals high bias (use a richer model / better features). **Validation curves** plot score vs. a single *hyperparameter* to find its sweet spot.

8.2 A/B testing for production models

Offline metrics do not always predict online impact. The gold standard is a randomized A/B test: split traffic, run the new model against the incumbent, and compare a pre-registered business metric with proper statistics (mind the multiple-comparisons and peeking problems from Phase 0). This bridges directly to Phase 7.

Phase 1

- **Hands-On Machine Learning** (Géron), Part I — the single best practical companion to this phase.
- Andrew Ng — *Machine Learning Specialization* (Coursera).
- **StatQuest** — trees, random forests, boosting, SVM, PCA series.
- **scikit-learn** user guide — exemplary documentation; read the "common pitfalls" page.
- **Kaggle** — read winning solution write-ups; they teach feature engineering better than any course.

9 Checkpoint project

End-to-end tabular case study

Take a real tabular dataset (e.g. Kaggle house prices or a customer-churn set) and produce a polished mini case study.

1. **EDA:** distributions, missingness, correlations, target relationship. Write down hypotheses.
2. **Feature engineering:** imputation, encoding, scaling, a few derived features — all inside a `Pipeline`.
3. **Modeling:** compare ≥4 models (linear/logistic + regularization, random forest, gradient boosting, and one more) via stratified cross-validation on a metric you justify.
4. **Tuning:** optimize the best model with Optuna; show the validation curve for at least one hyperparameter.
5. **Interpretation:** SHAP summary + dependence plots; explain the top drivers and sanity-check for leakage.
6. **Write-up:** a README framing the problem, decisions, results (with error bars), and limitations.

Definition of done: a reproducible repo, a single command to train, a held-out test score reported *once*, and a paragraph on what you would deploy and monitor (foreshadowing Phase 7).

10 Phase 1 self-check

- Choose and justify an evaluation metric for an imbalanced business problem.
- Explain bias–variance using a learning curve you drew.
- State when Ridge vs. Lasso vs. ElasticNet is appropriate and why.
- Explain why gradient boosting usually beats a single tree and a random forest on tabular data.
- Build a cross-validated, leakage-free pipeline and explain every transform's placement.
- Read a SHAP summary plot and identify a suspicious (leaky) feature.

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

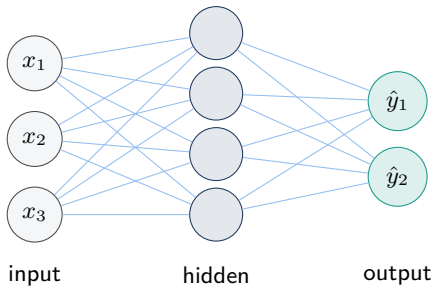
A neural network is just a big differentiable function $f_{\boldsymbol{\theta}}(\mathbf{x})$ with millions of parameters $\boldsymbol{\theta}$. We pick a loss, compute $\nabla_{\boldsymbol{\theta}} L$ by the chain rule (backprop), and descend. That is the *entire* algorithm. Phases 3–6 only change the *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 ϕ. A **multilayer perceptron (MLP)** stacks layers of these:

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

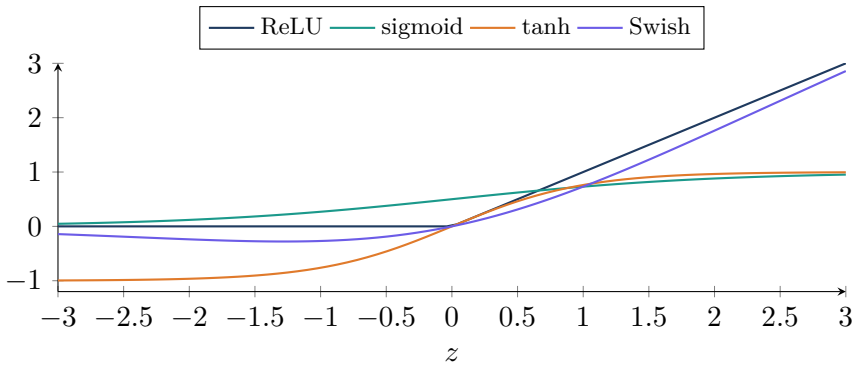


Intuition

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 α 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

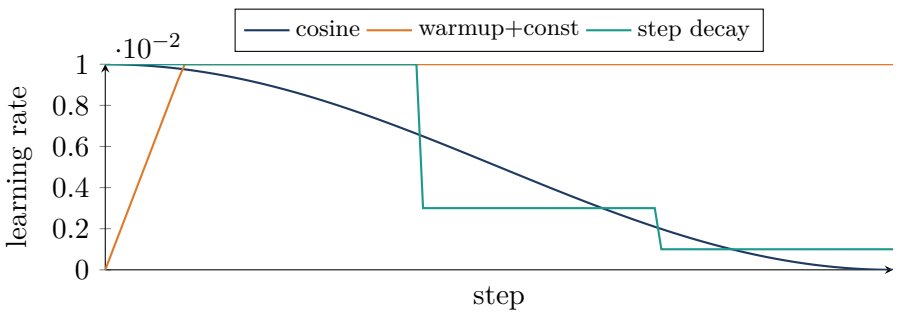
$$\begin{aligned}
\text{Momentum:}\quad & \mathbf{v}_t = \beta \mathbf{v}_{t-1} + \nabla L, \quad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \mathbf{v}_t \\
\text{RMSProp:}\quad & s_t = \rho s_{t-1} + (1-\rho)(\nabla L)^2, \quad \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \eta \nabla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

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.

PHASE 3
Computer Vision

Convolutions, CNN architectures, transfer learning, detection, segmentation, and ViTs

Phase goal

Understand how neural networks see: the convolution operation, the architectural lineage from LeNet to ResNet to ViT, and the practical craft of transfer learning. By the end you can fine-tune a pretrained model and serve it behind an API.

Estimated time: 4–6 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 Why images need special architectures

An image is a grid of pixels with strong *spatial structure*: nearby pixels are correlated, and patterns (edges, textures, objects) can appear anywhere. A plain MLP ignores this — it would need a separate weight for every pixel-position pair (a $224 \times 224 \times 3$ image flattened is ∼150,000 inputs) and would not generalize across translations. Convolutional networks bake in two priors that match images: **locality** and **translation equivariance**, with massive **weight sharing**.

Key idea

A convolution slides a small set of learnable weights (a *filter*) across the image, computing the same dot product at every location. This gives (1) far fewer parameters, (2) the ability to detect a pattern wherever it occurs, and (3) a hierarchy: early layers learn edges, later layers learn textures, then parts, then objects.

2 The convolution operation

2.1 Filters, stride, padding

A filter (kernel) of size $k \times k$ is convolved with the input: at each position, multiply the overlapping pixels by the filter weights and sum. **Stride** s is the step size of the slide (larger stride ⇒ smaller output). **Padding** adds a border of zeros so the output can keep the input's spatial size ("same" padding).

Definition

For an input of size W, kernel k, padding p, stride s, the output spatial size is

$$W_{\text{out}} = \left\lfloor \frac{W - k + 2p}{s} \right\rfloor + 1.$$

A convolutional layer with C_{in} input channels and C_{out} filters has $k \cdot k \cdot C_{\text{in}} \cdot C_{\text{out}}$ weights — independent of the image size.

Counting parameters

A 3×3 conv from 64 to 128 channels has $3 \cdot 3 \cdot 64 \cdot 128 = 73{,}728$ weights (plus 128 biases) — regardless of whether the feature map is 56×56 or 7×7. Compare to a fully-connected layer between two $56 \times 56 \times 64$ maps: over 4×10^{10} weights. Weight sharing is the difference between trainable and impossible.

2.2 Pooling and receptive field

Pooling (max or average) downsamples feature maps, adding translation invariance and reducing computation. The **receptive field** is the region of the input that influences one output unit; it grows with depth, so deep layers "see" large parts of the image. Modern nets increasingly replace pooling with strided convolutions.

Intuition

Think of a CNN as a funnel: spatial resolution shrinks ($224 \to 112 \to 56 \to \dots$) while channel depth grows ($3 \to 64 \to 128 \to \dots$). The network trades "where" for "what" — progressively discarding precise location while building richer semantic features.

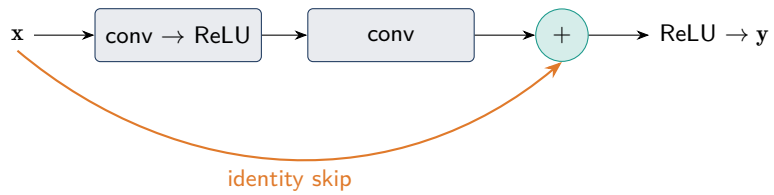
3 CNN architectures: a lineage

Model	Year	Key contribution
LeNet-5	1998	first practical CNN (digit recognition)
AlexNet	2012	ReLU, dropout, GPUs; ignited the deep-learning era
VGG	2014	simplicity: stacks of 3×3 convs; very deep
GoogLeNet/Inception	2014	multi-scale "inception" blocks; 1×1 convs
ResNet	2015	**skip connections** enable 100+ layer training
EfficientNet	2019	principled *compound scaling* of depth/width/resolution

3.1 ResNet and the skip connection

The central problem before ResNet: stacking more layers made networks *harder* to optimize (degradation), not just prone to overfitting. The fix is the **residual block**, which learns a residual $F(\mathbf{x})$ added to the input:

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x}.$$



Key idea

Skip connections give gradients a "highway" straight back to early layers, defeating the vanishing-gradient problem and making very deep networks trainable. This idea — add the input back — recurs everywhere: it is also at the heart of the Transformer (Phase 4) and diffusion U-Nets (Phase 5). If you remember one architectural trick, remember the residual connection.

3.2 1×1 convolutions and EfficientNet scaling

A 1×1 conv mixes channels without touching spatial dimensions — a cheap way to change depth and add nonlinearity (the "bottleneck" in ResNet). EfficientNet's lesson is that depth, width, and input resolution should be scaled *together* by a single compound coefficient, yielding better accuracy per FLOP.

4 Transfer learning & fine-tuning

You will almost never train a vision model from scratch. Instead, start from weights pretrained on a large dataset (ImageNet) and adapt them — this works because early features (edges, textures) are universal.

Definition

Feature extraction: freeze the pretrained backbone, replace and train only the final classifier head. **Fine-tuning:** unfreeze some or all backbone layers and continue training at a *small* learning rate so you refine rather than destroy the learned features.

Intuition

Rule of thumb by data size and similarity to ImageNet: *little data, similar domain* → freeze backbone, train head; *lots of data* → fine-tune more layers; *very different domain* (e.g. medical, satellite) → fine-tune deeply or train longer. Always use a much smaller LR for pretrained layers than for the new head ("discriminative learning rates").

```python
import torch, torch.nn as nn
from torchvision import models

model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V2)
for p in model.parameters():           # 1) freeze backbone
    p.requires_grad = False
model.fc = nn.Linear(model.fc.in_features, num_classes)   # 2) new head

opt = torch.optim.AdamW(model.fc.parameters(), lr=1e-3)    # train head only
# Later: unfreeze last block(s) and fine-tune at lr=1e-5 for a few epochs.
```

Common pitfall

Use the *same* preprocessing (resize, normalization mean/std) that the pretrained model expects — mismatched normalization silently wrecks accuracy. And when fine-tuning, a too-large learning rate causes "catastrophic forgetting": the pretrained features are overwritten before they can help.

5 Beyond classification

5.1 Object detection

Detection predicts *what* and *where* (bounding boxes + classes). Two families:

- **Two-stage** (Faster R-CNN): propose regions, then classify/refine them — accurate, slower.
- **One-stage** (YOLO, SSD, RetinaNet): predict boxes and classes in a single pass — fast, real-time. YOLO is the practical default for most applications.

Key concepts: anchor boxes, Intersection-over-Union (IoU), non-maximum suppression (NMS), and mean Average Precision (mAP) as the evaluation metric.

5.2 Image segmentation

Segmentation labels *every pixel*. **Semantic** segmentation classes each pixel; **instance** segmentation also separates object instances. **U-Net** (encoder–decoder with skip connections between matching resolutions) is the workhorse, especially in medical imaging; **Mask R-CNN** extends Faster R-CNN with a mask head. Note U-Net's skip connections recover the spatial detail lost during downsampling — and reappear in diffusion models.

6 Vision Transformers (ViT)

A ViT splits an image into fixed patches (e.g. 16×16), linearly embeds each patch as a token, adds positional embeddings, and feeds the sequence to a standard Transformer encoder (Phase 4). With enough data (or strong pretraining), ViTs match or beat CNNs.

Intuition

CNNs bake in locality and translation equivariance, so they are data-efficient. ViTs have weaker built-in priors but more flexibility, so they shine with large-scale pretraining and can model long-range relationships from layer one. In practice: CNNs (or hybrids like ConvNeXt) for smaller datasets; ViTs when you can leverage big pretrained checkpoints. Both are commodities on Hugging Face / timm today.

7 Practical skills

7.1 Data augmentation

Label-preserving transforms multiply your effective data and are the cheapest accuracy boost in vision: random crops, flips, rotations, color jitter, and stronger schemes (RandAugment, Mixup, CutMix). Use `torchvision.transforms` or **albumentations** (fast, rich, great for detection/segmentation where boxes/masks must transform too).

Common pitfall

Augment the *training* set only; validation/test must see clean, deterministic preprocessing. And keep augmentations realistic — vertically flipping street-sign images or digits teaches the model wrong invariances.

7.2 Efficient data pipelines

Use `Dataset` + `DataLoader` with multiple `num_workers`, pinned memory, and sensible batch sizes. Data loading — not the GPU — is often the bottleneck; cache decoded images, use efficient formats, and profile. On GPU, enable mixed precision (`torch.cuda.amp`) for a large speed/memory win.

8 Checkpoint project

Fine-tune and deploy an image classifier

Goal: fine-tune a pretrained ResNet (or ViT) on a custom dataset and serve it behind a simple API — playing directly to your backend strengths.

1. **Data:** pick or assemble a 3–10 class image dataset; build a `DataLoader` with train-time augmentation and correct normalization.
2. **Train:** feature-extract first (train the head), then fine-tune the last block(s) at a small LR. Track loss/accuracy in Weights & Biases or TensorBoard.
3. **Evaluate:** confusion matrix, per-class accuracy, and a look at the worst misclassifications (error analysis matters more than the single accuracy number).
4. **Interpret:** Grad-CAM heatmaps to confirm the model attends to the object, not the background (a classic "clever Hans" check).
5. **Deploy:** wrap the model in a FastAPI endpoint that accepts an uploaded image and returns top-k predictions; containerize it (foreshadowing Phase 7).

Definition of done: a repo with training script, an evaluation report with the confusion

matrix and Grad-CAM examples, and a running `/predict` endpoint with a sample `curl` command.

Phase 3

- **Stanford CS231n** — the definitive vision course (lecture videos + assignments, free online).
- **Hands-On Machine Learning** (Géron) — the CNN chapters.
- **fast.ai** — Practical Deep Learning, vision lessons (excellent transfer-learning craft).
- Papers (read abstracts + figures at least): *ResNet*, *EfficientNet*, *An Image is Worth 16×16 Words* (ViT).
- Libraries: **torchvision**, **timm** (huge model zoo), **albumentations**.

9 Phase 3 self-check

- Compute conv output sizes and parameter counts from memory.
- Explain why weight sharing and skip connections matter.
- Decide between feature extraction and fine-tuning given a dataset's size and domain.
- Contrast one-stage vs. two-stage detection and explain IoU/NMS/mAP.
- Explain when you would reach for a ViT vs. a CNN.
- Fine-tune a pretrained model and serve it behind an endpoint.

PHASE 4

Sequence Models, NLP & Transformers

From RNNs to attention to LLMs, tokenization, fine-tuning, and RAG

Phase goal

Master the architecture behind modern AI. Build a Transformer from first principles, understand tokenization (including its CJK quirks), and stand up a retrieval-augmented generation system. This is the highest-leverage phase in the curriculum.

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 Why this is the highest-leverage phase

The Transformer is the single most important architecture in modern machine learning. It powers large language models, underpins modern vision (ViT, Phase 3), audio, code, and multimodal systems, and is where the overwhelming majority of current research and commercial value concentrates. If your time is limited, this phase plus Phase 7 (deployment) is the highest-return path. We build up to it through classical sequence models so the design choices feel inevitable rather than arbitrary.

2 Classical sequence models

2.1 RNNs and the vanishing-gradient problem

A recurrent network processes a sequence one step at a time, maintaining a hidden state $\mathbf{h}_t$ that summarizes the past:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}).$$

Backpropagation through time multiplies many Jacobians together; if their norms are < 1 the gradient vanishes (the network forgets long-range context), if > 1 it explodes. This is the central limitation of vanilla RNNs.

Intuition

The vanishing-gradient problem means a plain RNN struggles to connect "The *cat* … [50 words] … *was* hungry" — by the time it reaches "was," the signal from "cat" has decayed. Every later innovation (gates, attention) is fundamentally about preserving information across long distances.

2.2 LSTMs and GRUs

LSTMs add a **cell state** and multiplicative **gates** (input, forget, output) that let the network learn what to keep, forget, and expose — creating a gradient highway across time (the same "additive path" insight as ResNet's skip connection). GRUs are a simpler two-gate variant. These dominated NLP from roughly 2014–2017.

2.3 Sequence-to-sequence and the bottleneck

Encoder–decoder (seq2seq) models compress an input sequence into a single fixed vector, then decode the output (machine translation, summarization). The flaw: cramming an entire sentence into one vector is a bottleneck. The fix — letting the decoder *look back* at all encoder states — is **attention**, and it changed everything.

3 Attention & Transformers

3.1 The attention mechanism

Attention lets a model, for each position, retrieve a weighted combination of *all* positions based on relevance. Cast it as a soft dictionary lookup with **queries**, **keys**, and **values**.

Definition

Scaled dot-product attention. Given queries $\mathbf{Q}$, keys $\mathbf{K}$, values $\mathbf{V}$ (rows are tokens, dimension d_k):

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}.$$

$\mathbf{QK}^\top$ scores every query against every key; the $\sqrt{d_k}$ scaling keeps the softmax in a sensible range; softmax turns scores into weights that sum to 1; multiplying by $\mathbf{V}$ returns a weighted blend of values.

Intuition

For each word, attention asks "which other words should I pay attention to?" and builds a context-aware representation accordingly. "It" in "the trophy didn't fit in the suitcase because *it* was too big" can attend to "trophy." The scaling by $\sqrt{d_k}$ matters: without it, large dot products push softmax into saturation where gradients vanish.

3.2 Self-attention and multi-head attention

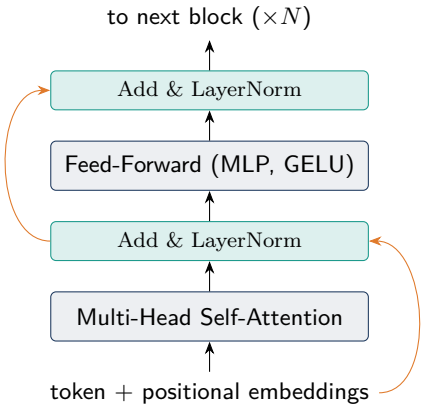
In **self-attention**, $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ all come from the same sequence (each token attends to the others). **Multi-head** attention runs several attention operations in parallel with different learned projections, then concatenates them — so different heads can specialize (one tracks syntax, another coreference, another position).

Key idea

Self-attention's key advantages over RNNs: (1) it connects any two positions in *one* step (no long-range decay), and (2) it is fully *parallel* across positions (no sequential dependency), which is what made training on internet-scale data feasible. The cost is $O(n^2)$ in sequence length n — the motivation behind FlashAttention and long-context research.

3.3 The Transformer, end to end

The original "Attention Is All You Need" (2017) architecture stacks identical blocks. Each block has two sublayers — multi-head self-attention and a position-wise feed-forward network — each wrapped with a **residual connection** and **layer normalization**.



Positional encoding. Because attention is permutation-invariant (it has no inherent notion of order), we inject position information — via fixed sinusoids (original) or learned/rotary embeddings (RoPE, now standard in LLMs). **LayerNorm placement** matters: modern LLMs use *pre-norm* (normalize before each sublayer) for more stable deep training.

```
import torch, torch.nn.functional as F
def scaled_dot_product_attention(Q, K, V, mask=None):
    d_k = Q.size(-1)
    scores = (Q @ K.transpose(-2, -1)) / d_k**0.5      # (.., n, n)
```

```
    if mask is not None:                                    # causal/padding mask
        scores = scores.masked_fill(mask == 0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights @ V                                      # (.., n, d_v)
```

4 Modern NLP & LLMs

4.1 Tokenization

Models operate on integer token IDs, not raw text. Subword tokenizers strike a balance between character-level (long sequences) and word-level (huge vocab, out-of-vocabulary words):

- **BPE** (Byte-Pair Encoding): iteratively merge the most frequent symbol pair. Used by GPT.
- **WordPiece**: similar, merges by likelihood. Used by BERT.
- **SentencePiece / Unigram**: language-agnostic, operates on raw bytes/characters — no pre-tokenization by spaces.

CJK tokenization has real quirks

English is conveniently delimited by spaces; Chinese is not. The sentence 我喜欢机器学习 ("I like machine learning") has no spaces, so a tokenizer cannot rely on whitespace to find word boundaries. Practical consequences for your bilingual document work:

- Most modern LLM tokenizers are **byte-level BPE**, so they handle Chinese, but often split a single character into multiple byte-tokens — inflating token counts (and API cost) for CJK text relative to English.
- A word like 机器学习 ("machine learning") may become several tokens; segmentation is implicit and learned, not linguistic.
- Always inspect how *your* tokenizer treats your corpus: token counts, whether rare characters round-trip, and whether mixed Chinese/English/code text fragments cleanly.

Inspecting tokenization

Tokenize the Chinese phrase 机器学习很有趣 ("machine learning is fun") and its English equivalent, then compare how each is split:

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-multilingual-cased")
zh = "..."   # the Chinese phrase above
print(tok.tokenize(zh))                    # CJK: roughly per-character pieces
print(tok.tokenize("machine learning"))  # English: subword pieces
# Compare token COUNTS across languages for the same meaning -> cost/latency.
```

The Chinese string typically yields about one token per character (six tokens here), while a byte-level BPE may emit several byte-tokens *per* character. Run this on your own Chinese↔English documents and compare token counts; it directly affects context-window budgeting and API spend.

4.2 Pretraining objectives

- **Masked LM (BERT-style, encoder):** mask random tokens and predict them from both sides — bidirectional context, great for *understanding* tasks (classification, NER, embeddings).

- **Autoregressive LM (GPT-style, decoder):** predict the next token from the left context only (causal mask) — great for *generation*. This is the basis of modern chat LLMs.

4.3 Adapting models: fine-tune vs. prompt vs. PEFT

Approach	When to use
Prompting / in-context	zero/few-shot; no training; fastest to try; great baseline
Full fine-tuning	lots of task data, need maximum quality, have the compute
PEFT (LoRA/QLoRA)	adapt a large model cheaply by training small low-rank adapters

LoRA / QLoRA

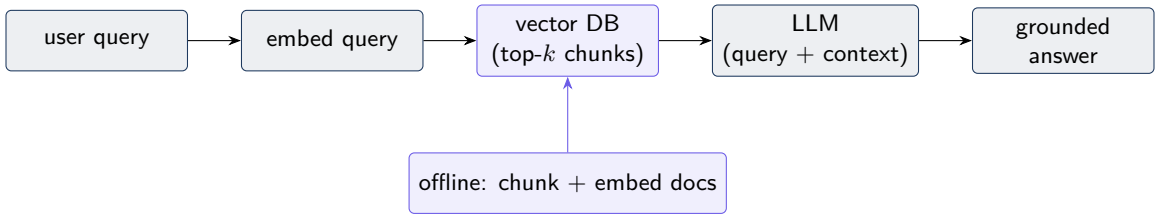
LoRA freezes the pretrained weights and learns a low-rank update $\Delta\mathbf{W} = \mathbf{BA}$ (with $\mathbf{A}, \mathbf{B}$ tiny) for each weight matrix — training $<1\%$ of the parameters while nearly matching full fine-tuning. **QLoRA** adds 4-bit quantization of the frozen base, letting you fine-tune a multi-billion-parameter model on a single consumer GPU. This is the dominant practical way to specialize LLMs today, and it is a direct application of the low-rank idea you met via SVD in Phase 0.

4.4 Embeddings & vector search

An embedding maps text to a dense vector where semantic similarity is geometric proximity (cosine similarity — the Phase 0 dot product again). Embeddings power semantic search, clustering, deduplication, recommendation, and retrieval. Store them in a vector index (FAISS, Chroma, pgvector) and query by nearest neighbors — approximate (ANN) algorithms like HNSW make this fast at scale.

5 Retrieval-Augmented Generation (RAG)

LLMs hallucinate and have a knowledge cutoff. RAG grounds them in your data: retrieve relevant chunks, then have the model answer *using* that context.



Intuition

RAG = open-book exam for an LLM. The quality ceiling is set by *retrieval*, not the model: if the right chunk is not retrieved, no amount of prompting saves you. Most "my RAG is bad" problems are retrieval problems — chunking strategy, embedding quality, and ranking — so instrument and evaluate retrieval separately from generation.

Common pitfall

Common RAG failure modes: chunks too large (dilute relevance) or too small (lose context); no metadata filtering; embedding the question and documents with mismatched models; and no reranking. Add a cross-encoder reranker, tune chunk size/overlap, and consider hybrid (keyword + vector) search. Evaluate with retrieval metrics (recall@k) and answer-faithfulness checks.

5.1 Evaluating generative models

Generation is hard to score. Classic n-gram metrics (BLEU, ROUGE) are weak proxies. Modern practice combines: task-specific metrics, *LLM-as-judge* (with care for bias), human evaluation on a curated set, and for RAG, **faithfulness/groundedness** and **answer relevance** (e.g. the RAGAS framework). Always keep a small, hand-built "golden" eval set.

6 The Hugging Face ecosystem

- **transformers**: thousands of pretrained models behind one API (`AutoModel`, `AutoTokenizer`, `Trainer`, `pipeline`).
- **datasets**: streaming, memory-mapped datasets; easy preprocessing.
- **tokenizers**: fast Rust-backed tokenizers; train your own.
- **PEFT** & **bitsandbytes**: LoRA/QLoRA and quantization.
- Deployment quantization: **GGUF** (llama.cpp, CPU/edge), **bitsandbytes/GPTQ/AWQ** (GPU).

```
from transformers import pipeline
clf = pipeline("sentiment-analysis")          # downloads a pretrained model
print(clf("This curriculum is fantastic!"))    # [{'label': 'POSITIVE', ...}]

# Fine-tuning skeleton with the Trainer API:
from transformers import AutoModelForSequenceClassification, TrainingArguments, Trainer
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
args = TrainingArguments(output_dir="out", eval_strategy="epoch",
                         per_device_train_batch_size=16, num_train_epochs=3)
trainer = Trainer(model=model, args=args, train_dataset=ds_train, eval_dataset=ds_val)
trainer.train()
```

7 Checkpoint project

Build a RAG pipeline over your own docs

Goal: a question-answering system grounded in a document set you care about (e.g. your own technical/API docs) — a natural fit for your backend skill set.

1. **Ingest:** load documents, chunk them (experiment with size/overlap), attach metadata.
2. **Embed & index:** embed chunks with a sentence-embedding model; store in FAISS, Chroma, or **pgvector** (the latter slots neatly into an existing Postgres backend).

3. **Retrieve:** top-k nearest neighbors for a query; add a reranker and optional hybrid keyword search.
4. **Generate:** construct a prompt with the retrieved context and call an LLM; cite sources in the answer.
5. **Serve:** expose a `/ask` REST endpoint (FastAPI); return the answer plus the source chunks.
6. **Evaluate:** build a small golden Q&A set; measure retrieval recall@k and answer faithfulness; iterate on chunking and retrieval.

Stretch: make it bilingual — test Chinese and English queries over mixed-language docs and compare retrieval quality and token costs (ties back to the tokenization section).

Phase 4

- **Stanford CS224n** — NLP with Deep Learning (the definitive course).
- **Andrej Karpathy** — "Let's build GPT" (builds a Transformer from scratch; pair with his earlier backprop video).
- **Hugging Face** free NLP course (huggingface.co/learn).
- **Jay Alammar** — "The Illustrated Transformer" (the clearest visual explainer).
- Papers: *Attention Is All You Need*; *BERT*; *GPT-1/2/3*; *LoRA*.

8 Phase 4 self-check

- Explain the vanishing-gradient problem and how LSTMs and attention each address it.
- Write scaled dot-product attention from memory and explain every term, including $\sqrt{d_k}$.
- Describe the full Transformer block and why residual connections and positional encodings are needed.
- Contrast encoder (BERT) vs. decoder (GPT) pretraining and when to use each.
- Explain BPE/WordPiece/SentencePiece and the specific quirks of CJK tokenization.
- Explain LoRA/QLoRA and why low-rank updates work.
- Draw the RAG architecture and name the most common failure mode.

PHASE 5
Generative Models

Autoencoders, VAEs, GANs, and diffusion — how machines create

Phase goal

Understand how models learn to *generate* data: the latent-variable view (VAEs), the adversarial game (GANs), and the denoising process (diffusion) behind modern image and audio generation.

Estimated time: 4–5 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 Discriminative vs. generative

Everything so far has been mostly *discriminative*: learn $p(y \mid \mathbf{x})$ to predict a label. **Generative** models learn the data distribution $p(\mathbf{x})$ itself, so you can sample new examples that look like the training data — images, audio, molecules, text. The central difficulty: $p(\mathbf{x})$ for real data (say, 256×256 images) is absurdly high-dimensional and complex. The three model families in this phase are three different bargains for making that tractable.

Key idea

All three families exploit the same observation: real data lives near a low-dimensional *manifold* inside its high-dimensional space (the SVD/PCA intuition from Phase 0). VAEs learn an explicit latent code for that manifold; GANs learn to map noise onto it; diffusion learns to walk back onto it from noise. Master the manifold picture and the rest follows.

2 Autoencoders

An autoencoder learns to compress and reconstruct: an **encoder** maps $\mathbf{x}$ to a low-dimensional latent $\mathbf{z}$, a **decoder** reconstructs $\hat{\mathbf{x}}$, trained to minimize reconstruction error $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$. The bottleneck forces the network to keep only the essential structure — a nonlinear generalization of PCA, useful for denoising, anomaly detection, and representation learning.

Common pitfall

A plain autoencoder is *not* a good generator. Its latent space has "holes" — sampling a random $\mathbf{z}$ usually decodes to garbage because the encoder never organized the latent space to be continuous or complete. Fixing exactly this is the motivation for the VAE.

3 Variational Autoencoders (VAEs)

A VAE makes the latent space *probabilistic and well-structured* so you can sample from it. The encoder outputs a distribution $q(\mathbf{z} \mid \mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ rather than a point, and we regularize that distribution toward a standard Gaussian prior.

Definition

VAEs maximize the **Evidence Lower Bound (ELBO)**:

$$\mathcal{L} = \underbrace{\mathbb{E}_{q(\mathbf{z}|\mathbf{x})}[\log p(\mathbf{x} \mid \mathbf{z})]}_{\text{reconstruction}} - \underbrace{D_{\mathrm{KL}}\big(q(\mathbf{z} \mid \mathbf{x}) \,\|\, p(\mathbf{z})\big)}_{\text{regularize latent to prior}}.$$

The KL term pulls each encoded distribution toward $\mathcal{N}(\mathbf{0}, \mathbf{I})$, filling the latent space so that sampling $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and decoding produces plausible new data.

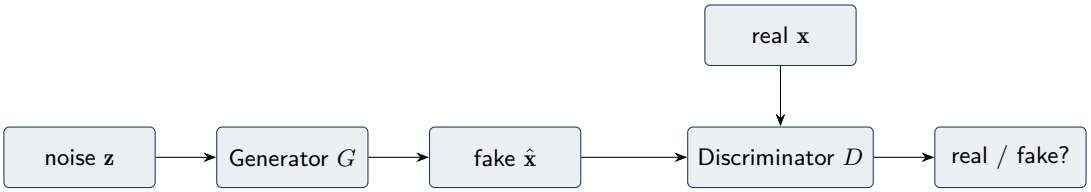
Intuition

The **reparameterization trick** — writing $\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$ with $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ — moves the randomness "outside" the network so gradients can flow through $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. This is the key engineering trick that makes VAEs trainable by ordinary backprop. VAEs produce smooth, interpolatable latent spaces but somewhat blurry samples (a consequence of the Gaussian likelihood and the averaging it encourages).

4 Generative Adversarial Networks (GANs)

A GAN pits two networks against each other. The **generator** G maps noise $\mathbf{z}$ to fake samples; the **discriminator** D tries to distinguish real from fake. They play a minimax game:

$$\min_G \max_D \; \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))].$$



Intuition

As D gets better at spotting fakes, G is pushed to make more convincing ones — an arms race that drives sample quality up. At the theoretical optimum, G's distribution matches the data and D is reduced to guessing (50%). GANs produce sharp, high-fidelity images but are notoriously finicky to train.

Common pitfall

GAN failure modes you must recognize: **mode collapse** (the generator produces only a few outputs, ignoring data diversity); **training instability / non-convergence** (the two losses oscillate rather than settle); and **vanishing discriminator gradients** (if D wins too easily, G gets no signal). Mitigations developed over years: Wasserstein loss (WGAN-GP), spectral normalization, two-timescale updates (TTUR), and careful architecture (DCGAN/StyleGAN). Evaluate with FID, not loss values.

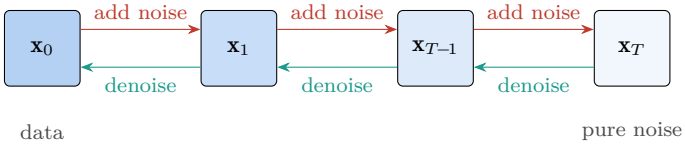
5 Diffusion models

Diffusion is the backbone of modern image and audio generation (Stable Diffusion, DALL · E, Imagen, Midjourney). The idea is strikingly simple: **gradually add noise to data, then learn to reverse it**.

Definition

Forward process: over T steps, progressively add Gaussian noise to a sample until it is pure noise — a fixed, parameter-free Markov chain $q(\mathbf{x}_t \mid \mathbf{x}_{t-1})$. **Reverse process:** train a neural network $\epsilon_\theta(\mathbf{x}_t, t)$ to predict the noise added at each step, so you can iteratively *denoise* from random noise back to a clean sample. The DDPM training loss is beautifully simple:

$$\mathcal{L} = \mathbb{E}_{\mathbf{x}_0,t,\boldsymbol{\epsilon}}\left[\; \|\boldsymbol{\epsilon} - \epsilon_\theta(\mathbf{x}_t, t)\|^2 \;\right].$$



Key idea

Diffusion trades the GAN's unstable adversarial game for a *stable regression* problem (predict the noise), which is why it scaled so well and largely displaced GANs for high-quality image generation. The price is slow sampling — many denoising steps — which spawned fast samplers (DDIM, DPM-Solver) and distillation. The denoising network is typically a **U-Net** (Phase 3) or, increasingly, a Transformer (DiT).

5.1 Conditional and text-to-image generation

To control generation, condition the model on a signal: a class label, or text. **Latent diffusion** (Stable Diffusion) runs the diffusion process in a compressed VAE latent space — far cheaper than pixel space. Text conditioning uses a text encoder (often **CLIP**, which aligns image and text embeddings in a shared space) and **classifier-free guidance** to steer samples toward the prompt. Conceptually: CLIP says "how well does this image match the text," and guidance nudges each denoising step to increase that match.

	VAE	**GAN**	**Diffusion**
Training	stable	unstable	stable
Sample quality	blurry	sharp	state-of-the-art
Sampling speed	fast	fast	slow (many steps)
Latent space	smooth, usable	less structured	(in latent diffusion)
Mode coverage	good	can collapse	good

6 Checkpoint project

Train a small generative model

Goal: watch generation emerge firsthand on a simple dataset (Fashion-MNIST or MNIST).

- **Track A — VAE:** build encoder/decoder, implement the ELBO with the reparameterization trick, train, then (1) sample new images from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ and (2) interpolate between two latent codes to see the smooth manifold.
- **Track B — DDPM:** implement the forward noising schedule and a small U-Net noise predictor; train on the simple ϵ-prediction loss; sample by iterative denoising and visualize the reverse trajectory.

Definition of done: a grid of generated samples, a latent interpolation (VAE) or a denoising trajectory (DDPM), the loss curve, and a paragraph comparing what each approach was like to train.

Stretch: add class conditioning (generate a specified digit/garment) and, for DDPM, compare DDPM vs. DDIM sampling speed/quality.

Phase 5

- **Lilian Weng**'s blog — "What are Diffusion Models?" and her GAN/VAE posts (exceptionally clear).
- **Hugging Face** *diffusers* library + its tutorials (the practical entry point).
- Papers (skim for intuition): *Auto-Encoding Variational Bayes* (VAE); *Generative Adversarial Networks*; *Denoising Diffusion Probabilistic Models* (DDPM); *High-Resolution Image Synthesis with Latent Diffusion* (Stable Diffusion).

- Karpathy / fast.ai diffusion-from-scratch walkthroughs.

7 Phase 5 self-check

- Explain why a plain autoencoder cannot generate and how the VAE fixes it.
- Write the ELBO and explain the reconstruction vs. KL terms and the reparameterization trick.
- Describe the GAN minimax game and name three failure modes with mitigations.
- Explain the forward/reverse diffusion processes and the simple ϵ-prediction loss.
- Compare VAE/GAN/diffusion on quality, stability, and sampling speed.
- Explain latent diffusion and classifier-free guidance at a conceptual level.

PHASE 6
Reinforcement Learning

MDPs, value functions, Q-learning, policy gradients, PPO, and RLHF

Phase goal

Understand learning by interaction and reward: Markov decision processes, the value/policy duality, deep Q-networks, policy gradients, and how PPO powers the RLHF that aligns modern LLMs.

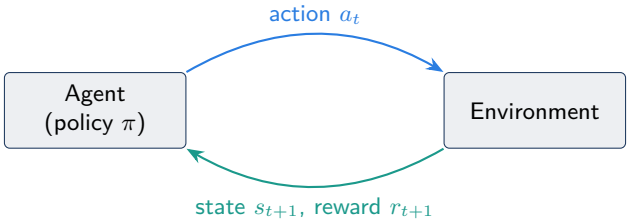
Estimated time: 3–4 weeks (optional) | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 A different learning paradigm

Supervised learning has labeled answers; reinforcement learning (RL) has only *reward*. An **agent** takes **actions** in an **environment**, which returns a new **state** and a scalar **reward**; the agent must learn a behavior (**policy**) that maximizes cumulative reward over time. There is no teacher saying "the right action was X" — only delayed, often sparse, feedback. This phase is optional for most ML careers but conceptually beautiful, and it is essential for understanding RLHF (how chat LLMs are aligned).



Key idea

The defining challenges of RL: (1) the **credit-assignment problem** — which past action caused this reward? (2) the **exploration vs. exploitation** tradeoff — try new actions to learn, or exploit what you know works? (3) **delayed reward** — a move in chess pays off twenty moves later. Every RL algorithm is a different answer to these three questions.

2 Markov Decision Processes

RL problems are formalized as an **MDP**: states $\mathcal{S}$, actions $\mathcal{A}$, transition dynamics $P(s' \mid s, a)$, reward $R(s, a)$, and discount $\gamma \in [0, 1)$. The **Markov property** assumes the next state depends only on the current state and action — the present summarizes the past.

Definition

The agent seeks a policy $\pi(a \mid s)$ maximizing the expected **discounted return**

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}.$$

The discount γ makes the infinite sum finite and encodes "prefer sooner rewards": γ near 1 is far-sighted, near 0 is myopic.

2.1 Value functions and the Bellman equation

The **state-value** $V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$ is the expected return from state s under π. The **action-value** $Q^\pi(s, a)$ is the expected return from taking a in s, then following π. Both satisfy a recursive **Bellman equation**:

$$Q^\pi(s, a) = \mathbb{E}\big[r + \gamma\, Q^\pi(s', a')\big].$$

Intuition

The Bellman equation says "the value of here = immediate reward + discounted value of where I land next." This recursion — the same self-referential structure as dynamic programming — is the foundation of nearly every value-based RL method. The optimal Q^* replaces $Q^\pi(s', a')$ with $\max_{a'} Q^*(s', a')$: act greedily with respect to the best future.

3 Q-learning and Deep Q-Networks

Q-learning learns Q^* directly by bootstrapping toward the Bellman target:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[\underbrace{r + \gamma \max_{a'} Q(s',a')}_{\text{TD target}} - Q(s,a)\big].$$

It is **off-policy** (learns the optimal policy while behaving exploratorily, e.g. ϵ-greedy). For small discrete problems Q is a table; for large/continuous states we approximate it with a neural network — a **Deep Q-Network (DQN)**.

Key idea

DQN (the 2015 Atari breakthrough) made neural-network Q-learning stable with two tricks: **experience replay** (store transitions and train on random minibatches, breaking correlation between consecutive samples) and a **target network** (a slowly-updated copy used for the TD target, preventing the "chasing a moving target" instability). These two ideas are the practical core of value-based deep RL.

Common pitfall

Naively plugging a neural net into Q-learning diverges — this is the "deadly triad" (function approximation + bootstrapping + off-policy). Replay and target networks are not optional polish; they are what makes it work. Other gotchas: reward scaling matters a lot, and DQN handles only *discrete* action spaces.

4 Policy gradient methods

Instead of learning values and acting greedily, **policy-gradient** methods directly parameterize and optimize the policy $\pi_\theta(a \mid s)$ — naturally handling continuous actions and stochastic policies.

Definition

The **REINFORCE** estimator ascends the expected return:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t\big].$$

Intuitively: increase the probability of actions that led to high return, decrease it for low return — weighted by how good the outcome was.

4.1 Actor–Critic

REINFORCE has high variance (it uses the noisy full return G_t). **Actor–Critic** reduces variance by learning both a policy (*actor*) and a value estimate (*critic*), replacing G_t with an **advantage** $A(s,a) = Q(s,a) - V(s)$ — "how much better than average was this action?" This is the workhorse structure behind modern policy-gradient algorithms.

5 PPO — the workhorse

Proximal Policy Optimization is the most widely used policy-gradient algorithm: reliable, reasonably sample-efficient, and relatively easy to tune. Its key idea is to take the biggest improvement step possible *without* moving the policy too far from the previous one (large updates destabilize RL).

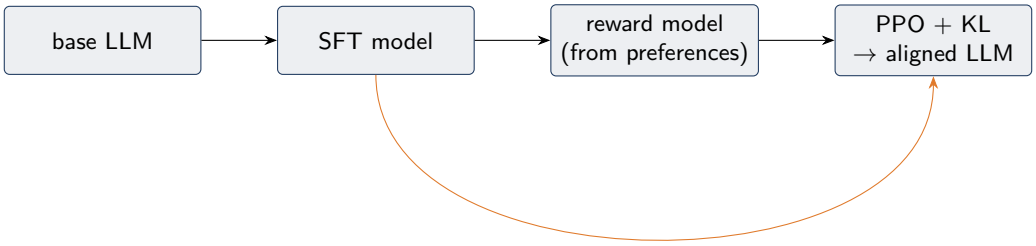
Key idea

PPO optimizes a **clipped surrogate objective**: it maximizes the advantage-weighted probability ratio $r_t(\theta) = \pi_\theta / \pi_{\theta_{\text{old}}}$, but *clips* that ratio to $[1-\epsilon, 1+\epsilon]$ so a single update cannot change the policy drastically. This "trust-region-lite" behavior is why PPO is robust enough to be the default in everything from robotics to RLHF.

6 RLHF — how modern LLMs get aligned

Reinforcement Learning from Human Feedback is how a raw, next-token-predicting LLM becomes a helpful, harmless assistant. The classic three-stage recipe:

1. **Supervised fine-tuning (SFT):** fine-tune the base LLM on high-quality demonstration data.
2. **Reward model:** collect human *preference* comparisons ("response A is better than B") and train a model to predict that preference — a learned reward.
3. **RL optimization:** use **PPO** to optimize the LLM against the reward model, with a KL penalty that keeps it close to the SFT model (so it does not "hack" the reward into gibberish).



Intuition

RLHF is why this curriculum's Phase 6 is worth skimming even if you never train a game-playing agent: the alignment of the assistants you use daily is a PPO loop over a learned reward. Newer methods like **DPO** (Direct Preference Optimization) skip the explicit reward model and RL loop, optimizing preferences directly with a simple classification-style loss — simpler and increasingly popular, but the RLHF framing remains the conceptual anchor.

7 Checkpoint project

Train a DQN on a Gym environment

Goal: get an agent to solve a classic control task and internalize the RL loop.

1. Start with **CartPole** (`gymnasium`); it should be solvable in minutes.
2. Implement DQN with **experience replay** and a **target network**; use an ϵ-greedy policy with decay.
3. Plot the episode-reward curve; confirm it rises to the solved threshold and discuss its variance.
4. **Stretch:** move to **LunarLander**; try a policy-gradient method (PPO via Stable-Baselines3) and compare sample efficiency and stability.

Definition of done: a training script, a reward-vs-episode plot, a short note on how replay and the target network affected stability, and (ideally) a rendered GIF of the trained agent.

```python
# Core DQN update step (PyTorch sketch)
states, actions, rewards, next_states, dones = replay.sample(batch)
q = qnet(states).gather(1, actions)                       # Q(s, a)
with torch.no_grad():
    q_next = target_net(next_states).max(1, keepdim=True).values
    target = rewards + gamma * q_next * (1 - dones)       # TD target
loss = F.smooth_l1_loss(q, target)                        # Huber loss
opt.zero_grad(); loss.backward(); opt.step()
# Periodically: target_net.load_state_dict(qnet.state_dict())
```

Phase 6

- **Sutton & Barto** — *Reinforcement Learning: An Introduction* (free PDF; the canonical text).
- **David Silver** — RL course (DeepMind/UCL, YouTube; the classic lectures).
- **OpenAI Spinning Up in Deep RL** — the best practical bridge to code, with clean PPO/VPG implementations.
- **Stable-Baselines3** — reliable reference implementations for experimentation.
- For RLHF: the InstructGPT paper, and the DPO paper for the modern simplification.

8 Phase 6 self-check

- Define an MDP and explain the discount factor and the Markov property.
- Write the Bellman equation and explain bootstrapping.
- Explain why DQN needs experience replay and a target network.
- Contrast value-based (DQN) and policy-gradient (REINFORCE/PPO) methods.
- Explain PPO's clipped objective in one sentence.
- Describe the three stages of RLHF and what the KL penalty prevents.

PHASE 7

MLOps, Deployment & Production Systems

Experiment tracking, serving, containers, monitoring, optimization, and CI/CD for ML

Phase goal

Turn models into reliable products. Track experiments, package and serve models, monitor for drift, optimize for latency/cost, and automate the whole lifecycle. This phase plays directly to your backend and infrastructure strengths.

Estimated time: 4–6 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 Why MLOps is your edge

A model in a notebook creates zero value. Value comes from a model that serves predictions reliably, cheaply, and correctly — and keeps doing so as data shifts. This is software and systems engineering with a few ML-specific twists, which means your existing strengths (APIs, nginx, OAuth, containers, CI/CD) transfer almost wholesale. This is the phase where you can differentiate fastest from people who can only train models.

Key idea

The "ops" problems unique to ML: (1) the artifact is *data + code + model*, not just code, so all three must be versioned and reproducible; (2) models *decay* silently as the world changes (drift) — there is no exception or stack trace, just quietly worse predictions; (3) training and serving can subtly disagree (training/serving skew). Most of MLOps exists to manage these three realities.

2 Experiment tracking & reproducibility

Before deployment comes the discipline of knowing *what you trained and how*. Log every run's parameters, metrics, code version, data version, and artifacts.

- **MLflow** (open-source, self-hostable) or **Weights & Biases** (hosted, great UI) for tracking and the model registry.
- **Reproducibility hygiene:** pin dependencies, set random seeds, version data (DVC or dataset hashes), and capture the git SHA with each run.

```
import mlflow
with mlflow.start_run():
    mlflow.log_params({"lr": 1e-3, "depth": 6})
    mlflow.log_metric("val_auc", auc)
    mlflow.sklearn.log_model(model, "model")      # versioned artifact
    mlflow.set_tag("git_sha", current_git_sha())
```

Common pitfall

"It worked on my machine / in my notebook" is the enemy. If you cannot reproduce a result from a logged run — same data, same code, same environment — you do not really have that result. Build reproducibility in from the first experiment; retrofitting it after a model is in production is painful.

3 Packaging & serving

3.1 Inference patterns

Pattern	Use when
Batch / offline	predictions can be precomputed (e.g. nightly scores written to a DB)
Real-time / online	low-latency per-request prediction (an API behind your app)
Streaming	continuous event scoring (fraud on a transaction stream)
Edge / on-device	privacy, offline, or latency needs (mobile CoreML/TFLite)

3.2 Serving frameworks

Wrap the model in an API (**FastAPI**/Flask) for full control, or use a dedicated server (**TorchServe**, **Triton**, BentoML, KServe) for batching, versioning, and metrics out of the box. Export to **ONNX** for a framework-agnostic, optimized runtime.

```python
from fastapi import FastAPI
from pydantic import BaseModel
import torch

app = FastAPI()
model = torch.jit.load("model.pt").eval()        # TorchScript artifact

class Req(BaseModel):
    features: list[float]

@app.post("/predict")
def predict(req: Req):
    x = torch.tensor([req.features])
    with torch.no_grad():
        prob = model(x).softmax(-1)[0].tolist()
    return {"probabilities": prob}

@app.get("/health")
def health():                                     # liveness/readiness probe
    return {"status": "ok"}
```

3.3 Containerization

Docker is the unit of deployment. For ML, mind image size (multi-stage builds), CUDA/-driver compatibility for GPU images, and pinning the exact inference dependencies. The same container runs in CI, staging, and production — eliminating environment skew.

Intuition

The serving stack should feel familiar: it is a stateless service behind a load balancer, with health checks, autoscaling, observability, and a CI/CD pipeline — exactly your backend wheelhouse. The ML-specific additions are the model artifact, its preprocessing, and drift monitoring.

4 Versioning & registries

A **model registry** (MLflow Registry, W&B, SageMaker) is the source of truth: it stores versioned model artifacts with stage labels (staging/production), lineage (which run/data produced it), and enables one-click rollback. Treat model promotion like a deploy: reviewed, gated, and reversible.

5 Monitoring

Once live, a model needs more monitoring than ordinary services, because it can fail *silently*.

- **Operational:** latency, throughput, error rate, resource use (the usual SRE signals).
- **Data drift:** input distribution shifts from training (monitor feature statistics; tests like PSI or KS).
- **Concept drift:** the input→output relationship changes (the world moved; yesterday's patterns no longer hold).
- **Performance:** the real metric (accuracy/AUC) once ground-truth labels arrive — often delayed, so use proxies meanwhile.
- **Prediction monitoring:** output distribution, confidence, and (for LLMs) safety/quality checks.

Common pitfall

The classic production failure is silent degradation: no errors, no alerts, just a model whose accuracy quietly slid because the input data drifted (a new product category, a changed upstream feature, seasonality). Alert on *input drift* as an early warning, since true performance labels often arrive days or weeks late. Also guard against the **training/serving skew** where preprocessing differs between offline training and online serving — a leading cause of "great offline, bad online."

6 Optimization for production

Making models fast and cheap is often as valuable as making them accurate.

- **Quantization:** use int8/fp16 instead of fp32 — big speed/memory wins with small accuracy loss (post-training or quantization-aware).
- **Distillation:** train a small "student" to mimic a large "teacher" — much smaller, nearly as good.
- **Pruning:** remove redundant weights/structures.
- **Graph/runtime optimization:** ONNX Runtime, TensorRT (fuse ops, pick optimal kernels).
- **Batching:** dynamic batching of concurrent requests raises throughput dramatically (tune against your latency SLO).

Intuition

A 4-bit quantized model that fits on one GPU and answers in 50 ms can be worth far more than a 2%-more-accurate model that needs four GPUs and 400 ms. In production, latency, cost, and reliability are first-class metrics — not afterthoughts. This is also where your earlier SVD/low-rank (Phase 0) and LoRA (Phase 4) intuitions pay off.

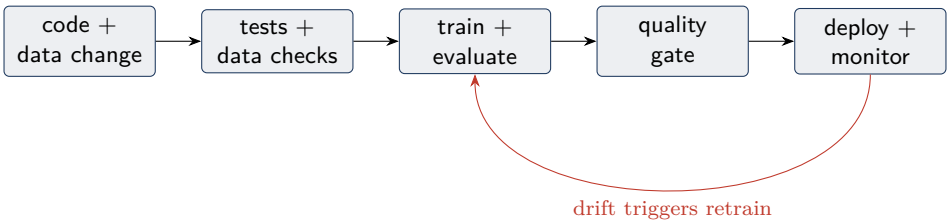
7 Feature stores & pipelines

A **feature store** (e.g. Feast) centralizes feature definitions and serves them consistently to both training (offline) and serving (online) — the standard cure for training/serving skew. You do

not need one for small projects, but understand the concept: "compute a feature once, use it everywhere, consistently."

8 CI/CD for ML

Extend familiar CI/CD with ML stages:



Add: data validation (Great Expectations), automated retraining triggers (on drift or schedule), a quality gate (don't deploy if the candidate underperforms the incumbent on a held-out set), and progressive rollout (canary / shadow / A-B, connecting back to Phase 1).

9 Checkpoint project

Productionize an earlier model end to end

Goal: take any model from Phases 1–4 and build the full production loop.

1. **Track:** retrain it under MLflow/W&B with logged params, metrics, and a registered model artifact.
2. **Serve:** wrap it in a FastAPI service with `/predict` and `/health`; validate inputs with Pydantic.
3. **Containerize:** a slim, pinned Docker image; run it locally and hit it with `curl`.
4. **Monitor:** log requests/predictions; compute a simple input-drift metric and expose Prometheus-style metrics.
5. **Automate:** a CI pipeline (GitHub Actions) that runs tests, builds the image, and (optionally) deploys; gate on a minimum eval score.
6. **Document:** a README with the architecture diagram, API contract, and a runbook (how to roll back, what alerts mean).

Definition of done: one command brings the service up; a load test reports latency/throughput; a simulated data shift trips your drift metric. **Stretch:** add ONNX export + quantization and report the latency improvement; add a canary deploy.

Phase 7

- **Designing Machine Learning Systems** (Chip Huyen) — the practitioner's bible for this phase.
- **Machine Learning Engineering** (Andriy Burkov) — free online; broad and practical.
- **Made With ML** (madewithml.com) — strong, free, code-first MLOps course.
- Google's *Rules of ML* and the *MLOps: Continuous delivery* whitepaper.
- Tools to get hands-on with: MLflow, FastAPI, Docker, ONNX Runtime, Prometheus/-Grafana, GitHub Actions.

10 Phase 7 self-check

- Explain why ML systems version data + code + model, not just code.
- Choose an inference pattern (batch/online/streaming/edge) for a given use case.
- Stand up a containerized prediction API with health checks.

- Distinguish data drift, concept drift, and training/serving skew, and say how you'd detect each.
- Pick an optimization (quantization/distillation/batching) for a latency or cost target.
- Describe a CI/CD pipeline for ML with a quality gate and a retraining trigger.

PHASE 8
Capstone Projects

Compounding value by building end-to-end systems in your own domains

Phase goal

Convert knowledge into demonstrable capability. Ship 2–3 end-to-end projects that intersect your existing domains (logistics, bilingual documents, mobile, backend) so each new skill compounds with what you already do well.

Estimated time: Ongoing | **Track:** Comprehensive ML & Deep Learning Curriculum

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

Contents

1 The purpose of capstones

Everything before this phase built capability; capstones prove and compound it. A finished, deployed project that solves a real problem teaches more — and signals more — than another course completed. The strategic move is to pick projects at the *intersection* of machine learning and domains you already understand deeply. There your background is an unfair advantage: you can judge data quality, frame the problem correctly, and ship the production parts faster than a pure ML newcomer.

Key idea

Depth beats breadth here. Two or three *finished, deployed, documented* projects are worth more than ten notebooks that stop at "0.92 validation accuracy." For each capstone, the deliverable is a system someone could use, a repository someone could read, and a write-up that explains the decisions and the results honestly — including what did not work.

2 What "done" means for a capstone

Dimension	Bar to clear
Problem framing	a real user/decision; a metric tied to value; a baseline
Data	sourced, cleaned, versioned; leakage checked
Modeling	a strong baseline first, then iteration justified by results
Evaluation	honest, with error bars and error analysis (not one number)
Deployment	served behind an API or app; reproducible; monitored
Communication	README + short write-up: problem, approach, results, limits

Intuition

A reviewer (or future employer, or future you) should be able to clone the repo, read a one-page README, run one command to reproduce the core result, and hit a running endpoint. If they can, the project is "done." Polish the seams — they are what people actually see.

3 Recommended capstones (pick 2–3)

1 · Shipping / logistics ML

Predictive delivery-time estimation or shipment anomaly detection — ties directly into Atravesar / Canada Post work.

- *ETA regression:* predict delivery time from route, carrier, origin/destination, weight, season, and historical performance. Strong fit for gradient boosting (Phase 1); report MAE with calibrated prediction intervals.
- *Anomaly detection:* flag shipments likely to be lost/delayed/mis-scanned (isolation forest / autoencoder reconstruction error, Phases 1 & 5).
- *Production:* serve as an API your existing backend can call; monitor drift as carrier behavior and seasons change (Phase 7).

Why it compounds: you already understand the domain, the data sources, and the

integration surface — so you can focus on the ML and ship fast.

2 · NLP + your bilingual skill set

Chinese↔English document intelligence — leverage a genuinely rare combination.

- *Bilingual document classification* (e.g. routing or tagging mixed-language documents) with a multilingual Transformer (Phase 4).
- *Translation-quality estimation* or a *semantic search* tool over your own translated API docs.
- Mind the CJK tokenization quirks (Phase 4): measure token counts/costs and retrieval quality across languages.

Why it compounds: bilingual CJK/English ability plus ML is a differentiated niche; few engineers can build and *evaluate* this well.

3 · Mobile / on-device DL

A lightweight on-device model for "Tips of Canada" — e.g. receipt OCR + amount extraction.

- A small vision model for detection/recognition (Phase 3), converted to **CoreML** for iOS.
- Emphasize the on-device constraints: quantization, latency, size (Phase 7); compare cloud vs. on-device tradeoffs.

Why it compounds: exercises the full edge-deployment path and produces a tangible mobile demo.

4 · End-to-end RAG assistant

A retrieval-augmented assistant over internal documentation (e.g. SolvoSync docs).

- Full RAG pipeline (Phase 4): chunk → embed → vector store (pgvector fits an existing Postgres) → retrieve + rerank → generate with citations.
- Build a real evaluation harness (retrieval recall@k, answer faithfulness) and a simple chat UI or API.
- Deploy and monitor (Phase 7): track query patterns, latency, and answer quality.

Why it compounds: the highest-relevance modern skill, and a natural fit for your API/backend strengths.

5 · Kaggle for benchmarking

Compete to calibrate yourself against the field.

- Start with a "Getting Started" competition (Titanic) to learn the workflow, then enter a live competition for honest benchmarking.
- Study and reproduce top solutions afterward — reading winning write-ups teaches feature engineering and validation better than any course.

Why it compounds: forces rigorous validation and exposes you to techniques you would not invent alone.

4 Choosing your set

Intuition

A strong portfolio of three: **(2) the bilingual NLP project** (differentiated, modern), **(4) the RAG assistant** (highest current demand, plays to your backend), and **(1) the logistics project** (domain depth, clear business value). This trio spans classical ML, transformers, retrieval, and production deployment — exactly the compressed high-value path (Phases 0→1→2→4→7) made tangible.

5 Execution playbook

1. **Week 1 — frame & baseline.** Define the user, the metric, and a dumb baseline (majority class / simple heuristic / BM25 for RAG). Never skip the baseline; it is your reality check.
2. **Weeks 2–3 — iterate.** Improve data and model; every change justified by the validation metric. Keep an experiment log (Phase 7).
3. **Week 4 — productionize.** Serve, containerize, monitor, document. Resist endless modeling; shipping is the point.
4. **Throughout — write it down.** A running README and a final write-up (problem, data, approach, results with error bars, limitations, next steps).

Common pitfall

The two ways capstones die: **scope creep** (the project balloons until it is never finished) and **the 90% trap** (the model works in a notebook but is never deployed or documented). Fight both by fixing a deadline, defining "done" up front (the table in §2), and treating deployment and the write-up as first-class deliverables, not optional extras.

6 Staying current after the curriculum

Keep the momentum

- **Kaggle** — competitions and winning-solution write-ups.
- **Papers With Code** — papers paired with implementations; track SOTA on tasks you care about.
- **Hugging Face Papers / arXiv** — a steady, curated reading habit beats binge-then-forget.
- Re-read the reference books as references, not cover-to-cover: Géron, Huyen, Goodfellow et al.
- Build in public — a blog post or repo per capstone compounds reputation alongside skill.

7 Closing

You started at vectors and gradients and ended at deployed, monitored systems that learn. The throughline never changed: represent data with linear algebra, improve parameters with calculus, reason about uncertainty with probability, and engineer the system around it with care. From here, the field will keep moving — but the foundations you built will let every new paper, model, and tool read as a variation on themes you now own. Pick a capstone, set a deadline, and ship.

Key idea

Knowledge you can't apply fades; systems you've shipped stay with you. Make something real, write down how and why, and let each project make the next one easier. That compounding — not any single model — is the whole point of this curriculum.