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:

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

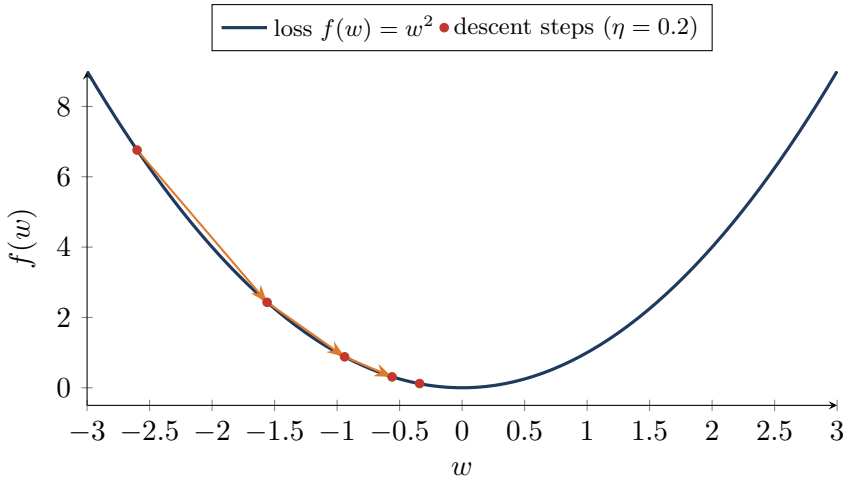
Differentiate term by term using the identities above:

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

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.