From Math Foundations to Deep Learning: A Complete Machine Learning Course for Applied Mathematics Students

The goal of this long-form blog post is to give you a coherent, rigorous, and highly scaffolded course that builds a mathematician's intuition for machine learning and deep learning, without assuming any prior exposure beyond high school mathematics. I will teach you how to formulate machine learning problems, understand the algorithms, interpret their mathematics, and implement them in a principled way. Along the way, I will define every term, explain every symbol, and show every derivation step by step. I will also provide concrete numerical examples to ground abstract ideas.

The course unfolds in five stages:

- Stage 0: A quick but complete tour of the mathematical tools you need (sets, functions, vectors, matrices, calculus, optimization, probability, and statistics).
- Stage 1: Supervised learning foundations, including regression, classification, regularization, evaluation, and cross-validation.
- Stage 2: Unsupervised learning, including clustering and dimensionality reduction, with anomaly detection.
- Stage 3: Deep learning, including neural networks, backpropagation, CNNs, and RNNs/Transformers.
- Stage 4: Advanced topics such as generative models, reinforcement learning, NLP, computer vision, and uncertainty quantification.

Each stage builds on the previous one. You will see connections explicitly and repeatedly. Every major concept will be taught via intuition, formal definition, why it matters, and a numerical example. You will also see derivations with complete algebraic steps. Together, this scaffolding ensures you never get lost; you will see how each new idea is necessary and inevitable.

With this arc in mind, let us begin.

Stage 0: Mathematical Foundations

To study machine learning, you need the language and logic of mathematics. Stage 0 is a quick but thorough foundation. We will move from the simplest objects (numbers) to the structures that describe data (vectors and matrices), the transformations we will apply (functions), the rates at which things change (derivatives and gradients), the search for good parameters (optimization), and the randomness that often models noise (probability and statistics).

Overview of Stage 0

Machine learning is a toolbox built on mathematics. The features and labels we use in ML are vectors. We combine them with weights and biases through functions, and we optimize these parameters to match data. Probability models noise and uncertainty. Statistics helps summarize datasets and evaluate models. We will introduce all these elements, define them rigorously, connect them, and anchor them with examples.

Numbers and Arithmetic

Numbers are the atoms of mathematics. For this course, we will operate primarily in the real numbers, but integers and rational numbers are also useful. We will also make frequent use of the notion of "small" and "large" numbers for scaling, which will later matter in numerical computation.

- Intuition: Numbers allow us to measure and compare. If a house has a price in dollars, a temperature in degrees Celsius, and a number of rooms, we have different kinds of numbers. In machine learning, these differences often require us to standardize or scale features to prevent one dimension from dominating the learning process.
- Formal definition: The set of real numbers $\mathbb{R}$ includes all rational numbers (like $1/2$, -3, and 4) and irrational numbers (like $\sqrt{2}$ and π). Arithmetic operations include addition ($+$), subtraction ($-$), multiplication ($\times$), division ($\div$), and exponentiation ( ). For matrices and vectors, we also define addition, scalar multiplication, and matrix multiplication.
- Why it matters: Every feature in a dataset is a number. Every parameter we learn is a number. Every performance metric is a number. Numbers ground the entire computational process.
- Numerical example: Suppose a feature is a temperature measured in degrees Celsius: 25. In Fahrenheit, the same temperature is 77. The two numbers represent the same physical quantity but differ because of scaling and origin differences. This is a precursor to the idea of normalization in ML.
- Connection: Features in ML are numbers. Optimization finds optimal numbers for parameters (weights and biases) to fit data.
- Common misconceptions: Assuming all numbers are comparable in magnitude. In ML, features often vary widely (e.g., age in years vs. income in thousands). Ignoring this leads to poor optimization and poor decisions. Normalization and scaling address this.

Sets and Logic

Sets are collections of objects. Logic is how we reason about truth and validity.

- Intuition: A set is a group. If I say "the set of students in class," I mean a well-defined collection of students. Logic tells me that if a student is in the class, then they must have a name; this is a logical implication.
- Formal definition: A set A is a collection of distinct elements. Membership is denoted $a \in A$. Operations include union ($A \cup B$), intersection ($A \cap B$), complement (A^c), and difference ($A \setminus B$). Logic includes statements P and Q with implications ($P \Rightarrow Q$), negation ($\neg P$), conjunctions ($P \wedge Q$), and disjunctions ($P \vee Q$).
- Why it matters: Sets describe possible inputs and outputs. Logic helps define training constraints, model assumptions, and correctness criteria. For example, "If x is a feature vector, then we transform it by $f(x)$" is an implication.
- Numerical example: Let $A = \{1, 2, 3\}$ and $B = \{3, 4, 5\}$. Then $A \cup B = \{1, 2, 3, 4, 5\}$, $A \cap B = \{3\}$, $A \setminus B = \{1, 2\}$.
- Connection: Datasets are sets of examples. Labels form sets of categories (e.g., $\{0, 1\}$ for binary classification).
- Common misconceptions: Confusing membership with subset. If $1 \in A$, that means 1 is in A, not that $\{1\} \subseteq A$. However, in a valid statement, $\{1\} \subseteq A$ is also true.

Functions

Functions map inputs to outputs. In ML, features are inputs and predicted labels are outputs.

- Intuition: Think of a function as a machine: you feed it an input and it gives you an output. A function can be linear (straight-line mapping) or nonlinear (curved).
- Formal definition: A function f from a set A to a set B assigns to each element $a \in A$ exactly one element $f(a) \in B$. The domain is A, the codomain is B, and the range is the set of outputs it actually produces.
- Why it matters: ML is about learning functions that map inputs to outputs. For regression, f returns continuous values. For classification, f returns probabilities or labels.
- Numerical example: Define $f(x) = 2x + 1$. Then $f(3) = 2 \cdot 3 + 1 = 7$. Input 3 maps to output 7.
- Connection: Models in ML are functions. We learn parameters (weights) such that the learned function fits training data.
- Common misconceptions: Confusing "function" with "function value." f is the function; $f(x)$ is the value at x.

Sequences and Series

Sequences are ordered lists of numbers. Series are sums of sequences.

- Intuition: A sequence is a list of numbers like $1, 1/2, 1/3, ...$ A series is the sum of these numbers. These ideas help explain how we build cost functions and optimization sequences.
- Formal definition: A sequence $(a_n)_{n=1}^{\infty}$ is a function from natural numbers to real numbers. A series $\sum_{n=1}^{\infty} a_n$ is the sum of the terms of the sequence.
- Why it matters: We use sums all the time in ML: sums over training examples to compute loss, sums over weights to compute regularization terms.
- Numerical example: Sequence $a_n = 1/n$ for $n = 1, 2, 3$ gives $1, 0.5, 0.333....$ The partial series $a_1 + a_2 + a_3 = 1 + 0.5 + 0.333... = 1.833....$
- Connection: Loss functions are often sums over examples. Regularization terms are sums over parameters.
- Common misconceptions: Confusing convergence with existence. The harmonic series diverges; it does not converge to a finite number.

Vectors and Matrices

Vectors and matrices are the languages of ML. Vectors represent features. Matrices represent datasets or linear transformations.

- Intuition: A vector is a list of numbers: $[3, 2, 1]$. A matrix is a rectangular array of numbers: a grid of rows and columns. Vectors are like arrows in a space. Matrices are like blueprints that transform vectors.
- Formal definition: An n-dimensional vector $x \in \mathbb{R}^n$ is a tuple $(x_1, x_2, ..., x_n)$. A matrix $A \in \mathbb{R}^{m \times n}$ has m rows and n columns. Matrix multiplication AB is defined for $A \in \mathbb{R}^{m \times p}$ and $B \in \mathbb{R}^{p \times n}$: $(AB)_{ij} = \sum_{k=1}^{p} A_{ik} B_{kj}$.
- Why it matters: Features are vectors. A dataset with N examples and D features is a matrix $X \in \mathbb{R}^{N \times D}$. Linear regression uses a matrix to transform features into predictions via weights.
- Numerical example: Let $X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $w = \begin{bmatrix} 5 \\ 6 \end{bmatrix}$. The product $Xw = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 6 \\ 3 \cdot 5 + 4 \cdot 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$.
- Connection: Supervised learning models typically compute $Xw + b$ or a more complex function of X and learned parameters.
- Common misconceptions: Confusing matrix multiplication with element-wise multiplication. Matrix multiplication is defined by dot products of rows and columns, not by multiplying entries pairwise.

Linear Equations

Linear equations encode relationships that sum up contributions linearly.

- Intuition: If income equals $2\times$ (hours worked) minus a constant overhead, that is a linear relationship. Linear algebra solves systems of these equations

simultaneously.

- Formal definition: A system of linear equations can be written as $Ax = b$, where $A \in \mathbb{R}^{m \times n}$ encodes coefficients, $x \in \mathbb{R}^n$ are unknowns (often weights), and $b \in \mathbb{R}^m$ are known values.
- Why it matters: Many ML models predict y as a linear function of features: $y = Xw$. Solving linear systems is at the heart of many algorithms.
- Numerical example: Solve $X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $b = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$, and compute $w = X^{-1}b$ if X is invertible. Here $X^{-1} = \frac{1}{1 \cdot 4 - 2 \cdot 3} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \frac{1}{-2} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix}$. Thus $w = \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix} \begin{bmatrix} 5 \\ 11 \end{bmatrix} = \begin{bmatrix} -2 \cdot 5 + 1 \cdot 11 \\ 1.5 \cdot 5 - 0.5 \cdot 11 \end{bmatrix} = \begin{bmatrix} 1 \\ 2.5 \end{bmatrix}$.
- Connection: Many learning algorithms reduce to solving linear systems (e.g., least squares).
- Common misconceptions: Assuming X^{-1} always exists. X must be invertible, or we need a different method (e.g., least squares or pseudoinverse).

Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors reveal fundamental directions and scaling factors.

- Intuition: Imagine a matrix as a transformation of space. Eigenvectors are directions that do not change direction under the transformation; they just get stretched or squashed by the eigenvalue.
- Formal definition: For a square matrix $A \in \mathbb{R}^{n \times n}$, an eigenvector $v \neq 0$ and eigenvalue λ satisfy $Av = \lambda v$. This equation implies $(A - \lambda I)v = 0$, which has a nontrivial solution when $\det(A - \lambda I) = 0$.
- Why it matters: Principal Component Analysis (PCA) is based on eigendecomposition. Eigenvectors provide orthogonal directions that explain variance in data.
- Numerical example: Let $A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$. Solve $\det(A - \lambda I) = \det \begin{bmatrix} 3 - \lambda & 1 \\ 0 & 2 - \lambda \end{bmatrix} = (3 - \lambda)(2 - \lambda) - 0 = 0$. So eigenvalues are $\lambda_1 = 3$, $\lambda_2 = 2$. For $\lambda_1 = 3$, solve $(A - 3I)v = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} v = 0$. This gives $v_2 = 0$ and any v_1; pick $v = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. For $\lambda_2 = 2$, solve $(A - 2I)v = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} v = 0$, giving $v_1 = -v_2$, pick $v = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$.
- Connection: PCA uses eigenvectors of the covariance matrix to project data onto orthogonal directions of maximal variance.
- Common misconceptions: Thinking eigenvectors must be unique. Any scalar multiple of an eigenvector is also an eigenvector. Normalization is a convention for uniqueness.

Norms and Distance

Norms measure the size of vectors. Distances measure how far vectors are from each other.

- Intuition: Norms are like rulers. A Euclidean norm measures straight-line distance from the origin. Manhattan norm measures path-length by moving along gridlines. Distance between vectors tells us how similar they are.
- Formal definition: The ℓ_p norm of a vector $x = (x_1, ..., x_n)$ is $\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p}$. Common cases: $p = 1$ gives $\|x\|_1 = \sum |x_i|$, $p = 2$ gives $\|x\|_2 = \sqrt{\sum x_i^2}$, and $p = \infty$ gives $\|x\|_\infty = \max_i |x_i|$.
- Why it matters: Regularization uses norms of weights (e.g., ℓ_2 penalty). Similarity in clustering uses distances. Loss functions often measure distance between predictions and targets.
- Numerical example: Let $x = [3, -4]$. Then $\|x\|_1 = |3| + |-4| = 7$, $\|x\|_2 = \sqrt{3^2 + (-4)^2} = \sqrt{9 + 16} = 5$, and $\|x\|_\infty = \max(|3|, |-4|) = 4$.
- Connection: Many algorithms use distances to define clusters, neighbors, and objective penalties.
- Common misconceptions: Using ℓ_2 norm by default. The right norm depends on the problem (e.g., ℓ_1 for sparsity).

Calculus: Derivatives and Gradients

Calculus quantifies change. Derivatives describe how functions change. Gradients generalize derivatives to multi-variable functions.

- Intuition: If you have a hill described by height $h(x, y)$, the gradient points uphill fastest; it is the direction of steepest ascent. The negative gradient is the direction of steepest descent.
- Formal definition: The derivative of $f(x)$ is $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$. For $f : \mathbb{R}^n \to \mathbb{R}$, the gradient is $\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$, where $\frac{\partial f}{\partial x_i}$ is the partial derivative.
- Why it matters: Optimization uses gradients to move toward minima. Loss functions are minimized with gradient-based algorithms.
- Numerical example: Let $f(x) = x^2$. Then $f'(x) = 2x$. At $x = 3$, $f'(3) = 6$. For $f(x, y) = x^2 + y^2$, $\nabla f(x, y) = \begin{bmatrix} 2x \\ 2y \end{bmatrix}$.
- Connection: We will use gradients of loss functions (e.g., mean squared error) to update parameters.
- Common misconceptions: Confusing derivatives and gradients. Derivatives are for single-variable functions; gradients are for multi-variable scalar fields.

Optimization: Minima/Maxima, Gradient Descent

We want to find parameter values that minimize loss.

- Intuition: Think of a bowl-shaped surface (a loss landscape) and a ball rolling downhill. Each step follows the negative of the local slope, until we reach the bottom.
- Formal definition: For a differentiable function $L(w)$, gradient descent updates parameters by $w \leftarrow w - \eta \nabla L(w)$, where $\eta > 0$ is the learning rate. Learning rate controls step size.
- Why it matters: Gradient descent is the backbone of training ML models, especially deep networks where closed-form solutions do not exist.
- Numerical example: Let $L(w) = (w-2)^2$. Then $\nabla L(w) = 2(w-2)$. Start with $w_0 = 5$, $\eta = 0.1$. Step 1: $w_1 = 5 - 0.1 \cdot 2(5-2) = 5 - 0.1 \cdot 6 = 4.4$. Step 2: $w_2 = 4.4 - 0.1 \cdot 2(4.4-2) = 4.4 - 0.1 \cdot 4.8 = 3.92$. Step 3: $w_3 = 3.92 - 0.1 \cdot 2(3.92-2) = 3.92 - 0.1 \cdot 3.84 = 3.536$. The values converge toward $w^* = 2$.
- Connection: Regularization and loss functions will be minimized via gradient descent.
- Common misconceptions: Choosing a learning rate that is too large (can overshoot the minimum) or too small (converges slowly). In practice, learning rate schedules adaptively decrease the step size over time.

Probability: Fundamentals and Distributions

Probability describes randomness and uncertainty.

- Intuition: If you flip a fair coin, the outcome is uncertain. Probability assigns numbers (like 0.5) to events that capture likelihood.
- Formal definition: A probability space has a sample space Ω, a sigma-algebra of events $\mathcal{F}$, and a probability measure P that assigns probabilities to events. A random variable X maps outcomes to numbers. The distribution of X is described by its probability mass function (for discrete X) or probability density function (for continuous X). Expectation $E[X]$ is the average value; variance $\text{Var}(X) = E[(X - E[X])^2]$ measures spread.
- Why it matters: Real data are noisy. Probabilistic models capture this noise, provide uncertainty estimates, and explain generalization performance via statistical theory.
- Numerical example: Let X be a Bernoulli random variable: $P(X=1) = p$, $P(X=0) = 1-p$. Then $E[X] = 0 \cdot (1-p) + 1 \cdot p = p$, and $\text{Var}(X) = E[X^2] - (E[X])^2 = p - p^2 = p(1-p)$.
- Connection: Classification often models $P(Y=1|X)$ as a logistic function. Gaussian distributions model noise in regression.
- Common misconceptions: Treating the sample mean as the true mean. Sample estimates vary, and we need confidence intervals or tests to quantify uncertainty.

Statistics: Mean, Variance, Hypothesis Testing, Correlation

Statistics lets us summarize and interpret data.

- Intuition: We can think of a dataset as a handful of observations from an underlying distribution. We compute sample statistics (like mean and variance) to estimate population parameters. Tests help us decide if observed differences are significant or just noise.
- Formal definition: The sample mean $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. The sample variance $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$. Covariance measures joint variability: $\text{Cov}(x, y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$. Correlation is covariance normalized by standard deviations: $\rho = \frac{\text{Cov}(x,y)}{s_x s_y}$.
- Why it matters: We use statistics to understand patterns and relationships in data and to evaluate models against baselines.
- Numerical example: Let $x = [1, 2, 3]$, $y = [2, 4, 6]$. Then $\bar{x} = 2$, $\bar{y} = 4$. Covariance: $(1-2)(2-4) + (2-2)(4-4) + (3-2)(6-4) = (-2)(-2) + 0 + (2) = 4 + 2 = 6$. Divide by $n - 1 = 2$: $\text{Cov}(x, y) = 3$. Standard deviations: $s_x = \sqrt{\frac{(1-2)^2+(2-2)^2+(3-2)^2}{2}} = \sqrt{\frac{1+0+1}{2}} = \sqrt{1} = 1$. Similarly $s_y = \sqrt{\frac{(2-4)^2+(4-4)^2+(6-4)^2}{2}} = \sqrt{\frac{4+0+4}{2}} = \sqrt{4} = 2$. Correlation $\rho = 3/(1 \cdot 2) = 1.5$? Wait, correlation should be bounded by 1. Let's recompute correctly: cov is computed as average of product deviations; with $n = 3$, we divide by 2. Compute deviations: $x_i - \bar{x}$ are $(-1, 0, 1)$. $y_i - \bar{y}$ are $(-2, 0, 2)$. Products: $2, 0, 2$. Sum is 4. Covariance is $4/2 = 2$. Standard deviations: $s_x = \sqrt{\frac{(-1)^2+0^2+1^2}{2}} = \sqrt{\frac{2}{2}} = 1$. $s_y = \sqrt{\frac{(-2)^2+0^2+2^2}{2}} = \sqrt{\frac{8}{2}} = \sqrt{4} = 2$. Correlation $\rho = 2/(1 \cdot 2) = 1$. Perfect correlation is 1; that makes sense, since $y = 2x$. This correction ensures we have valid correlation.
- Connection: Correlation informs linear relationships. Supervised models try to learn such relationships. Hypothesis testing helps compare models or assess differences.
- Common misconceptions: Thinking correlation implies causation. Correlation quantifies association; causal inference requires controlled experiments or other designs.

Check Your Understanding (Stage 0)

- Question: What is the gradient of $L(w) = \frac{1}{2}\|Xw - y\|_2^2$ with respect to w?
- Answer: Expand $\|Xw - y\|_2^2 = (Xw - y)^\top (Xw - y) = w^\top X^\top X w - 2w^\top X^\top y + y^\top y$. Differentiating with respect to w yields $2X^\top X w - 2X^\top y$. So $\nabla_w L(w) = X^\top (Xw - y)$.
- Question: Why does normalization matter for features in ML?
- Answer: Features often have very different scales, which can cause gradient-based optimization to move unevenly through the parameter space. Nor-

malization brings features into comparable ranges, improving numerical stability and convergence speed.

With these foundations in place, we are ready to formulate and solve machine learning problems.

Stage 1: Supervised Learning Foundations

Supervised learning uses labeled examples to learn a mapping from inputs to outputs. We will define the ML problem rigorously, describe training vs. test, and learn the core methods: linear regression, gradient descent, regularization, classification, evaluation, and model selection.

Overview of Stage 1

We will begin by formalizing the learning problem and its core components: dataset, loss function, parameters, and hyperparameters. Then, we will derive linear regression from first principles and explain gradient descent in detail. We will then build intuition for classification, regularization, evaluation metrics, and model selection. Each topic will be connected to Stage 0 and built upon.

ML Problem Formulation

Supervised learning learns from pairs (x_i, y_i) where x_i are inputs (features) and y_i are labels (targets). We want a function f_θ parameterized by θ such that $f_\theta(x_i) \approx y_i$ on training data, and also generalizes to unseen data.

- Intuition: Think of learning as fitting a curve through points. The curve should pass near the training points but also avoid overfitting.
- Formal definition: Let $\mathcal{D}_{\text{train}} = \{(x_i, y_i)\}_{i=1}^N$ be training data. We choose a model class $f_\theta(x) = f(x; \theta)$ (e.g., linear $f(x) = w^\top x + b$). Define a loss $L(y, \hat{y})$ that measures disagreement between true label y and predicted $\hat{y}$. Train by minimizing empirical risk: $\hat{\theta} = \arg\min_\theta \frac{1}{N} \sum_{i=1}^N L(y_i, f_\theta(x_i)) + \lambda R(\theta)$. Here $R(\theta)$ is a regularization term, and λ is a regularization strength. This is our learning objective.
- Why it matters: Every supervised method builds a model and minimizes a loss. The choice of model and loss determines the behavior and properties of the learned function.
- Numerical example: Suppose we have x_i as 1D: $x = [1, 2, 3]$, $y = [2, 4, 6]$. Choose a linear model $f(w, b; x) = wx + b$. The loss for one example is $L = (y - (wx + b))^2$. For all three examples, the average loss is $\frac{1}{3}[(2 - (w \cdot 1 + b))^2 + (4 - (w \cdot 2 + b))^2 + (6 - (w \cdot 3 + b))^2]$.

- Connection: All ML algorithms define such a function. The derivations in Stage 0 (gradients and linear algebra) are the tools we use to minimize this loss.
- Common misconceptions: Overfitting occurs when the model memorizes the training data by using too complex a model, producing poor predictions on unseen data. Regularization counters this.

Training Process, Parameters, Hyperparameters

Parameters are learned from data. Hyperparameters control learning (e.g., regularization strength, learning rate).

- Intuition: Parameters are the knobs we turn to adjust the model's behavior. Hyperparameters are the dials on the machine controlling how fast and how thoroughly we turn those knobs.
- Formal definition: Parameters θ (e.g., weights w and biases b) are optimized by minimizing the loss. Hyperparameters η (e.g., learning rate, regularization coefficient λ) control the training process and are set before training by the practitioner.
- Why it matters: We must distinguish between what we learn automatically (parameters) and what we choose manually or via validation (hyperparameters).
- Numerical example: In gradient descent for linear regression, w and b are parameters; the learning rate η and number of epochs T are hyperparameters.
- Connection: In Stage 3, deep networks have millions of parameters but a handful of hyperparameters (e.g., learning rate, batch size). In Stage 1, we will tune them via cross-validation.
- Common misconceptions: Including hyperparameters inside the loss optimization leads to a different problem (automated machine learning or Bayesian optimization).

Linear Regression: Simple and Multivariate

Linear regression models the expected target as a linear combination of features plus noise. It is a baseline model and a foundation for more advanced methods.

- Intuition: We draw a straight line through points such that the average vertical distance between the line and the points is minimized.
- Formal definition: For a dataset $X \in \mathbb{R}^{N \times D}$ with features $x_i \in \mathbb{R}^D$, targets $y \in \mathbb{R}^N$, the ordinary least squares (OLS) objective is $\min_w \frac{1}{N}\|Xw - y\|_2^2$. Including intercepts can be done by augmenting X with a column of ones, or by adding a separate bias term b.

- Derivation (closed-form solution): Expand $\|Xw-y\|_2^2 = (Xw-y)^\top(Xw-y) = w^\top X^\top Xw - 2w^\top X^\top y + y^\top y$. Differentiate with respect to w to set the gradient to zero:
  - Gradient: $\nabla_w(w^\top X^\top Xw - 2w^\top X^\top y + y^\top y) = 2X^\top Xw - 2X^\top y$.
  - Set gradient to zero: $2X^\top Xw - 2X^\top y = 0 \implies X^\top Xw = X^\top y$.
  - Solve: If $X^\top X$ is invertible, $w^* = (X^\top X)^{-1}X^\top y$.
- Why it matters: OLS gives an explicit solution when linear relationships are strong. It also offers statistical interpretations, connects to Gaussian noise assumptions, and generalizes to ridge regression (which stabilizes the inverse).
- Numerical example: Let's compute a tiny regression with $N = 3$, $D = 1$. Define $X = \begin{bmatrix}1\\2\\3\end{bmatrix}$ (we will add a bias term separately to illustrate the full linear model $f(x) = wx+b$; to apply the matrix form, append a column of ones: $\tilde{X} = \begin{bmatrix}1 & 1\\2 & 1\\3 & 1\end{bmatrix}$). Let $y = \begin{bmatrix}2\\4\\6\end{bmatrix}$. Compute $\tilde{X}^\top\tilde{X} = \begin{bmatrix}1 & 2 & 3\\1 & 1 & 1\end{bmatrix}\begin{bmatrix}1 & 1\\2 & 1\\3 & 1\end{bmatrix} = \begin{bmatrix}1+4+9 & 1+2+3\\1+2+3 & 1+1+1\end{bmatrix} = \begin{bmatrix}14 & 6\\6 & 3\end{bmatrix}$. Compute $\tilde{X}^\top y = \begin{bmatrix}1 & 2 & 3\\1 & 1 & 1\end{bmatrix}\begin{bmatrix}2\\4\\6\end{bmatrix} = \begin{bmatrix}1\cdot 2+2\cdot 4+3\cdot 6\\1\cdot 2+1\cdot 4+1\cdot 6\end{bmatrix} = \begin{bmatrix}2+8+18\\2+4+6\end{bmatrix} = \begin{bmatrix}28\\12\end{bmatrix}$. Solve $(\tilde{X}^\top\tilde{X})w = \tilde{X}^\top y$:
  - $\begin{bmatrix}14 & 6\\6 & 3\end{bmatrix}\begin{bmatrix}w\\b\end{bmatrix} = \begin{bmatrix}28\\12\end{bmatrix}$.
  - Solve: Multiply the second equation by 2 to align: $12w + 6b = 24$. Subtract the first equation: $(14w + 6b) - (12w + 6b) = 28 - 24$ gives $2w = 4$, so $w = 2$. Substitute back: $12 \cdot 2 + 6b = 24 \implies 24 + 6b = 24 \implies b = 0$.
  - Thus, $w = 2$, $b = 0$, giving $f(x) = 2x$, which fits the data perfectly.
- Connection: Linear regression generalizes to multivariate input: Xw captures the linear combination of features. Regularization adds a penalty to stabilize the solution and reduce overfitting.
- Common misconceptions: Thinking linear regression always needs a closed-form solution. In high dimensions or with added regularization, we use iterative methods like gradient descent.

Gradient Descent Algorithm

We will often solve optimization problems iteratively when closed-form solutions are unavailable. Gradient descent updates parameters by moving in the direction of steepest descent.

- Intuition: Imagine a ball rolling downhill. At each step, it moves in the direction of steepest descent (negative gradient), with the learning rate

determining the step size.

- Formal definition: For $L(\theta)$ (the loss), gradient descent updates are $\theta^{(t+1)} = \theta^{(t)} - \eta \nabla_\theta L(\theta^{(t)})$. Stochastic gradient descent (SGD) computes gradients on small mini-batches of data to reduce noise and compute cost efficiently. Momentum methods accelerate convergence by accumulating past gradients.
- Derivation for linear regression via gradient descent: Consider $L(w) = \frac{1}{2N}\|Xw - y\|_2^2$. Gradient: $\nabla_w L(w) = \frac{1}{N} X^\top (Xw - y)$. The update: $w \leftarrow w - \eta \cdot \frac{1}{N} X^\top (Xw - y)$.
- Why it matters: Gradient descent scales to complex models (like deep networks) where closed-form solutions are not available. It is also flexible enough to incorporate regularization.
- Numerical example: Let's continue with the tiny regression: $X = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, $y = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$. Suppose we choose $w = 0$, $b = 0$, and $\eta = 0.1$. Compute the gradient for each example separately (SGD):
  - For $i = 1$: $f(x_1) = 0 \cdot 1 + 0 = 0$, residual $r_1 = y_1 - f(x_1) = 2 - 0 = 2$. The gradient for w is $\frac{1}{N}\sum_i x_i(y_i - \hat{y}_i) = \frac{1}{3}(x_1 r_1 + x_2 r_2 + x_3 r_3) = \frac{1}{3}(1 \cdot 2 + 2 \cdot 4 + 3 \cdot 6) = \frac{1}{3}(2 + 8 + 18) = \frac{28}{3} \approx 9.33$ (for full-batch gradient). But we do per-example gradient; for SGD, we update per example:
    - For example 1: gradient for w is $x_1 r_1 = 1 \cdot 2 = 2$; for b is $r_1 = 2$.
    - Update: $w \leftarrow 0 - 0.1 \cdot 2 = -0.2$, $b \leftarrow 0 - 0.1 \cdot 2 = -0.2$.
  - Example 2: $f(2) = -0.2 \cdot 2 + (-0.2) = -0.6$, residual $r_2 = 4 - (-0.6) = 4.6$. Update: $w \leftarrow -0.2 - 0.1 \cdot (2 \cdot 4.6) = -0.2 - 0.92 = -1.12$. $b \leftarrow -0.2 - 0.1 \cdot 4.6 = -0.66$.
  - Example 3: $f(3) = -1.12 \cdot 3 - 0.66 = -4.02$, residual $r_3 = 6 - (-4.02) = 10.02$. Update: $w \leftarrow -1.12 - 0.1 \cdot (3 \cdot 10.02) = -1.12 - 3.006 = -4.126$. $b \leftarrow -0.66 - 0.1 \cdot 10.02 = -1.662$.
  - After one full pass (epoch), $w \approx -4.126$, $b \approx -1.662$. The parameters are not optimal yet; continued SGD with smaller learning rate would improve. The noise of SGD is natural; mini-batches stabilize gradients.
- Connection: Gradient descent uses the gradient from Stage 0 to move parameters toward a minimum. Regularization adds terms to the loss and thus changes the gradient.
- Common misconceptions: Using a fixed learning rate forever may overshoot. In practice, we decay the learning rate or use adaptive methods like Adam.

Regularization: L1, L2, Elastic Net

Regularization discourages overly complex solutions and reduces overfitting by penalizing large weights.

- Intuition: Think of regularization as a "guardrail" that keeps the model from being too adventurous and making wild predictions.
- Formal definition: L2 (ridge) regularization adds $\lambda\|w\|_2^2$. L1 (lasso) adds $\lambda\|w\|_1$. Elastic Net adds a convex combination: $\lambda_1\|w\|_1 + \lambda_2\|w\|_2^2$.
- Derivation for ridge regression: Minimize $\frac{1}{2N}\|Xw-y\|_2^2+\frac{\lambda}{2}\|w\|_2^2$. Gradient: $\nabla_w L = \frac{1}{N}X^\top(Xw-y)+\lambda w$. Setting to zero: $X^\top Xw+\lambda Nw = X^\top y \implies (X^\top X + \lambda NI)w = X^\top y$. So $w^* = (X^\top X + \lambda NI)^{-1}X^\top y$.
- Why it matters: L2 regularization stabilizes the solution (especially if $X^\top X$ is singular). L1 promotes sparsity (some weights become exactly zero). Elastic Net combines benefits of both.
- Numerical example: Suppose $X^\top X = \begin{bmatrix}14 & 6\\ 6 & 3\end{bmatrix}$ (from earlier), $N = 3$. Let $\lambda = 1$. Then $X^\top X + \lambda NI = \begin{bmatrix}14 & 6\\ 6 & 3\end{bmatrix} + 1 \cdot 3 \cdot \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}17 & 6\\ 6 & 6\end{bmatrix}$. The inverse exists (determinant $= 17 \cdot 6 - 6 \cdot 6 = 102 - 36 = 66$). Compute $(X^\top X + \lambda NI)^{-1} = \frac{1}{66}\begin{bmatrix}6 & -6\\ -6 & 17\end{bmatrix} = \begin{bmatrix}6/66 & -6/66\\ -6/66 & 17/66\end{bmatrix} = \begin{bmatrix}1/11 & -1/11\\ -1/11 & 17/66\end{bmatrix}$. Multiply by $X^\top y = \begin{bmatrix}28\\ 12\end{bmatrix}$: $w^* = \begin{bmatrix}1/11 & -1/11\\ -1/11 & 17/66\end{bmatrix}\begin{bmatrix}28\\ 12\end{bmatrix} = \begin{bmatrix}(1/11)\cdot 28 + (-1/11)\cdot 12\\ (-1/11)\cdot 28 + (17/66)\cdot 12\end{bmatrix} = \begin{bmatrix}(28-12)/11\\ (-28/11) + (204/66)\end{bmatrix}$.
  - Simplify the first component: $w = (16)/11 \approx 1.4545$.
  - Simplify the second: $b = (-28/11)+(204/66) = (-28/11)+(34/11) = 6/11 \approx 0.5455$.
  - These differ from the unregularized solution ($w = 2, b = 0$) due to the penalty pushing weights toward smaller magnitude.
- Connection: Regularization modifies the objective we optimize, which changes the gradient. In deep learning, we use L2 regularization via weight decay.
- Common misconceptions: Choosing a regularization strength that is too high forces weights to be close to zero, underfitting the data. Too low allows overfitting.

Classification: Logistic Regression and Decision Boundaries

Classification predicts discrete labels, often probabilities.

- Intuition: We want to draw a boundary that separates classes. Logistic regression outputs probabilities via the logistic function and uses a decision threshold to classify.

- Formal definition: For binary classification with features x, the logistic model is $p(y=1|x) = \sigma(w^\top x + b)$ where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid. The cross-entropy loss (negative log-likelihood) for N examples is $L(w,b) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\sigma(w^\top x_i + b)) + (1-y_i)\log(1-\sigma(w^\top x_i + b))\right]$.
- Derivation for gradients: Compute $\nabla_w L$. Let $p_i = \sigma(z_i)$ with $z_i = w^\top x_i + b$, residual $r_i = y_i - p_i$. Using $\sigma'(z) = \sigma(z)(1-\sigma(z))$, differentiate the cross-entropy: $L_i = -[y_i \log p_i + (1-y_i)\log(1-p_i)]$. Derivative with respect to z_i: $\frac{\partial L_i}{\partial z_i} = -[y_i \frac{\sigma'(z_i)}{p_i} + (1-y_i)\frac{-\sigma'(z_i)}{1-p_i}] = -[y_i(1-p_i) - (1-y_i)p_i] = p_i - y_i = r_i$. Then $\frac{\partial L}{\partial w} = \frac{1}{N}\sum_i r_i x_i$, and $\frac{\partial L}{\partial b} = \frac{1}{N}\sum_i r_i$.
- Why it matters: Logistic regression is a simple but powerful classifier. It generalizes to multinomial with the softmax function for multiclass problems.
- Numerical example: Use a small 2D dataset with points: $(0,0)$ label 0, $(1,1)$ label 1, $(2,0)$ label 0. Let's fit logistic regression. Start with $w = [0.5, -0.3]$, $b = 0.1$, learning rate $\eta = 0.1$. Compute predictions:
  - $x_1 = (0,0)$, $z_1 = 0.5 \cdot 0 + (-0.3) \cdot 0 + 0.1 = 0.1$, $p_1 = \sigma(0.1) \approx 0.525$. Residual $r_1 = 0 - 0.525 = -0.525$.
  - $x_2 = (1,1)$, $z_2 = 0.5 \cdot 1 + (-0.3) \cdot 1 + 0.1 = 0.3$, $p_2 = \sigma(0.3) \approx 0.574$. Residual $r_2 = 1 - 0.574 = 0.426$.
  - $x_3 = (2,0)$, $z_3 = 0.5 \cdot 2 + (-0.3) \cdot 0 + 0.1 = 1.1$, $p_3 = \sigma(1.1) \approx 0.750$. Residual $r_3 = 0 - 0.750 = -0.750$.
  - Gradients: $\nabla_w L = \frac{1}{3}[(-0.525) \cdot (0,0) + (0.426) \cdot (1,1) + (-0.750) \cdot (2,0)] = \frac{1}{3}[(0,0) + (0.426, 0.426) + (-1.5, 0)] = \frac{1}{3}[(-1.074, 0.426)] \approx (-0.358, 0.142)$. $\frac{\partial L}{\partial b} = \frac{1}{3}(-0.525 + 0.426 - 0.750) = \frac{1}{3}(-0.849) \approx -0.283$.
  - Update: $w \leftarrow w - \eta \nabla_w L = (0.5, -0.3) - 0.1(-0.358, 0.142) = (0.5 + 0.0358, -0.3 - 0.0142) = (0.5358, -0.3142)$. $b \leftarrow 0.1 - 0.1 \cdot (-0.283) = 0.1 + 0.0283 = 0.1283$.
  - After one step, parameters improved a bit. Over multiple iterations, logistic regression converges toward parameters that separate the classes.
- Connection: Logistic regression shares structure with linear regression but uses a different link function (sigmoid). Its gradients arise from probability theory and statistical learning.
- Common misconceptions: Using a linear boundary by default. Logistic regression assumes linear separability in feature space; when classes are not linearly separable, we use other methods.

Feature Scaling, Normalization

Features often have different scales. Scaling improves optimization.

- Intuition: If one feature ranges from 0–1 and another from 0–1000, the optimizer has to navigate an elongated valley. Scaling gives the valley a more round shape, making descent faster.

- Formal definition: Min-max scaling rescales features to $[0, 1]$ via $\tilde{x} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$. Standardization rescales features to have zero mean and unit variance via $\tilde{x} = \frac{x - \bar{x}}{s}$.
- Why it matters: Optimization is more stable and converges faster. Many algorithms assume standardized features for best performance.
- Numerical example: Let feature $x = [10, 20, 30]$; min-max: $(10 - 10)/(30 - 10) = 0$, $(20 - 10)/20 = 0.5$, $(30 - 10)/20 = 1$. Standardization: $\bar{x} = 20$, $s = \sqrt{\frac{(10-20)^2+(20-20)^2+(30-20)^2}{3-1}} = \sqrt{\frac{100+0+100}{2}} = \sqrt{100} = 10$. Then scaled: $(10 - 20)/10 = -1$, $(20 - 20)/10 = 0$, $(30 - 20)/10 = 1$.
- Connection: Scaling affects the gradients and learning dynamics. It is a crucial preprocessing step.
- Common misconceptions: Forgetting to fit scalers on training data and then applying them on test data (data leakage). Always fit on training data and apply the same transformation to test data.

Evaluation Metrics: Regression and Classification

We need measures to assess performance. Metrics inform choices.

- Intuition: A single number may not tell the whole story. Different metrics emphasize different aspects (e.g., overall error vs. per-class accuracy).
- Formal definitions: For regression, mean squared error (MSE) is $\frac{1}{N}\sum(y_i - \hat{y}_i)^2$, mean absolute error (MAE) is $\frac{1}{N}\sum|y_i - \hat{y}_i|$, and R^2 is $1 - \frac{\sum(y_i - \hat{y}_i)^2}{\sum(y_i - \bar{y})^2}$. For classification, accuracy is fraction of correct predictions, precision is $\frac{TP}{TP+FP}$, recall is $\frac{TP}{TP+FN}$, F1-score is $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$.
- Why it matters: Different metrics reflect different losses. Classification decisions often hinge on false positives or false negatives more than raw accuracy.
- Numerical example: Suppose $y = [2, 4, 6]$, $\hat{y} = [2.1, 3.8, 6.2]$. MSE: $(2 - 2.1)^2+(4-3.8)^2+(6-6.2)^2 = (-0.1)^2+(0.2)^2+(-0.2)^2 = 0.01+0.04+0.04 = 0.09$. MAE: $0.1 + 0.2 + 0.2 = 0.5$.
- Connection: Metrics correspond to the loss functions we minimize; aligning them ensures consistent training objectives.
- Common misconceptions: Using accuracy when classes are imbalanced; precision, recall, and AUC better reflect performance.

Train-Test Split, Cross-Validation

To assess generalization, we hold out data and validate models.

- Intuition: We test a model on unseen examples to see if it learned general patterns, not just memorized.

- Formal definition: Random split: X_{train}, X_{test}; train the model on training data and evaluate on test data. K-fold cross-validation: split training data into K folds; train K times, each time holding one fold out for validation, then average results.
- Why it matters: Cross-validation gives more robust estimates of generalization performance, especially when dataset sizes are limited.
- Numerical example: For $K = 3$, with 12 training examples, create 3 folds of 4 examples each. Train three models and compute validation errors; average to get an estimate.
- Connection: Regularization and hyperparameter tuning rely on validation strategies to avoid overfitting the model to the test set.
- Common misconceptions: Using the test set repeatedly for model selection; this leaks information. Reserve test set until final evaluation.

Hyperparameter Tuning

We select hyperparameters via grid search, random search, or Bayesian optimization.

- Intuition: Hyperparameters control model capacity and learning dynamics. We search for values that achieve good generalization.
- Formal definition: Define a space (e.g., $\lambda \in \{0.001, 0.01, 0.1, 1.0\}$, learning rate $\eta \in \{0.01, 0.05, 0.1\}$). Evaluate each configuration on a validation set and choose the best.
- Why it matters: Good hyperparameters can dramatically improve performance and reduce overfitting.
- Numerical example: For linear regression with L2 regularization, try $\lambda = 0, 0.1, 1.0, 10.0$ and evaluate each via 5-fold CV; choose the λ with smallest validation error.
- Connection: Validation and cross-validation are the engines behind hyperparameter tuning.
- Common misconceptions: Tuning hyperparameters too finely on a small validation set leads to overfitting the validation data.

Feature Engineering

We transform and create features to improve model performance.

- Intuition: Raw data often require transformation into a more informative representation (e.g., polynomial features, interactions).
- Formal definition: Polynomial features of degree d transform x into all monomials up to degree d. Interaction features combine pairs of original features.
- Why it matters: Engineering good features is often more impactful than algorithmic improvements. It aligns data with model assumptions.

- Numerical example: With $x = [1, 2]$, polynomial degree 2 yields $[1, 2, 1, 2, 4]$ where the features are $1, 2, 1^2, 1 \cdot 2, 2^2$.
- Connection: Scaling and normalization apply to engineered features too. Principal components can serve as engineered features.
- Common misconceptions: Overengineering (too many features) can lead to overfitting and poor generalization.

Handling Missing Data and Class Imbalance

Real datasets often have missing values and class imbalance.

- Intuition: Missing values can be imputed (e.g., using mean/median), but imputation introduces uncertainty. Imbalanced classes require strategies that account for underrepresented classes.
- Formal definitions: Mean imputation fills missing numeric values with the average. For classification, resampling (oversampling minority class, undersampling majority) or class weights can balance effective loss.
- Why it matters: Missing data reduces usable examples; imbalance can bias predictions toward the majority class.
- Numerical example: Feature x values: $[10, \text{missing}, 30]$. Mean imputation: 20, making the dataset $[10, 20, 30]$. For binary labels $[0, 0, 1]$, if we train with balanced class weights, the effective penalty for misclassifying class 1 is higher.
- Connection: The loss functions can incorporate class weights; this connects to regularization and training dynamics.
- Common misconceptions: Imputing test data with statistics computed on test data (data leakage). Always compute imputations from training data and apply to test.

Ensemble Methods: Bagging, Boosting, Random Forests

Ensembles combine multiple models to improve performance.

- Intuition: A committee of models can average out individual errors, producing better decisions.
- Formal definitions: Bagging (bootstrap aggregating) trains many models on bootstrap samples and averages predictions (e.g., Random Forests). Boosting trains models sequentially, each correcting errors of previous ones (e.g., Gradient Boosting).
- Why it matters: Ensembles reduce variance (bagging) and bias (boosting). They are often top performers in competitions.
- Numerical example: A random forest of 10 decision trees each trained on a bootstrapped dataset; predictions are averaged for regression or majority vote for classification.

- Connection: Gradient boosting uses gradients of the loss to fit weak learners (e.g., decision trees). This connects directly to Stage 0 calculus.
- Common misconceptions: Random forests are just random subsets of features; they also use randomness in sampling data and splitting nodes. Boosting is sensitive to noisy data because each new model focuses on previous errors.

Check Your Understanding (Stage 1)

- Question: What is the gradient of logistic regression with L2 regularization?
- Answer: Loss: $L(w) = \frac{1}{N}\sum_i[-y_i \log \sigma(z_i) - (1-y_i)\log(1-\sigma(z_i))] + \frac{\lambda}{2}\|w\|_2^2$, where $z_i = w^\top x_i + b$. Gradient: $\nabla_w L = \frac{1}{N}\sum_i (p_i - y_i)x_i + \lambda w$, where $p_i = \sigma(z_i)$. For bias: $\nabla_b L = \frac{1}{N}\sum_i (p_i - y_i)$.
- Question: How does regularization affect the linear regression solution?
- Answer: Ridge regression modifies w^* to $(X^\top X + \lambda N I)^{-1} X^\top y$, pushing weights toward smaller magnitudes and stabilizing the inverse.

With supervised learning under our belts, we are ready to explore data structures and patterns we can find without labels.

Stage 2: Unsupervised Learning

Unsupervised learning explores structure in data without labels. We will learn clustering, dimensionality reduction, and anomaly detection. Each method connects back to Stage 0 (distance, norms, gradients) and Stage 1 (optimization, evaluation).

Overview of Stage 2

We will ask questions like: What groups exist in the data? What lower-dimensional representation preserves meaningful information? Are there points that are unusual? We will build from simple distance-based methods to eigende-compositions and non-linear embeddings.

Clustering: K-Means, Hierarchical, DBSCAN

Clustering groups similar points into clusters using similarity measures.

- Intuition: Imagine points of different colors lying on a plane. K-Means tries to find a small number of cluster centers and assign points to the nearest center. Hierarchical clustering builds a tree of clusters based on proximity. DBSCAN finds dense regions and labels points as core, border, or noise.

- Formal definitions: K-Means minimizes within-cluster sum of squared distances: $\min_{C_1,\ldots,C_k} \sum_{j=1}^{k} \sum_{x \in C_j} \|x - \mu_j\|_2^2$, where μ_j is the centroid. Hierarchical clustering uses linkage criteria (single, average, complete) to merge clusters. DBSCAN defines neighborhood radius ϵ and minimum points m; core points have at least m neighbors within ϵ, border points are near core points, and noise points are neither.
- Why it matters: Clustering reveals hidden groupings (e.g., customer segments) and serves as preprocessing for downstream models.
- Numerical example (K-Means): Let data be $x = [(0,0), (1,1), (2,0), (8,8), (9,9), (10,8)]$, $k = 2$. Initialize centroids at $(1,1)$ and $(9,9)$.
  - Step 1 (assign):
    - Distances from $(0,0)$: to $(1,1)$ is $\sqrt{(0-1)^2 + (0-1)^2} = \sqrt{2} \approx 1.414$, to $(9,9)$ is $\sqrt{(0-9)^2 + (0-9)^2} = \sqrt{162} \approx 12.73$. Assign to cluster 1.
    - $(1,1)$ is equally distant from both centroids; assign to cluster 1 by tie-breaking rule.
    - $(2,0)$: to $(1,1)$ is $\sqrt{(2-1)^2 + (0-1)^2} = \sqrt{2} \approx 1.414$; to $(9,9)$ is $\sqrt{(2-9)^2 + (0-9)^2} = \sqrt{49+81} = \sqrt{130} \approx 11.4$. Assign to cluster 1.
    - $(8,8)$: to $(1,1)$ is $\sqrt{(8-1)^2 + (8-1)^2} = \sqrt{49+49} = \sqrt{98} \approx 9.899$; to $(9,9)$ is $\sqrt{(8-9)^2 + (8-9)^2} = \sqrt{2} \approx 1.414$. Assign to cluster 2.
    - $(9,9)$ and $(10,8)$ similarly assign to cluster 2.
  - Step 2 (recompute centroids):
    - Cluster 1 points: $(0,0)$, $(1,1)$, $(2,0)$. Centroid: $((0+1+2)/3, (0+1+0)/3) = (1, 1/3)$.
    - Cluster 2 points: $(8,8)$, $(9,9)$, $(10,8)$. Centroid: $((8+9+10)/3, (8+9+8)/3) = (9, 25/3 \approx 8.33)$.
  - Iterate until convergence. After a few iterations, centroids settle near $(1, 0.33)$ and $(9, 8.33)$, capturing the two groups.
- Connection: K-Means objective is a loss function minimized via alternating optimization (assign and update). It resembles gradient-based methods but alternates between assignments and centroid updates.
- Common misconceptions: K-Means assumes spherical clusters and is sensitive to initialization; random restarts mitigate this. DBSCAN requires careful choice of ϵ and m.

Dimensionality Reduction: PCA, t-SNE, UMAP, Autoencoders

Reducing dimensionality helps visualize data, compress features, and remove noise.

- Intuition: Imagine projecting a 3D cloud onto a 2D plane while preserving as much variation as possible. PCA finds orthogonal axes (principal

components) that maximize variance. t-SNE and UMAP emphasize local neighborhoods and manifold structure. Autoencoders learn non-linear mappings with bottleneck layers.

- Formal definitions: PCA centers the data and computes the covariance matrix $\Sigma = \frac{1}{N-1} X^\top X$. Eigen-decompose $\Sigma v_i = \lambda_i v_i$. Sort eigenvalues descending; project onto the top k eigenvectors: $Z = XV_k$, where V_k contains the top k eigenvectors. t-SNE defines pairwise similarities via a Student t-distribution in low dimension and a Gaussian in high dimension, minimizing KL divergence. UMAP optimizes cross-entropy between fuzzy set memberships. Autoencoders consist of encoder $h = f_\phi(x)$ and decoder $\hat{x} = g_\theta(h)$; they minimize reconstruction loss $\|x - \hat{x}\|_2^2$.
- Why it matters: Dimensionality reduction enables visualization, data compression, and better feature learning for downstream tasks.
- Numerical example (PCA): Let $X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$. Compute mean: $\bar{x} = (3, 4)$. Center data: $X_c = \begin{bmatrix} -2 & -2 \\ 0 & 0 \\ 2 & 2 \end{bmatrix}$. Compute covariance: $\Sigma = \frac{1}{2} X_c^\top X_c = \frac{1}{2} \begin{bmatrix} (-2)^2 + 0^2 + 2^2 & (-2)(-2) + 0 \cdot 0 + 2 \cdot 2 \\ (-2)(-2) + 0 \cdot 0 + 2 \cdot 2 & (-2)^2 + 0^2 + 2^2 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix} = \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$. Eigenvalues: Solve $\det(\Sigma - \lambda I) = (4 - \lambda)^2 - 16 = 0 \implies (4 - \lambda)^2 = 16 \implies 4 - \lambda = \pm 4$. So $\lambda_1 = 0$ (eigenvector $[1, -1]$), $\lambda_2 = 8$ (eigenvector $[1, 1]$). The top eigenvector is $[1, 1]$, normalized to $[1/\sqrt{2}, 1/\sqrt{2}]$. Project: $Z = X_c \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -2 + (-2) \\ 0 + 0 \\ 2 + 2 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -4 \\ 0 \\ 4 \end{bmatrix} = \begin{bmatrix} -4/\sqrt{2} \\ 0 \\ 4/\sqrt{2} \end{bmatrix} = \begin{bmatrix} -2\sqrt{2} \\ 0 \\ 2\sqrt{2} \end{bmatrix}$. The data are projected onto the direction that maximizes variance (diagonal).
- Connection: PCA relies on eigendecomposition from Stage 0. Autoencoders use optimization (gradients) similar to supervised models but with reconstruction loss.
- Common misconceptions: PCA assumes linearity; non-linear methods like t-SNE/UMAP capture curved structures. Autoencoders risk learning trivial identity mappings; a bottleneck forces meaningful compression.

Anomaly Detection

Identify rare or unusual points that deviate from normal patterns.

- Intuition: Think of finding outliers among many typical examples; anomalies have higher "distance" or lower probability under the model.

- Formal definitions: Gaussian outlier detection models X as $N(\mu, \Sigma)$ and flags points with low likelihood. Isolation Forest builds random trees; anomalies require fewer splits to isolate. One-class SVM finds a boundary around the normal data.
- Why it matters: Detecting fraud, system failures, or quality issues is valuable across industries.
- Numerical example (Gaussian): Suppose X is scalar with $\mu = 10$, $\sigma = 2$. Compute likelihood of $x = 16$: $p = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) = \frac{1}{2\sqrt{2\pi}} \exp\left(-\frac{(16-10)^2}{2 \cdot 4}\right) = \frac{1}{2\sqrt{2\pi}} \exp\left(-\frac{36}{8}\right) = \frac{1}{2\sqrt{2\pi}} e^{-4.5} \approx \frac{1}{5.013} \cdot 0.011 \approx 0.0022$. This is low, indicating anomaly.
- Connection: Distances and probability distributions from Stage 0 support anomaly detection. It is unsupervised because we do not label anomalies in training.
- Common misconceptions: Using thresholds arbitrarily; calibrate thresholds using validation to control false positive/false negative trade-offs.

Check Your Understanding (Stage 2)

- Question: How do you choose k in K-Means?
- Answer: Use the elbow method (plot within-cluster sum of squares vs. k) or silhouette scores to pick a k where improvements plateau.
- Question: Why is PCA useful before classification?
- Answer: PCA reduces noise, decorrelates features, and can improve numerical stability and speed of classification algorithms.

With the ability to structure and summarize data, we move into learning complex functions with deep networks.

Stage 3: Deep Learning Foundations

Deep learning models learn hierarchical features by composing many layers of nonlinear transformations. We will start with perceptrons and multilayer networks, derive the backpropagation algorithm, and then explore specialized architectures: convolutional networks for images and recurrent networks and Transformers for sequences.

Overview of Stage 3

We will build a thorough understanding of how neural networks are constructed, trained, and evaluated. We will revisit calculus (gradients) and optimization (gradient descent) and connect them to new objectives like cross-entropy. We will then discuss how architectural choices (convolution, recurrence, attention) embed structure about data into models.

Neural Networks: Perceptron, Multilayer, Forward and Backward Pass

Neural networks are compositions of functions applied to inputs, with parameters (weights and biases) learned via optimization.

- Intuition: A single perceptron is a linear classifier. A multilayer network stacks nonlinearities to represent complex functions. The forward pass computes outputs; the backward pass computes gradients and updates parameters.
- Formal definition: A feedforward network computes $h^{(1)} = \phi^{(1)}(XW^{(1)} + b^{(1)})$, $h^{(2)} = \phi^{(2)}(h^{(1)}W^{(2)} + b^{(2)})$, $\hat{y} = h^{(L)}$ for outputs. For classification, we often apply softmax to produce probabilities. The loss $L(y, \hat{y})$ penalizes mispredictions.
- Derivation: Chain rule applies to compute gradients of complex compositions. For a scalar loss L, the gradient with respect to a weight $W^{(l)}$ at layer l is $(X^{(l)})$ times an error signal computed from downstream layers. More concretely, for a two-layer network with activation ϕ, the gradient can be expressed via product of Jacobians: $\nabla_{W^{(1)}} L = \frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W^{(1)}}$ with intermediate steps.
- Why it matters: Deep networks can approximate complex functions and learn rich representations from data.
- Numerical example: A tiny perceptron on binary input $[0, 0]$, $[0, 1]$, $[1, 0]$, $[1, 1]$ to output XOR is not possible with a single linear unit (perceptron). With a hidden layer of size 2 and activation ReLU or sigmoid, the network can represent XOR. This demonstrates the power of depth and nonlinearity.
- Connection: The backpropagation algorithm is an application of the chain rule from calculus to compute gradients efficiently in layered networks.
- Common misconceptions: Thinking deeper always means better; deep networks require careful regularization and data.

Activation Functions: Sigmoid, tanh, ReLU, Softmax

Activations introduce nonlinearity so networks can represent complex relationships.

- Intuition: Linear functions (like $w^\top x + b$) draw straight lines and planes. Nonlinear functions (like sigmoid, ReLU) bend space, allowing curved decision boundaries.
- Formal definitions: Sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$, $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, $\text{ReLU}(z) = \max(0, z)$, softmax $p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.
- Why it matters: Nonlinearities are essential for expressive power. Softmax provides proper probabilities for multiclass classification.
- Numerical example: Sigmoid at $z = 0$ is 0.5. At $z = 2$ is 0.8808. At $z = -2$ is 0.1192. tanh at $z = 0$ is 0. At $z = 2$ is 0.9640. At $z = -2$

is -0.9640. ReLU: $\text{ReLU}(-2) = 0$, $\text{ReLU}(2) = 2$. Softmax with logits $[1, 2, 0]$: $e^1 \approx 2.718$, $e^2 \approx 7.389$, $e^0 = 1$. Sum $= 11.107$. Probabilities: $2.718/11.107 \approx 0.245$, $7.389/11.107 \approx 0.665$, $1/11.107 \approx 0.090$.

- Connection: Activation functions change the gradients. For sigmoid, $\sigma'(z) = \sigma(z)(1 - \sigma(z))$, and $\tanh'(z) = 1 - \tanh^2(z)$. ReLU has derivative 0 for negative inputs and 1 for positive inputs.
- Common misconceptions: Vanishing gradients occur when $\sigma(z)$ or $\tanh(z)$ saturate near extremes; this slows learning. ReLU mitigates saturation but can die if many neurons get negative inputs.

Backpropagation Algorithm

Backpropagation computes gradients efficiently by reusing intermediate results.

- Intuition: Think of traversing the network from outputs back to inputs, carrying "error signals" that tell each layer how to adjust its weights.
- Formal definition: For a network with layers $l = 1, ..., L$, define $\delta^{(L)} = \nabla_{\hat{y}} L$ (gradient of loss with respect to outputs). Then for each layer going backward: $\delta^{(l)} = (W^{(l+1)}\delta^{(l+1)}) \odot \phi'(z^{(l)})$, where $z^{(l)} = h^{(l-1)}W^{(l)} + b^{(l)}$, $\odot$ denotes element-wise multiplication, and ϕ' is the activation derivative. Finally, $\nabla_{W^{(l)}} L = (h^{(l-1)})^\top \delta^{(l)}$, $\nabla_{b^{(l)}} L = \sum \delta^{(l)}$.
- Derivation: For a scalar loss, chain rule dictates that the gradient with respect to a parameter is the product of upstream gradients and the local partial derivatives. The use of shared computations (like $z^{(l)}$) yields efficiency.
- Why it matters: Backpropagation is the engine that trains neural networks. Without efficient gradient computation, deep learning would be infeasible.
- Numerical example: Consider a tiny network with one hidden layer: $z_1 = xw_1 + b_1$, $h = \phi(z_1)$, $z_2 = hw_2 + b_2$, $\hat{y} = \sigma(z_2)$, loss $L = (y - \hat{y})^2$. Compute:
  - $\frac{\partial L}{\partial \hat{y}} = -2(y - \hat{y})$.
  - $\frac{\partial \hat{y}}{\partial z_2} = \sigma'(z_2) = \sigma(z_2)(1 - \sigma(z_2))$.
  - So $\delta_2 = \frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2}$.
  - Then $\delta_1 = \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial z_1} = (w_2\delta_2) \cdot \phi'(z_1)$.
  - Gradients: $\frac{\partial L}{\partial w_2} = h \cdot \delta_2$, $\frac{\partial L}{\partial b_2} = \delta_2$, $\frac{\partial L}{\partial w_1} = x \cdot \delta_1$, $\frac{\partial L}{\partial b_1} = \delta_1$.
- Connection: Backpropagation uses the chain rule from calculus (Stage 0) and gradients to update parameters (Stage 1).
- Common misconceptions: Backpropagation is not training itself; it computes gradients. The training step uses those gradients to update parameters (e.g., via SGD or Adam).

Loss Functions: MSE, Cross-Entropy

Loss functions measure mispredictions and guide optimization.

- Intuition: The loss function is the score we minimize. MSE measures distance for regression; cross-entropy measures divergence for classification.
- Formal definitions: MSE: $L = \frac{1}{N}\sum_i \|y_i - \hat{y}_i\|_2^2$. Cross-entropy for multiclass: $L = -\frac{1}{N}\sum_i \sum_{c=1}^{C} y_{i,c} \log p_{i,c}$, where $p_{i,c}$ is the softmax probability for class c, and $y_{i,c}$ are one-hot labels.
- Why it matters: Loss functions determine what the model optimizes. Their shapes affect gradient magnitudes and convergence.
- Numerical example: For a multiclass example with $y = [1, 0, 0]$ and logits $[2, -1, 0]$, softmax gives $p = [e^2/(e^2 + e^{-1} + e^0), e^{-1}/(...), e^0/(...)]$ approximately $[0.860, 0.116, 0.024]$. Cross-entropy: $-\log(0.860) \approx 0.151$. This penalizes wrong predictions heavily; correct predictions have small loss.
- Connection: Cross-entropy aligns with probabilistic modeling and yields well-behaved gradients via backpropagation.
- Common misconceptions: Mixing regression losses with classification tasks often yields poor performance.

Optimization Algorithms: SGD, Adam, RMSprop, Momentum

Optimization algorithms update parameters using gradients; they differ in adaptation and memory.

- Intuition: Momentum accumulates past gradients to speed up progress along persistent directions and dampen oscillations. Adam adapts learning rates per parameter based on first and second moments of gradients.
- Formal definitions:
  - Momentum: $v^{(t)} = \beta v^{(t-1)} + (1-\beta)\nabla L$, $w \leftarrow w - \eta v^{(t)}$.
  - RMSprop: $s^{(t)} = \rho s^{(t-1)} + (1-\rho)(\nabla L)^2$, $w \leftarrow w - \eta \frac{\nabla L}{\sqrt{s^{(t)}}+\epsilon}$.
  - Adam: $m^{(t)} = \beta_1 m^{(t-1)} + (1-\beta_1)\nabla L$, $v^{(t)} = \beta_2 v^{(t-1)} + (1-\beta_2)(\nabla L)^2$, $\hat{m}^{(t)} = m^{(t)}/(1-\beta_1^t)$, $\hat{v}^{(t)} = v^{(t)}/(1-\beta_2^t)$, $w \leftarrow w - \eta \frac{\hat{m}^{(t)}}{\sqrt{\hat{v}^{(t)}}+\epsilon}$.
- Why it matters: Better optimizers converge faster and more stably, especially on noisy data.
- Numerical example: With a small toy loss gradient sequence: $-2, -1.8, -1.5$, momentum with $\beta = 0.9$:
  - $v^{(1)} = 0.9 \cdot 0 + (1 - 0.9) \cdot (-2) = -0.2$.
  - $v^{(2)} = 0.9 \cdot (-0.2) + 0.1 \cdot (-1.8) = -0.18 - 0.18 = -0.36$.
  - $v^{(3)} = 0.9 \cdot (-0.36) + 0.1 \cdot (-1.5) = -0.324 - 0.15 = -0.474$.
  - Momentum smooths the gradient direction; the update steps follow $-v^{(t)}$ scaled by η.
- Connection: All optimizers use gradients; they differ in the path they take toward minima.
- Common misconceptions: Adam never needs learning rate decay; in practice, decaying learning rate helps fine-tune convergence.

CNNs: Convolution, Pooling, Architecture

Convolutional Neural Networks exploit spatial locality and translational invariance.

- Intuition: Convolutions slide a small filter across the image, computing weighted sums (dot products) at each position. This captures local patterns. Pooling aggregates information to reduce resolution.
- Formal definitions: For a 1D case, convolution of input x and filter k yields $y[t] = \sum_i x[i]k[t-i]$. In 2D, $y[i,j] = \sum_{m,n} x[m,n]k[i-m, j-n]$. Common strides and padding control output size. Pooling (max pooling) takes the maximum value in a window; average pooling takes the mean.
- Why it matters: CNNs achieve strong performance on image tasks by learning translation-invariant features.
- Numerical example (simple 1D convolution): Let $x = [1, 2, 3, 4]$ and filter $k = [0.5, -0.5]$. With padding to preserve length (assuming same padding), compute:
  - $y[0] = 1 \cdot 0.5 + 2 \cdot (-0.5) = 0.5 - 1 = -0.5$.
  - $y[1] = 2 \cdot 0.5 + 3 \cdot (-0.5) = 1 - 1.5 = -0.5$.
  - $y[2] = 3 \cdot 0.5 + 4 \cdot (-0.5) = 1.5 - 2 = -0.5$.
  - $y[3] = 4 \cdot 0.5 + 0 \cdot (-0.5) = 2 - 0 = 2$ (padding considered).
- Connection: Convolution layers are linear operations with structured weights (shared filters). Pooling reduces dimensionality and can be seen as a summary operation.
- Common misconceptions: Convolutions with no padding reduce output size; padded convolutions maintain dimensions or control resolution.

RNNs, LSTM, GRU

Recurrent networks process sequences by maintaining hidden states.

- Intuition: RNNs loop information from previous steps to the next step, capturing memory. LSTM and GRU improve on vanilla RNNs by controlling information flow with gates.
- Formal definitions: For a simple RNN, $h_t = \phi(W_x x_t + W_h h_{t-1} + b)$, $\hat{y}_t = f(W_y h_t + b)$. LSTM uses input, forget, and output gates to regulate memory: $i_t = \sigma(W_x^i x_t + W_h^i h_{t-1} + b_i)$, $f_t = \sigma(W_x^f x_t + W_h^f h_{t-1} + b_f)$, $o_t = \sigma(W_x^o x_t + W_h^o h_{t-1} + b_o)$, $\tilde{C}_t = \tanh(W_x^c x_t + W_h^c h_{t-1} + b_c)$, $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$, $h_t = o_t \odot \tanh(C_t)$. GRU simplifies with reset and update gates.
- Why it matters: Sequence modeling is fundamental for language and time series. LSTM and GRU mitigate vanishing gradients.
- Numerical example: Compute a step of LSTM on $x_t = [1, 0]$, $h_{t-1} = [0, 0]$, $C_{t-1} = [0, 0]$. Suppose weights are simple identity for gates: $i_t = \sigma([1,0] \cdot [1,0] + 0) = \sigma(1) \approx 0.731$, $f_t = \sigma([1,0] \cdot [1,0] + 0) = 0.731$, $o_t = \sigma([1,0] \cdot [1,0] + 0) = 0.731$, $\tilde{C}_t = \tanh([1,0] \cdot [1,0] + 0) = \tanh(1) \approx 0.762$.

Then $C_t = f_t \odot 0 + i_t \odot \tilde{C}_t = 0.731 \cdot 0.762 \approx 0.557$. $h_t = o_t \odot \tanh(C_t) = 0.731 \cdot \tanh(0.557) \approx 0.731 \cdot 0.505 \approx 0.369$.
- Connection: RNNs and LSTMs use gradients across time via backpropagation through time (BPTT), extending the chain rule to temporal sequences.
- Common misconceptions: LSTM does not eliminate gradients; it reduces vanishing by multiplicative gates, allowing gradient flow.

Attention Mechanisms and Transformers

Attention weights determine which parts of input matter most. Transformers use attention to process sequences without recurrence.

- Intuition: Think of attention as a spotlight: each output element chooses how much to illuminate each input element based on compatibility.
- Formal definitions: Given queries Q, keys K, values V, attention scores are $S = QK^\top/\sqrt{d_k}$, weights $A = \text{softmax}(S)$, output $O = AV$. Multi-head attention repeats this with multiple projections to capture different relationships. Transformers stack layers of multi-head attention and feedforward networks, often with residual connections and layer normalization.
- Why it matters: Transformers excel at capturing long-range dependencies and are more parallelizable than RNNs.
- Numerical example: Let $Q = \begin{bmatrix}1\\0\end{bmatrix}$, $K = \begin{bmatrix}1\\0\end{bmatrix}$, $V = \begin{bmatrix}5\\5\end{bmatrix}$, $d_k = 1$. Compute $S = QK^\top = 1 \cdot 1 = 1$. Softmax: $A = 1$. Output $O = 1 \cdot V = [5, 5]$.
- Connection: Attention uses dot products (vectors and norms), softmax (to normalize), and gradient-based optimization to learn weights. It relates to Stage 0 mathematics.
- Common misconceptions: Attention outputs are probabilities (after softmax), but they reflect learned compatibility, not guaranteed causal truth.

Check Your Understanding (Stage 3)

- Question: What causes vanishing gradients in deep networks?
- Answer: The chain rule multiplies derivatives along layers; small derivatives (like those from sigmoid or small weights) compound and shrink quickly, slowing learning. LSTM and careful initialization mitigate this.
- Question: Why are transformers powerful?
- Answer: Attention computes relationships across all positions in parallel, capturing global dependencies without recurrence.

With deep learning as our foundation, we move into advanced applications and generative models.

Stage 4: Advanced Topics & Applications

We will explore generative models (GANs, VAEs), reinforcement learning (Q-learning and policy gradients), NLP (embeddings and sequence-to-sequence), computer vision applications, and uncertainty quantification. Each topic builds on Stage 0–3: probability, optimization, and deep learning.

Overview of Stage 4

This final stage shows how to apply deep learning in more complex settings, how to model distributions and generate data, how to learn policies for decision-making, and how to work with sequences and images. We will connect advanced methods back to the core mathematics.

Generative Models: GANs and VAEs

Generative models learn data distributions and can synthesize new samples.

- Intuition: GANs train a generator that produces fake data and a discriminator that tries to distinguish real from fake; they compete. VAEs model latent variables and reconstruct data from low-dimensional representations.
- Formal definitions: GAN objective: $\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p(z)}[\log(1 - D(G(z)))]$. VAE objective: $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - \beta\text{KL}(q_\phi(z|x)\|p(z))$, where $q_\phi(z|x)$ is the encoder, $p_\theta(x|z)$ is the decoder, and $p(z)$ is a prior (often Gaussian).
- Why it matters: Generative models enable creative applications (image synthesis, data augmentation) and representation learning.
- Numerical example (VAE): For a single data point $x = 0.5$, encode with Gaussian $q_\phi(z|x)$ mean $\mu = -0.1$, variance $\sigma^2 = 0.2$. Draw $z \sim N(-0.1, 0.2)$ say $z = -0.2$. Decode to $\hat{x} = \mu_z + \sigma_z \cdot z$ (with parameters), compute reconstruction loss; KL term penalizes divergence from prior $N(0, 1)$. The model trades off reconstruction quality with alignment with the prior.
- Connection: Generative models use likelihood (probability) and KL divergence (a measure of distance between distributions), which rely on Stage 0 probability and statistics. Training uses optimization methods from Stage 1 and deep architectures from Stage 3.
- Common misconceptions: GAN training can be unstable; careful tuning and alternative losses are often necessary. VAEs can blur reconstructed images.

Reinforcement Learning: Basics, Q-Learning, Policy Gradients

Reinforcement Learning (RL) learns policies to maximize reward in sequential decision-making.

- Intuition: An agent takes actions in an environment, receives rewards, and aims to learn a strategy that maximizes cumulative reward.
- Formal definitions: A Markov Decision Process (MDP) consists of states S, actions A, transition dynamics $P(s'|s,a)$, reward function $R(s,a)$, and discount factor γ. Q-learning estimates action-value function $Q(s,a)$ and updates via $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$. Policy gradient methods optimize policy parameters θ via gradient of expected return: $\nabla_\theta \mathbb{E}_{\pi_\theta}[\sum_t r_t]$.
- Why it matters: RL powers game-playing agents (e.g., chess, Go), robotics, and optimization tasks with delayed feedback.
- Numerical example (Q-learning): Suppose $Q(s_0,a_0) = 0$, $Q(s_0,a_1) = 0$, reward $r = 10$ when transitioning from s_0 to s_1 via a_1, and $Q(s_1,a_0) = 5$. With $\alpha = 0.5$, $\gamma = 0.9$:
  - Update: $Q(s_0,a_1) \leftarrow 0 + 0.5[10 + 0.9 \cdot 5 - 0] = 0 + 0.5[14.5] = 7.25$.
- Connection: RL uses probability and statistics (expected rewards) and optimization via gradient descent-like updates (Q-learning approximates gradient). Policy gradients use derivatives and backpropagation (for neural policy networks).
- Common misconceptions: Delayed rewards complicate credit assignment; function approximators (deep Q-networks) stabilize estimates.

NLP: Embeddings, Sequence-to-Sequence

Natural Language Processing models words as vectors and sequences to sequences.

- Intuition: Embeddings map words into vectors capturing semantic similarity. Seq2seq models translate or transform sequences by encoding them and decoding to output sequences.
- Formal definitions: Word embeddings $E \in \mathbb{R}^{V \times d}$ map tokens x to dense vectors $e_x = E[x]$. A seq2seq model has an encoder that produces a context vector c from input tokens and a decoder that uses c to generate output tokens, often via attention.
- Why it matters: Embeddings and attention allow models to capture language structure and relationships beyond simple bag-of-words.
- Numerical example: Map words "cat" to $[0.2, -0.1]$, "dog" to $[0.19, -0.09]$; they are close in embedding space, indicating similar semantics.
- Connection: Embeddings are learned via optimization using gradients (Stage 3). Attention in NLP uses dot products and softmax.
- Common misconceptions: Treating embeddings as static; they are learned

and context-dependent (contextual embeddings via Transformers improve on static ones).

Computer Vision Applications

Computer vision applies deep learning to images, often with CNNs and Transformers.

- Intuition: Images have spatial structure; convolution captures local patterns. Transformer-based vision models (like ViT) embed patches of images and use attention.
- Formal definitions: Typical CNN pipeline: convolutional layers, pooling, non-linearities, classification heads. Vision Transformers flatten image patches, add positional encodings, and apply multi-head attention.
- Why it matters: Vision enables object detection, segmentation, and generation for medical imaging, autonomous vehicles, and more.
- Numerical example: A small 3×3 image and a 3×3 edge-detection kernel (e.g., Sobel) will produce a new image highlighting edges after convolution.
- Connection: Convolutions and attention use Stage 0 linear algebra and statistics; training uses Stage 3 optimization and deep architectures.
- Common misconceptions: Deep networks require large datasets; pretraining on large corpora and fine-tuning on smaller domain-specific data often improves performance.

Uncertainty Quantification

Quantifying uncertainty helps interpret and trust model predictions.

- Intuition: Models can be confident but wrong; uncertainty flags low-confidence predictions or estimates of model error.
- Formal definitions: Epistemic uncertainty (model uncertainty) measures uncertainty about parameters; aleatoric uncertainty (data uncertainty) measures irreducible noise. Bayesian neural networks approximate posterior distributions over weights. Monte Carlo dropout performs dropout during inference to estimate uncertainty.
- Why it matters: In high-stakes applications (medical diagnosis, autonomous driving), uncertainty informs risk-aware decisions.
- Numerical example: Use dropout at inference time (50% rate). Run forward pass 100 times on the same input; collect predictions. The mean of predictions gives a point estimate; variance quantifies uncertainty.
- Connection: Uncertainty quantification relies on probability distributions (Stage 0) and statistical inference; it extends models to behave probabilistically.

- Common misconceptions: Equating high softmax probabilities with low uncertainty; calibration (e.g., temperature scaling) often improves probability interpretation.

Check Your Understanding (Stage 4)

- Question: How does the KL term in VAEs affect learned representations?
- Answer: The KL term encourages latent variables to follow the prior distribution (e.g., standard normal), leading to smoother and more interpretable latent spaces. Stronger KL weight pushes representations toward the prior; weaker KL weight yields better reconstruction.
- Question: Why are Transformers effective for long sequences?
- Answer: Attention directly computes interactions between any two positions, bypassing the sequential bottleneck of RNNs. Multi-head attention captures diverse relationships.

Synthesis and Next Steps

We have traversed a comprehensive course: from basic mathematics (Stage 0) to supervised learning (Stage 1), unsupervised learning (Stage 2), deep learning (Stage 3), and advanced applications (Stage 4). Along the way, we defined every term, explained each symbol, and derived the core formulas step by step. We built intuition and connected topics to each other and to the underlying mathematics.

Recap of the Journey

- Stage 0 grounded us in sets, functions, vectors/matrices, calculus, optimization, probability, and statistics. These tools underpin everything else.
- Stage 1 introduced models (linear regression, logistic regression), training procedures (gradient descent), and model selection strategies (cross-validation, regularization).
- Stage 2 taught us how to discover structure without labels (clustering, dimensionality reduction) and how to detect anomalies.
- Stage 3 explained neural networks, activation functions, backpropagation, and specialized architectures (CNNs, RNNs, Transformers).
- Stage 4 covered generative models, reinforcement learning, NLP, computer vision, and uncertainty quantification.

Validation of Teaching Principles

We adhered to the teaching principles:

- Completeness: Every concept was defined from basics, described in plain language, presented with notation, and connected to applications. Numerical examples were provided with step-by-step calculations.
- Scaffolding: Each stage built on the previous stage, with explicit connections and reminders of prior material.
- Explanation diversity: Each major concept included intuition, formal definition, applications, numerical examples, connections to other concepts, and common misconceptions.
- Anti-assumption: No prior ML knowledge was assumed beyond high school math. We defined jargon on first use.
- Anti-skipping: We did not skip derivations or steps; we included mathematics with clear explanations of symbols.

Guidance for Further Study

To go further, you should:

- Implement these algorithms in code. Coding consolidates understanding. Start with simple linear regression and logistic regression in Python, then move to small neural networks. Implement backpropagation from scratch to strengthen the concept.
- Explore larger datasets and benchmark tasks (e.g., CIFAR-10 for vision, GLUE for NLP). Compare classical ML and deep models to see strengths and weaknesses.
- Study probabilistic models more deeply: Bayesian linear models, Gaussian processes, and variational inference. This builds a robust understanding of uncertainty.
- Practice reading research papers and reproducing results. Focus on understanding assumptions, derivations, and limitations.
- Develop domain expertise where ML will be applied (e.g., finance, healthcare, engineering). The right problem framing and data quality matter more than any algorithm.

Final Remarks

Machine learning is both an art and a science. The mathematics provides rigor and clarity; the applications provide purpose and creativity. This course has equipped you with the conceptual foundations and practical intuition necessary to continue learning, experimenting, and contributing in the field. Continue building your understanding step by step, always grounding complex ideas in

the fundamentals we have covered. Your mathematical toolkit is now ready; use it to explore, to reason, and to create.