Welcome to the Machine

Prequisites

Yinbo Wang

School of Statistics, RUC

May 9, 2025

Prequisites

Learning by doing

Math

- **Linear Algebra**
- **Matrix Calculus**
- **Probability Theory**
- **Mathematical Statistics**

Coding

- **Python**
- **Numpy**
- **PyTorch**

0-0 NumPy (‘numpy‘)

Numerical Python

What is it?

- Fundamental package for scientific computing in Python.
- Provides a high-performance **multidimensional array object** and tools for working with these arrays.

Core Object: ‘ndarray‘

- Fast, memory-efficient multidimensional array for *homogeneous* data.
- Dimensions:
    - 1D: Vector
    - 2D: Matrix
    - 3D+: Tensor

0-1 NumPy: Key Features & Why Use It

Key Features

- **Vectorized Operations**: Element-wise ops on arrays without Python loops (much faster).
- **Broadcasting**: Operations on arrays of different shapes.
- **Rich Functionality**: Math functions, linear algebra, random numbers.
- **Interoperability**: Base for Pandas, SciPy, Scikit-learn, PyTorch.

Why Use It?

- **Performance**: Significantly faster than pure Python.
- **Convenience**: Concise syntax for numerical operations.
- **Foundation**: Bedrock of scientific Python.

0-2 NumPy: Simple Example

Basic Operations

```python
import numpy as np

# Create a NumPy array from a Python list
a = np.array([1, 2, 3, 4, 5])
b = np.array([6, 7, 8, 9, 10])

# Element-wise addition (vectorized)
c = a + b
print(f"a: {a}")
print(f"b: {b}")
print(f"a + b: {c}")

# Scalar multiplication
d = a * 2
print(f"a * 2: {d}")

# Create a 2D array (matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(f"Matrix:\n{matrix}")
print(f"Matrix shape: {matrix.shape}")
```

0-3 PyTorch (‘torch‘)

Machine Learning Framework

What is it?

- Open-source machine learning framework (Meta AI).
- Known for flexibility, ease of use, Python integration.
- Widely used for deep learning research and production.

Core Object: ‘Tensor‘

- Multi-dimensional arrays, similar to NumPy’s ‘ndarray‘.
- **Key Difference**: Can be moved to GPUs for parallel computation.
- Basis of **Automatic Differentiation (Autograd)**.

0-4 PyTorch: Key Features

- **GPU Acceleration**: Seamlessly run computations on GPUs.
- **Automatic Differentiation (‘torch.autograd‘)**:
  - Automatically computes gradients.
  - Backbone of training neural networks via backpropagation.
  - Builds a "computational graph" dynamically.
- **Neural Network Module (‘torch.nn‘)**: Pre-defined layers, loss functions.
- **Optimization Algorithms (‘torch.optim‘)**: SGD, Adam, etc.
- **Utilities**: Data loading, distributed training.
- **Dynamic Computational Graphs**: Graph built "on the fly", easier debugging, flexible architectures.

0-5 PyTorch: Why Use It for ML/DL?

Advantages

- **Python-first**: Natural for Python developers.
- **Flexibility & Control**: Balance of high-level abstractions and low-level control.
- **Strong Research Community**: Rapid adoption of new ideas.
- **Ease of Debugging**: Dynamic graphs aid inspection.

0-6 PyTorch: Simple Example

Basic Operations

```python
import torch
a = torch.tensor([1, 2, 3, 4, 5], dtype=torch.float32)
b = torch.tensor([6, 7, 8, 9, 10], dtype=torch.float32)

# Element-wise addition
c = a + b
print(f"a + b: {c}")

# Scalar multiplication
d = a * 2
print(f"a * 2: {d}")

# Create a 2D tensor (matrix)
matrix = torch.tensor([[1, 2, 3], [4, 5, 6]], dtype=torch.float32)
print(f"Matrix:\n{matrix}")
print(f"Matrix shape: {matrix.shape}")

# Automatic differentiation example
x = torch.tensor(2.0, requires_grad=True)
y = x**2 + 3*x + 1
y.backward() # Computes gradients dy/dx
print(f"x: {x}")
print(f"y = x^2 + 3x + 1: {y}")
print(f"dy/dx at x=2: {x.grad}") # Expected: 2*x + 3 = 2*2 + 3 = 7
```

0-7 NumPy vs. PyTorch Tensors

- PyTorch tensors can be easily converted to NumPy arrays and vice-versa.
- `tensor.numpy()`: PyTorch Tensor → NumPy array.
- `torch.from_numpy(ndarray)`: NumPy array → PyTorch Tensor.

Interoperability

```
numpy_arr = np.array([10, 20, 30])
pytorch_tensor = torch.from_numpy(numpy_arr)
print(f"From NumPy to PyTorch: {pytorch_tensor}")

new_numpy_arr = pytorch_tensor.numpy()
print(f"From PyTorch to NumPy: {new_numpy_arr}")

# If a tensor is on GPU, you need to move it to CPU first
# if torch.cuda.is_available():
#     gpu_tensor = torch.tensor([1,2,3], device="cuda")
#     # cpu_tensor = gpu_tensor.cpu() # Move to CPU
#     # numpy_arr_from_gpu = cpu_tensor.numpy()
```

1-0 Linear Algebra

Overview

Linear algebra is the branch of mathematics concerning vector spaces and linear mappings between them. In ML, it's fundamental for **representing data, defining transformations, and solving systems of equations**.

Why it's important for ML/DL?

- **Data Representation**: Datasets as matrices, multimedia as tensors.
- **Model Parameters**: Weights and biases as matrices/vectors.
- **Transformations**: Core operations in NNs (matrix multiplications).
- **Dimensionality Reduction**: E.g., PCA.
- **Optimization**: Solving linear systems.

1-1 Scalars, Vectors, Matrices, Tensors

- **Scalar**: A single number (e.g., 5, 3.14).
- **Vector**: An ordered array of numbers.
  - In ML: Represents a single data point or feature vector.
- **Matrix**: A 2D array of numbers.
  - In ML: Represents a dataset or model parameters (NN layer weights).
- **Tensor**: Generalization to N dimensions.
  - 0D: Scalar, 1D: Vector, 2D: Matrix.
  - 3D: RGB image (height, width, channels).
  - 4D: Batch of RGB images (batch, height, width, channels).

NumPy/PyTorch Implementation

Both libraries provide powerful tools for these:

- NumPy: `np.array()`, `.shape`, `.ndim`
- PyTorch: `torch.tensor()`, `.shape`, `.ndim`, `dtype`

1-2 Dot Product (Inner Product)

- For vectors $\mathbf{a}, \mathbf{b}$ of length n: $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$. Result is a scalar.
- Geometric: $\mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}|| \cdot ||\mathbf{b}|| \cos(\theta)$.
  - Measures similarity/alignment. Orthogonal if $\mathbf{a} \cdot \mathbf{b} = 0$.

NumPy/PyTorch Implementation

- NumPy: `np.dot(a,b)`, `a @ b`, `np.sum(a*b)`
- PyTorch: `torch.dot(a,b)`, `torch.matmul(a,b)`, `a @ b`

1-3: Matrix Multiplication

- $\mathbf{A}$ $(m \times n)$, $\mathbf{B}$ $(n \times p)$ $\implies$ $\mathbf{C} = \mathbf{AB}$ is $m \times p$.
- $C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$ (dot product of i-th row of $\mathbf{A}$ and j-th col of $\mathbf{B}$).
- Order matters: $\mathbf{AB} \neq \mathbf{BA}$ generally.
- Fundamental in NNs: `output = activation(weights @ inputs + bias)`.

NumPy/PyTorch Implementation

- NumPy: `np.matmul(A,B)`, `A @ B`
- PyTorch: `torch.matmul(A,B)`, `A @ B`, `torch.mm(A,B)` (for 2D only)

1-4 Transpose

- $\mathbf{A}^T$: Flips matrix over main diagonal. Rows become columns.
- If $\mathbf{A}$ is $m \times n$, $\mathbf{A}^T$ is $n \times m$. $(A^T)_{ij} = A_{ji}$.
- Property: $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$.

NumPy/PyTorch Implementation

- NumPy: `A.T`, `np.transpose(A)`
- PyTorch: `A.T`, `torch.transpose(A, 0, 1)`

1-5 Inverse & Pseudo-inverse

- **Inverse ($\mathbf{A}^{-1}$)**: For square $\mathbf{A}$, if it exists.
  - $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$ (identity).
  - Only exists if $\mathbf{A}$ is non-singular (determinant $\neq 0$).
  - Solves $\mathbf{A}\mathbf{x} = \mathbf{b} \implies \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.
- **Pseudo-inverse ($\mathbf{A}^{+}$)**: Generalization for non-square or singular matrices.
  - Finds "best fit" (least squares) solution to $\mathbf{A}\mathbf{x} = \mathbf{b}$.
  - Used in Linear Regression (Normal Eq: $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$).

NumPy/PyTorch Implementation

- NumPy: `numpy.linalg.inv()`, `numpy.linalg.pinv()`
- PyTorch: `torch.linalg.inv()`, `torch.linalg.pinv()`

1-6 Determinant

- Scalar value from a square matrix.
- Geometrically: Signed area (2D) or volume (3D) of parallelogram/parallelepiped formed by column/row vectors.
- If determinant is 0, matrix is singular (no inverse, transformation collapses space).

NumPy/PyTorch Implementation

- NumPy: `numpy.linalg.det()`
- PyTorch: `torch.linalg.det()`

1-7 Eigenvalues & Eigenvectors

- For square $\mathbf{A}$, eigenvector $\mathbf{v}$ and eigenvalue λ: $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$.
- Vector $\mathbf{v}$'s direction is unchanged by $\mathbf{A}$, only scaled by λ.
- Eigenvectors: Principal axes of transformation by $\mathbf{A}$.
- Eigenvalues: Scaling factor along these axes.
- Crucial for PCA, matrix powers, stability analysis.

NumPy/PyTorch Implementation

- NumPy: `numpy.linalg.eig()`
- PyTorch: `torch.linalg.eig()` (complex), `torch.linalg.eigh()` (real symmetric)

1-8: Singular Value Decomposition (SVD)

- Factorization of *any* $m \times n$ matrix $\mathbf{A}$ into $\mathbf{A} = \mathbf{U\Sigma V}^T$.
  - $\mathbf{U}$: $m \times m$ orthogonal (left singular vectors).
  - $\mathbf{\Sigma}$: $m \times n$ diagonal (singular values $\sigma_i \geq 0$).
  - $\mathbf{V}^T$: $n \times n$ orthogonal (rows are right singular vectors).
- Singular values $\approx$ "strengths" of principal components.
- **Applications**: PCA, dimensionality reduction, matrix approximation, pseudo-inverse computation.
- Reconstructing $\mathbf{A}$ from U, S, Vh is common: `U @ Sigma_matrix @ Vh`.

NumPy/PyTorch Implementation

- NumPy: `numpy.linalg.svd()`
- PyTorch: `torch.linalg.svd()`

2-0 Matrix Calculus

Overview

Extends calculus concepts (derivatives, gradients) to functions involving matrices and vectors. Essential for optimizing ML models.

Why it's important for ML/DL?

- **Optimization**: Most ML models are trained by minimizing a loss function. Gradient Descent requires computing gradients of loss w.r.t. model parameters (matrices/vectors).
- **Backpropagation**: Algorithm for training NNs, an application of chain rule from matrix calculus.

2-1 Gradient

- For a **scalar-valued function** $f(\mathbf{x})$ of a vector $\mathbf{x} = [x_1, \dots, x_n]^T$.
- Gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$ is a vector of partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

- Points in direction of steepest ascent. $-\nabla_{\mathbf{x}} f(\mathbf{x})$ points in direction of steepest descent.

2-2 Jacobian

- For a **vector-valued function** $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}^m$.
- Jacobian matrix $\mathbf{J}$ is an $m \times n$ matrix of all first-order partial derivatives: $J_{ij} = \frac{\partial f_i}{\partial x_j}$.

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} (\nabla_{\mathbf{x}} f_1(\mathbf{x}))^T \\ \vdots \\ (\nabla_{\mathbf{x}} f_m(\mathbf{x}))^T \end{bmatrix}$$

- If $m = 1$ (scalar function), Jacobian is transpose of gradient vector.

2-3 Hessian

- For a **scalar-valued function** $f(\mathbf{x})$ of a vector $\mathbf{x}$.
- Hessian matrix $\mathbf{H}$ is an $n \times n$ matrix of second-order partial derivatives: $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

- Describes local curvature of the function. Used in second-order optimization (e.g., Newton's method).

2-4 Chain Rule (for vectors/matrices)

- Scalar case: If $y = f(u)$ and $u = g(\mathbf{x})$, then $\frac{\partial y}{\partial x_i} = \frac{df}{du}\frac{\partial g}{\partial x_i}$.
- Vector case: If $\mathbf{z} = f(\mathbf{y})$ and $\mathbf{y} = g(\mathbf{x})$, then the Jacobian of $\mathbf{z}$ w.r.t. $\mathbf{x}$ is:

$$\mathbf{J_x(z)} = \mathbf{J_y(z)J_x(y)}$$

- Fundamental principle behind backpropagation in neural networks.

2-5 PyTorch Implementation (Autograd)

Core Idea: ‘autograd‘

PyTorch’s ‘autograd‘ package automatically computes gradients.

- Tensors can track operations if `requires_grad=True`.
- A computation graph is built dynamically.
- `loss.backward()` computes gradients of ‘loss‘ w.r.t. all parameters with `requires_grad=True`.
- Gradients are stored in the `.grad` attribute of tensors.

Scalar Output Example (Conceptual)

```
x = torch.tensor([1.,2.,3.])
w = torch.tensor([0.1,0.2,0.3], requires_grad=True)

y = (w @ x)**2  # Forward pass, builds graph
y.backward()    # Backward pass, computes dy/dw

print(w.grad)   # Access gradient dy/dw
```

2-5 PyTorch Implementation (Autograd)

- For full Jacobians/Hessians: `torch.autograd.functional.jacobian`, `torch.autograd.functional.hessian`.
- 'backward()' computes gradients essential for optimization via gradient descent.

Vector Output Example (Conceptual)

```
x = torch.tensor([1.,2.,3.])
w = torch.tensor([0.1,0.2,0.3], requires_grad=True)

# For Jacobian
# Define the function whose Jacobian is sought
def compute_model_output(weights_arg):
# 'x' is captured from the outer scope where it's defined.
# 'weights_arg' will be the 'w' tensor when 'jacobian' is called.
return torch.stack([weights_arg @ x, (weights_arg**2).sum()])

jacobian = torch.autograd.functional.jacobian(compute_model_output, w)
print(jacobian)  # Access Jacobian
```

3-0 Probability Theory

Quantifying Uncertainty and Randomness

Concept

The mathematical framework for **quantifying uncertainty and randomness**. In its modern, axiomatic formulation, probability is a type of measure.

Why it's important for ML/DL?

- **Modeling Uncertainty**: Data is noisy, processes are stochastic.
- **Generative Models**: Learn probability distributions (GANs, VAEs).
- **Loss Functions**: Derived from probabilistic principles (e.g., cross-entropy).
- **Bayesian Methods**: Probabilistic approach to inference.

3-1 Probability Space $(\Omega, \mathcal{F}, P)$

Models a random process with three components:

- **Sample Space (Ω)**: Set of *all possible outcomes*.
  - Coin flip: $\Omega = \{H, T\}$. Dice roll: $\Omega = \{1, 2, 3, 4, 5, 6\}$.
- **Event Space ($\mathcal{F}$ or Sigma-Algebra)**: Collection of subsets of Ω (events) to which we assign probabilities. Must satisfy:
  1. $\Omega \in \mathcal{F}$
  2. If $A \in \mathcal{F}$, then $A^c \in \mathcal{F}$ (closed under complement)
  3. If $A_1, A_2, \cdots \in \mathcal{F}$, then $\bigcup_{i=1}^{\infty} A_i \in \mathcal{F}$ (closed under countable unions)
- **Probability Measure (P)**: Function $P : \mathcal{F} \to [0, 1]$ satisfying Kolmogorov Axioms:
  1. Non-negativity: $P(A) \geq 0$ for any event A.
  2. Normalization: $P(\Omega) = 1$.
  3. Countable Additivity: For disjoint events A_i, $P(\bigcup A_i) = \sum P(A_i)$.

3-2 Random Variables (RV)

- **Intuition**: Assigns a numerical value to each outcome in Ω.
- **Formal Definition**: A function $X : \Omega \to \mathbb{R}$ such that for every Borel set $B \subset \mathbb{R}$ (any "reasonable" subset), its pre-image $X^{-1}(B) = \{\omega \in \Omega : X(\omega) \in B\}$ is an event in $\mathcal{F}$.
    - This is **measurability**. Ensures we can ask $P(X \in B)$.
    - Sufficient to check for $B = (-\infty, x]$, i.e., $\{\omega : X(\omega) \leq x\} \in \mathcal{F}$. This allows defining CDF.
- **Types**:
    - **Discrete RV**: Takes finite or countably infinite distinct values.
    - **Continuous RV**: Can take any value in a continuous interval.

3-3 Cumulative Distribution Function (CDF)

- For any RV X, its CDF $F_X(x) = P(X \leq x)$.
- **Properties**:
  1. Non-decreasing: $a < b \implies F_X(a) \leq F_X(b)$.
  2. Limits: $\lim_{x \to -\infty} F_X(x) = 0$, $\lim_{x \to +\infty} F_X(x) = 1$.
  3. Right-continuous: $\lim_{h \to 0^+} F_X(x + h) = F_X(x)$.
- $P(a < X \leq b) = F_X(b) - F_X(a)$.

3-4 Probability Mass Function (PMF) (Discrete RVs)

- For a discrete RV X taking values in $S = \{x_1, x_2, \dots\}$.
- PMF $p_X(x) = P(X = x)$.
- **Properties**:
  1. $p_X(x) \geq 0$ for $x \in S$, and $p_X(x) = 0$ if $x \notin S$.
  2. $\sum_{x \in S} p_X(x) = 1$.
- Examples: Bernoulli(p), Binomial(n, p), Poisson(λ).

3-5 Probability Density Function (PDF) (Continuous RVs)

- For a continuous RV X, its PDF $f_X(x)$ is such that $F_X(x) = \int_{-\infty}^{x} f_X(t)dt$.
- $f_X(x) = \frac{dF_X(x)}{dx}$ where $F_X(x)$ is differentiable.
- Important: $f_X(x)$ is NOT $P(X = x)$. For continuous RVs, $P(X = x) = 0$.
- $P(a < X \leq b) = \int_{a}^{b} f_X(x)dx$.
- **Properties**:
  1. $f_X(x) \geq 0$. (PDF values can be > 1).
  2. $\int_{-\infty}^{\infty} f_X(x)dx = 1$.

3-6 Expected Value, Variance, Covariance

- **Expected Value (Mean $E[X]$ or μ_X)**:
  - Discrete: $E[g(X)] = \sum_{x \in S} g(x) p_X(x)$.
  - Continuous: $E[g(X)] = \int_{-\infty}^{\infty} g(x) f_X(x) dx$.
- **Variance ($Var(X)$ or σ_X^2)**:
  - $Var(X) = E[(X - E[X])^2] = E[X^2] - (E[X])^2$.
  - Standard Deviation: $\sigma_X = \sqrt{Var(X)}$.
- **Covariance ($Cov(X, Y)$)**: (Requires joint distribution)
  - $Cov(X, Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$.
- **Correlation (ρ_{XY})**: $\frac{Cov(X,Y)}{\sigma_X \sigma_Y}$.
- PyTorch distributions often have `.mean`, `.variance` properties.

3-7 Joint, Marginal, Conditional Distributions

- **Joint Distribution**: Describes simultaneous behavior of multiple RVs.
  - $F_{X,Y}(x,y) = P(X \leq x, Y \leq y)$ (Joint CDF).
  - $p_{X,Y}(x,y) = P(X = x, Y = y)$ (Joint PMF).
  - $f_{X,Y}(x,y)$ (Joint PDF).
- **Marginal Distribution**: Distribution of a subset of RVs, "averaging out" others.
  - $p_X(x) = \sum_y p_{X,Y}(x,y)$.
  - $f_X(x) = \int f_{X,Y}(x,y)dy$.
- **Conditional Distribution**: Distribution of one RV given specific values for others.
  - $p_{Y|X}(y|x) = P(Y = y|X = x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$. (Similar for PDF).
- **Chain Rule**: $p(x,y) = p(y|x)p(x)$. Generalizes to many variables, key for graphical models.

3-8 Independence and Conditional Independence

- **Independence of RVs ($X \perp Y$)**:
  - Joint distribution factors: $p_{X,Y}(x,y) = p_X(x)p_Y(y)$ (or $f_{X,Y}(x,y) = f_X(x)f_Y(y)$).
  - Knowing one gives no info about the other: $p_{Y|X}(y|x) = p_Y(y)$.
  - If independent, $Cov(X,Y) = 0$. (Converse not always true unless jointly Normal).
- **Conditional Independence ($X \perp Y|Z$)**:
  - $p_{X,Y|Z}(x,y|z) = p_{X|Z}(x|z)p_{Y|Z}(y|z)$.
  - $p_{X|Y,Z}(x|y,z) = p_{X|Z}(x|z)$ (Given Z, Y gives no extra info about X).
  - Cornerstone of probabilistic graphical models (e.g., Naive Bayes assumes features are conditionally independent given class).

3-9 Limit Theorems: LLN and CLT

Sum/average of many i.i.d. RVs tends towards Normal distribution.

Law of Large Numbers (LLN)

- Let $X_1, \dots, X_n$ be i.i.d. with mean μ. Sample mean $\bar{X}_n = \frac{1}{n}\sum X_i$.
- **Weak LLN**: $\lim_{n\to\infty} P(|\bar{X}_n - \mu| < \epsilon) = 1$. (Convergence in probability).
- **Strong LLN**: $P(\lim_{n\to\infty} \bar{X}_n = \mu) = 1$. (Almost sure convergence).
- **Importance**: Justifies Monte Carlo, param estimation, empirical risk.

Central Limit Theorem (CLT)

- Let X_i be i.i.d. with mean μ, variance σ^2.
- Standardized sum/mean: $Z_n = \frac{\sum X_i - n\mu}{\sigma\sqrt{n}} = \frac{\bar{X}_n - \mu}{\sigma/\sqrt{n}} \xrightarrow{d} N(0,1)$.
- So, $\bar{X}_n \approx N(\mu, \sigma^2/n)$ for large n.
- **Importance**: Foundation for CIs, explains ubiquity of Normal dist.

3-10 Concentration Inequalities

Non-asymptotic bounds on deviation from expectation.

- **Markov's Inequality**: For non-negative RV X, $a > 0$:

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$$

  - General but often loose. X must be non-negative.

- **Chebyshev's Inequality**: For RV X with mean μ, variance σ^2, $k > 0$:

$$\mathbb{P}(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

  - Stronger than Markov, uses variance. Applies to any distribution with finite mean/var.

3-10 Concentration Inequalities

Non-asymptotic bounds on deviation from expectation.

- **Extended Markov's Inequality**:
  - If φ is a nondecreasing nonnegative function, X is a (not necessarily nonnegative) random variable, and $\varphi(a) > 0$, then

$$\mathbb{P}(X \geq a) \leq \frac{\mathbb{E}[\varphi(X)]}{\varphi(a)}.$$

  - **Corollary (Higher Moments)**: For any RV X, integer $n \geq 1$, and $a > 0$:

$$\mathbb{P}(|X| \geq a) \leq \frac{\mathbb{E}[|X|^n]}{a^n}.$$

    (This follows by applying the extended form to the RV $|X|$ with $\varphi(x) = x^n$ for $x \geq 0$).

3-10 Concentration Inequalities

Further bounds and extensions.

- **Hoeffding's Inequality**: For sum S_n of n independent *bounded* RVs $X_i \in [a_i, b_i]$:

$$\mathbb{P}(S_n - \mathbb{E}[S_n] \geq t) \leq \exp\left(-\frac{2t^2}{\sum(b_i - a_i)^2}\right)$$

  For i.i.d. Bernoulli(p) variables, $\bar{X}_n = \hat{p}$: $\mathbb{P}(|\hat{p} - p| \geq \epsilon) \leq 2\exp(-2n\epsilon^2)$.
  - Tighter for bounded RVs, exponential decay. Crucial for generalization bounds in ML.
- **Others**: Chernoff, Bernstein, McDiarmid (more specialized/tighter).

Core Idea

Using data to infer properties of an unknown underlying probability distribution or model.

Probability Theory:

- Model $\rightarrow$ Data
- Known model, predict outcomes.
- Deductive.
- "If coin is fair, $P(7H$ in $10)$?"

Statistics:

- Data $\rightarrow$ Model
- Unknown model, infer from data.
- Inductive.
- "Given $7H$ in 10, is coin fair?"

Why it's important for ML/DL?

- Parameter Estimation, Model Building/Selection, Model Evaluation, Uncertainty Quantification, Understanding Generalization.

4-1 Statistical Inference: The Goal

Deduce properties of an underlying probability distribution from data.

- **Point Estimation**: Single "best" value for an unknown parameter (e.g., mean μ).
- **Interval Estimation**: Range of plausible values (e.g., confidence interval for μ).
- **Hypothesis Testing**: Decide between competing claims (e.g., is $\mu = 0$ vs $\mu \neq 0$?).
- **Prediction**: Predict future observations.

We primarily focus on point estimation for ML model training.

4-2 Samples, Statistics, Estimators

- **Random Sample**: $X_1, \ldots, X_n$ i.i.d. from $f(x|\theta)$ or $p(x|\theta)$.
  - Observed data $x_1, \ldots, x_n$ are realizations.
- **Statistic**: Function $T(X_1, \ldots, X_n)$ of sample, not depending on unknown θ.
  - E.g., sample mean $\bar{X}$, sample variance S^2.
- **Estimator** $\hat{\theta}$: Statistic used to estimate θ. It's an RV.
- **Estimate**: Specific value $\hat{\theta}(x_1, \ldots, x_n)$ from data.

Desirable Properties of Estimators

- **Unbiasedness**: $E[\hat{\theta}] = \theta$. (Bias: $E[\hat{\theta}] - \theta$).
- **Efficiency**: Smaller variance among unbiased estimators.
- **Consistency**: $\hat{\theta}_n \xrightarrow{P} \theta$ as $n \to \infty$.
- **Sufficiency**: Captures all info about θ from sample.

4-3 Point Estimation: Maximum Likelihood Estimation (MLE)

Find parameter value that makes observed data most probable.

- **Likelihood Function** $L(\theta|\mathbf{x})$: Joint PMF/PDF of sample, viewed as function of θ for fixed data $\mathbf{x}$.

$$L(\theta|\mathbf{x}) = \prod_{i=1}^{n} f(x_i|\theta) \quad (\text{or } p(x_i|\theta))$$

  - $L(\theta|\mathbf{x})$ is NOT a probability dist. for θ.
- **MLE Principle**: $\hat{\theta}_{MLE} = \arg\max_\theta L(\theta|\mathbf{x})$.
- **Log-Likelihood** $\ell(\theta|\mathbf{x})$: $\log L(\theta|\mathbf{x}) = \sum_{i=1}^{n} \log f(x_i|\theta)$.
  - Easier to work with (products to sums, numerical stability).
  - $\hat{\theta}_{MLE} = \arg\max_\theta \ell(\theta|\mathbf{x})$.
- **Finding MLE**: Differentiate $\ell(\theta|\mathbf{x})$ w.r.t θ, set to 0, solve. (Or numerical opt.)

4-4 MLE Examples

Example 1: Bernoulli parameter p

Data $X_1, \ldots, X_n \sim \text{Bernoulli}(p)$. $k = \sum x_i$.
$\ell(p|\mathbf{x}) = k \log p + (n-k)\log(1-p)$. $\implies \hat{p}_{MLE} = k/n = \bar{x}$ (sample proportion).

Example 2: Mean μ of Normal $N(\mu, \sigma^2)$ (σ^2 known)

$\ell(\mu|\mathbf{x}, \sigma^2) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum(x_i - \mu)^2$. $\implies \hat{\mu}_{MLE} = \frac{1}{n}\sum x_i = \bar{x}$ (sample mean).

4-5 MLE Properties

Properties of MLEs (under regularity conditions)

Consistent, Asymptotically Normal, Asymptotically Efficient, Invariance.

- May be biased for finite samples (e.g., $\hat{\sigma}^2_{MLE}$ for Normal).

MLE in Machine Learning

Many loss functions are Negative Log-Likelihoods (NLL). Minimizing NLL $\equiv$ MLE.

- MSE Loss (LinReg with Gaussian noise) $\equiv$ MLE.
- Cross-Entropy Loss (LogReg/Classification) $\equiv$ MLE.

4-6 Bayesian Inference & Maximum A Posteriori (MAP)

Incorporating Prior Beliefs

Frequentist (MLE):

- θ is fixed, unknown.
- Data is random.

Bayesian (MAP):

- θ is a random variable with a distribution.
- Data is observed, fixed.

Bayes' Theorem for Parameters

$$\text{Posterior } P(\theta|\mathbf{x}) = \frac{\text{Likelihood } P(\mathbf{x}|\theta) \times \text{Prior } P(\theta)}{\text{Evidence } P(\mathbf{x})}$$

- **Prior** $P(\theta)$: Beliefs about θ *before* data.
- **Posterior** $P(\theta|\mathbf{x})$: Updated beliefs about θ *after* data.
- **MAP Estimation**: Point estimate maximizing posterior.

$$\begin{aligned}\hat{\theta}_{MAP} &= \arg\max_{\theta} P(\theta|\mathbf{x}) = \arg\max_{\theta} [P(\mathbf{x}|\theta)P(\theta)] \\ &= \arg\max_{\theta} [\ell(\theta|\mathbf{x}) + \log P(\theta)]\end{aligned}$$

4-7 MAP, MLE, and Regularization

- **MAP vs MLE**:
  - If prior $P(\theta)$ is uniform (flat), $\log P(\theta)$ is constant $\implies \hat{\theta}_{MAP} = \hat{\theta}_{MLE}$.
  - As data $n \to \infty$, likelihood dominates prior $\implies \hat{\theta}_{MAP} \to \hat{\theta}_{MLE}$.
- **MAP and Regularization in ML**:
  - Minimizing (NLL + Regularizer) $\equiv$ MAP estimation.
  - **L2 Regularization (Ridge/Weight Decay)**:

    $$\mathrm{NLL}(\mathbf{w}|\mathbf{x}) + \lambda||\mathbf{w}||_2^2$$

    Equivalent to MAP with Gaussian prior $N(0, \sigma_p^2)$ on weights $\mathbf{w}$. $(\lambda \propto 1/\sigma_p^2)$
  - **L1 Regularization (Lasso)**:

    $$\mathrm{NLL}(\mathbf{w}|\mathbf{x}) + \lambda||\mathbf{w}||_1$$

    Equivalent to MAP with Laplacian prior on weights $\mathbf{w}$. Promotes sparsity.
- Conceptual PyTorch MAP: Add `log_prior` term to loss: `loss = -(log_likelihood + log_prior)`.

4-8 Hypothesis Testing & Confidence Intervals (Briefly)

- **Hypothesis Testing**: Formal procedure to decide between two competing statements (H_0: null, H_A: alternative) about a population based on sample data.
    - Calculate test statistic $\rightarrow$ p-value.
    - **p-value**: P(observed data or more extreme $|H_0$ is true).
    - If p-value ¡ significance level α (e.g., 0.05), reject H_0.
    - ML: Model comparison, feature significance.
- **Confidence Intervals (CI)**: Range of values likely to contain true parameter value with certain confidence (e.g., 95% CI).
    - If repeated sampling, 95% of CIs would contain true parameter.
    - Measures uncertainty around point estimate.
- *Crucial for rigorous model evaluation, A/B testing, scientific reporting in ML.*

4-9 Bias-Variance Tradeoff (Statistical Perspective)

Decomposing Expected Prediction Error

Assume $Y = f_{true}(\mathbf{X}) + \epsilon$, with $E[\epsilon] = 0, Var(\epsilon) = \sigma_\epsilon^2$. Our learned model is $\hat{f}(\mathbf{X})$. Expected Prediction Error (EPE) at $\mathbf{x}_0$ for squared error loss:

$$EPE(\mathbf{x}_0) = \underbrace{(E_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(\mathbf{x}_0)] - f_{true}(\mathbf{x}_0))^2}_{\text{Bias}^2(\hat{f}(\mathbf{x}_0))} + \underbrace{E_{\mathcal{D}}[(\hat{f}_{\mathcal{D}}(\mathbf{x}_0) - E_{\mathcal{D}}[\hat{f}_{\mathcal{D}}(\mathbf{x}_0)])^2]}_{\text{Variance}(\hat{f}(\mathbf{x}_0))} + \sigma_\epsilon^2$$

The final term is the Irreducible Error σ_ϵ^2.

- **Bias**: Systematic error. Average difference between model's prediction and true function. (Underfitting: high bias).
- **Variance**: Model's sensitivity to specific training data. Variability of predictions if retrained on different datasets. (Overfitting: high variance).
- **Irreducible Error**: Noise inherent in data. Cannot be reduced.

4-9 Bias-Variance Tradeoff (Statistical Perspective)

The Tradeoff

- Increasing model complexity $\implies \downarrow$ Bias, $\uparrow$ Variance.
- Decreasing model complexity $\implies \uparrow$ Bias, $\downarrow$ Variance.
- Goal: Find balance to minimize total error. (Cross-validation, regularization, ensembles).

Questions?

Thank You!