模式识别与机器学习

Lecture Notes

笔记一

机器学习基础与数学工具

目录

1 机器学习基本定义与损失函数

1.1 什么是机器学习？

机器学习的核心目标，是从数据中学习到输入到输出的映射规律，从而实现对未知数据的预测与决策。

1.1.1 核心定义

机器学习的本质是寻找最优预测函数的过程：

- 预测函数 f 接收输入 $\boldsymbol{x}$，并产生对应的输出 y，即 $y = f(\boldsymbol{x})$。
- 我们的目标是在所有可能的函数中，找到最贴合数据规律的最优映射关系。

1.1.2 函数族与参数空间

在实际建模中，我们预先定义一个**函数族** F，所有候选函数都属于这个集合：

$$F = \{f \mid Y = f(X)\}$$

在绝大多数机器学习场景中，函数族由参数向量决定，因此可以进一步表示为**参数化函数族**：

$$F = \{f \mid Y = f_\theta(X), \theta \in \mathbb{R}^n\}$$

其中：

- $\theta = (\theta_1, \theta_2, \ldots, \theta_n)^T$ 是 n 维参数向量；
- $\mathbb{R}^n$ 是 n 维欧氏空间，也称为**参数空间**，所有参数的可能取值都在这个空间内。

直观示例：线性回归的函数族是所有线性函数 $f_\theta(x) = \theta_0 + \theta_1 x$，参数空间为二维平面 $\mathbb{R}^2$，任务是在该平面中找到最优的 (θ_0, θ_1)。

1.2 如何在参数空间找到最佳的参数

定义好函数族后，核心问题转化为：如何在参数空间 $\mathbb{R}^n$ 中，找到让模型预测效果最优的参数 θ^*。

1.2.1 定义“最佳”：损失函数

“最佳”的直观含义是：模型预测值 $f_\theta(X)$ 与真实标签 Y 越接近越好。为了量化这个“接近程度”，我们引入**损失函数（Loss Function）** $L(f_\theta(X), Y)$，它衡量了模型预测的错误程度：

- 损失函数值越小，代表模型预测越准确；
- 损失函数的设计由任务目标决定（如回归任务用均方误差，分类任务用交叉熵）。

1.2.2 最优参数的数学表达

最优参数 θ^* 是让全体训练数据的总损失最小的参数，其数学表达式为：

$$\theta^* = \arg\min_\theta \sum_{i=1}^{m} L(f_\theta(x_i), y_i)$$

其中：

- m 是训练样本的数量；
- (x_i, y_i) 是第 i 个训练样本；
- $\arg\min_\theta$ 表示“找到使后面表达式取最小值的参数 θ”。

1.2.3 经典实例：均方误差（MSE）

对于回归任务，最常用的损失函数是**均方误差（Mean Squared Error, MSE）**，此时最优参数的求解公式为：

$$\theta^* = \arg\min_\theta \sum_{i=1}^{m} \left(f_\theta(x_i) - y_i\right)^2$$

该公式的直观含义是：找到一组参数 θ，让所有样本的预测值与真实值的平方差之和最小，从而让模型整体预测误差最小。

2 信息熵、交叉熵与 KL 散度

2.1 引言

在模式识别与机器学习中，我们经常要回答一个问题：模型学到了什么？它对自己的预测有多确定？而信息论为我们提供了量化不确定性的工具：熵、交叉熵与 KL 散度。理解这些概念，可以帮助我们更好理解模型行为、改进模型设计。

2.2 自信息 (Self-information)

2.2.1 直观理解

自信息衡量的是**一个具体事件发生所带来的信息量**。一条信息的信息量大小和它的不确定性有很大的关系，其核心思想是：越罕见的事件，发生后带来的震撼（信息量）越大。比如“太阳从东边升起”是必然事件，这条信息的信息量就很小。而“考倒数第一的同学突然考了第一名”是小概率事件，一旦发生，所带来的信息量是巨大的。

2.2.2 数学定义

对于概率为 $p(x)$ 的事件 x，其自信息为：

$$I(x) = -\log p(x)$$

- 概率越小，$-\log p(x)$ 越大。
- 底数通常取 2，单位为 bit（比特）；若以 e 为底，单位为 nat（奈特）；若以 10 为底，单位为 dit（迪特）或 hartley（哈特莱）。

2.3 信息熵 (Shannon Entropy)

2.3.1 直观理解

如果说自信息是单个事件的信息量，那么信息熵就是**所有可能事件的平均信息量**，它衡量的是一个随机变量**整体的不确定性**。熵越高，系统越混乱、越难预测；熵越低，系统越确定。

2.3.2 数学定义

对于离散随机变量 X 及其分布 $p(x)$：

$$H(X) = \mathbb{E}_{x\sim P}[I(x)] = -\sum_{x} p(x)\log p(x)$$

2.3.3 示例

- **抛一枚公平硬币：**正反概率各 0.5，熵 $H = -0.5\log_2 0.5 - 0.5\log_2 0.5 = \log_2 2 = 1$ bit，不确定性高。
- **抛一枚作弊硬币：**正面概率 0.9，反面 0.1，熵 $H = -0.9\log_2 0.9 - 0.1\log_2 0.1 \approx 0.47$ bit，不确定性低。
- **均匀分布：**当所有事件等概率时，熵最大，最混乱。
- **确定性事件：**熵为 0，某个结果必然发生。

在机器学习中，我们常希望模型输出的熵保持适中：熵太高说明模型过于犹豫、无法做出判断；熵太低则说明模型过度自信、容易犯错。

2.4 交叉熵 (Cross-Entropy)

2.4.1 直观理解

在许多实际问题中，我们通常有一个真实的未知分布 p 和模型预测分布 q。交叉熵正是**衡量这两个分布之间差异**的一种方式。

交叉熵也可以衡量我们基于某种主观认识 q 去感受客观世界 p 时，产生的平均“惊奇度”。若两者严重不匹配，平均惊奇度就高，交叉熵就大；当 $q = p$ 时，产生的惊奇度最低，此时交叉熵等于信息熵。

2.4.2 数学定义

对于两个定义在相同样本空间上的离散概率分布 p（真实分布）和 q（近似分布），它们之间的交叉熵定义为：

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

如果存在某个 x 使得 $p(x) > 0$ 但 $q(x) = 0$，则交叉熵 $H(p, q)$ 会是无穷大，因此实践中常通过平滑技术避免模型预测概率为零。

在深度学习中，更常用以下公式计算交叉熵损失：

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \log \hat{y}_t^{(i)}$$

其中 m 为样本数，t 为该样本的真实标签对应的类别索引，$\hat{y}_t^{(i)}$ 表示第 i 个样本在真实类别 t 上的模型预测概率。该损失只取决于模型对正确类别预测概率的对数值，模型表现越好其值越小，常用于分类问题。

2.4.3 示例

二分类问题中，真实标签 $p = [1, 0]$：

- 模型预测 $q_1 = [0.9, 0.1]$：$H(p, q_1) = -1 \times \log 0.9 - 0 \times \log 0.1 \approx 0.105$，拟合效果好。
- 模型预测 $q_2 = [0.6, 0.4]$：$H(p, q_2) = -1 \times \log 0.6 - 0 \times \log 0.4 \approx 0.511$，拟合效果差。

2.5 KL 散度 (Kullback-Leibler Divergence)

2.5.1 直观理解

KL 散度也称为相对熵 (Relative Entropy)，是信息熵与交叉熵的差值，用于衡量两个分布 p 和 q 之间的差异，可以理解为“用 q 近似 p 时**额外付出的信息量**”。

2.5.2 数学定义

对于离散随机变量，分布 p 和 q 的 KL 散度的定义如下：

$$\begin{aligned} D_{\mathrm{KL}}(p \parallel q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\ &= \sum_x p(x) \left[\log p(x) - \log q(x)\right] \\ &= -\sum_x p(x) \log q(x) + \sum_x p(x) \log p(x) \\ &= H(p,q) - H(p). \end{aligned}$$

其中 $H(p,q)$ 为交叉熵，$H(p)$ 为信息熵。当 $p(x) = 0$ 时，约定 $p(x) \log \frac{p(x)}{q(x)} = 0$。

2.5.3 示例

用 q 近似 p（$p = [1, 0]$）：

- 模型预测 $q_1 = [0.9, 0.1]$：$D_{\mathrm{KL}}(p \parallel q_1) = 1 \times \log \frac{1}{0.9} + 0 \times \log \frac{0}{0.1} \approx 0.105$，损失小。
- 模型预测 $q_2 = [0.6, 0.4]$：$D_{\mathrm{KL}}(p \parallel q_2) = 1 \times \log \frac{1}{0.6} + 0 \times \log \frac{0}{0.4} \approx 0.511$，损失大。

2.6 总结归纳

通过定义直接变形，可得到三者最核心的关系：

$$\underbrace{D_{\mathrm{KL}}(p \parallel q)}_{\text{相对熵}} = \underbrace{H(p,q)}_{\text{交叉熵}} - \underbrace{H(p)}_{\text{信息熵}}$$

- 信息熵：描述**单个概率分布**本身的不确定性。
- 交叉熵：衡量**两个分布之间的拟合误差**。
- KL 散度：衡量**两个分布之间的真实差异**。

通过最小化相对熵，我们可以让主观模型分布 q 不断逼近客观真实分布 p；当相对熵为 0 时，交叉熵等于信息熵，模型与真实世界完全一致。当真实分布 p 固定时，$H(p)$ 是常数，此时最小化交叉熵等价于最小化 KL 散度。这正是分类任务中优先使用交叉熵作为损失函数的原因：计算更简单，且优化效果与 KL 散度等价。

在强化学习中，智能体的行动会动态改变环境状态，因此常将 KL 散度加入目标函数，以此限制策略更新幅度、避免训练过早收敛，从而保证策略具备足够的探索能力。

3 NumPy 核心原理与应用

3.1 NumPy 是什么？

3.1.1 核心定义与生态定位

NumPy（全称 **Numeric Python**）是 Python 语言的一款开源数值计算扩展库，是 Python 科学计算与机器学习生态的底层基石。它为 Python 提供了高性能的多维数组对象、数值运算工具与数学函数体系，几乎所有 Python 机器学习相关库（包括 Scikit-learn、Pandas、PyTorch、TensorFlow、Matplotlib 等）均构建于 NumPy 的数组结构与运算能力之上。

NumPy 的官方文档地址为：https://docs.scipy.org/doc/numpy/reference/，是学习与使用 NumPy 的权威参考资料。

3.1.2 诞生背景：Python 原生数据结构的缺陷

Python 原生提供了列表（list）结构，可以模拟数组的功能，但在数值计算场景中存在致命缺陷，这也是 NumPy 诞生的核心动因：

1. **数据类型无约束**：Python 列表中的元素可以是任意类型的对象，每个元素都需要存储类型指针、引用计数等额外信息，造成了极大的内存浪费，也无法保证数值计算的类型一致性。

2. **运算效率极低**：Python 是解释型语言，原生循环需要逐行进行类型检查、解释执行，无法利用 CPU 的并行计算能力，在处理大规模数值数据时，运算速度会出现数量级的下降。

3. **缺乏数值运算原生支持**：Python 列表不支持矩阵乘法、维度变换、线性代数运算等数值计算核心操作，手动实现不仅代码冗余，还极易出现错误。

3.1.3 NumPy 的两大核心对象

NumPy 的核心能力围绕两大基础对象构建，二者共同构成了 NumPy 数值计算体系的核心：

ndarray：n 维数组对象 ndarray 全称 **n-dimensional array object**，是 NumPy 的核心数据结构，用于存储单一数据类型的多维数组，是所有数值运算的载体。

- **核心特性**：数组内所有元素数据类型一致，内存连续存储，支持任意维度的扩展，具备固定的形状（shape）属性。

- **维度与形状定义**：

$$\text{ndarray.shape} = (d_1, d_2, ..., d_n)$$

  其中 d_i 代表第 i 个维度的元素个数，例如：

  - 一维数组：shape=(4,)，对应 4 个元素的向量；
  - 二维数组：shape=(3,2)，对应 3 行 2 列的矩阵；
  - 三维数组：shape=(4,3,2)，对应 4 个 3 行 2 列的矩阵构成的张量。

ufunc：通用函数对象 ufunc 全称 **universal function object**，是能够对 ndarray 进行逐元素运算的函数，解决了 Python 循环效率低下的问题。

- 所有 ufunc 均底层由 C/C++ 实现，避免了 Python 循环的解释开销，支持向量化运算；
- 涵盖了算术运算、三角函数、比较运算、位运算等基础操作，同时支持广播机制，适配不同形状数组的运算需求；
- 是 NumPy 实现高性能数值计算的核心载体。

广播机制示例：NumPy 允许不同形状的数组进行运算，自动进行维度扩展和对齐。

Listing 1: 广播机制示例

```
import numpy as np
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([10, 20, 30]) # 形状为 (3,)
print(a + b)  # 自动广播对齐维度
```

3.1.4 NumPy 的核心功能体系

NumPy 围绕多维数组与通用函数，构建了完整的数值计算功能体系，核心能力包括：

1. **多维数组创建与操作：**支持任意维度数组的创建、切片、索引、维度变换、拼接与拆分，灵活适配不同数据结构需求；
2. **基础数值函数运算：**提供逐元素的算术、逻辑、三角函数、指数对数等函数运算，无需循环即可完成全数组计算；
3. **线性代数运算：**内置矩阵乘法、矩阵求逆、特征值分解、奇异值分解（SVD）、行列式计算、线性方程组求解等核心线性代数功能，是机器学习算法的数学基础；
4. **随机数生成：**提供服从均匀分布、正态分布、二项分布等多种概率分布的随机数生成器，支持模型参数初始化、数据采样等机器学习核心场景；

5. **傅里叶变换与信号处理：**内置快速傅里叶变换（FFT）等工具，支持时序数据、信号数据的特征提取与处理；

3.2 机器学习任务中为什么要使用 NumPy？

机器学习的核心是从数据中学习输入到输出的映射规律，本质是大规模的矩阵运算、数值优化与统计计算。NumPy 的所有特性都与机器学习的核心需求高度契合，是 Python 机器学习任务的必备工具。

3.2.1 极致的运算性能：适配机器学习大规模数据计算需求

机器学习任务通常需要处理数万甚至数百万样本的高维特征数据，对运算效率有极高的要求，而 NumPy 的运算性能相比 Python 原生代码有数量级的提升，核心优势来自于：

1. **底层编译型语言实现：**NumPy 的核心运算逻辑均由 C/C++ 编写，编译后直接执行机器码，避免了 Python 解释型语言的逐行解释开销，运算速度可提升 10 - 1000 倍。

2. **向量化运算机制：**NumPy 将” 循环” 交给底层高效执行，无需编写 Python 显式循环即可对整个数组进行批量运算，并支持 CPU 的 SIMD（单指令多数据流）并行计算，充分利用多核 CPU 的运算能力。

以机器学习中最基础的特征与权重点积运算为例：

Listing 2: 向量化运算示例

```
# Python原生循环实现（极慢）
def python_dot(x, w):
    n = len(x)
    y = 0
    for i in range(n):
        y += x[i] * w[i]
    return y
```

```
# NumPy向量化实现（极快）
import numpy as np
def numpy_dot(x, w):
    return np.dot(x, w)
```

3.2.2 高效的内存管理：降低大规模数据集的内存开销

机器学习任务中的数据集通常规模庞大，高维特征矩阵会占用大量内存，而 NumPy 的内存管理机制相比 Python 列表有极大的优化：

1. **连续内存存储**：ndarray 的所有元素在内存中连续存储，而 Python 列表的元素是分散的对象指针，连续存储可以极大提升 CPU 缓存的命中率，减少内存寻址开销。

2. **固定数据类型，无冗余开销**：ndarray 要求所有元素为同一数据类型，仅需存储一次类型信息，无需为每个元素单独存储类型指针、引用计数等冗余信息。以存储 100 万个整数为例：

   - Python 列表：每个 int 对象约占用 28 字节，总内存开销约 28MB；
   - NumPy 数组（int32 类型）：每个元素仅占用 4 字节，总内存开销约 4MB，内存占用仅为 Python 列表的约 1/7。

3. **视图机制避免内存拷贝**：NumPy 的切片操作默认返回原数组的视图（View）而非副本，无需额外分配内存即可完成数组的子区域访问，大幅降低了大规模数据处理时的内存峰值。

3.2.3 完备的线性代数支持：机器学习算法的数学基石

无论是传统机器学习的线性回归、逻辑回归、主成分分析（PCA），还是深度学习的神经网络前向传播与反向传播，其底层数学本质都是线性代数运算，而 NumPy 提供了完备、高效的线性代数工具集。

以本课程核心的线性回归算法为例，其正规方程求解完全依赖 NumPy 的线性代数能力：

$$w^* = (X^\top X)^{-1} X^\top y \tag{1}$$

其中 X 为样本设计矩阵，y 为真实标签向量，w^* 为最优参数向量。

Listing 3: 线性回归求解的稳定写法

```
# 不推荐：直接求逆（数值不稳定，矩阵接近奇异时精度差、易报错）
# w_opt = np.linalg.inv(X.T @ X) @ X.T @ y

# 推荐1：最小二乘（最稳定）
w_opt = np.linalg.lstsq(X, y, rcond=None)[0]

# 推荐2：直接解方程（更快）
w_opt = np.linalg.solve(X.T @ X, X.T @ w)
```

除此之外，梯度下降算法的迭代更新、损失函数的批量计算、特征值分解与降维、聚类算法的距离矩阵计算等几乎所有机器学习核心算法，都可以通过 NumPy 的线性代数工具简洁、高效地实现。

3.2.4 向量化编程范式：简化机器学习算法实现

机器学习算法中存在大量的批量样本运算，若使用 Python 原生循环实现，不仅代码冗长、可读性差，还极易出现索引错误。NumPy 的向量化编程范式，将批量运算抽象为数组操作，带来了极大的工程优势：

1. **代码极简，可读性强**：向量化代码将循环逻辑隐藏到底层，用数学表达式直接对应代码实现，与机器学习的公式推导一一对应，降低了算法实现的理解门槛。

2. **减少人为错误**：Python 循环需要手动处理样本索引、维度匹配等问题，在高维数组运算中极易出现边界错误、维度不匹配等问题；而 NumPy 的向量化操作自动处理全数组元素，同时内置严格的维度检查，大幅降低了算法实现的出错概率。

3. **统一的编程范式**：从传统机器学习的线性模型、树模型，到深度学习的张量运算，

均基于向量化编程范式。掌握 NumPy 的向量化思维，是理解与实现各类机器学习算法的通用基础。

例如批量梯度下降的参数更新公式：

$$w := w + \alpha \frac{2}{n} X^\top (y - Xw) \tag{2}$$

其 NumPy 实现与公式完全一致，无需任何循环，代码简洁且不易出错。

3.2.5 全链路生态兼容：机器学习全流程工具链的通用底层

机器学习项目是一个完整的流水线，涵盖数据加载、数据预处理、特征工程、模型训练、模型评估、可视化全流程，而 NumPy 是 Python 机器学习全生态的通用数据格式标准，实现了全流程工具链的无缝衔接：

1. **数据处理层**：Pandas 的 DataFrame 底层基于 NumPy 数组实现，数据清洗、特征工程的结果可直接转换为 NumPy 数组输入模型；
2. **模型训练层**：Scikit-learn 等传统机器学习库的所有模型，均原生支持 NumPy 数组作为输入，模型的参数、预测结果也均以 NumPy 数组格式返回；
3. **深度学习框架**：PyTorch、TensorFlow 等深度学习框架的张量（Tensor）与 NumPy 数组可无缝转换，模型的预处理、后处理逻辑均可通过 NumPy 实现；
4. **可视化与评估层**：Matplotlib、Seaborn 等可视化库原生支持 NumPy 数组，模型评估指标（准确率、RMSE、F1 值等）的计算也均基于 NumPy 实现。

使用 NumPy 作为核心数据结构，可实现机器学习全流程的无缝打通，无需进行复杂的数据格式转换，大幅提升了项目开发效率。

3.2.6 稳定的数值计算能力：保障模型训练的数值稳定性

机器学习模型的训练过程，尤其是深度学习的多轮迭代优化，对数值计算的稳定性有极高的要求。微小的数值误差经过多轮迭代后，可能会被无限放大，导致模型梯度爆

炸、不收敛等问题。NumPy 经过数十年的工业界验证，具备极强的数值稳定性：

1. **严格的浮点数计算规范**：NumPy 严格遵循 IEEE 754 浮点数计算标准，内置了对数值下溢、上溢、除零等异常情况的处理机制，避免数值计算过程中出现非数值（NaN）、无穷大（Inf）等问题；
2. **高精度的数值算法实现**：NumPy 的线性代数、数值优化等底层算法，均基于业界成熟的 LAPACK、BLAS 数学库实现，经过了数十年的工业界验证，保证了复杂数值计算的精度与稳定性；
3. **可复现的随机数体系**：NumPy 的随机数生成器支持固定随机种子，可保证模型参数初始化、数据随机划分等操作的可复现性，这是机器学习实验可验证、可复现的核心基础。

3.3 总结

NumPy 不是机器学习算法本身，但它是 Python 机器学习生态的” 数字地基”。从最基础的线性回归、梯度下降，到复杂的大模型训练与微调，机器学习的每一个环节都离不开数值计算，而 NumPy 为这些计算提供了高性能、高稳定、高兼容的底层支持。

理解 NumPy 的核心原理与使用方法，不仅是为了掌握一个工具，更是为了深入理解机器学习算法的底层数值逻辑，将课堂上学习的损失函数、参数优化、线性代数公式，转化为可运行、可验证的代码实现。

4 SIMD 原理与 GPU 并行计算的对比分析

摘要

本文档详细介绍了 SIMD（单指令多数据流）加速计算的硬件原理、核心限制，并与 GPU 的 SIMT（单指令多线程）架构进行对比分析，最后总结了机器学习中高度依赖 SIMD 加速的关键操作。

4.1 SIMD 加速计算的原理

4.1.1 核心思想：数据级并行

SIMD 的核心思想是：**一条指令同时处理多个数据**，而非传统标量处理器那样一条指令只处理一个数据。

传统标量方式	SIMD 向量化方式
循环 4 次	1 条指令
4 条 load 指令	1 条向量 load
4 条 mul 指令	1 条向量 mul
4 条 store 指令	1 条向量 store

表 1: 标量与 SIMD 的指令数量对比（4 个浮点数运算）

4.1.2 硬件基础：向量寄存器

向量寄存器是 SIMD 的物理基础，不同架构支持的位宽不同：

- **128-bit**：SSE 指令集（Intel/AMD）
- **256-bit**：AVX2 指令集（主流 CPU）
- **512-bit**：AVX-512 指令集（服务器/高性能计算）

存储容量示例：

- 256-bit 寄存器 = 8 个 float32 / 4 个 float64 / 32 个 int8
- 512-bit 寄存器 = 16 个 float32 / 8 个 float64

4.1.3 加速比分析

理想情况下，SIMD 的理论加速比等于向量宽度：

$$\text{加速比} = \frac{\text{向量寄存器位宽}}{\text{单个数据位宽}} \tag{3}$$

以 AVX2（256-bit）处理 float32 为例：

$$\text{加速比} = \frac{256}{32} = 8 \tag{4}$$

实际应用中，受内存访问和数据对齐影响，通常可达到 5-7 倍加速。

4.2 SIMD 的主要限制

1. **数据对齐要求严格**

   所有数据必须类型一致、大小相同、内存连续且对齐。未对齐访问会导致性能显著下降甚至错误。

2. **分支处理效率低**

   遇到 `if/else` 分支时，SIMD 需要串行执行不同路径，部分计算单元被屏蔽，效率明显下降。

3. **编译器自动向量化不可靠**

   GCC、Clang 等对循环的向量化策略存在差异，常需使用 `#pragma omp simd` 等指令手动干预。

4. **指令集不统一**

   x86（SSE/AVX）、ARM（NEON/SVE）、RISC-V（RVV）指令集互不兼容，跨平台移植成本高。

5. **不适合不规则计算**

   稀疏矩阵、图算法、树结构等访问模式不规则的任务，SIMD 难以有效加速。

4.3 SIMD 与 GPU 并行计算（SIMT）的区别

GPU 采用的 **SIMT**（Single Instruction Multiple Threads）可以看作是 SIMD 的推广。

4.3.1 详细对比

维度	SIMD（CPU）	SIMT（GPU）
执行模型	一个线程处理多个数据元素	多个线程各处理一个数据（32 线程/Warp）
硬件结构	单个 ALU + 宽数据通路	多核 + 每核独立寄存器文件
分支处理	串行执行不同路径，效率下降严重	硬件 mask 串行化，对程序员透明
数据寻址	必须连续 + 严格对齐	每个线程可独立寻址
寄存器资源	少量（16-32 个通用寄存器）	大量（每个线程独立寄存器）
典型场景	CPU 上的向量化数学运算	大规模稠密并行计算
延迟容忍	低延迟设计	通过线程切换隐藏延迟

表 2: SIMD vs SIMT 详细对比

4.3.2 关键理解：SIMT 不是取代 SIMD

- GPU 硬件底层仍然使用 SIMD 执行单元
- SIMT 在 SIMD 之上增加了**线程抽象层**
- Warp（32 线程）通过硬件调度隐藏了向量宽度和分支复杂度
- 程序员写 GPU 代码时通常不需要关心 SIMD 宽度

4.4 机器学习中依赖 SIMD 的操作

SIMD 加速在现代 ML 框架（PyTorch、TensorFlow、NumPy、JAX）中无处不在。

核心算子

1. **矩阵乘法（GEMM）**

   深度学习最核心的算子，SIMD 可同时计算多个输出元素，配合 cache 优化可获得 10× 以上加速。

2. **卷积运算**

   现代 CNN 中的卷积通常通过 im2col + GEMM 实现，本质依赖 SIMD 加速。

3. **逐元素运算**

   ReLU、Sigmoid、Tanh、GELU、BatchNorm、Dropout、张量加减乘除等。

4. **规约操作**

   求和、最大值、最小值、均值计算，在 bfloat16 上使用 AVX2/AVX-512 可获得 3-4× 加速。

5. **Embedding 查找与注意力机制**

   大规模向量相似度计算（余弦相似度、点积注意力），SIMD 加速可达 8×。

6. **梯度计算**

   反向传播中的逐元素梯度（如 ReLU、Softmax 梯度）。

操作	标量速度	SIMD 加速后
向量点积（长度 256）	1×	6-8×
矩阵乘（64×64）	1×	8-10×
ReLU（1M elements）	1×	5-7×

表 3: SIMD 典型加速效果

实际性能数据

4.5 小结

- **原理：**SIMD 通过单指令多数据实现数据级并行，加速比理论上等于向量宽度
- **限制：**数据对齐要求高、分支效率低、编译器不可靠、指令集不统一、不适合不规则计算
- **与 GPU 对比：**SIMT 在 SIMD 硬件上增加了线程抽象，更灵活、对程序员更友好
- **ML 依赖：**矩阵乘法、卷积、激活函数、规约操作、注意力机制高度依赖 SIMD 加速

5 Pandas 数据处理与分析

5.1 Pandas 概述

在深度学习任务中，现实世界的原始数据（如表格、CSV、数据库数据）无法直接转换为张量（Tensor）用于模型训练，必须先进行数据预处理。

Pandas (Python Data Analysis Library) 是 Python 的一个开源数据分析与操作库，提供了高效的数据结构和数据分析工具，在深度学习任务中，它是连接原始异构数据与张量（Tensor）之间的核心桥梁。Pandas 是基于 NumPy 构建的数据处理与分析工具库，专为表格型、异构型数据设计，是机器学习与深度学习数据 pipeline 中不可或缺的核心工具。

就像向量是标量的推广，矩阵是向量的推广一样，在处理数据时可以构建具有更多轴的数据结构，张量是描述具有任意数量轴的 n 维数组的通用方法。例如，向量是一阶张量，矩阵是二阶张量。比如处理图像时，张量将变得非常重要，图像以 n 维数组形式出现，其中 3 个轴对应于高度、宽度，以及一个通道（channel）轴，用于表示颜色通道（红色、绿色、蓝色）。

5.1.1 异构性与灵活性

- **NumPy**：仅支持同质数值数组，`ndarray` 内所有元素必须为同一数据类型。
- **Pandas**：完美支持异构数据存储，`DataFrame` 可同时包含整数、浮点数、字符串、时间、类别等多种类型，与真实业务数据格式高度匹配。

5.1.2 数据清洗与缺失值处理

原始数据普遍存在缺失值（NaN）、异常值、格式不统一等问题：

- NumPy 缺失值处理能力弱，操作底层且繁琐。
- Pandas 提供 `dropna()`、`fillna()`、`interpolate()` 等高级方法，可快速完成脏数

据清洗，是数据转张量的必要步骤。

5.1.3 标签化与直观性

- NumPy 仅支持位置索引，代码可读性差，易出错。
- Pandas 使用自定义索引与列名，可通过字段名称直接访问数据，大幅提升代码可读性与工程稳定性。

5.2 Series 和 DataFrame 的数据结构

Pandas 有两大核心数据结构：一维的 `Series` 和二维的 `DataFrame`。

5.2.1 Series (一维标记数组)

`Series` 是带索引的一维数组，可以看作是由一组数据和一组索引（标签）组成的结构，类似于带名字的一维 NumPy 数组，能够存储整数、浮点数、字符串等任意类型数据，每一个值都可以通过索引标签快速访问，是 Pandas 中最基础的数据单元。

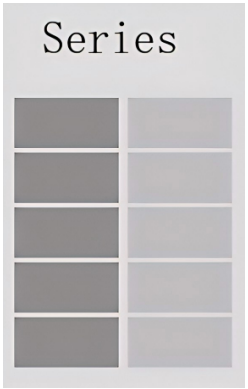

图 1: Pandas Series 结构示意图（1D）

Listing 4: 创建 Series 示例

```
import pandas as pd
import numpy as np

data = np.arange(3)
s = pd.Series(data)
```

```
# 查看索引与值
print(s.index)
print(s.values)

# 自定义索引创建 Series
s2 = pd.Series(data, index=['a', 'b', 'c'])
```

```
RangeIndex(start=0, stop=3, step=1)
[0 1 2]
```

5.2.2 DataFrame (二维标记数据结构)

DataFrame 是表格型数据结构，由多个列（可异构）和行索引构成，可以理解为多个 Series 按列组合而成。它拥有行索引和列名，结构与 Excel 表格、数据库表非常相似，每一列可以是不同的数据类型。DataFrame 支持对行、列进行灵活的增删改查，能够方便地完成数据筛选、分组、聚合、缺失值处理等操作，是机器学习数据预处理中最常用的数据载体。

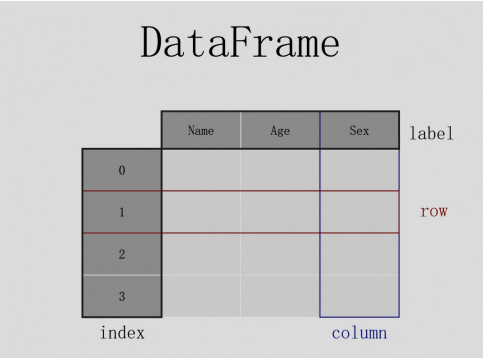

图 2: Pandas DataFrame 结构示意图（2D）

Listing 5: 构建异构 DataFrame

```
df2 = pd.DataFrame({
    "A": 1.0,
    "B": pd.Timestamp("20130102"),
    "C": pd.Series(1, index=list(range(4)), dtype="float32"),
    "D": np.array([3] * 4, dtype="int32"),
    "E": pd.Categorical(["test", "train", "test", "train"]),
    "F": "foo",
})
```

```
# 查看数据结构与类型
print(df2)
print(df2.dtypes)
```

注意：DataFrame 天然支持异构数据，每列可独立设置数据类型，适配真实数据集。

```
  A          B         C    D      E      F
  0  1.0  2013-01-02   1.0   3   test   foo
  1  1.0  2013-01-02   1.0   3  train   foo
  2  1.0  2013-01-02   1.0   3   test   foo
  3  1.0  2013-01-02   1.0   3  train   foo

  A           float64
  B    datetime64[s]
  C           float32
  D             int32
  E          category
  F            object
  dtype: object
```

表 4: Series 与 DataFrame 的主要区别

特性	Series	DataFrame
维度	一维 (1D)	二维 (2D)
索引	仅行索引 (index)	行索引 (index) + 列索引 (columns)
数据同构性	列内（唯一一列）必须同构	列内同构，列间可异构
应用场景	单一变量数据	多变量表格数据
操作重点	一维数组运算（排序、统计）	表格级操作（筛选、分组、合并）
转换关系	可作为 DataFrame 的一列/一行	可通过 `df[" 列名"]` 提取 Series

5.3 Pandas 在机器学习中的作用

在深度学习任务中，现实世界的原始数据（如表格、CSV、数据库数据）无法直接转换为张量（Tensor）用于模型训练，必须先进行数据预处理，Pandas 是这一流程的核心工具。

5.3.1 核心数据操作：数据选择与索引

Listing 6: 基础数据选取

```
# 构建测试 DataFrame
dates = pd.date_range('20130101', periods=6)
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=['A','B'
    ,'C','D'])
# 1. 选择单列, 返回 Series
df["A"]
df.A
# 2. 行切片选择 (按位置)
df[0:3]
# 3. 高级索引: loc (标签) / iloc (位置)
df.loc[:, ['A', 'B']]
df.iloc[0:3, 0:2]
# 4. 时间序列切片
df["20130102":"20130104"]
```

```
1. 选择单列, 返回 Series
2013-01-01    0.185468
2013-01-02    2.097908
2013-01-03    0.823570
2013-01-04    0.556451
2013-01-05   -1.858341
2013-01-06   -0.717175
Freq: D, Name: A, dtype: float64

2. 行切片选择 (按位置)
A         B         C         D
2013-01-01 0.185468 0.920085 1.223819 -1.840856
2013-01-02 2.097908 -1.434812 0.231906 0.034359
2013-01-03 0.823570 -0.398397 -1.563884 -0.019314

3. 高级索引: loc (标签) / iloc (位置)
A         B
2013-01-01 0.185468 0.920085
2013-01-02 2.097908 -1.434812
2013-01-03 0.823570 -0.398397

4. 时间序列切片
A          B         C         D
2013-01-02 2.097908 -1.434812 0.231906 0.034359
```

```
  2013-01-03 0.823570 -0.398397 -1.563884 -0.019314
  2013-01-04 0.556451 -0.368262 -1.332349 0.323835
```

5.3.2 数据清洗：脏数据处理

原始数据普遍存在缺失值（NaN）、异常值、格式不统一等问题，Pandas 提供了解决方案。

示例：房价数据预处理（来自《动手学深度学习》）

Listing 7: D2L 中房价数据预处理

```
# 1. 模拟创建数据集
import os
os.makedirs('data', exist_ok=True)
data_file = os.path.join('data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\nNA,Pave,127500\n2,NA,106000\n4,NA
        ,178100\nNA,NA,140000\n')

# 读取并划分数据
data = pd.read_csv(data_file)
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
print(f"␣原始读取的数据:\n{data}\n")

# 处理缺失值
inputs = inputs.fillna(inputs.mean(numeric_only=True))
print(f"␣均值填充缺失值后␣(NumRooms␣NaN␣变为␣3.0):\n{inputs}\n")

# 独热编码 (One-hot Encoding)
inputs = pd.get_dummies(inputs, dummy_na=True)
print(f"␣独热编码后:\n{inputs}\n")

# 转换为 PyTorch 张量
import torch
X = torch.tensor(inputs.to_numpy(dtype=float))
y = torch.tensor(outputs.to_numpy(dtype=float))
print(f"␣最终␣PyTorch␣张量␣X:\n{X}")
print(f"␣最终␣PyTorch␣张量␣y:␣{y}")
```

```
  原始读取的数据:
   NumRooms Alley  Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000

  均值填充缺失值后 (NumRooms NaN 变为 3.0):
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN

  独热编码后:
   NumRooms  Alley_Pave  Alley_nan
0       3.0        True      False
1       2.0       False       True
2       4.0       False       True
3       3.0       False       True

  最终 PyTorch 张量 X:
tensor([[3., 1., 0.],
        [2., 0., 1.],
        [4., 0., 1.],
        [3., 0., 1.]], dtype=torch.float64)
  最终 PyTorch 张量 y: tensor([127500., 106000., 178100.,
   140000.], dtype=torch.float64)
```

处理逻辑说明

- **均值填充**：对于数值特征，均值能最大程度保留统计特性而不引入剧烈偏差。
- **独热编码 (One-hot)**：神经网络无法理解“Pave”或“NaN”等字符串。通过 `pd.get_dummies`，我们将一列离散值展开为多列 0 或 1，让模型能进行数学运算。

5.3.3 数据转换：从 DataFrame 到张量

Pandas 处理完成后，可直接转为 NumPy 数组，再转换为张量用于模型训练，完成数据 pipeline 的最后一步：

Listing 8: Pandas 转 Tensor 流程

```
# DataFrame → NumPy 数组
numpy_data = df.values

# NumPy → PyTorch 张量
import torch
tensor_data = torch.tensor(numpy_data, dtype=torch.float32)
```

```
原始的 Pandas DataFrame
当前类型: <class 'pandas.core.frame.DataFrame'>
带有行索引 (0,1,2) 和列名 (Feature_1, Feature_2):
   Feature_1  Feature_2
0        1.5        4.2
1        2.0        5.0
2        1.5        6.8
--------------------------------------------------
转换后的 NumPy 数组
当前类型: <class 'numpy.ndarray'>
行号和列名都被剥离了, 变成了纯粹的二维数组
[[1.5 4.2]
 [2.  5. ]
 [1.5 6.8]]
--------------------------------------------------
PyTorch 张量 (Tensor)
当前类型: <class 'torch.Tensor'>
内部数据精度: torch.float32
这是深度学习模型真正能接收和计算的格式:
tensor([[1.5000, 4.2000],
        [2.0000, 5.0000],
        [1.5000, 6.8000]])
```

5.3.4 在机器学习中的作用

Pandas 专注于数据预处理阶段，是连接原始数据与深度学习模型的核心桥梁，完整流程如下：

1. **数据加载：**读取 CSV、Excel、SQL、JSON 等格式数据。
2. **数据清洗：**缺失值、异常值、重复值处理。
3. **特征工程：**连续变量分箱、类别变量编码、特征聚合。
4. **数据转换：**清洗后的数据转为 NumPy 数组 $\rightarrow$ 张量，输入模型训练。

5.4 总结

Pandas 凭借异构数据支持、强大的清洗能力、标签化索引，成为深度学习数据预处理的标准工具。掌握 Series、DataFrame 数据结构、数据选取、缺失值处理，是完成高质量数据处理、搭建稳定深度学习模型的基础。

6 极大似然估计 (Maximum Likelihood Estimation)

6.1 定义与直观理解

什么是极大似然估计 (MLE)？

核心思想：模型已定，参数未知。根据已经观察到的数据结果，去反推哪一组参数能让这个“结果”发生的概率最大。这就是“极大似然”。

6.2 核心数学公式

假设我们有一组观察到的数据样本 $X = \{x_1, x_2, \ldots, x_n\}$，且这些样本是**独立同分布**的 (i.i.d)。假设生成这些数据的模型参数为 θ。

数据的**似然函数** (Likelihood Function) $L(\theta)$ 定义为所有样本联合出现的概率：

$$L(\theta) = P(X|\theta) = \prod_{i=1}^{n} P(x_i|\theta) \tag{5}$$

为了方便计算（将连乘变为连加，避免计算机浮点数下溢），我们通常对似然函数取自然对数，得到**对数似然函数** (Log-Likelihood)：

$$\ln L(\theta) = \sum_{i=1}^{n} \ln P(x_i|\theta) \tag{6}$$

极大似然估计的目标就是找到那个让 $\ln L(\theta)$ 最大的参数 $\hat{\theta}$：

$$\hat{\theta}_{MLE} = \arg\max_{\theta} \ln L(\theta) \tag{7}$$

6.3 举例说明：带入真实数据的计算

场景：假设我们抛一枚硬币 5 次，观测到的真实数据结果为：**正、正、反、正、正**。设抛出正面的概率为未知参数 θ。我们要用 MLE 求出最合理的 θ 值。这批数据可以表示为 $X = \{1, 1, 0, 1, 1\}$ （1 代表正，0 代表反）。

- **第一步：写出似然函数**（即这 5 次结果同时发生的概率）。因为每次抛硬币是独立的，所以联合概率就是各自概率相乘：

$$L(\theta) = \theta \times \theta \times (1-\theta) \times \theta \times \theta = \theta^4(1-\theta)^1 \tag{8}$$

- **第二步：取对数得到对数似然函数。**

$$\ln L(\theta) = \ln\left(\theta^4(1-\theta)^1\right) = 4\ln(\theta) + 1\ln(1-\theta) \tag{9}$$

- **第三步：对参数求导并令其为 0，求最大值。**

$$\frac{d \ln L(\theta)}{d\theta} = \frac{4}{\theta} - \frac{1}{1-\theta} = 0 \tag{10}$$

解得：$4(1-\theta)=\theta \implies 5\theta=4 \implies \hat{\theta}_{MLE}=0.8$

最终结论：根据这 5 个具体数据，推导得出这枚硬币抛出正面的概率最可能是 **0.8**。完美契合直觉（5 次有 4 次正面）。

6.4 MLE 与损失函数的关系

在机器学习中，**“最大化似然”往往等价于“最小化损失函数”**。这是理解很多经典算法底层逻辑的钥匙。

- **回归问题与均方误差 (MSE)**

  在回归问题（如线性回归）中，我们通常假设预测误差服从**高斯分布（正态分布）**。即真实值 y 围绕预测值 $f(x)$ 波动。将高斯分布的概率密度函数代入 MLE 的对数似然公式中，忽略常数项并加负号翻转，最终的优化目标等价于：

$$\arg\min_{\theta}\sum_{i=1}^{n}(y_i-f(x_i))^2 \tag{11}$$

  结论：假设误差服从高斯分布时，MLE 的目标函数正是**均方误差 (MSE)**。

- **分类问题与交叉熵损失 (Cross-Entropy)**

  在二分类问题（如逻辑回归）中，结果非 0 即 1，服从**伯努利分布**。设模型预测样本为正类的概率为 $\hat{y}$，真实标签为 y。单个样本的似然可以写成：$\hat{y}^y(1-\hat{y})^{(1-y)}$。对其整体取对数并加负号，得到：

$$\text{Loss}=-\sum_{i=1}^{n}[y_i\ln\hat{y}_i+(1-y_i)\ln(1-\hat{y}_i)] \tag{12}$$

  结论：假设数据服从伯努利分布时，MLE 的目标函数正是**交叉熵损失函数**。

6.5 极大似然估计的局限性

虽然 MLE 逻辑严密且应用广泛，但它在实际应用中也有几个明显的局限性：

- **严重依赖数据量，极易过拟合：** MLE 完全信任当前的观测数据。如果你抛了 3 次硬币全是正面，MLE 会武断地得出“抛出正面的概率是 100%”的结论。在数据量极少时，结果极不可靠。
- **无法融入先验知识：**即使根据常识你知道硬币通常是均匀的，只要观测数据有偏差，MLE 就会无视常识。相比之下，**最大后验估计 (MAP)** 通过引入先验概率，可以很好地缓解这个问题。
- **强依赖于模型分布假设：** MLE 的成立前提是你对数据分布的假设（如假设回归误差是高斯分布）必须正确。如果现实中真实的分布与假设不符，基于此推导出的结果会产生严重偏差。