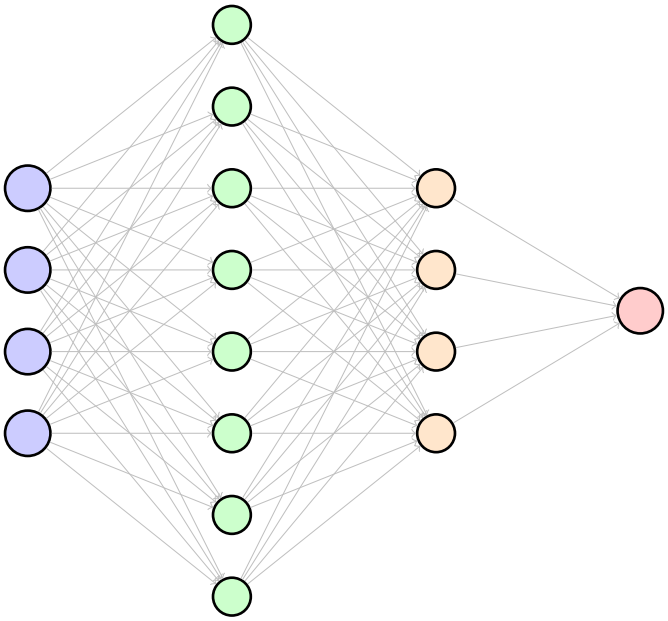
AI/LLM 基础教程

数学基础 · Python 编程 · 机器学习 · 深度学习 · 大语言模型

从基础理论到前沿应用，系统掌握人工智能和大语言模型核心技术



孙豪　中国人民大学

引言

欢迎阅读《AI/LLM 基础教程》!

本教程旨在为读者提供人工智能和大语言模型领域的系统性基础知识，涵盖从数学基础到前沿应用的完整知识体系。本教程包括以下六个核心部分：

- **数学基础**：线性代数、概率论、优化理论、信息论、图论等核心数学工具
- **Python 编程基础**：Python 语法、NumPy 数值计算、Pandas 数据处理
- **机器学习**：监督学习、无监督学习、强化学习、特征工程、模型评估
- **深度学习**：神经网络、CNN、RNN、Transformer、优化技术
- **大语言模型基础**：Transformer 架构、注意力机制、预训练微调、提示工程、RAG
- **大语言模型先进技术**：参数高效微调、推理加速、模型部署、评估方法

教程定位与使用建议：

本教程采用**提纲挈领**的方式组织内容，重点在于构建知识框架、阐明核心概念和基本原理，而非详尽展开每一个技术细节。读者在学习过程中如遇到不理解的概念或需要更深入的细节，建议：

- **查阅相关资料**：利用传统搜索引擎查找相关论文、技术文档和教程
- **使用 AI 工具**：借助现代 AI 搜索引擎和对话系统（如 ChatGPT、Claude 等）进行交互式学习，这些工具能够提供即时的解释和示例
- **实践验证**：通过编写代码、运行实验来验证和理解所学知识

重要说明：

- **技术快速迭代**：大语言模型领域发展迅速，新技术、新方法不断涌现。本教程涵盖当前主流的基础理论和技术，但无法涵盖所有最新进展。读者应保持对前沿研究的关注。
- **持续学习的重要性**：完成本教程的学习后，建议读者：
  - 持续关注最新的研究论文和技术博客
  - 积极参与开源项目，在实践中掌握最新技术
  - 关注行业动态，了解技术在实际应用中的发展

  - 建立知识更新机制，保持与领域发展同步

- **理论与实践并重**：本教程提供了理论基础和代码示例，但真正的掌握需要通过实际项目来巩固和深化。建议读者在学习过程中同步进行编程实践。
- **培养批判性思维**：在学习过程中，应保持批判性思维，深入理解技术的原理、适用场景和局限性，避免盲目使用。

希望本教程能够为您的 AI/LLM 学习之旅打下坚实的基础，并激发您继续深入探索的兴趣！

目录

Part I

第一部分：数学基础

1 引言

数学是人工智能和机器学习的理论基础，为算法设计、模型构建和性能分析提供了严格的数学框架。从线性代数到概率论，从优化理论到信息论，数学工具贯穿于 AI 研究的各个层面。

数学在 AI 中的核心作用：

- **数据表示**：线性代数提供了向量和矩阵等数据结构，用于表示高维数据
- **不确定性建模**：概率论和统计学提供了处理不确定性和噪声的数学工具
- **模型优化**：优化理论提供了寻找最优参数的方法
- **信息度量**：信息论提供了量化信息和不确定性的方法
- **关系建模**：图论提供了表示和处理复杂关系结构的数学框架

本文档系统性地介绍 AI 领域中最常用的数学理论和方法，涵盖线性代数、概率论与统计学、优化理论、信息论和图论等核心内容。每个概念都配有清晰的定义、数学表达式、在 AI/ML 中的应用场景，以及生动的比喻帮助理解。

2 线性代数基础

线性代数是机器学习的数学语言，提供了表示和处理高维数据的工具。从数据表示到模型计算，线性代数无处不在。

2.1 向量空间

定义 2.1 (向量空间). 设 V 是一个非空集合，$\mathbb{F}$ 是一个数域（通常是实数域 $\mathbb{R}$ 或复数域 $\mathbb{C}$）。如果 V 上定义了加法和数乘运算，且满足以下 8 条公理，则称 V 为 $\mathbb{F}$ 上的向量空间：

1. 加法交换律：$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$
2. 加法结合律：$(\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$

3. 存在零向量：$\mathbf{0} + \mathbf{v} = \mathbf{v}$

4. 存在负向量：$\mathbf{v} + (-\mathbf{v}) = \mathbf{0}$

5. 数乘单位元：$1 \cdot \mathbf{v} = \mathbf{v}$

6. 数乘结合律：$(ab)\mathbf{v} = a(b\mathbf{v})$

7. 数乘分配律 1：$a(\mathbf{u} + \mathbf{v}) = a\mathbf{u} + a\mathbf{v}$

8. 数乘分配律 2：$(a + b)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$

通俗解释：向量空间就像一个”数学宇宙”，其中的”点”（向量）可以相加、可以伸缩，但必须遵循特定的规则。就像在三维空间中，我们可以将两个箭头相加，也可以将箭头拉长或缩短。

在 AI/ML 中的应用：

- **特征空间**：在机器学习中，每个样本可以表示为一个向量，所有样本构成一个向量空间（特征空间）
- **词嵌入空间**：在自然语言处理中，词嵌入将词汇映射到高维向量空间，语义相似的词在空间中距离较近
- **图像表示**：图像可以展平为向量，所有可能的图像构成一个巨大的向量空间

例 2.1 (图像特征空间). 一张 28×28 的灰度图像可以表示为一个 784 维向量 $\mathbf{x} \in \mathbb{R}^{784}$。所有可能的图像构成一个 784 维的向量空间。在这个空间中，相似的图像（如都是手写数字”5”）会聚集在一起。

2.2 矩阵运算

矩阵是线性代数中的核心对象，用于表示线性变换和存储数据。

定义 2.2 (矩阵). 一个 $m \times n$ 矩阵 $\mathbf{A}$ 是一个由 m 行 n 列元素排列成的矩形阵列：

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{1}$$

矩阵乘法：对于矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$ 和 $\mathbf{B} \in \mathbb{R}^{n \times p}$，其乘积 $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times p}$ 定义为：

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{2}$$

矩阵转置：矩阵 $\mathbf{A}$ 的转置 $\mathbf{A}^T$ 定义为：

$$(\mathbf{A}^T)_{ij} = a_{ji} \tag{3}$$

矩阵的逆：对于方阵 $\mathbf{A} \in \mathbb{R}^{n\times n}$，如果存在矩阵 $\mathbf{A}^{-1}$ 使得 $\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$（单位矩阵），则称 $\mathbf{A}$ 可逆，$\mathbf{A}^{-1}$ 为其逆矩阵。

在 AI/ML 中的应用：

- **神经网络计算**：前向传播本质上是矩阵乘法，$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$
- **数据变换**：主成分分析（PCA）使用矩阵分解降维
- **图像处理**：卷积操作可以表示为矩阵乘法
- **推荐系统**：用户-物品评分矩阵用于协同过滤

例 2.2 (神经网络中的矩阵乘法). *在多层感知机中，第 l 层的计算可以表示为：*

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{4}$$

其中，$\mathbf{W}^{(l)} \in \mathbb{R}^{n_l\times n_{l-1}}$ 是权重矩阵，$\mathbf{h}^{(l-1)} \in \mathbb{R}^{n_{l-1}}$ 是上一层的输出，σ 是激活函数。矩阵乘法 $\mathbf{W}^{(l)}\mathbf{h}^{(l-1)}$ 计算了所有神经元之间的连接。

2.3 特征值与特征向量

特征值和特征向量揭示了矩阵的本质结构，在降维、主成分分析等领域有重要应用。

定义 2.3 (特征值与特征向量). *对于方阵 $\mathbf{A} \in \mathbb{R}^{n\times n}$，如果存在标量 λ 和非零向量 $\mathbf{v}$ 使得：*

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{5}$$

则称 λ 为 $\mathbf{A}$ 的特征值，$\mathbf{v}$ 为对应的特征向量。

特征值分解：如果矩阵 $\mathbf{A}$ 有 n 个线性无关的特征向量，则可以分解为：

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} \tag{6}$$

其中，$\mathbf{V}$ 的列是特征向量，$\mathbf{\Lambda}$ 是对角矩阵，对角线元素是特征值。

奇异值分解（SVD）：对于任意矩阵 $\mathbf{A} \in \mathbb{R}^{m\times n}$，可以分解为：

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{7}$$

其中，$\mathbf{U} \in \mathbb{R}^{m\times m}$ 和 $\mathbf{V} \in \mathbb{R}^{n\times n}$ 是正交矩阵，$\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ 是对角矩阵（奇异值矩阵）。

通俗解释：特征向量是矩阵作用下的” 不变方向”，特征值表示在这个方向上的” 伸缩倍数”。就像拉伸一个弹性物体，有些方向会被拉伸（特征值大于 1），有些方向会被压缩（特征值小于 1），而特征向量就是这些特殊的方向。

在 AI/ML 中的应用：

- **主成分分析（PCA）**：使用特征值分解找到数据的主要变化方向
- **降维**：保留最大的几个特征值对应的特征向量，实现数据降维
- **推荐系统**：SVD 用于矩阵分解，发现潜在因子
- **图像压缩**：使用 SVD 压缩图像数据
- **自然语言处理**：潜在语义分析（LSA）使用 SVD

例 2.3 (主成分分析). *给定数据矩阵 $\mathbf{X} \in \mathbb{R}^{n \times d}$（$n$ 个样本，d 个特征），PCA 的步骤如下：*

1. *中心化数据：$\tilde{\mathbf{X}} = \mathbf{X} - \bar{\mathbf{X}}$*
2. *计算协方差矩阵：$\mathbf{C} = \frac{1}{n-1}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}}$*
3. *特征值分解：$\mathbf{C} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$*
4. *选择前 k 个最大特征值对应的特征向量，构成投影矩阵 $\mathbf{W} \in \mathbb{R}^{d \times k}$*
5. *降维：$\mathbf{Y} = \tilde{\mathbf{X}}\mathbf{W}$*

主成分就是协方差矩阵的特征向量，特征值表示该方向上的方差大小。

2.4 范数与距离

范数用于度量向量的大小，距离用于度量向量之间的差异。

定义 2.4 (向量范数). *向量 $\mathbf{x} \in \mathbb{R}^n$ 的 p-范数定义为：*

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{8}$$

常用的范数：

- *L_1 范数（曼哈顿距离）：$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$*
- *L_2 范数（欧氏距离）：$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$*

- L_∞ 范数：$\|\mathbf{x}\|_\infty = \max_i |x_i|$

距离度量：两个向量 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ 之间的距离：

- 欧氏距离：$d_2(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_2$
- 曼哈顿距离：$d_1(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|_1$
- 余弦相似度：$\cos(\theta) = \frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|_2\|\mathbf{y}\|_2}$

在 AI/ML 中的应用：

- **正则化**：L_1 正则化（Lasso）和 L_2 正则化（Ridge）用于防止过拟合
- **聚类**：K-means 使用欧氏距离度量样本相似性
- **相似度计算**：余弦相似度用于文本相似度、推荐系统
- **损失函数**：均方误差使用 L_2 范数，平均绝对误差使用 L_1 范数

2.4.1 线性代数在神经网络中的应用

前向传播的矩阵表示：

在神经网络中，每一层的计算都可以表示为矩阵乘法：

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}) \tag{9}$$

其中：

- $\mathbf{W}^{(l)} \in \mathbb{R}^{d_l \times d_{l-1}}$：第 l 层的权重矩阵
- $\mathbf{h}^{(l-1)} \in \mathbb{R}^{d_{l-1}}$：第 $l-1$ 层的输出（第 l 层的输入）
- $\mathbf{b}^{(l)} \in \mathbb{R}^{d_l}$：偏置向量
- $\sigma(\cdot)$：激活函数

批量处理：

对于批量大小为 B 的输入，可以并行计算：

$$\mathbf{H}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{H}^{(l-1)} + \mathbf{b}^{(l)}\mathbf{1}^T) \tag{10}$$

其中 $\mathbf{H}^{(l)} \in \mathbb{R}^{d_l \times B}$ 是批量输出矩阵，$\mathbf{1} \in \mathbb{R}^B$ 是全 1 向量。

在 LLM 中的应用：

注意力机制中的矩阵运算：

Transformer 中的缩放点积注意力可以表示为：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{11}$$

其中：

- $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$：查询矩阵
- $\mathbf{K} \in \mathbb{R}^{m \times d_k}$：键矩阵
- $\mathbf{V} \in \mathbb{R}^{m \times d_v}$：值矩阵
- $\mathbf{Q}\mathbf{K}^T$：计算注意力分数矩阵（$n \times m$）
- 矩阵乘法的复杂度：$O(n \cdot m \cdot d_k)$

多头注意力的矩阵分割：

在多头注意力中，通过矩阵重塑实现并行计算：

$$\mathbf{Q} \to [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_h] \quad \text{（分割为 } h \text{ 个头）} \tag{12}$$

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \tag{13}$$

$$\text{MultiHead} = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{14}$$

这种设计充分利用了矩阵运算的并行性，是现代 GPU 加速的基础。

3 概率论与统计学

概率论和统计学为机器学习提供了处理不确定性的数学框架，从数据分布建模到参数估计，都离不开概率统计方法。

3.1 概率分布

概率分布描述了随机变量的取值规律，是统计学习的基础。

定义 3.1 (随机变量与概率分布). 随机变量 X 是一个函数，将样本空间映射到实数集。概率分布函数 $P(X = x)$ 或概率密度函数 $p(x)$ 描述了随机变量取各个值的概率。

离散分布:

伯努利分布：单次试验的成功概率分布

$$P(X = k) = \begin{cases} p & \text{if } k = 1 \\ 1 - p & \text{if } k = 0 \end{cases} \tag{15}$$

期望：$\mathbb{E}[X] = p$，方差：$\mathrm{Var}(X) = p(1 - p)$

二项分布：n 次独立伯努利试验的成功次数

$$P(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}, \quad k = 0, 1, \ldots, n \tag{16}$$

期望：$\mathbb{E}[X] = np$，方差：$\mathrm{Var}(X) = np(1 - p)$

多项分布：n 次独立试验中各类别出现的次数

$$P(X_1 = k_1, \ldots, X_m = k_m) = \frac{n!}{k_1! \cdots k_m!} p_1^{k_1} \cdots p_m^{k_m} \tag{17}$$

其中，$\sum_{i=1}^{m} k_i = n$，$\sum_{i=1}^{m} p_i = 1$。

连续分布:

正态分布（高斯分布）:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2} \right) \tag{18}$$

记作 $X \sim \mathcal{N}(\mu, \sigma^2)$，其中 μ 是均值，σ^2 是方差。

多元正态分布:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}|^{1/2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right) \tag{19}$$

其中，$\boldsymbol{\mu}$ 是均值向量，$\boldsymbol{\Sigma}$ 是协方差矩阵。

在 AI/ML 中的应用:

- **分类问题**：多项分布用于多分类，伯努利分布用于二分类
- **回归问题**：假设误差服从正态分布，使用最大似然估计
- **生成模型**：变分自编码器（VAE）、生成对抗网络（GAN）使用概率分布
- **贝叶斯方法**：使用先验分布和似然函数进行推理

例 3.1 (逻辑回归中的概率建模). 在逻辑回归中，假设 $P(Y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b)$，其中 σ 是 *sigmoid* 函数。这实际上是在建模 Y 服从伯努利分布，参数为 $\sigma(\mathbf{w}^T \mathbf{x} + b)$。

3.2 贝叶斯定理

贝叶斯定理是概率推理的核心，提供了在观察到新证据后更新信念的方法。

定理 3.1 (贝叶斯定理). *对于事件 A 和 B，如果 $P(B) > 0$，则：*

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{20}$$

在连续情况下，对于随机变量：

$$p(\theta|\mathbf{x}) = \frac{p(\mathbf{x}|\theta)p(\theta)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|\theta)p(\theta)}{\int p(\mathbf{x}|\theta)p(\theta)d\theta} \tag{21}$$

术语解释：

- $P(\theta)$：先验概率（Prior），在观察到数据前的信念
- $P(\mathbf{x}|\theta)$：似然函数（Likelihood），给定参数下数据的概率
- $P(\theta|\mathbf{x})$：后验概率（Posterior），观察到数据后的信念
- $P(\mathbf{x})$：证据（Evidence），数据的边际概率

通俗解释：贝叶斯定理就像”根据结果反推原因”。比如，如果知道某种疾病的症状（结果），可以反推患病的概率（原因）。先验概率是”一般人群中患病的概率”，后验概率是”出现症状后患病的概率”。

在 AI/ML 中的应用：

- **朴素贝叶斯分类器**：用于文本分类、垃圾邮件检测
- **贝叶斯网络**：表示变量之间的条件依赖关系
- **贝叶斯优化**：用于超参数优化
- **变分推理**：近似计算后验分布
- **贝叶斯神经网络**：为权重分配概率分布，量化不确定性

例 3.2 (垃圾邮件分类). *假设：*

- $P(\text{垃圾邮件}) = 0.3$（先验）
- $P(\text{”免费”}|\text{垃圾邮件}) = 0.8$（似然）
- $P(\text{”免费”}|\text{正常邮件}) = 0.1$

如果一封邮件包含”免费”，则：

$$P(\text{垃圾邮件}|\text{”免费”}) = \frac{P(\text{”免费”}|\text{垃圾邮件})P(\text{垃圾邮件})}{P(\text{”免费”})} \tag{22}$$

$$= \frac{0.8 \times 0.3}{0.8 \times 0.3 + 0.1 \times 0.7} = \frac{0.24}{0.31} \approx 0.77 \tag{23}$$

因此，这封邮件是垃圾邮件的概率约为 77%。

3.3 最大似然估计

最大似然估计（Maximum Likelihood Estimation, MLE）是参数估计的重要方法。

定义 3.2 (最大似然估计). 给定独立同分布的样本 $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$，其联合概率密度为 $p(\mathbf{x}_1, \ldots, \mathbf{x}_n|\theta) = \prod_{i=1}^{n} p(\mathbf{x}_i|\theta)$。似然函数定义为：

$$L(\theta) = \prod_{i=1}^{n} p(\mathbf{x}_i|\theta) \tag{24}$$

对数似然函数为：

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{n} \log p(\mathbf{x}_i|\theta) \tag{25}$$

最大似然估计 $\hat{\theta}_{MLE}$ 是使似然函数最大的参数值：

$$\hat{\theta}_{MLE} = \arg\max_{\theta} L(\theta) = \arg\max_{\theta} \ell(\theta) \tag{26}$$

求解方法：通常对对数似然函数求导并令其为零：

$$\frac{\partial \ell(\theta)}{\partial \theta} = 0 \tag{27}$$

在 AI/ML 中的应用：

- **线性回归**：假设误差服从正态分布，MLE 等价于最小二乘法
- **逻辑回归**：使用 MLE 估计参数
- **神经网络**：交叉熵损失函数对应多项分布的 MLE
- **高斯混合模型**：使用 EM 算法进行 MLE

例 3.3 (正态分布的 MLE). 假设样本 $x_1, \ldots, x_n$ 来自正态分布 $\mathcal{N}(\mu, \sigma^2)$，则：

$$\ell(\mu, \sigma^2) = \sum_{i=1}^{n} \log \left[\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x_i - \mu)^2}{2\sigma^2} \right) \right] \tag{28}$$

$$= -\frac{n}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \sum_{i=1}^{n} (x_i - \mu)^2 \tag{29}$$

对 μ 和 σ^2 求导并令其为零：

$$\frac{\partial \ell}{\partial \mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i - \mu) = 0 \Rightarrow \hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{30}$$

$$\frac{\partial \ell}{\partial \sigma^2} = -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{i=1}^{n}(x_i - \mu)^2 = 0 \Rightarrow \hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 \tag{31}$$

因此，样本均值和样本方差分别是均值和方差的最大似然估计。

3.4 期望、方差与协方差

这些统计量描述了随机变量的重要特征。

定义 3.3 (期望). 离散随机变量 X 的期望：

$$\mathbb{E}[X] = \sum_{x} xP(X = x) \tag{32}$$

连续随机变量 X 的期望：

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} xp(x)dx \tag{33}$$

定义 3.4 (方差). 随机变量 X 的方差：

$$Var(X) = \mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2 \tag{34}$$

定义 3.5 (协方差). 两个随机变量 X 和 Y 的协方差：

$$Cov(X, Y) = \mathbb{E}[(X - \mathbb{E}[X])(Y - \mathbb{E}[Y])] = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y] \tag{35}$$

在 AI/ML 中的应用：

- **特征选择**：使用方差筛选低方差特征
- **主成分分析**：协方差矩阵的特征值分解
- **批量归一化**：归一化均值和方差
- **损失函数**：均方误差是方差的估计

3.4.1 概率分布在机器学习中的应用

高斯分布在回归问题中的应用：

线性回归假设误差服从高斯分布：

$$y = \mathbf{w}^T\mathbf{x} + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{36}$$

这导致最大似然估计等价于最小化均方误差（MSE）：

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i - b)^2 \tag{37}$$

多项分布在分类问题中的应用：

多分类问题中，softmax 函数将 logits 转换为概率分布：

$$P(y = k|\mathbf{x}) = \frac{\exp(z_k)}{\sum_{j=1}^{C}\exp(z_j)} = \text{softmax}(\mathbf{z})_k \tag{38}$$

其中 $\mathbf{z} = [z_1, z_2, \ldots, z_C]^T$ 是模型的原始输出（logits），C 是类别数。

在 LLM 中的应用：

语言模型的概率建模：

自回归语言模型（如 GPT）将文本生成建模为条件概率：

$$P(w_1, w_2, \ldots, w_n) = \prod_{i=1}^{n} P(w_i|w_1, w_2, \ldots, w_{i-1}) \tag{39}$$

每一步生成都是基于前面所有词的条件概率分布：

$$P(w_i|w_{<i}) = \text{softmax}(\mathbf{W}\mathbf{h}_i) \tag{40}$$

其中 $\mathbf{h}_i$ 是位置 i 的隐藏状态，$\mathbf{W}$ 是输出投影矩阵。

困惑度（Perplexity）：

困惑度是语言模型的重要评估指标，定义为：

$$\text{Perplexity} = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log P(w_i|w_{<i})\right) \tag{41}$$

困惑度越小，模型对序列的预测越确定，性能越好。

4 优化理论

优化理论提供了寻找函数最小值或最大值的方法，是机器学习算法训练的核心。

4.1 梯度下降

梯度下降是最基本的优化算法，通过沿着梯度的反方向迭代更新参数。

定义 4.1 (梯度). *对于多元函数 $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$，其梯度定义为：*

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]^T \tag{42}$$

梯度指向函数值增加最快的方向。

梯度下降算法：

Algorithm 1 梯度下降算法

Require: 目标函数 $f(\mathbf{x})$，学习率 $\eta > 0$，初始点 $\mathbf{x}_0$

Ensure: 局部最优解 $\mathbf{x}^*$

1: 初始化 $\mathbf{x} = \mathbf{x}_0$
2: **repeat**
3: 计算梯度：$\mathbf{g} = \nabla f(\mathbf{x})$
4: 更新参数：$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathbf{g}$
5: **until** 收敛（如 $\|\mathbf{g}\| < \epsilon$）

学习率的选择：

- 学习率太小：收敛慢
- 学习率太大：可能发散，无法收敛
- 自适应学习率：Adam、RMSprop 等算法自动调整学习率

在 AI/ML 中的应用：

- **神经网络训练**：反向传播算法使用梯度下降更新权重
- **线性回归**：最小化均方误差
- **逻辑回归**：最小化对数似然函数
- **所有监督学习算法**：本质上都是优化问题

例 4.1 (线性回归的梯度下降). *对于线性回归 $y = \mathbf{w}^T\mathbf{x} + b$，损失函数为：*

$$\mathcal{L}(\mathbf{w}, b) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i - b)^2 \tag{43}$$

梯度为：

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = -\frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i - b)\mathbf{x}_i \tag{44}$$

$$\frac{\partial \mathcal{L}}{\partial b} = -\frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}^T\mathbf{x}_i - b) \tag{45}$$

梯度下降更新规则：

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{w}} \tag{46}$$

$$b \leftarrow b - \eta\frac{\partial \mathcal{L}}{\partial b} \tag{47}$$

4.2 凸优化

凸优化问题具有良好的性质，可以保证找到全局最优解。

定义 4.2 (凸函数). 函数 $f : \mathbb{R}^n \to \mathbb{R}$ 是凸函数，如果对于任意 $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$ 和 $\lambda \in [0, 1]$，有：

$$f(\lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2) \leq \lambda f(\mathbf{x}_1) + (1-\lambda)f(\mathbf{x}_2) \tag{48}$$

凸优化问题：

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ \text{s.t.} \quad & g_i(\mathbf{x}) \leq 0, \quad i = 1, \ldots, m \\ & h_j(\mathbf{x}) = 0, \quad j = 1, \ldots, p \end{aligned} \tag{49}$$

其中，f 和 g_i 是凸函数，h_j 是仿射函数。

凸优化的性质：

- 局部最优解就是全局最优解
- 可以使用高效的算法求解（如内点法）
- 对偶理论提供了另一种求解视角

在 AI/ML 中的应用：

- **支持向量机（SVM）**：凸优化问题
- **逻辑回归**：凸优化问题（对数似然函数是凹函数，取负号后是凸函数）
- **Lasso 和 Ridge 回归**：凸优化问题
- **线性规划**：用于资源分配等问题

4.2.1 梯度下降在机器学习中的应用

批量梯度下降（BGD）：

对于损失函数 $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f(\mathbf{x}_i;\theta), y_i)$，批量梯度下降的更新规则为：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) = \theta_t - \frac{\eta}{n}\sum_{i=1}^{n} \nabla_\theta \ell(f(\mathbf{x}_i;\theta_t), y_i) \tag{50}$$

随机梯度下降（SGD）：

每次只使用一个样本计算梯度：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \ell(f(\mathbf{x}_i;\theta_t), y_i) \tag{51}$$

小批量梯度下降（Mini-batch GD）：

使用小批量样本（通常 $B = 32, 64, 128$）：

$$\theta_{t+1} = \theta_t - \frac{\eta}{B}\sum_{i\in\mathcal{B}_t} \nabla_\theta \ell(f(\mathbf{x}_i;\theta_t), y_i) \tag{52}$$

Adam 优化器：

Adam 结合了动量和自适应学习率：

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t \quad \text{（一阶矩估计）} \tag{53}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 \quad \text{（二阶矩估计）} \tag{54}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \quad \text{（偏差修正）} \tag{55}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{56}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t \tag{57}$$

其中 $\mathbf{g}_t = \nabla_\theta \mathcal{L}(\theta_t)$ 是梯度，$\beta_1 = 0.9$，$\beta_2 = 0.999$，$\epsilon = 10^{-8}$。

在 LLM 训练中的应用：

大语言模型的训练通常使用：

- **混合精度训练**：使用 FP16 降低内存占用，FP32 保证数值稳定性
- **梯度累积**：当 GPU 内存有限时，累积多个小批量的梯度再更新

- **学习率调度**：使用 warmup 和学习率衰减策略
- **权重衰减**：L_2 正则化防止过拟合

4.3 拉格朗日乘数法

拉格朗日乘数法用于求解带约束的优化问题。

定理 4.1 (拉格朗日乘数法)**.** *考虑优化问题*：

$$\begin{aligned} \min_{\mathbf{x}} \quad & f(\mathbf{x}) \\ s.t. \quad & g_i(\mathbf{x}) = 0, \quad i = 1, \ldots, m \end{aligned} \tag{58}$$

构造拉格朗日函数：

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i g_i(\mathbf{x}) \tag{59}$$

最优解满足：

$$\nabla_{\mathbf{x}} \mathcal{L} = \mathbf{0} \tag{60}$$

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L} = \mathbf{0} \tag{61}$$

即：

$$\nabla f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}) = \mathbf{0} \tag{62}$$

$$g_i(\mathbf{x}) = 0, \quad i = 1, \ldots, m \tag{63}$$

KKT 条件：对于不等式约束，使用 Karush-Kuhn-Tucker (KKT) 条件：

$$\nabla f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla g_i(\mathbf{x}) + \sum_{j=1}^{p} \nu_j \nabla h_j(\mathbf{x}) = \mathbf{0} \tag{64}$$

$$g_i(\mathbf{x}) \leq 0 \tag{65}$$

$$\lambda_i \geq 0 \tag{66}$$

$$\lambda_i g_i(\mathbf{x}) = 0 \tag{67}$$

在 AI/ML 中的应用：

- **支持向量机**：使用拉格朗日乘数法推导对偶问题
- **正则化**：可以看作带约束的优化问题

- **最大熵模型**：在约束条件下最大化熵

例 4.2 (支持向量机的对偶问题). *SVM 的原始问题*：

$$\begin{aligned} \min_{\mathbf{w},b} \quad & \frac{1}{2}\|\mathbf{w}\|^2 \\ s.t. \quad & y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \quad i = 1, \ldots, n \end{aligned} \tag{68}$$

拉格朗日函数：

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\mathbf{w}\|^2 - \sum_{i=1}^{n} \alpha_i [y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1] \tag{69}$$

对偶问题：

$$\begin{aligned} \max_{\boldsymbol{\alpha}} \quad & \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ s.t. \quad & \sum_{i=1}^{n} \alpha_i y_i = 0 \\ & \alpha_i \geq 0, \quad i = 1, \ldots, n \end{aligned} \tag{70}$$

5 信息论基础

信息论提供了量化信息和不确定性的数学工具，在机器学习中用于特征选择、模型评估和正则化。

5.1 熵

熵度量了随机变量的不确定性。

定义 5.1 (信息熵). *离散随机变量 X 的熵定义为*：

$$H(X) = -\sum_{x} P(x) \log P(x) \tag{71}$$

连续随机变量 X 的微分熵定义为：

$$h(X) = -\int p(x) \log p(x) dx \tag{72}$$

熵的性质：

- 非负性：$H(X) \geq 0$
- 最大值：当分布均匀时，熵最大

- 可加性：$H(X,Y) = H(X) + H(Y|X)$

通俗解释：熵就像”惊喜程度”。如果一件事总是发生（概率为 1），没有惊喜，熵为 0。如果所有结果等可能，最不确定，熵最大。就像抛硬币，如果硬币总是正面，没有不确定性；如果正反面各 50%，不确定性最大。

在 AI/ML 中的应用：

- **决策树**：使用信息增益（熵的减少）选择分裂特征
- **特征选择**：选择能最大程度减少目标变量不确定性的特征
- **模型评估**：交叉熵作为分类任务的损失函数
- **正则化**：最大熵原理用于防止过拟合

例 5.1 (决策树中的信息增益). *在决策树中，选择特征 A 分裂节点，信息增益定义为：*

$$IG(D, A) = H(D) - \sum_{v} \frac{|D_v|}{|D|} H(D_v) \tag{73}$$

其中，D 是当前节点的数据集，D_v 是特征 A 取值为 v 的子集。选择信息增益最大的特征进行分裂。

5.2 互信息

互信息度量了两个随机变量之间的相关性。

定义 5.2 (互信息). *两个随机变量 X 和 Y 的互信息定义为：*

$$I(X;Y) = \sum_{x,y} P(x,y) \log \frac{P(x,y)}{P(x)P(y)} = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{74}$$

其中，$H(X|Y)$ 是条件熵：

$$H(X|Y) = -\sum_{x,y} P(x,y) \log P(x|y) \tag{75}$$

互信息的性质：

- 对称性：$I(X;Y) = I(Y;X)$
- 非负性：$I(X;Y) \geq 0$，当且仅当 X 和 Y 独立时等于 0
- 上界：$I(X;Y) \leq \min(H(X), H(Y))$

在 AI/ML 中的应用：

- **特征选择**：选择与目标变量互信息大的特征
- **独立成分分析（ICA）**：最大化互信息
- **信息瓶颈理论**：在压缩和预测之间权衡

5.2.1 信息论在机器学习中的应用

交叉熵损失函数：

在分类问题中，交叉熵损失是最常用的损失函数：

$$\mathcal{L}_{CE} = -\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log \hat{y}_{i,c} \tag{76}$$

其中 $y_{i,c}$ 是真实标签的 one-hot 编码，$\hat{y}_{i,c}$ 是模型预测的概率。

交叉熵可以理解为真实分布和预测分布之间的" 距离"：

$$H(P,Q) = -\sum_{x} P(x)\log Q(x) = H(P) + D_{KL}(P\|Q) \tag{77}$$

在 LLM 中的应用：

语言建模中的交叉熵：

对于序列 $\mathbf{w} = [w_1, w_2, \ldots, w_n]$，语言模型的损失函数为：

$$\mathcal{L} = -\sum_{i=1}^{n} \log P(w_i|w_{<i}) \tag{78}$$

这等价于最大化序列的似然概率。

困惑度与熵的关系：

困惑度是交叉熵的指数形式：

$$\text{Perplexity} = 2^{H(P,Q)} = \exp(H(P,Q)) \tag{79}$$

其中 $H(P,Q)$ 是真实分布 P 和模型分布 Q 之间的交叉熵。

KL 散度在模型评估中的应用：

KL 散度用于衡量两个概率分布的差异：

$$D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} \tag{80}$$

在模型评估中：

- $D_{KL}(P_{\text{data}}\|P_{\text{model}})$ 衡量模型分布与真实数据分布的差异
- 在变分推断中，用于优化变分后验分布
- 在知识蒸馏中，用于衡量教师模型和学生模型的差异

互信息在特征选择中的应用：

互信息用于衡量特征与目标变量之间的相关性：

$$I(X;Y) = H(Y) - H(Y|X) \tag{81}$$

互信息越大，说明特征 X 对预测 Y 的贡献越大，常用于特征选择。

5.3 KL 散度

KL 散度（Kullback-Leibler Divergence）度量了两个概率分布之间的差异。

定义 5.3 (KL 散度). *两个概率分布 P 和 Q 的 KL 散度定义为*：

$$D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)} = \mathbb{E}_{x\sim P}\left[\log \frac{P(x)}{Q(x)}\right] \tag{82}$$

连续情况：

$$D_{KL}(P\|Q) = \int p(x) \log \frac{p(x)}{q(x)} dx \tag{83}$$

KL 散度的性质：

- 非负性：$D_{KL}(P\|Q) \geq 0$，当且仅当 $P = Q$ 时等于 0
- 不对称性：$D_{KL}(P\|Q) \neq D_{KL}(Q\|P)$
- 不满足三角不等式，因此不是真正的距离度量

通俗解释：KL 散度就像” 用分布 Q 来编码分布 P 的额外成本”。如果 P 和 Q 相同，没有额外成本，KL 散度为 0。差异越大，额外成本越高。

在 AI/ML 中的应用：

- **变分推理**：最小化 KL 散度，用简单分布近似复杂后验分布
- **变分自编码器（VAE）**：使用 KL 散度作为正则项
- **模型比较**：比较不同模型的分布
- **知识蒸馏**：让学生模型学习教师模型的分布

例 5.2 (变分自编码器中的 KL 散度). *在 VAE 中，编码器学习后验分布 $q_\phi(\mathbf{z}|\mathbf{x})$，先验分布是 $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$。损失函数包含重构误差和 KL 散度：*

$$\mathcal{L} = \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})] - D_{KL}(q_\phi(\mathbf{z}|\mathbf{x}) \| p(\mathbf{z})) \tag{84}$$

KL 散度项确保学习到的潜在分布接近先验分布。

6 图论基础

图论提供了表示和处理关系结构的数学框架，在社交网络分析、推荐系统、图神经网络等领域有重要应用。

6.1 图的基本概念

定义 6.1 (图). *图 $G = (V, E)$ 由顶点集 V 和边集 E 组成，其中每条边 $e \in E$ 连接两个顶点。*

- **无向图**：*边没有方向，$(u, v) = (v, u)$*
- **有向图**：*边有方向，$(u, v) \neq (v, u)$*
- **加权图**：*边有权重 $w : E \to \mathbb{R}$*
- **简单图**：*没有自环和多重边*

图的表示：

- **邻接矩阵**：$\mathbf{A} \in \{0, 1\}^{|V| \times |V|}$，$A_{ij} = 1$ 表示顶点 i 和 j 之间有边
- **邻接表**：为每个顶点存储其邻居列表
- **边列表**：存储所有边的列表

度（Degree）：顶点 v 的度 $d(v)$ 是与 v 相连的边的数量。对于有向图，分为入度（in-degree）和出度（out-degree）。

在 AI/ML 中的应用：

- **社交网络**：用户是顶点，关注关系是边
- **知识图谱**：实体是顶点，关系是边
- **推荐系统**：用户-物品二部图
- **分子结构**：原子是顶点，化学键是边

6.2 路径搜索

路径搜索是图论中的基本问题，用于找到两个顶点之间的路径。

最短路径问题：找到从顶点 s 到顶点 t 的路径，使得路径上边的权重之和最小。

Dijkstra 算法：用于非负权重图的最短路径问题。

Algorithm 2 Dijkstra 最短路径算法

Require: 图 $G = (V, E)$，起点 s，权重函数 w

Ensure: 从 s 到所有顶点的最短距离 d

初始化：$d[s] = 0$，$d[v] = \infty$ 对于 $v \neq s$，$S = \emptyset$
while $S \neq V$ **do**
  选择 $u \notin S$ 使得 $d[u]$ 最小
  $S \leftarrow S \cup \{u\}$
  for 所有与 u 相邻的顶点 $v \notin S$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
      $d[v] \leftarrow d[u] + w(u, v)$
    end if
  end for
end while

在 AI/ML 中的应用：

- **路径规划**：机器人导航、GPS 导航
- **网络路由**：互联网数据包路由
- **社交网络分析**：找到两个用户之间的最短路径（六度分隔理论）

6.3 图神经网络相关概念

图神经网络（Graph Neural Network, GNN）扩展了神经网络以处理图结构数据。

图卷积：图上的卷积操作，聚合邻居节点的信息。

消息传递框架：GNN 的核心思想是通过消息传递更新节点表示：

$$\mathbf{h}_v^{(l+1)} = \text{UPDATE}^{(l)}\left(\mathbf{h}_v^{(l)}, \text{AGGREGATE}^{(l)}\left(\{\mathbf{h}_u^{(l)} : u \in \mathcal{N}(v)\}\right)\right) \tag{85}$$

其中，$\mathbf{h}_v^{(l)}$ 是节点 v 在第 l 层的表示，$\mathcal{N}(v)$ 是 v 的邻居集合。

图注意力网络（GAT）：使用注意力机制聚合邻居信息：

$$\mathbf{h}_v^{(l+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(l)} \mathbf{W}^{(l)} \mathbf{h}_u^{(l)}\right) \tag{86}$$

其中，α_{vu} 是注意力权重：

$$\alpha_{vu} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_v \| \mathbf{W}\mathbf{h}_u]))}{\sum_{w \in \mathcal{N}(v)} \exp(\text{LeakyReLU}(\mathbf{a}^T[\mathbf{W}\mathbf{h}_v \| \mathbf{W}\mathbf{h}_w]))} \tag{87}$$

在 AI/ML 中的应用：

- **节点分类**：预测节点的类别（如社交网络中的用户分类）
- **链接预测**：预测两个节点之间是否存在边
- **图分类**：对整个图进行分类（如分子性质预测）
- **推荐系统**：用户-物品图上的推荐
- **知识图谱**：实体和关系的表示学习

例 6.1 (社交网络中的节点分类). 在社交网络中，每个用户是一个节点，关注关系是边。使用 *GNN* 可以：

- 学习用户的嵌入表示
- 根据用户的朋友特征预测用户的兴趣
- 发现社区结构

7 数学在 ML 和 LLM 中的综合应用

7.1 机器学习中的数学工具链

完整的 ML 流程中的数学应用：

1. **数据预处理**：

- **线性代数**：数据标准化、PCA 降维、特征变换
- **概率论**：异常值检测（3σ 原则）、数据分布分析

2. **模型设计**：

- **线性代数**：定义权重矩阵维度、网络结构
- **概率论**：选择输出分布（高斯分布用于回归，多项分布用于分类）

3. **模型训练**：

- **优化理论**：梯度下降、Adam 等优化算法
- **线性代数**：矩阵乘法实现前向和反向传播
- **信息论**：交叉熵损失函数

4. **模型评估**：

- **概率论**：ROC 曲线、AUC 值
- **信息论**：困惑度、KL 散度
- **线性代数**：混淆矩阵、特征重要性分析

7.2 大语言模型中的数学工具链

Transformer 架构中的数学：

1. **词嵌入**：

- **线性代数**：将词映射到高维向量空间 $\mathbf{e}_i \in \mathbb{R}^d$
- **概率论**：学习词共现的统计规律

2. **位置编码**：

- **三角函数**：使用正弦和余弦函数编码位置信息

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{88}$$

3. **注意力机制**：

- **线性代数**：矩阵乘法 $\mathbf{QK}^T$ 计算注意力分数
- **概率论**：Softmax 归一化得到注意力权重（概率分布）
- **信息论**：注意力权重可以理解为信息分配

4. **前馈网络**:

   - **线性代数**: $\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$

5. **训练过程**:

   - **优化理论**: Adam 优化器、学习率调度
   - **信息论**: 交叉熵损失、困惑度评估
   - **概率论**: 语言建模的概率框架

6. **推理过程**:

   - **概率论**: 采样策略（贪心、top-k、top-p）
   - **信息论**: 困惑度计算、生成质量评估

数学工具的综合运用示例:

例 7.1 (Transformer 中的一次前向传播)**.** *给定输入序列* $\mathbf{X} \in \mathbb{R}^{n \times d}$:

1. **线性变换**（线性代数）:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \in \mathbb{R}^{n \times d_k} \tag{89}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K \in \mathbb{R}^{n \times d_k} \tag{90}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V \in \mathbb{R}^{n \times d_v} \tag{91}$$

2. **计算注意力分数**（线性代数）:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \in \mathbb{R}^{n \times n} \tag{92}$$

3. ***Softmax* 归一化**（概率论）:

$$\mathbf{A} = softmax(\mathbf{S}) \in \mathbb{R}^{n \times n} \tag{93}$$

其中 $\mathbf{A}_{ij}$ 表示位置 i 对位置 j 的注意力权重（概率）。

4. **加权求和**（线性代数）:

$$\mathbf{Z} = \mathbf{A}\mathbf{V} \in \mathbb{R}^{n \times d_v} \tag{94}$$

5. **损失计算**（信息论）:

$$\mathcal{L} = -\sum_{i=1}^{n} \log P(w_i | w_{<i}) = -\sum_{i=1}^{n} \log softmax(\mathbf{W}_o \mathbf{z}_i)_{w_i} \tag{95}$$

8 总结

本文档系统性地介绍了 AI 领域中最常用的数学理论和方法：

- **线性代数**：提供了数据表示和变换的工具，是神经网络计算的基础
- **概率论与统计学**：提供了处理不确定性的框架，是统计学习的理论基础
- **优化理论**：提供了寻找最优解的方法，是模型训练的核心
- **信息论**：提供了量化信息和不确定性的工具，用于特征选择和模型评估
- **图论**：提供了表示和处理关系结构的框架，是图神经网络的基础

这些数学工具相互交织，共同构成了 AI 和机器学习的坚实数学基础。在机器学习中，从数据预处理到模型训练、评估，数学工具贯穿始终；在大语言模型中，从词嵌入到注意力机制，从训练到推理，数学提供了精确的描述和高效的计算方法。掌握这些数学知识，有助于深入理解算法原理，设计新的方法，并解决实际问题。

9 作业与练习

9.1 计算题

1. **矩阵运算**：
   - 给定矩阵 $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 和 $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$，计算 $\mathbf{AB}$、$\mathbf{A}^T$ 和 $\mathbf{A}^{-1}$（如果存在）。
   - 计算向量 $\mathbf{x} = [1, 2, 3]^T$ 的 L_1、L_2 和 L_∞ 范数。
   - 给定两个向量 $\mathbf{u} = [1, 0, 1]^T$ 和 $\mathbf{v} = [0, 1, 1]^T$，计算它们的点积和余弦相似度。
2. **概率计算**：
   - 抛一枚公平硬币 3 次，计算恰好出现 2 次正面的概率。
   - 假设 $X \sim \mathcal{N}(0, 1)$，计算 $P(-1 < X < 1)$（可以使用标准正态分布表）。
   - 给定先验概率 $P(A) = 0.3$，$P(B|A) = 0.8$，$P(B|\neg A) = 0.2$，使用贝叶斯定理计算 $P(A|B)$。
3. **最大似然估计**：
   - 给定样本 $x_1 = 1, x_2 = 2, x_3 = 3$，假设它们来自泊松分布 $P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}$，求参数 λ 的最大似然估计。

- 给定样本 $x_1, \ldots, x_n$ 来自指数分布 $p(x) = \lambda e^{-\lambda x}$（$x \geq 0$），求参数 λ 的最大似然估计。

4. **信息论**：

- 计算伯努利分布 $P(X = 1) = p$，$P(X = 0) = 1 - p$ 的熵 $H(X)$，并找出使熵最大的 p 值。
- 给定联合分布：

$X \backslash Y$	0	1
0	0.3	0.2
1	0.1	0.4

  计算 $H(X)$、$H(Y)$、$H(X|Y)$ 和 $I(X;Y)$。

5. **优化问题**：

- 使用梯度下降法求解 $f(x) = x^2 + 2x + 1$ 的最小值，初始值 $x_0 = 0$，学习率 $\eta = 0.1$，迭代 5 次。
- 使用拉格朗日乘数法求解约束优化问题：

$$\begin{aligned} \min \quad & x^2 + y^2 \\ \text{s.t.} \quad & x + y = 1 \end{aligned} \tag{96}$$

9.2 概念题

1. **线性代数**：

- 解释特征值和特征向量的几何意义。
- 为什么在机器学习中经常使用矩阵的转置?
- 解释为什么矩阵乘法不满足交换律，但在神经网络中这通常不是问题。

2. **概率论**：

- 解释先验概率、似然函数和后验概率的区别和联系。
- 为什么在机器学习中经常假设误差服从正态分布?
- 最大似然估计和最大后验估计（MAP）的区别是什么?

3. **优化理论**：

- 解释为什么梯度下降可能陷入局部最优，而凸优化可以保证全局最优。
- 学习率的选择对梯度下降有什么影响?

- 解释拉格朗日乘数的物理意义或几何意义。

4. **信息论**:

   - 解释熵、互信息和 KL 散度的直观含义。
   - 为什么 KL 散度不是真正的距离度量?
   - 在决策树中，为什么使用信息增益而不是直接使用熵?

5. **图论**:

   - 解释邻接矩阵和邻接表的优缺点。
   - 为什么图神经网络需要特殊的消息传递机制，而不能直接使用传统的卷积?
   - 解释图注意力网络中注意力机制的作用。

9.3 应用题

1. **PCA 实现**:

   - 给定一个数据矩阵，实现主成分分析（PCA）算法
   - 使用特征值分解计算主成分
   - 将数据投影到前两个主成分，可视化结果

2. **朴素贝叶斯分类器**:

   - 实现一个简单的朴素贝叶斯分类器
   - 在文本分类任务上测试（如垃圾邮件检测）
   - 分析先验概率对分类结果的影响

3. **梯度下降可视化**:

   - 实现梯度下降算法
   - 在二维函数上可视化梯度下降的路径
   - 比较不同学习率对收敛速度的影响

4. **信息增益计算**:

   - 实现信息熵和信息增益的计算
   - 在简单的数据集上计算各特征的信息增益
   - 解释为什么信息增益大的特征更适合用于分裂

5. **图的基本操作**：

   - 实现图的邻接矩阵和邻接表表示
   - 实现图的遍历算法（深度优先搜索、广度优先搜索）
   - 实现简单的图神经网络消息传递

9.4 综合思考题

1. **数学工具的综合应用**：

   - 选择一个机器学习算法（如逻辑回归、支持向量机、神经网络），分析其中用到了哪些数学工具。
   - 解释这些数学工具在算法中各自的作用。
   - 讨论如果缺少某个数学工具，算法会如何变化。

2. **数学与直觉**：

   - 选择一个数学概念（如熵、梯度、特征值），用非数学的语言解释其直观含义。
   - 举一个生活中的例子说明这个概念。
   - 解释这个概念在机器学习中的重要性。

3. **数学证明**：

   - 证明信息熵的非负性。
   - 证明 KL 散度的非负性（使用 Jensen 不等式）。
   - 证明梯度指向函数值增加最快的方向。

通过完成以上作业和练习，读者可以深入理解 AI 相关的数学理论基础，掌握数学工具在机器学习中的应用，为进一步的学习和研究打下坚实的基础。

Part II

第二部分：Python 编程基础

10 Python 基础语法

10.1 变量与数据类型

概念解释：Python 是动态类型语言，变量不需要显式声明类型，类型在运行时确定。

基本数据类型：

- **整数（int）**：任意大小的整数
- **浮点数（float）**：双精度浮点数
- **字符串（str）**：不可变字符序列
- **布尔值（bool）**：True 或 False
- **列表（list）**：可变有序序列
- **元组（tuple）**：不可变有序序列
- **字典（dict）**：键值对映射
- **集合（set）**：无序不重复元素集

```
# 整数
x = 42
print(type(x))  # <class 'int'>

# 浮点数
y = 3.14
print(type(y))  # <class 'float'>

# 字符串
name = "Python"
print(type(name))  # <class 'str'>

# 列表
```

```
numbers = [1, 2, 3, 4, 5]
print(type(numbers))  # <class 'list'>

# 字典
person = {"name": "Alice", "age": 30}
print(type(person))  # <class 'dict'>

# 集合
unique_numbers = {1, 2, 3, 4, 5}
print(type(unique_numbers))  # <class 'set'>
```

Listing 1: Python 基本数据类型示例

在 AI/ML 中的应用:

- 列表用于存储特征向量、样本索引等
- 字典用于存储模型参数、配置信息
- 字符串用于处理文本数据、文件路径

10.2 控制流

条件语句:

```
# if-elif-else
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
else:
    grade = "D"

print(f"分数 {score} 对应的等级是 {grade}")

# 在数据处理中的应用：数据分类
def categorize_age(age):
```

```
    if age < 18:
        return "未成年"
    elif age < 65:
        return "成年"
    else:
        return "老年"
```

Listing 2: 条件语句示例

循环语句:

```
# for 循环
numbers = [1, 2, 3, 4, 5]
squared = []
for num in numbers:
    squared.append(num ** 2)
print(squared)  # [1, 4, 9, 16, 25]

# while 循环
count = 0
while count < 5:
    print(count)
    count += 1

# 在数据处理中的应用：遍历数据
data = [{"name": "Alice", "age": 25}, {"name": "Bob", "age": 30}]
for person in data:
    print(f"{person['name']} is {person['age']} years old")
```

Listing 3: 循环语句示例

10.3 函数定义

概念解释：函数是组织代码的基本单元，可以接受参数并返回结果。

```
# 基本函数定义
def add(a, b):
    """计算两个数的和"""
    return a + b
```

```
result = add(3, 5)
print(result)  # 8

# 默认参数
def greet(name, greeting="Hello"):
    """打招呼函数，带有默认参数"""
    return f"{greeting}, {name}!"

print(greet("Alice"))  # Hello, Alice!
print(greet("Bob", "Hi"))  # Hi, Bob!

# 可变参数
def sum_all(*args):
    """计算所有参数的和"""
    total = 0
    for num in args:
        total += num
    return total

print(sum_all(1, 2, 3, 4, 5))  # 15

# 关键字参数
def create_person(name, age, **kwargs):
    """创建人员信息字典"""
    person = {"name": name, "age": age}
    person.update(kwargs)
    return person

person = create_person("Alice", 25, city="Beijing", job="Engineer")
print(person)  # {'name': 'Alice', 'age': 25, 'city': 'Beijing', 'job': '
   Engineer'}
```

Listing 4: 函数定义示例

在 AI/ML 中的应用:

- 定义数据预处理函数
- 实现评估指标函数
- 封装模型训练流程

10.4 面向对象编程

概念解释：面向对象编程（OOP）通过类和对象组织代码，实现封装、继承和多态。

类与对象：

```
class Dataset:
    """数据集类，用于管理机器学习数据"""

    def __init__(self, data, labels=None):
        """
        初始化数据集

        参数:
            data: 特征数据
            labels: 标签数据（可选）
        """
        self.data = data
        self.labels = labels
        self.size = len(data)

    def __len__(self):
        """返回数据集大小"""
        return self.size

    def __getitem__(self, index):
        """支持索引访问"""
        if self.labels is not None:
            return self.data[index], self.labels[index]
        return self.data[index]

    def get_batch(self, batch_size):
        """获取一个批次的数据"""
        indices = list(range(0, self.size, batch_size))
        for i in indices:
            end = min(i + batch_size, self.size)
            if self.labels is not None:
                yield self.data[i:end], self.labels[i:end]
            else:
                yield self.data[i:end]
```

```
# 使用示例
data = [[1, 2], [3, 4], [5, 6]]
labels = [0, 1, 0]
dataset = Dataset(data, labels)

print(f"数据集大小: {len(dataset)}")  # 数据集大小: 3
print(f"第一个样本: {dataset[0]}")  # 第一个样本: ([1, 2], 0)

# 获取批次
for batch_data, batch_labels in dataset.get_batch(2):
    print(f"批次数据: {batch_data}, 批次标签: {batch_labels}")
```

Listing 5: 类定义示例

继承:

```
class BaseModel:
    """基础模型类"""

    def __init__(self):
        self.trained = False

    def train(self):
        """训练模型"""
        self.trained = True
        print("模型训练完成")

    def predict(self, x):
        """预测（需要在子类中实现）"""
        raise NotImplementedError("子类必须实现 predict 方法")

class LinearModel(BaseModel):
    """线性模型，继承自 BaseModel"""

    def __init__(self):
        super().__init__()
        self.weights = None

    def train(self, X, y):
        """训练线性模型"""
```

```
        # 简化的训练过程
        self.weights = [0.5, 0.3]  # 示例权重
        super().train()

    def predict(self, x):
        """预测"""
        if not self.trained:
            raise ValueError("模型尚未训练")
        # 简化的预测: w0 * x[0] + w1 * x[1]
        return self.weights[0] * x[0] + self.weights[1] * x[1]

# 使用示例
model = LinearModel()
model.train([[1, 2], [3, 4]], [1, 2])
prediction = model.predict([2, 3])
print(f"预测结果: {prediction}")
```

Listing 6: 继承示例

10.5 常用内置模块

os 模块: 操作系统接口

```
import os

# 获取当前工作目录
current_dir = os.getcwd()
print(f"当前目录: {current_dir}")

# 列出目录内容
files = os.listdir('.')
print(f"目录内容: {files}")

# 检查文件是否存在
file_path = "data.csv"
if os.path.exists(file_path):
    print(f"{file_path} 存在")
else:
    print(f"{file_path} 不存在")
```

```
# 路径操作
base_path = "/home/user"
data_path = os.path.join(base_path, "data", "dataset.csv")
print(f"完整路径: {data_path}")
```

Listing 7: os 模块示例

json 模块：JSON 数据处理

```
import json

# 将 Python 对象转换为 JSON 字符串
data = {
    "name": "Alice",
    "age": 25,
    "scores": [85, 90, 88]
}
json_string = json.dumps(data, ensure_ascii=False, indent=2)
print(json_string)

# 将 JSON 字符串转换为 Python 对象
parsed_data = json.loads(json_string)
print(parsed_data["name"])

# 读写 JSON 文件
# 写入
with open("data.json", "w", encoding="utf-8") as f:
    json.dump(data, f, ensure_ascii=False, indent=2)

# 读取
with open("data.json", "r", encoding="utf-8") as f:
    loaded_data = json.load(f)
    print(loaded_data)
```

Listing 8: json 模块示例

datetime 模块：日期时间处理

```
from datetime import datetime, timedelta
```

```
# 获取当前时间
now = datetime.now()
print(f"当前时间: {now}")

# 格式化时间
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print(f"格式化时间: {formatted}")

# 解析时间字符串
time_str = "2024-01-15 10:30:00"
parsed_time = datetime.strptime(time_str, "%Y-%m-%d %H:%M:%S")
print(f"解析的时间: {parsed_time}")

# 时间计算
future_time = now + timedelta(days=7)
print(f"7天后: {future_time}")

# 在时间序列数据处理中的应用
dates = [datetime(2024, 1, i) for i in range(1, 6)]
print("日期列表:", dates)
```

Listing 9: datetime 模块示例

collections 模块：特殊容器类型

```
from collections import Counter, defaultdict, deque

# Counter: 计数器
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
word_count = Counter(words)
print(f"词频统计: {word_count}")
print(f"最常见的2个: {word_count.most_common(2)}")

# defaultdict: 带默认值的字典
dd = defaultdict(list)
dd["fruits"].append("apple")
dd["fruits"].append("banana")
print(f"默认字典: {dict(dd)}")

# deque: 双端队列
```

```
queue = deque([1, 2, 3])
queue.append(4)  # 右端添加
queue.appendleft(0)  # 左端添加
print(f"双端队列: {queue}")
```

Listing 10: collections 模块示例

10.6 文件操作

CSV 文件处理:

```
import csv

# 写入 CSV 文件
data = [
    ["姓名", "年龄", "城市"],
    ["Alice", 25, "北京"],
    ["Bob", 30, "上海"],
    ["Charlie", 35, "广州"]
]

with open("people.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(data)

# 读取 CSV 文件
with open("people.csv", "r", encoding="utf-8") as f:
    reader = csv.reader(f)
    header = next(reader)  # 读取表头
    print(f"表头: {header}")
    for row in reader:
        print(f"数据: {row}")

# 使用字典方式读写 (更常用)
with open("people.csv", "r", encoding="utf-8") as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(f"姓名: {row['姓名']}, 年龄: {row['年龄']}")
```

Listing 11: CSV 文件处理示例

异常处理:

```
# 基本异常处理
try:
    result = 10 / 0
except ZeroDivisionError:
    print("除数不能为零! ")

# 多个异常类型
try:
    value = int("abc")
    result = 10 / value
except ValueError:
    print("无法转换为整数")
except ZeroDivisionError:
    print("除数不能为零")
except Exception as e:
    print(f"发生错误: {e}")

# finally 子句
def read_file_safely(filename):
    """安全读取文件"""
    try:
        with open(filename, "r", encoding="utf-8") as f:
            content = f.read()
        return content
    except FileNotFoundError:
        print(f"文件 {filename} 不存在")
        return None
    except Exception as e:
        print(f"读取文件时出错: {e}")
        return None
    finally:
        print("文件操作完成")

# 自定义异常
class DataValidationError(Exception):
```

```
    """数据验证错误"""
    pass

def validate_age(age):
    """验证年龄"""
    if age < 0 or age > 150:
        raise DataValidationError(f"年龄 {age} 不在有效范围内")
    return True

try:
    validate_age(200)
except DataValidationError as e:
    print(f"验证失败: {e}")
```

Listing 12: 异常处理示例

10.7 高级特性

列表推导式:

```
# 基本列表推导式
squares = [x**2 for x in range(10)]
print(f"平方数: {squares}")

# 带条件的列表推导式
even_squares = [x**2 for x in range(10) if x % 2 == 0]
print(f"偶数的平方: {even_squares}")

# 嵌套列表推导式
matrix = [[i*j for j in range(3)] for i in range(3)]
print(f"矩阵: {matrix}")

# 在数据处理中的应用: 特征提取
data = [{"age": 25, "score": 85}, {"age": 30, "score": 90}]
ages = [person["age"] for person in data]
scores = [person["score"] for person in data if person["score"] > 80]
print(f"年龄列表: {ages}")
print(f"高分列表: {scores}")
```

Listing 13: 列表推导式示例

生成器:

```
# 生成器函数
def fibonacci(n):
    """生成斐波那契数列"""
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# 使用生成器
for num in fibonacci(10):
    print(num, end=" ")
print()

# 生成器表达式
squares_gen = (x**2 for x in range(10))
print(f"生成器对象: {squares_gen}")
print(f"转换为列表: {list(squares_gen)}")

# 在数据处理中的应用: 大数据流处理
def read_large_file(filename):
    """逐行读取大文件"""
    with open(filename, "r", encoding="utf-8") as f:
        for line in f:
            yield line.strip()

# 使用生成器处理大文件, 节省内存
# for line in read_large_file("large_data.txt"):
#     process(line)
```

Listing 14: 生成器示例

装饰器:

```
import time
```

```python
from functools import wraps

# 基本装饰器
def timer(func):
    """计时装饰器"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} 执行时间: {end - start:.4f} 秒")
        return result
    return wrapper

@timer
def slow_function():
    """慢速函数"""
    time.sleep(1)
    return "完成"

result = slow_function()

# 带参数的装饰器
def repeat(times):
    """重复执行装饰器"""
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            results = []
            for _ in range(times):
                results.append(func(*args, **kwargs))
            return results
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))
```

Listing 15: 装饰器示例

11 NumPy 基础

NumPy（Numerical Python）是 Python 科学计算的基础库，提供了高效的多维数组对象和数学运算函数。

11.1 NumPy 基础概念

概念解释：

- **数组（ndarray）**：NumPy 的核心数据结构，是同质多维数组
- **维度（dimension）**：数组的轴数，1 维数组有 1 个轴，2 维数组有 2 个轴
- **形状（shape）**：描述数组每个维度大小的元组
- **数据类型（dtype）**：数组中元素的数据类型

学术解释：NumPy 数组是连续内存块中的同质数据集合，支持向量化操作，比 Python 列表快数十到数百倍。

通俗解释：NumPy 数组就像一个整齐排列的表格，所有数据都是同一种类型，可以快速进行数学运算。

11.2 数组创建与初始化

```
import numpy as np

# 从列表创建数组
arr1 = np.array([1, 2, 3, 4, 5])
print(f"一维数组：{arr1}")
print(f"形状：{arr1.shape}")  # (5,)
print(f"维度：{arr1.ndim}")  # 1
print(f"数据类型：{arr1.dtype}")  # int64

# 二维数组
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
```

```
print(f"二维数组:\n{arr2}")
print(f"形状: {arr2.shape}")  # (2, 3)

# 创建全零数组
zeros = np.zeros((3, 4))
print(f"全零数组:\n{zeros}")

# 创建全一数组
ones = np.ones((2, 3))
print(f"全一数组:\n{ones}")

# 创建单位矩阵
identity = np.eye(3)
print(f"单位矩阵:\n{identity}")

# 使用 arange 创建数组
arr_range = np.arange(0, 10, 2)
print(f"arange(0, 10, 2): {arr_range}")  # [0 2 4 6 8]

# 使用 linspace 创建等间距数组
arr_linspace = np.linspace(0, 1, 5)
print(f"linspace(0, 1, 5): {arr_linspace}")  # [0.   0.25 0.5  0.75 1.  ]

# 随机数组
random_arr = np.random.rand(3, 3)  # [0, 1) 均匀分布
print(f"随机数组:\n{random_arr}")

normal_arr = np.random.randn(3, 3)  # 标准正态分布
print(f"正态分布随机数组:\n{normal_arr}")

# 指定数据类型的数组
int_arr = np.array([1, 2, 3], dtype=np.float32)
print(f"浮点数组: {int_arr}, 类型: {int_arr.dtype}")
```

Listing 16: 数组创建示例

API 说明:

- `np.array(object, dtype=None)`: 从列表或其他数组创建数组
- `np.zeros(shape, dtype=float)`: 创建全零数组

- `np.ones(shape, dtype=float)`：创建全一数组
- `np.arange(start, stop, step)`：创建等间距数组
- `np.linspace(start, stop, num)`：创建指定数量的等间距数组
- `np.random.rand(*dims)`：创建 [0,1) 均匀分布随机数组
- `np.random.randn(*dims)`：创建标准正态分布随机数组

11.3 数组操作

索引与切片：

概念解释：索引用于访问数组的特定元素，切片用于获取数组的子数组。

数学表达式演示：

对于矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$：

$$\mathbf{A} = \begin{bmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \end{bmatrix} \tag{97}$$

基本索引：访问第 i 行第 j 列的元素：

$$A_{ij} = \mathbf{A}[i, j] \tag{98}$$

切片操作：

- 第 i 行：$\mathbf{A}[i, :] = [A_{i0}, A_{i1}, A_{i2}, A_{i3}]$
- 第 j 列：$\mathbf{A}[:, j] = [A_{0j}, A_{1j}, A_{2j}]^T$
- 前 k 行：$\mathbf{A}[: k, :] = \begin{bmatrix} A_{00} & \cdots & A_{0(n-1)} \\ \vdots & \ddots & \vdots \\ A_{(k-1)0} & \cdots & A_{(k-1)(n-1)} \end{bmatrix}$

布尔索引：对于条件 $C(\mathbf{A})$，布尔索引定义为：

$$\mathbf{A}[C(\mathbf{A})] = \{A_{ij} : C(A_{ij}) = \text{True}\} \tag{99}$$

例如，$C(x) = x > 5$，则：

$$\mathbf{A}[\mathbf{A} > 5] = \{A_{ij} : A_{ij} > 5\} \tag{100}$$

```python
import numpy as np

arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])

print("原始数组:")
print(arr)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]

# 基本索引
print(f"\narr[0, 0] = {arr[0, 0]}")  # 1
print(f"arr[1, 2] = {arr[1, 2]}")  # 7

# 切片
print(f"\n第一行 arr[0, :] = {arr[0, :]}")  # [1 2 3 4]
print(f"第一列 arr[:, 0] = {arr[:, 0]}")  # [1 5 9]
print(f"\n前两行 arr[:2, :]:\n{arr[:2, :]}")
# [[1 2 3 4]
#  [5 6 7 8]]
print(f"\n前两列 arr[:, :2]:\n{arr[:, :2]}")
# [[ 1  2]
#  [ 5  6]
#  [ 9 10]]

# 布尔索引
mask = arr > 5
print(f"\n布尔掩码 (arr > 5):\n{mask}")
# [[False False False False]
#  [False  True  True  True]
#  [ True  True  True  True]]
print(f"\n大于5的值 arr[mask] = {arr[mask]}")  # [6 7 8 9 10 11 12]

# 花式索引 (Fancy Indexing)
indices = [0, 2]
print(f"\n选择第0和第2行 arr[[0, 2], :]:\n{arr[indices, :]}")
# [[ 1  2  3  4]
```

```
#  [ 9 10 11 12]]
```

Listing 17: 数组索引与切片示例

数组重塑与转置:

概念解释: 重塑（reshape）改变数组的形状而不改变数据，转置（transpose）交换数组的维度。

数学表达式演示:

对于一维数组 $\mathbf{a} = [a_0, a_1, \ldots, a_{11}]$，重塑为 3×4 矩阵:

$$\mathbf{a} = [a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}]^T \tag{101}$$

重塑为 3×4 矩阵:

$$\text{reshape}(\mathbf{a}, (3, 4)) = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \\ a_8 & a_9 & a_{10} & a_{11} \end{bmatrix} = \mathbf{A} \tag{102}$$

转置操作:

$$\mathbf{A}^T = \begin{bmatrix} a_0 & a_4 & a_8 \\ a_1 & a_5 & a_9 \\ a_2 & a_6 & a_{10} \\ a_3 & a_7 & a_{11} \end{bmatrix} \tag{103}$$

对于矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$，转置定义为:

$$(\mathbf{A}^T)_{ij} = A_{ji} \tag{104}$$

```
import numpy as np

arr = np.arange(12)
print(f"原始数组: {arr}")  # [0 1 2 3 4 5 6 7 8 9 10 11]

# 重塑为 3x4 数组
reshaped = arr.reshape(3, 4)
print(f"重塑为 3x4:\n{reshaped}")
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

```
# 转置
transposed = reshaped.T
print(f"转置:\n{transposed}")
# [[ 0  4  8]
#  [ 1  5  9]
#  [ 2  6 10]
#  [ 3  7 11]]

# 展平
flattened = reshaped.flatten()
print(f"展平: {flattened}")  # [0 1 2 3 4 5 6 7 8 9 10 11]

# 改变形状（不复制数据）
arr_2d = arr.reshape(3, 4)
arr_2d[0, 0] = 999
print(f"修改后原数组: {arr}")  # 原数组也被修改
```

Listing 18: 数组重塑示例

数组拼接:

概念解释: 拼接（concatenation）将多个数组沿指定轴组合成一个数组。

数学表达式演示:

对于矩阵 $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ 和 $\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$:

垂直拼接（沿轴 0）:

$$\text{vstack}([\mathbf{A}, \mathbf{B}]) = \begin{bmatrix} \mathbf{A} \\ \mathbf{B} \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{105}$$

水平拼接（沿轴 1）:

$$\text{hstack}([\mathbf{A}, \mathbf{B}]) = \begin{bmatrix} \mathbf{A} & \mathbf{B} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 3 & 4 & 7 & 8 \end{bmatrix} \tag{106}$$

一般地，对于数组 $\mathbf{A} \in \mathbb{R}^{m \times n}$ 和 $\mathbf{B} \in \mathbb{R}^{p \times n}$，沿轴 0 拼接:

$$\text{concatenate}([\mathbf{A}, \mathbf{B}], \text{axis} = 0) \in \mathbb{R}^{(m+p) \times n} \tag{107}$$

```
import numpy as np
```

```
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# 垂直拼接（沿轴0）
vstacked = np.vstack([arr1, arr2])
print(f"垂直拼接:\n{vstacked}")
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]

# 水平拼接（沿轴1）
hstacked = np.hstack([arr1, arr2])
print(f"水平拼接:\n{hstacked}")
# [[1 2 5 6]
#  [3 4 7 8]]

# 使用 concatenate
concatenated = np.concatenate([arr1, arr2], axis=0)
print(f"沿轴0拼接:\n{concatenated}")

# 在 AI/ML 中的应用：合并特征
features1 = np.random.rand(100, 10)
features2 = np.random.rand(100, 5)
combined_features = np.hstack([features1, features2])
print(f"合并后的特征形状：{combined_features.shape}")  # (100, 15)
```

Listing 19: 数组拼接示例

11.4 数组运算

数学运算:

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
```

```python
# 基本运算（逐元素）
print(f"a + b = {a + b}")  # [6 8 10 12]
print(f"a - b = {a - b}")  # [-4 -4 -4 -4]
print(f"a * b = {a * b}")  # [5 12 21 32]（逐元素乘法）
print(f"a / b = {a / b}")  # [0.2 0.333... 0.428... 0.5]
print(f"a ** 2 = {a ** 2}")  # [1 4 9 16]

# 矩阵乘法
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
C = np.dot(A, B)  # 或 A @ B
print(f"矩阵乘法:\n{C}")

# 数学函数
arr = np.array([1, 4, 9, 16])
print(f"sqrt: {np.sqrt(arr)}")  # [1. 2. 3. 4.]
print(f"exp: {np.exp([1, 2, 3])}")  # [2.718... 7.389... 20.085...]
print(f"log: {np.log([1, np.e, np.e**2])}")  # [0. 1. 2.]
```

Listing 20: 数组数学运算示例

数学表达式演示:

逐元素运算：对于数组 $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ 和 $\mathbf{b} = [b_1, b_2, \ldots, b_n]$，逐元素运算定义为：

$$\mathbf{a} + \mathbf{b} = [a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n] \tag{108}$$

$$\mathbf{a} \odot \mathbf{b} = [a_1 \cdot b_1, a_2 \cdot b_2, \ldots, a_n \cdot b_n] \quad \text{（逐元素乘法，Hadamard 积）} \tag{109}$$

$$\mathbf{a}^2 = [a_1^2, a_2^2, \ldots, a_n^2] \tag{110}$$

$$\frac{\mathbf{a}}{\mathbf{b}} = \left[\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_n}{b_n}\right] \tag{111}$$

矩阵乘法：对于矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$ 和 $\mathbf{B} \in \mathbb{R}^{n \times p}$，矩阵乘法 $\mathbf{C} = \mathbf{AB}$ 定义为：

$$C_{ij} = (\mathbf{AB})_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}, \quad i = 1, \ldots, m, \quad j = 1, \ldots, p \tag{112}$$

结果矩阵 $\mathbf{C} \in \mathbb{R}^{m \times p}$。

示例：设 $\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$，$\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$，则：

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{113}$$

$$= \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix} \tag{114}$$

数学函数：对于数组 $\mathbf{x} = [x_1, x_2, \ldots, x_n]$，数学函数逐元素应用：

$$\sqrt{\mathbf{x}} = [\sqrt{x_1}, \sqrt{x_2}, \ldots, \sqrt{x_n}] \tag{115}$$

$$e^{\mathbf{x}} = [e^{x_1}, e^{x_2}, \ldots, e^{x_n}] \tag{116}$$

$$\log(\mathbf{x}) = [\log(x_1), \log(x_2), \ldots, \log(x_n)] \tag{117}$$

11.5 广播机制

概念解释：广播（Broadcasting）是 NumPy 对不同形状数组进行算术运算的机制。当数组形状不匹配时，NumPy 会自动扩展较小的数组以匹配较大数组的形状。

广播规则：

1. 如果两个数组的维度数不同，在较小数组的形状前面补 1
2. 如果两个数组在某个维度上的大小相同，或其中一个为 1，则可以广播
3. 广播后，数组在每个维度上的大小等于两个数组在该维度上的最大值

数学表达式演示：

示例 1：标量与数组

$$\begin{aligned} a &= 5 \quad \text{（标量）} \\ \mathbf{b} &= \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad \text{（3×1 数组）} \\ a + \mathbf{b} &= \begin{bmatrix} 5 \\ 5 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 7 \\ 8 \end{bmatrix} \end{aligned} \tag{118}$$

标量 a 被广播为与 $\mathbf{b}$ 相同形状的数组。

示例 2：不同形状的数组

$$\begin{aligned}\mathbf{A} &= \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \quad (2\times3 \text{ 数组})\\ \mathbf{b} &= \begin{bmatrix}10\\ 20\end{bmatrix} \quad (2\times1 \text{ 数组})\\ \mathbf{A}+\mathbf{b} &= \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10 & 10 & 10\\ 20 & 20 & 20\end{bmatrix} = \begin{bmatrix}11 & 12 & 13\\ 24 & 25 & 26\end{bmatrix}\end{aligned} \tag{119}$$

$\mathbf{b}$ 被广播为 (2×3) 形状：$\begin{bmatrix}10 & 10 & 10\\ 20 & 20 & 20\end{bmatrix}$

示例 3：更复杂的广播

$$\begin{aligned}\mathbf{A} &= \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} \quad (2\times3)\\ \mathbf{b} &= \begin{bmatrix}10 & 20 & 30\end{bmatrix} \quad (1\times3)\\ \mathbf{A}+\mathbf{b} &= \begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix} + \begin{bmatrix}10 & 20 & 30\\ 10 & 20 & 30\end{bmatrix} = \begin{bmatrix}11 & 22 & 33\\ 14 & 25 & 36\end{bmatrix}\end{aligned} \tag{120}$$

$\mathbf{b}$ 被广播为 (2×3) 形状。

```python
import numpy as np

# 示例1: 标量与数组
scalar = 5
arr = np.array([1, 2, 3])
result1 = scalar + arr
print(f"标量 + 数组: {result1}")  # [6 7 8]
print(f"广播后的形状: {(scalar + arr).shape}")  # (3,)

# 示例2: 不同形状的数组
A = np.array([[1, 2, 3],
              [4, 5, 6]])  # (2, 3)
b = np.array([[10],
              [20]])  # (2, 1)
result2 = A + b
print(f"数组 + 列向量:\n{result2}")
# [[11 12 13]
#  [24 25 26]]
```

```
# 示例3: 行向量广播
A = np.array([[1, 2, 3],
              [4, 5, 6]])  # (2, 3)
b = np.array([10, 20, 30])  # (3,)
result3 = A + b
print(f"数组 + 行向量:\n{result3}")
# [[11 22 33]
#  [14 25 36]]

# 示例4: 三维广播
A = np.random.rand(5, 3, 4)  # (5, 3, 4)
b = np.random.rand(3, 4)     # (3, 4)
result4 = A + b  # b 被广播为 (1, 3, 4), 然后为 (5, 3, 4)
print(f"三维广播形状: {result4.shape}")  # (5, 3, 4)

# 在 AI/ML 中的应用: 特征归一化
features = np.random.rand(100, 10)  # 100个样本, 10个特征
mean = features.mean(axis=0)  # 每个特征的均值, 形状 (10,)
std = features.std(axis=0)    # 每个特征的标准差, 形状 (10,)
normalized = (features - mean) / std  # 广播: features (100,10) - mean
    (10,)
print(f"归一化后的形状: {normalized.shape}")  # (100, 10)
print(f"归一化后均值: {normalized.mean(axis=0)}")  # 接近 [0, 0, ..., 0]
print(f"归一化后标准差: {normalized.std(axis=0)}")  # 接近 [1, 1, ..., 1]
```

Listing 21: 广播机制代码示例

广播机制的优势:

- **代码简洁**: 无需显式扩展数组维度
- **内存高效**: 不实际复制数据，只是虚拟扩展
- **计算高效**: 向量化操作，比循环快得多

常见陷阱:

- 广播失败: 当数组形状不兼容时会报错
- 意外的维度扩展: 可能导致意外的结果
- 性能问题: 虽然广播高效，但某些情况下显式操作可能更快

11.6 线性代数操作

矩阵乘法:

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# 矩阵乘法
C = np.dot(A, B)  # 或 A @ B
print(f"矩阵乘法:\n{C}")

# 矩阵转置
A_T = A.T
print(f"转置:\n{A_T}")

# 矩阵的逆
A_inv = np.linalg.inv(A)
print(f"逆矩阵:\n{A_inv}")
print(f"验证: A @ A_inv =\n{A @ A_inv}")  # 应该接近单位矩阵

# 特征值和特征向量
eigenvalues, eigenvectors = np.linalg.eig(A)
print(f"特征值: {eigenvalues}")
print(f"特征向量:\n{eigenvectors}")

# 奇异值分解 (SVD)
U, s, Vt = np.linalg.svd(A)
print(f"U:\n{U}")
print(f"奇异值: {s}")
print(f"Vt:\n{Vt}")

# 在 AI/ML 中的应用: 主成分分析 (PCA)
# 数据矩阵
X = np.random.rand(100, 10)
# 中心化
X_centered = X - X.mean(axis=0)
# 协方差矩阵
cov_matrix = np.cov(X_centered.T)
```

```
# 特征值分解
eigenvals, eigenvecs = np.linalg.eig(cov_matrix)
# 选择前k个主成分
k = 3
top_k_indices = eigenvals.argsort()[-k:][::-1]
principal_components = eigenvecs[:, top_k_indices]
# 投影
X_pca = X_centered @ principal_components
print(f"PCA 投影后的形状: {X_pca.shape}")  # (100, 3)
```

Listing 22: 线性代数操作示例

数学表达式演示:

矩阵的逆: 对于可逆方阵 $\mathbf{A} \in \mathbb{R}^{n \times n}$，其逆矩阵 $\mathbf{A}^{-1}$ 满足:

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_n \tag{121}$$

其中 $\mathbf{I}_n$ 是 $n \times n$ 单位矩阵。

特征值分解: 对于方阵 $\mathbf{A} \in \mathbb{R}^{n \times n}$，如果存在标量 λ（特征值）和非零向量 $\mathbf{v}$（特征向量）使得:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{122}$$

如果 $\mathbf{A}$ 有 n 个线性无关的特征向量，则可以分解为:

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} \tag{123}$$

其中，$\mathbf{V}$ 的列是特征向量，$\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$ 是对角矩阵。

奇异值分解（SVD）: 对于任意矩阵 $\mathbf{A} \in \mathbb{R}^{m \times n}$，可以分解为:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{124}$$

其中:

- $\mathbf{U} \in \mathbb{R}^{m \times m}$ 是左奇异向量矩阵（列向量正交）
- $\mathbf{\Sigma} \in \mathbb{R}^{m \times n}$ 是对角矩阵，对角线元素 $s_1 \geq s_2 \geq \cdots \geq s_{\min(m,n)} \geq 0$ 是奇异值
- $\mathbf{V} \in \mathbb{R}^{n \times n}$ 是右奇异向量矩阵（列向量正交）

协方差矩阵: 对于数据矩阵 $\mathbf{X} \in \mathbb{R}^{n \times d}$（$n$ 个样本，d 个特征），中心化后为 $\tilde{\mathbf{X}}$，协方差矩阵为:

$$\mathbf{C} = \frac{1}{n-1}\tilde{\mathbf{X}}^T\tilde{\mathbf{X}} \in \mathbb{R}^{d \times d} \tag{125}$$

其中 $C_{ij} = \text{Cov}(X_i, X_j)$ 表示第 i 个和第 j 个特征的协方差。

11.7 统计函数

```
import numpy as np

data = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 10, 11, 12]])

# 基本统计量
print(f"均值: {np.mean(data)}")  # 6.5
print(f"标准差: {np.std(data)}")  # 3.452...
print(f"方差: {np.var(data)}")  # 11.916...
print(f"总和: {np.sum(data)}")  # 78
print(f"最大值: {np.max(data)}")  # 12
print(f"最小值: {np.min(data)}")  # 1

# 沿轴计算
print(f"每列均值: {np.mean(data, axis=0)}")  # [5. 6. 7. 8.]
print(f"每行均值: {np.mean(data, axis=1)}")  # [2.5 6.5 10.5]

# 百分位数
print(f"中位数: {np.median(data)}")  # 6.5
print(f"25%分位数: {np.percentile(data, 25)}")  # 3.75
print(f"75%分位数: {np.percentile(data, 75)}")  # 9.25

# 在 AI/ML 中的应用: 数据预处理
features = np.random.randn(1000, 20)  # 1000个样本, 20个特征

# 计算每个特征的统计量
feature_means = np.mean(features, axis=0)
feature_stds = np.std(features, axis=0)
feature_mins = np.min(features, axis=0)
feature_maxs = np.max(features, axis=0)

print(f"特征均值范围: [{feature_means.min():.3f}, {feature_means.max():.3
    f}]")
print(f"特征标准差范围: [{feature_stds.min():.3f}, {feature_stds.max():.3
    f}]")

# 检测异常值 (使用3$\sigma$原则)
```

```
outliers = np.abs(features - feature_means) > 3 * feature_stds
outlier_count = np.sum(outliers, axis=0)
print(f"每个特征的异常值数量: {outlier_count}")
```

Listing 23: 统计函数示例

数学表达式演示:

对于数组 $\mathbf{x} = [x_1, x_2, \ldots, x_n]$，统计量定义为:

均值:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n} \tag{126}$$

方差（总体方差）:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^2 \tag{127}$$

样本方差（无偏估计）:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{128}$$

标准差:

$$\sigma = \sqrt{\sigma^2} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{129}$$

中位数：将数据排序后，中位数定义为:

$$\text{median}(\mathbf{x}) = \begin{cases} x_{((n+1)/2)} & \text{如果} n \text{ 为奇数} \\ \frac{x_{(n/2)} + x_{(n/2+1)}}{2} & \text{如果} n \text{ 为偶数} \end{cases} \tag{130}$$

其中 $x_{(i)}$ 表示排序后的第 i 个元素。

百分位数：第 p 百分位数是使得至少 $p\%$ 的数据小于等于该值的数。对于排序后的数据 $\mathbf{x}_{sorted}$:

$$\text{percentile}(\mathbf{x}, p) = x_{(\lceil np/100 \rceil)} \tag{131}$$

沿轴计算：对于矩阵 $\mathbf{X} \in \mathbb{R}^{m \times n}$，沿轴 0（列）的均值为:

$$\bar{\mathbf{x}}_{\text{axis}=0} = \left[\frac{1}{m}\sum_{i=1}^{m} X_{i1}, \frac{1}{m}\sum_{i=1}^{m} X_{i2}, \ldots, \frac{1}{m}\sum_{i=1}^{m} X_{in}\right] \tag{132}$$

沿轴 1（行）的均值为:

$$\bar{\mathbf{x}}_{\text{axis}=1} = \left[\frac{1}{n}\sum_{j=1}^{n} X_{1j}, \frac{1}{n}\sum_{j=1}^{n} X_{2j}, \ldots, \frac{1}{n}\sum_{j=1}^{n} X_{mj}\right]^T \tag{133}$$

12 总结

第一部分介绍了 Python 基础语法和 NumPy 核心功能：

Python 基础：

- 变量、数据类型、控制流
- 函数定义和面向对象编程
- 常用内置模块（os、json、datetime、collections）
- 文件操作和异常处理
- 高级特性（列表推导式、生成器、装饰器）

NumPy 核心：

- 数组创建和初始化
- 数组操作（索引、切片、重塑、拼接）
- 数组运算和广播机制（包含详细的数学表达式和代码示例）
- 线性代数操作（矩阵乘法、特征值分解、SVD）
- 统计函数

这些基础知识是使用 Pandas 和进行数据科学工作的前提。在第二部分中，我们将介绍 Pandas 数据处理和分析功能。

13 NumPy 综合例题

本节通过综合例题展示 NumPy 在实际问题中的应用。

例 13.1 (例题 1：数据预处理). 给定一个包含 *1000* 个样本、*20* 个特征的数据集，完成以下任务：

1. 计算每个特征的均值和标准差
2. 进行 *Z-score* 标准化（均值为 *0*，标准差为 *1*）
3. 检测并处理异常值（使用 3σ 原则）

4. 将数据重塑为适合输入神经网络的形状

解答:

```
import numpy as np

# 1. 生成模拟数据
np.random.seed(42)
data = np.random.randn(1000, 20)  # 1000个样本, 20个特征
# 添加一些异常值
data[100, 5] = 10  # 添加异常值
data[200, 10] = -8

print(f"原始数据形状: {data.shape}")
print(f"原始数据统计:\n均值范围: [{data.mean(axis=0).min():.3f}, {data.
    mean(axis=0).max():.3f}]")
print(f"标准差范围: [{data.std(axis=0).min():.3f}, {data.std(axis=0).max
    ():.3f}]")

# 2. 计算每个特征的均值和标准差
feature_means = np.mean(data, axis=0)  # 形状: (20,)
feature_stds = np.std(data, axis=0)    # 形状: (20,)

print(f"\n每个特征的均值:\n{feature_means}")
print(f"\n每个特征的标准差:\n{feature_stds}")

# 3. Z-score标准化: (x - mean) / std
# 利用广播机制: data (1000,20) - feature_means (20,) 自动广播
normalized_data = (data - feature_means) / feature_stds

print(f"\n标准化后均值: {normalized_data.mean(axis=0)[:5]}")  # 应该接近0
print(f"标准化后标准差: {normalized_data.std(axis=0)[:5]}")  # 应该接近1

# 4. 检测异常值 (3原则)
# 计算每个样本到均值的距离 (使用标准化后的数据)
z_scores = np.abs(normalized_data)
outliers_mask = z_scores > 3  # 超过3个标准差
outlier_indices = np.where(outliers_mask)[0]  # 获取异常值的位置

print(f"\n检测到 {len(np.unique(outlier_indices))} 个样本包含异常值")
```

```python
print(f"异常值位置: {np.unique(outlier_indices)[:10]}")

# 处理异常值: 可以用中位数替换, 或者删除
# 方法1: 用特征的中位数替换异常值
for i in range(data.shape[1]):
    feature_outliers = outliers_mask[:, i]
    if np.any(feature_outliers):
        median_value = np.median(data[:, i])
        data[feature_outliers, i] = median_value

# 重新标准化处理后的数据
feature_means_new = np.mean(data, axis=0)
feature_stds_new = np.std(data, axis=0)
normalized_data_clean = (data - feature_means_new) / feature_stds_new

# 5. 重塑数据 (例如: 为CNN准备, 将每个样本重塑为4x5)
# 注意: 20 = 4 * 5
reshaped_data = normalized_data_clean.reshape(1000, 4, 5)
print(f"\n重塑后的数据形状: {reshaped_data.shape}")  # (1000, 4, 5)

# 或者展平为适合全连接网络的形状 (已经是扁平的了)
flattened_data = normalized_data_clean.flatten().reshape(1000, -1)
print(f"展平后的数据形状: {flattened_data.shape}")  # (1000, 20)
```

Listing 24: 数据预处理完整示例

例 13.2 (例题 2: 矩阵运算与线性代数). 给定矩阵 $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ 和向量 $\mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$, 完成以下任务:

1. 计算矩阵乘法 $\mathbf{Ab}$
2. 计算 $\mathbf{A}$ 的转置 $\mathbf{A}^T$
3. 计算 $\mathbf{A}^T\mathbf{A}$ 的特征值和特征向量
4. 对 $\mathbf{A}$ 进行 *SVD* 分解

解答:

数学计算:

1. 矩阵乘法:

$$\mathbf{Ab} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 \\ 4 \cdot 1 + 5 \cdot 2 + 6 \cdot 3 \end{bmatrix} = \begin{bmatrix} 14 \\ 32 \end{bmatrix} \tag{134}$$

2. 转置:

$$\mathbf{A}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \tag{135}$$

3. $\mathbf{A}^T\mathbf{A}$:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 17 & 22 & 27 \\ 22 & 29 & 36 \\ 27 & 36 & 45 \end{bmatrix} \tag{136}$$

```python
import numpy as np

# 定义矩阵和向量
A = np.array([[1, 2, 3],
              [4, 5, 6]])  # (2, 3)
b = np.array([[1],
              [2],
              [3]])  # (3, 1)

print("矩阵 A:")
print(A)
print("\n向量 b:")
print(b)

# 1. 矩阵乘法 A @ b
result = A @ b  # 或 np.dot(A, b)
print(f"\n1. A @ b = \n{result}")
# [[14]
#  [32]]

# 2. 转置
A_T = A.T
print(f"\n2. A^T = \n{A_T}")
# [[1 4]
#  [2 5]
```

```
#  [3 6]]

# 3. A^T @ A 的特征值和特征向量
ATA = A_T @ A
print(f"\n3. A^T @ A = \n{ATA}")

eigenvals, eigenvecs = np.linalg.eig(ATA)
print(f"\n特征值: {eigenvals}")
print(f"特征向量:\n{eigenvecs}")

# 验证: A^T @ A @ v = lambda * v
for i in range(len(eigenvals)):
    v = eigenvecs[:, i].reshape(-1, 1)
    lambda_v = eigenvals[i] * v
    Av = ATA @ v
    print(f"\n验证特征值 {eigenvals[i]:.6f}:")
    print(f"ATA @ v = \n{Av.flatten()}")
    print(f"lambda * v = \n{lambda_v.flatten()}")
    print(f"误差: {np.abs(Av - lambda_v).max():.10f}")

# 4. SVD分解
U, s, Vt = np.linalg.svd(A, full_matrices=False)
print(f"\n4. SVD分解:")
print(f"U:\n{U}")
print(f"奇异值 s: {s}")
print(f"V^T:\n{Vt}")

# 重构矩阵验证
Sigma = np.diag(s)
A_reconstructed = U @ Sigma @ Vt
print(f"\n重构的矩阵 A:\n{A_reconstructed}")
print(f"重构误差: {np.abs(A - A_reconstructed).max():.10f}")
```

Listing 25: 矩阵运算与线性代数示例

例 13.3 (例题 3：统计分析与数据探索). 给定一个包含学生成绩的数据集，完成以下分析：

1. 计算各科目的平均分、标准差、最高分、最低分
2. 找出总分最高的学生

3. 计算各科目的及格率（60 分以上）

4. 找出所有科目都及格的学生

解答：

```
import numpy as np

# 生成模拟数据：50个学生，5门课程
np.random.seed(42)
scores = np.random.randint(40, 100, size=(50, 5))  # 50个学生，5门课程
course_names = ['数学', '语文', '英语', '物理', '化学']

print(f"成绩数据形状：{scores.shape}")
print(f"前5个学生的成绩：\n{scores[:5]}")

# 1. 计算各科目的统计量
print("\n=== 各科目统计信息 ===")
for i, course in enumerate(course_names):
    course_scores = scores[:, i]
    mean_score = np.mean(course_scores)
    std_score = np.std(course_scores)
    max_score = np.max(course_scores)
    min_score = np.min(course_scores)
    median_score = np.median(course_scores)

    print(f"\n{course}:")
    print(f"  平均分：{mean_score:.2f}")
    print(f"  标准差：{std_score:.2f}")
    print(f"  最高分：{max_score}")
    print(f"  最低分：{min_score}")
    print(f"  中位数：{median_score:.2f}")

# 使用向量化操作一次性计算所有科目的统计量
means = np.mean(scores, axis=0)
stds = np.std(scores, axis=0)
maxs = np.max(scores, axis=0)
mins = np.min(scores, axis=0)
medians = np.median(scores, axis=0)

print("\n=== 向量化计算所有科目统计量 ===")
```

```
stats = np.array([means, stds, maxs, mins, medians])
print("统计量矩阵（行：均值、标准差、最高、最低、中位数）:")
print(stats)

# 2. 计算总分并找出最高分学生
total_scores = np.sum(scores, axis=1)  # 每个学生的总分
top_student_idx = np.argmax(total_scores)
print(f"\n=== 总分分析 ===")
print(f"最高总分: {total_scores[top_student_idx]}")
print(f"最高分学生索引: {top_student_idx}")
print(f"该学生各科成绩: {scores[top_student_idx]}")
print(f"该学生平均分: {np.mean(scores[top_student_idx]):.2f}")

# 3. 计算各科目的及格率（60分以上）
pass_threshold = 60
pass_rates = np.mean(scores >= pass_threshold, axis=0) * 100

print(f"\n=== 及格率分析（{pass_threshold}分）===")
for i, course in enumerate(course_names):
    print(f"{course}: {pass_rates[i]:.1f}%")

# 4. 找出所有科目都及格的学生
all_passed = np.all(scores >= pass_threshold, axis=1)
passed_students = np.where(all_passed)[0]
print(f"\n=== 全部及格学生 ===")
print(f"全部及格的学生数量: {len(passed_students)}")
print(f"全部及格的学生索引: {passed_students[:10]}")  # 显示前10个

# 统计信息
print(f"\n全部及格学生的平均总分: {np.mean(total_scores[all_passed]):.2f}
    ")
print(f"全部及格学生的各科平均分:")
for i, course in enumerate(course_names):
    avg = np.mean(scores[all_passed, i])
    print(f"  {course}: {avg:.2f}")
```

Listing 26: 统计分析示例

例 13.4 (例题 4: 广播机制的实际应用). 实现批量归一化（*Batch Normalization*）操作，这是深度学习中常用的技术。

要求:

1. 给定一个批次的数据 $\mathbf{X} \in \mathbb{R}^{B \times D}$（$B$ 个样本，D 个特征）
2. 对每个特征进行归一化：$\hat{x}_i = \frac{x_i - \mu}{\sigma}$
3. 应用缩放和平移：$y_i = \gamma \hat{x}_i + \beta$

解答:

数学表达式:

对于批次数据 $\mathbf{X} \in \mathbb{R}^{B \times D}$，批量归一化定义为：

1. 计算每个特征的均值和方差：

$$\mu_j = \frac{1}{B} \sum_{i=1}^{B} X_{ij}, \quad j = 1, \ldots, D \tag{137}$$

$$\sigma_j^2 = \frac{1}{B} \sum_{i=1}^{B} (X_{ij} - \mu_j)^2, \quad j = 1, \ldots, D \tag{138}$$

2. 归一化：

$$\hat{X}_{ij} = \frac{X_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \tag{139}$$

其中 ϵ 是小的常数（如 10^{-5}）防止除零。

3. 缩放和平移：

$$Y_{ij} = \gamma_j \hat{X}_{ij} + \beta_j \tag{140}$$

其中 γ_j 和 β_j 是可学习的参数。

```
import numpy as np

def batch_normalize(X, gamma=1.0, beta=0.0, epsilon=1e-5):
    """
    批量归一化实现

    参数:
        X: 输入数据 (batch_size, num_features)
        gamma: 缩放参数，形状 (num_features,)
        beta: 平移参数，形状 (num_features,)
        epsilon: 防止除零的小常数

    返回:
```

```
        Y: 归一化后的数据
        mean: 每个特征的均值
        std: 每个特征的标准差
    """
    # 1. 计算每个特征的均值和方差（沿batch维度）
    # X shape: (B, D)
    # mean shape: (D,) - 利用广播
    mean = np.mean(X, axis=0)  # 沿轴0（batch维度）求均值
    var = np.var(X, axis=0)      # 沿轴0求方差
    std = np.sqrt(var + epsilon)

    # 2. 归一化: (X - mean) / std
    # X (B, D) - mean (D,) 自动广播
    X_normalized = (X - mean) / std

    # 3. 缩放和平移
    # 如果 gamma 和 beta 是标量，会自动广播
    # 如果是数组，形状应该是 (D,)
    if np.isscalar(gamma):
        gamma = np.ones(X.shape[1]) * gamma
    if np.isscalar(beta):
        beta = np.zeros(X.shape[1]) + beta

    Y = gamma * X_normalized + beta

    return Y, mean, std

# 测试
np.random.seed(42)
batch_size = 32
num_features = 10

# 生成模拟数据（不同特征的分布不同）
X = np.random.randn(batch_size, num_features)
# 让不同特征有不同的均值和方差
for i in range(num_features):
    X[:, i] = X[:, i] * (i + 1) + i * 2

print("原始数据统计:")
print(f"形状: {X.shape}")
```

```
print(f"各特征均值: {np.mean(X, axis=0)}")
print(f"各特征标准差: {np.std(X, axis=0)}")

# 应用批量归一化
gamma = np.ones(num_features)  # 缩放参数
beta = np.zeros(num_features)  # 平移参数
Y, mean, std = batch_normalize(X, gamma, beta)

print("\n归一化后数据统计:")
print(f"各特征均值: {np.mean(Y, axis=0)}")  # 应该接近0
print(f"各特征标准差: {np.std(Y, axis=0)}")  # 应该接近1

# 验证: 手动计算第一个特征的归一化
first_feature = X[:, 0]
manual_mean = np.mean(first_feature)
manual_std = np.std(first_feature)
manual_normalized = (first_feature - manual_mean) / manual_std
print(f"\n验证第一个特征:")
print(f"手动计算归一化均值: {np.mean(manual_normalized):.10f}")
print(f"函数计算归一化均值: {np.mean(Y[:, 0]):.10f}")
print(f"差异: {np.abs(np.mean(manual_normalized) - np.mean(Y[:, 0])):.10f
    }")
```

Listing 27: 批量归一化实现

这些例题展示了 NumPy 在实际数据科学和机器学习任务中的应用，包括数据预处理、线性代数运算、统计分析和批量归一化等重要操作。

14 Pandas 基础

14.1 Pandas 简介

概念解释：Pandas 是 Python 中最重要的数据处理和分析库，提供了强大的数据结构和数据分析工具，特别适合处理结构化数据（如 CSV、Excel、数据库等）。

核心数据结构：

- **Series**：一维带标签数组，类似于带索引的 NumPy 数组
- **DataFrame**：二维表格型数据结构，类似于 Excel 表格或 SQL 表

数学表示：

对于 Series，可以表示为：

$$\mathbf{s} = [s_1, s_2, \ldots, s_n]^T \quad \text{带索引} \{i_1, i_2, \ldots, i_n\} \tag{141}$$

对于 DataFrame，可以表示为矩阵：

$$\mathbf{D} = \begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1m} \\ d_{21} & d_{22} & \cdots & d_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ d_{n1} & d_{n2} & \cdots & d_{nm} \end{bmatrix} \tag{142}$$

其中行索引为 $\{r_1, r_2, \ldots, r_n\}$，列索引为 $\{c_1, c_2, \ldots, c_m\}$。

14.2 Series 基础

概念解释：Series 是一维数据结构，由数据和索引组成。

创建 Series：

```python
import pandas as pd
import numpy as np

# 从列表创建
data = [10, 20, 30, 40, 50]
s1 = pd.Series(data)
print(s1)
# 0    10
# 1    20
# 2    30
# 3    40
# 4    50
# dtype: int64

# 指定索引
s2 = pd.Series(data, index=['a', 'b', 'c', 'd', 'e'])
print(s2)
# a    10
# b    20
# c    30
# d    40
```

```
# e    50
# dtype: int64

# 从字典创建
s3 = pd.Series({'Alice': 25, 'Bob': 30, 'Charlie': 35})
print(s3)
# Alice      25
# Bob        30
# Charlie    35
# dtype: int64

# 从 NumPy 数组创建
arr = np.array([1.5, 2.5, 3.5, 4.5])
s4 = pd.Series(arr, index=['x', 'y', 'z', 'w'])
print(s4)
```

Listing 28: Series 创建示例

Series 操作:

```
s = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])

# 访问元素
print(s['a'])  # 10
print(s[0])    # 10
print(s[['a', 'c', 'e']])  # 选择多个元素

# 切片
print(s['a':'c'])  # 包含两端

# 数学运算
print(s * 2)  # 每个元素乘以2
print(s + 10)  # 每个元素加10
print(s ** 2)  # 每个元素平方

# 统计函数
print(s.mean())  # 30.0
print(s.std())   # 15.81...
print(s.sum())   # 150
print(s.max())   # 50
```

```
print(s.min())    # 10
```

Listing 29: Series 基本操作

14.3 DataFrame 基础

概念解释：DataFrame 是二维表格型数据结构，是最常用的 Pandas 数据结构。

创建 DataFrame：

```
import pandas as pd
import numpy as np

# 从字典创建
data = {
    '姓名': ['Alice', 'Bob', 'Charlie', 'David'],
    '年龄': [25, 30, 35, 28],
    '城市': ['北京', '上海', '广州', '深圳'],
    '薪资': [8000, 12000, 15000, 10000]
}
df = pd.DataFrame(data)
print(df)

# 从列表的列表创建
data_list = [
    ['Alice', 25, '北京', 8000],
    ['Bob', 30, '上海', 12000],
    ['Charlie', 35, '广州', 15000],
    ['David', 28, '深圳', 10000]
]
df2 = pd.DataFrame(data_list, columns=['姓名', '年龄', '城市', '薪资'])

# 从 NumPy 数组创建
arr = np.random.randn(5, 4)
df3 = pd.DataFrame(arr, columns=['A', 'B', 'C', 'D'])

# 从 CSV 文件读取
df4 = pd.read_csv('data.csv')
```

Listing 30: DataFrame 创建示例

DataFrame 基本操作：

```
# 查看数据
print(df.head())  # 前5行
print(df.tail())  # 后5行
print(df.shape)   # (4, 4) - 形状
print(df.info())  # 数据信息
print(df.describe())  # 统计摘要

# 访问列
print(df['姓名'])  # 单列，返回 Series
print(df[['姓名', '年龄']])  # 多列，返回 DataFrame

# 访问行
print(df.iloc[0])  # 按位置访问第一行
print(df.loc[0])   # 按索引访问第一行
print(df.iloc[0:2])  # 前两行

# 条件筛选
print(df[df['年龄'] > 28])  # 年龄大于28的行
print(df[(df['年龄'] > 28) & (df['薪资'] > 10000)])  # 多条件

# 添加列
df['奖金'] = df['薪资'] * 0.1
df['总薪资'] = df['薪资'] + df['奖金']

# 删除列
df = df.drop('奖金', axis=1)  # 删除列
df = df.drop(0, axis=0)  # 删除行
```

Listing 31: DataFrame 基本操作

14.4 数据读取与写入

CSV 文件：

```
# 读取 CSV
df = pd.read_csv('data.csv', encoding='utf-8')
df = pd.read_csv('data.csv', sep=',', header=0, index_col=0)
```

```
# 写入 CSV
df.to_csv('output.csv', index=False, encoding='utf-8')

# 读取时指定参数
df = pd.read_csv('data.csv',
                 sep=',',           # 分隔符
                 header=0,          # 表头行
                 index_col=0,       # 索引列
                 skiprows=1,        # 跳过行数
                 nrows=1000,        # 读取行数
                 na_values=['NA', 'N/A'])  # 缺失值标识
```

Listing 32: CSV 文件读写

Excel 文件:

```
# 读取 Excel
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# 写入 Excel
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

Listing 33: Excel 文件读写

JSON 文件:

```
# 读取 JSON
df = pd.read_json('data.json')

# 写入 JSON
df.to_json('output.json', orient='records', force_ascii=False)
```

Listing 34: JSON 文件读写

14.5 数据查看与选择

查看数据:

```
# 基本信息
print(df.shape)          # (行数, 列数)
print(df.columns)        # 列名
```

```
print(df.index)          # 索引
print(df.dtypes)         # 数据类型
print(df.info())         # 详细信息

# 统计摘要
print(df.describe())     # 数值列的统计信息
print(df.describe(include='all'))  # 所有列的统计信息

# 查看数据
print(df.head(10))       # 前10行
print(df.tail(10))       # 后10行
print(df.sample(5))      # 随机5行
```

Listing 35: 数据查看方法

选择数据:

```
# 选择列
df['列名']               # 单列，返回 Series
df[['列1', '列2']]       # 多列，返回 DataFrame

# 选择行（按位置）
df.iloc[0]               # 第一行
df.iloc[0:5]             # 前5行
df.iloc[0:5, 0:3]        # 前5行，前3列

# 选择行（按索引）
df.loc['索引值']         # 按索引选择行
df.loc['索引1':'索引5'] # 索引范围

# 条件筛选
df[df['年龄'] > 30]                      # 单条件
df[(df['年龄'] > 30) & (df['薪资'] > 10000)]  # 多条件
df[df['城市'].isin(['北京', '上海'])]  # 包含在列表中

# 使用 query 方法
df.query('年龄 > 30 and 薪资 > 10000')
```

Listing 36: 数据选择方法

14.6 数据清洗

处理缺失值:

```
# 检测缺失值
print(df.isnull())          # 返回布尔 DataFrame
print(df.isnull().sum())    # 每列缺失值数量
print(df.isnull().any())    # 每列是否有缺失值

# 删除缺失值
df.dropna()                 # 删除包含缺失值的行
df.dropna(axis=1)           # 删除包含缺失值的列
df.dropna(subset=['列名']) # 删除指定列有缺失值的行

# 填充缺失值
df.fillna(0)                # 用0填充
df.fillna(df.mean())        # 用均值填充
df.fillna(method='ffill') # 前向填充
df.fillna(method='bfill') # 后向填充
df['列名'].fillna(df['列名'].median())  # 用中位数填充
```

Listing 37: 缺失值处理

处理重复值:

```
# 检测重复值
print(df.duplicated())     # 返回布尔 Series
print(df.duplicated().sum())  # 重复行数量

# 删除重复值
df.drop_duplicates()       # 删除完全重复的行
df.drop_duplicates(subset=['列1', '列2'])  # 基于指定列删除重复
```

Listing 38: 重复值处理

数据类型转换:

```
# 转换数据类型
df['列名'] = df['列名'].astype(int)
df['列名'] = df['列名'].astype(float)
df['列名'] = df['列名'].astype(str)
```

```
# 转换为日期时间
df['日期'] = pd.to_datetime(df['日期列'])

# 转换分类类型
df['类别'] = df['类别'].astype('category')
```

Listing 39: 数据类型转换

14.7 数据转换

添加和删除列:

```
# 添加列
df['新列'] = df['列1'] + df['列2']  # 基于现有列计算
df['新列'] = 100  # 常量值
df['新列'] = np.random.randn(len(df))  # 随机值

# 使用 assign 方法
df = df.assign(新列1=df['列1']*2, 新列2=df['列2']**2)

# 删除列
df = df.drop('列名', axis=1)
df = df.drop(['列1', '列2'], axis=1)

# 删除行
df = df.drop(索引值, axis=0)
df = df.drop([索引1, 索引2], axis=0)
```

Listing 40: 列操作

重命名:

```
# 重命名列
df = df.rename(columns={'旧名': '新名'})
df.columns = ['新列1', '新列2', '新列3']  # 重命名所有列

# 重命名索引
df = df.rename(index={旧索引: 新索引})
df.index = ['新索引1', '新索引2', ...]  # 重命名所有索引
```

Listing 41: 重命名操作

排序:

```
# 按列排序
df.sort_values('列名')                          # 升序
df.sort_values('列名', ascending=False)          # 降序
df.sort_values(['列1', '列2'])                   # 多列排序

# 按索引排序
df.sort_index()                                 # 按索引排序
```

Listing 42: 排序操作

14.8 分组与聚合

分组操作:

```
# 基本分组
grouped = df.groupby('城市')
print(grouped.groups)  # 查看分组

# 分组聚合
df.groupby('城市')['薪资'].mean()          # 按城市分组，计算平均薪资
df.groupby('城市')['薪资'].sum()           # 求和
df.groupby('城市')['薪资'].count()         # 计数
df.groupby('城市')['薪资'].std()           # 标准差

# 多列分组
df.groupby(['城市', '部门'])['薪资'].mean()

# 多个聚合函数
df.groupby('城市')['薪资'].agg(['mean', 'std', 'min', 'max'])

# 自定义聚合函数
def custom_agg(x):
    return x.max() - x.min()

df.groupby('城市')['薪资'].agg(custom_agg)
```

Listing 43: 分组操作示例

数学表示：

对于分组聚合，数学上可以表示为：

$$\bar{x}_g = \frac{1}{|G_g|} \sum_{i \in G_g} x_i \tag{143}$$

其中 G_g 是分组 g 的索引集合，$|G_g|$ 是该组的样本数。

14.9 数据合并

合并操作：

```
# 内连接（inner join）
result = pd.merge(df1, df2, on='键列', how='inner')

# 左连接（left join）
result = pd.merge(df1, df2, on='键列', how='left')

# 右连接（right join）
result = pd.merge(df1, df2, on='键列', how='right')

# 外连接（outer join）
result = pd.merge(df1, df2, on='键列', how='outer')

# 多个键合并
result = pd.merge(df1, df2, on=['键1', '键2'])

# 不同列名合并
result = pd.merge(df1, df2, left_on='列1', right_on='列2')

# 纵向拼接
result = pd.concat([df1, df2], axis=0)  # 垂直拼接
result = pd.concat([df1, df2], axis=1)  # 水平拼接

# 忽略索引
result = pd.concat([df1, df2], ignore_index=True)
```

Listing 44: 数据合并示例

数学表示：

对于内连接，结果集为：

$$R = \{r \in R_1 \times R_2 : r.\text{key}_1 = r.\text{key}_2\} \tag{144}$$

14.10 时间序列处理

时间序列基础：

```
# 创建时间索引
dates = pd.date_range('2024-01-01', periods=100, freq='D')
df = pd.DataFrame(np.random.randn(100, 4), index=dates, columns=['A', 'B'
    , 'C', 'D'])

# 时间索引操作
df['年'] = df.index.year
df['月'] = df.index.month
df['日'] = df.index.day
df['星期'] = df.index.dayofweek

# 重采样
df.resample('W').mean()    # 按周重采样，计算均值
df.resample('M').sum()    # 按月重采样，求和
df.resample('Q').mean()   # 按季度重采样

# 时间窗口
df.rolling(window=7).mean()    # 7天移动平均
df.rolling(window=30).std()    # 30天移动标准差
```

Listing 45: 时间序列处理

14.11 透视表

概念解释：透视表（Pivot Table）是一种数据汇总工具，可以按照多个维度对数据进行分组和聚合。

数学表示：

透视表可以表示为：

$$P_{ij} = \text{agg}(\{d_k : d_k.\text{row}_i = r_i \text{ and } d_k.\text{col}_j = c_j\}) \tag{145}$$

其中 agg 是聚合函数（如 sum、mean 等）。

```
# 基本透视表
df.pivot_table(values='薪资', index='城市', columns='部门', aggfunc='mean
    ')

# 多个值列
df.pivot_table(values=['薪资', '奖金'], index='城市', columns='部门',
    aggfunc='mean')

# 多个聚合函数
df.pivot_table(values='薪资', index='城市', columns='部门',
                aggfunc=['mean', 'std', 'count'])

# 填充缺失值
df.pivot_table(values='薪资', index='城市', columns='部门',
                aggfunc='mean', fill_value=0)
```

Listing 46: 透视表示例

14.12 Pandas 在 AI/ML 中的应用

数据预处理流程：

```
import pandas as pd
import numpy as np

# 1. 读取数据
df = pd.read_csv('dataset.csv')

# 2. 探索性数据分析
print(df.info())
print(df.describe())
print(df.isnull().sum())
```

```
# 3. 处理缺失值
df = df.fillna(df.mean())  # 数值列用均值填充
df = df.fillna(df.mode().iloc[0])  # 分类列用众数填充

# 4. 处理异常值
Q1 = df['数值列'].quantile(0.25)
Q3 = df['数值列'].quantile(0.75)
IQR = Q3 - Q1
df = df[(df['数值列'] >= Q1 - 1.5*IQR) & (df['数值列'] <= Q3 + 1.5*IQR)]

# 5. 特征工程
df['新特征'] = df['特征1'] * df['特征2']
df['类别编码'] = pd.Categorical(df['类别']).codes

# 6. 转换为 NumPy 数组用于机器学习
X = df[['特征1', '特征2', '特征3']].values
y = df['标签'].values
```

Listing 47: 完整的数据预处理流程

特征工程示例:

```
# 创建交互特征
df['特征1_特征2'] = df['特征1'] * df['特征2']
df['特征1_平方'] = df['特征1'] ** 2

# 分箱（Binning）
df['年龄组'] = pd.cut(df['年龄'], bins=[0, 30, 50, 100], labels=['青年',
    '中年', '老年'])

# 独热编码
df_encoded = pd.get_dummies(df, columns=['城市', '部门'])

# 标签编码
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df['城市编码'] = le.fit_transform(df['城市'])

# 时间特征提取
df['日期'] = pd.to_datetime(df['日期列'])
```

```
df['年'] = df['日期'].dt.year
df['月'] = df['日期'].dt.month
df['星期'] = df['日期'].dt.dayofweek
```

Listing 48: 特征工程示例

15 Pandas 综合例题

例 15.1 (例题 1：数据清洗与预处理). 给定一个包含学生信息的 *CSV* 文件，完成以下任务：

1. 读取数据并查看基本信息
2. 处理缺失值
3. 删除重复记录
4. 添加计算列（如总分、平均分）
5. 按班级分组统计

```
import pandas as pd
import numpy as np

# 读取数据
df = pd.read_csv('students.csv')

# 查看基本信息
print("数据形状:", df.shape)
print("缺失值统计:")
print(df.isnull().sum())
print("\n数据前5行:")
print(df.head())

# 处理缺失值
df['数学'] = df['数学'].fillna(df['数学'].mean())
df['英语'] = df['英语'].fillna(df['英语'].median())
df['班级'] = df['班级'].fillna('未知')

# 删除重复记录
df = df.drop_duplicates()
```

```
# 添加计算列
df['总分'] = df['数学'] + df['英语'] + df['语文']
df['平均分'] = df['总分'] / 3
df['等级'] = pd.cut(df['平均分'],
                    bins=[0, 60, 80, 100],
                    labels=['不及格', '良好', '优秀'])

# 按班级分组统计
class_stats = df.groupby('班级').agg({
    '总分': ['mean', 'std', 'min', 'max'],
    '数学': 'mean',
    '英语': 'mean',
    '语文': 'mean'
})

print("\n班级统计:")
print(class_stats)
```

Listing 49: 数据清洗完整示例

例 15.2 (例题 2: 数据合并与分析). 合并两个数据表，并进行综合分析:

```
# 学生信息表
students = pd.DataFrame({
    '学号': [1, 2, 3, 4, 5],
    '姓名': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    '班级': ['A', 'A', 'B', 'B', 'A']
})

# 成绩表
scores = pd.DataFrame({
    '学号': [1, 2, 3, 4, 5, 1, 2, 3],
    '科目': ['数学', '数学', '数学', '数学', '数学', '英语', '英语', '英
    语'],
    '分数': [85, 90, 75, 88, 92, 80, 85, 70]
})

# 合并数据
merged = pd.merge(students, scores, on='学号', how='inner')
```

```python
# 透视表分析
pivot = merged.pivot_table(values='分数', index='姓名', columns='科目',
    aggfunc='mean')
print("学生各科平均分:")
print(pivot)

# 班级平均分
class_avg = merged.groupby(['班级', '科目'])['分数'].mean().unstack()
print("\n班级各科平均分:")
print(class_avg)
```

Listing 50: 数据合并与分析示例

16 总结

第二部分介绍了 Pandas 数据处理和分析的核心功能：

Pandas 核心功能：

- **数据结构**：Series 和 DataFrame 的创建和操作
- **数据读取**：CSV、Excel、JSON 等格式的读写
- **数据查看**：查看数据基本信息、统计摘要
- **数据选择**：按列、按行、按条件选择数据
- **数据清洗**：处理缺失值、重复值、数据类型转换
- **数据转换**：添加删除列、重命名、排序
- **分组聚合**：groupby 操作和聚合函数
- **数据合并**：merge 和 concat 操作
- **时间序列**：时间索引、重采样、移动窗口
- **透视表**：多维度数据汇总

在 AI/ML 中的应用：

- 数据预处理和清洗

- 特征工程
- 数据探索性分析
- 数据格式转换（Pandas DataFrame → NumPy 数组）

Python、NumPy 和 Pandas 构成了数据科学和机器学习的核心工具链，掌握这些工具是进行 AI/ML 研究和应用的基础。

Part III

第三部分：机器学习

17 引言

机器学习（Machine Learning, ML）是人工智能的核心分支，通过从数据中自动学习模式和规律，使计算机系统能够完成特定任务而无需显式编程。机器学习在图像识别、自然语言处理、推荐系统、医疗诊断、金融风控等领域取得了显著成功。

机器学习的独特价值：

- **处理高维特征**：能够处理包含数千甚至数万个特征的数据，自动发现重要特征
- **捕捉非线性关系**：通过非线性模型（如神经网络、核方法）捕捉数据中的复杂模式
- **自适应学习**：能够从新数据中持续学习，适应数据分布的变化
- **端到端学习**：直接从原始数据学习到最终输出，减少人工特征工程的需求

本文档系统性地介绍机器学习的核心理论、经典算法和实际应用，涵盖监督学习、无监督学习、强化学习等主要范式，以及特征工程、模型评估、集成学习等重要技术。

18 机器学习基础

机器学习根据学习方式可以分为三大类：监督学习、无监督学习和强化学习。每种范式适用于不同类型的问题和场景。

18.1 监督学习

定义 18.1 (监督学习). 监督学习（*Supervised Learning*）是从带标签的训练数据中学习一个映射函数 $f : \mathcal{X} \to \mathcal{Y}$，使得对于新的输入 $\mathbf{x} \in \mathcal{X}$，能够预测其标签 $y \in \mathcal{Y}$。

给定训练数据集 $\mathcal{D} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$，其中 $\mathbf{x}_i \in \mathbb{R}^d$ 是特征向量，y_i 是对应的标签，目标是学习函数 f 使得 $f(\mathbf{x}_i) \approx y_i$。

通俗解释：监督学习就像有老师指导的学习。老师提供大量” 题目-答案” 对（训练数据），学生（模型）通过学习这些例子，掌握解题方法，然后能够解答新的题目。

18.1.1 分类问题

分类问题的目标是预测离散的类别标签。根据类别数量，可以分为：

- **二分类**：只有两个类别，如垃圾邮件分类（垃圾/正常）、疾病诊断（患病/健康）
- **多分类**：有多个类别，如图像分类（猫/狗/鸟等）、手写数字识别（0-9）

典型应用：

例 18.1 (垃圾邮件分类)**. 问题**：自动识别邮件是否为垃圾邮件。

数据：

- 特征：邮件内容中的词频、发件人信息、邮件标题等
- 标签：垃圾邮件（*1*）或正常邮件（*0*）

方法：使用朴素贝叶斯、逻辑回归或支持向量机等分类算法。

实际应用：*Gmail* 使用机器学习模型过滤垃圾邮件，准确率超过 *99%*。

18.1.2 回归问题

回归问题的目标是预测连续的数值。

典型应用：

例 18.2 (房价预测)**. 问题**：根据房屋特征预测房价。

数据：

- 特征：面积、卧室数、位置、建造年份等
- 标签：房价（连续值）

方法：使用线性回归、决策树回归或神经网络等算法。

实际应用：*Zillow* 的 *Zestimate* 使用机器学习模型预测房价，帮助用户了解房产价值。

18.1.3 监督学习的优势与局限性

优势：

- 有明确的优化目标（最小化预测误差）

- 可以使用大量标注数据进行训练
- 模型性能可以通过测试集准确评估
- 理论基础相对完善

局限性:

- 需要大量标注数据，标注成本高
- 对标注质量要求高，错误标注会影响模型性能
- 难以处理标注数据稀缺的场景
- 模型可能过拟合训练数据

18.2 无监督学习

定义 18.2 (无监督学习)**.** 无监督学习（*Unsupervised Learning*）是从无标签的数据中学习数据的内在结构和模式，不需要人工标注。

给定数据集 $\mathcal{D} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$，其中只有特征向量，没有标签，目标是发现数据中的隐藏模式、结构或分布。

通俗解释：无监督学习就像没有老师指导的自学。学生只能看到大量数据，需要自己发现其中的规律和模式，比如哪些数据相似、数据如何分组等。

18.2.1 聚类

聚类是将相似的数据点分组的过程。

典型应用:

例 18.3 (客户细分)**. 问题**：根据客户的购买行为将其分为不同的群体。

数据：客户的购买历史、浏览记录、消费金额等特征（无标签）。

方法：使用 *K-means*、层次聚类等算法将客户分为若干群体。

实际应用：电商平台使用聚类分析进行客户细分，为不同群体提供个性化推荐和营销策略。

18.2.2 降维

降维是将高维数据映射到低维空间，保留主要信息。

典型应用：

例 18.4 (数据可视化). **问题**：将高维数据（如 *1000* 维）可视化到 *2D* 或 *3D* 空间。

方法：使用 *PCA*（主成分分析)、*t-SNE*、*UMAP* 等降维算法。

实际应用：在生物信息学中，使用 *t-SNE* 将基因表达数据降维到 *2D*，可视化不同细胞类型的分布。

18.2.3 无监督学习的优势与局限性

优势：

- 不需要标注数据，数据获取成本低
- 可以发现人类未预见的模式和结构
- 适用于探索性数据分析
- 可以作为监督学习的预处理步骤

局限性：

- 没有明确的优化目标，评估困难
- 结果可能难以解释和验证
- 对参数选择敏感（如聚类数量）
- 可能发现无意义的模式

18.3 强化学习

定义 18.3 (强化学习). 强化学习（*Reinforcement Learning, RL*）是智能体（*Agent*）通过与环境的交互来学习最优策略，通过试错和奖励信号来指导学习过程。

强化学习问题可以建模为马尔可夫决策过程（*MDP*)：$(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$，其中：

- $\mathcal{S}$：状态空间
- $\mathcal{A}$：动作空间

- $\mathcal{P}$：状态转移概率
- $\mathcal{R}$：奖励函数
- γ：折扣因子

智能体的目标是学习策略 $\pi : \mathcal{S} \to \mathcal{A}$，最大化累积奖励：

$$R = \sum_{t=0}^{\infty} \gamma^t r_t$$

通俗解释：强化学习就像训练宠物。宠物（智能体）做出动作（如坐下），主人（环境）给出奖励（如给食物）或惩罚（如不给食物）。通过反复尝试，宠物学会在什么情况下做什么动作能获得最多奖励。

18.3.1 强化学习的核心概念

- **状态（State）**：环境在某个时刻的完整描述
- **动作（Action）**：智能体可以执行的操作
- **奖励（Reward）**：环境对智能体动作的反馈信号
- **策略（Policy）**：从状态到动作的映射
- **价值函数（Value Function）**：评估状态或动作的长期价值

18.3.2 典型应用

例 18.5 (游戏 AI). **问题**：训练 *AI* 在游戏中达到人类或超人类水平。

方法：

- *Deep Q-Network (DQN)*：使用深度神经网络近似 Q 函数
- *Policy Gradient*：直接优化策略函数
- *Actor-Critic*：结合价值函数和策略梯度

实际应用：

- ***AlphaGo***：*DeepMind* 开发的围棋 *AI*，*2016* 年击败世界冠军李世石
- ***AlphaStar***：在《星际争霸 *II*》中达到大师级水平

- ***OpenAI Five***：在 *Dota 2* 中击败职业战队

例 18.6 (机器人控制). **问题**：训练机器人完成复杂任务，如抓取物体、行走等。

方法：使用强化学习让机器人在仿真或真实环境中通过试错学习。

实际应用：*Boston Dynamics* 的机器人使用强化学习进行运动控制，实现复杂的平衡和移动能力。

18.3.3 强化学习的优势与局限性

优势：

- 适用于序列决策问题
- 可以通过试错学习，不需要大量标注数据
- 能够学习长期策略
- 适用于动态环境

局限性：

- 训练过程可能很慢，需要大量交互
- 奖励函数设计困难
- 探索与利用的平衡问题
- 安全性问题（在关键应用中）

19 经典算法

本节介绍机器学习中的经典算法，包括线性回归、决策树、随机森林和支持向量机。这些算法虽然相对简单，但在许多实际问题中仍然非常有效。

19.1 线性回归

线性回归是监督学习中最基础的算法之一，用于解决回归问题。

定义 19.1 (线性回归). 线性回归假设目标变量 y 与特征向量 $\mathbf{x}$ 之间存在线性关系：

$$y = \mathbf{w}^T\mathbf{x} + b + \epsilon$$

其中 $\mathbf{w} \in \mathbb{R}^d$ 是权重向量，$b \in \mathbb{R}$ 是偏置项，ϵ 是误差项（通常假设 $\epsilon \sim \mathcal{N}(0, \sigma^2)$）。

给定训练数据 $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$，线性回归的目标是找到参数 $\mathbf{w}$ 和 b，使得预测误差最小。

通俗解释：线性回归就像用一条直线去拟合数据点。例如，用一条直线拟合" 房屋面积-房价" 的关系，直线的斜率和截距就是学到的参数。

19.1.1 最小二乘法

最常用的方法是最小二乘法（Least Squares），最小化平方误差：

$$L(\mathbf{w}, b) = \sum_{i=1}^{n} (y_i - (\mathbf{w}^T \mathbf{x}_i + b))^2$$

通过求导并令导数为零，可以得到闭式解：

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

其中 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 是特征矩阵，$\mathbf{y} \in \mathbb{R}^n$ 是标签向量。

19.1.2 正则化

为了防止过拟合，可以加入正则化项：

- **Ridge 回归（L_2 正则化）**：

$$L_{\text{Ridge}} = \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|_2^2$$

  其中 $\lambda > 0$ 是正则化系数。Ridge 回归倾向于产生较小的权重。

- **Lasso 回归（L_1 正则化）**：

$$L_{\text{Lasso}} = \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|_1$$

  Lasso 回归可以产生稀疏解（许多权重为 0），实现特征选择。

19.1.3 应用场景

例 19.1 (房价预测). 使用线性回归预测房价，特征包括面积、卧室数、位置等。*Ridge* 回归可以防止过拟合，*Lasso* 回归可以自动选择重要特征。

例 19.2 (股票价格预测). 使用历史价格、交易量等特征预测未来股价。虽然股价受多种因素影响，线性回归可以作为基准模型。

19.1.4 优势与局限性

优势：

- 简单易懂，计算效率高
- 有闭式解，训练快速
- 可解释性强（权重表示特征重要性）
- 适合作为基准模型

局限性：

- 只能捕捉线性关系
- 对异常值敏感
- 假设特征之间相互独立
- 需要特征工程（如多项式特征）来捕捉非线性

19.2 决策树

决策树是一种基于树结构的分类和回归算法，通过一系列规则进行决策。

定义 19.2 (决策树). 决策树（*Decision Tree*）是一个树形结构，其中：

- 内部节点：表示特征测试
- 分支：表示测试结果
- 叶节点：表示类别标签或数值

从根节点到叶节点的路径对应一条决策规则。

通俗解释：决策树就像医生诊断疾病的流程。先问” 是否发烧？ ”，如果” 是” 再问” 是否咳嗽？ ”，根据一系列问题的答案，最终得出诊断结果。

19.2.1 构建决策树

决策树的构建是一个递归过程：

Algorithm 3 构建决策树

Require: 训练数据集 $\mathcal{D}$，特征集 $\mathcal{F}$

Ensure: 决策树 T

if $\mathcal{D}$ 中所有样本属于同一类别 **then**
  返回叶节点，标记为该类别
else if $\mathcal{F}$ 为空或 $\mathcal{D}$ 为空 **then**
  返回叶节点，标记为 $\mathcal{D}$ 中多数类
else
  选择最优特征 $f^* = \arg\max_{f \in \mathcal{F}} \mathrm{IG}(\mathcal{D}, f)$
  为 f^* 的每个可能值创建分支
  for 每个分支 **do**
    $D_v = \mathcal{D}$ 中 $f^* = v$ 的样本子集
    if D_v 为空 **then**
      添加叶节点，标记为 $\mathcal{D}$ 中多数类
    else
      递归构建子树：$\mathrm{BuildTree}(D_v, \mathcal{F} \setminus \{f^*\})$
    end if
  end for
end if

19.2.2 特征选择准则

常用的特征选择准则包括：

- **信息增益（Information Gain）**：

$$\text{IG}(\mathcal{D}, f) = H(\mathcal{D}) - \sum_{v} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} H(\mathcal{D}_v)$$

其中 $H(\mathcal{D})$ 是数据集的熵。

- **基尼不纯度（Gini Impurity）**：

$$G(\mathcal{D}) = 1 - \sum_{k} p_k^2$$

其中 p_k 是类别 k 在数据集中的比例。

- **基尼增益**：

$$\text{GiniGain}(\mathcal{D}, f) = G(\mathcal{D}) - \sum_{v} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} G(\mathcal{D}_v)$$

19.2.3 剪枝

为了防止过拟合，需要对决策树进行剪枝：

- **预剪枝**：在构建过程中提前停止（如限制树深度、最小样本数）
- **后剪枝**：构建完整树后，自底向上删除节点，用叶节点替代

19.2.4 应用场景

例 19.3 (医疗诊断). 使用决策树根据症状（发烧、咳嗽、头痛等）诊断疾病。决策树的可解释性使得医生可以理解诊断依据。

例 19.4 (信用评估). 银行使用决策树评估贷款申请人的信用风险，根据收入、工作年限、信用历史等特征做出决策。

19.2.5 优势与局限性

优势：

- 可解释性强，决策过程清晰

- 可以处理数值和类别特征
- 不需要特征缩放
- 可以捕捉非线性关系

局限性:

- 容易过拟合
- 对数据的小变化敏感（不稳定）
- 倾向于选择具有更多取值的特征
- 难以处理特征之间的交互

19.3 随机森林

随机森林（Random Forest）是决策树的集成方法，通过组合多个决策树来提高性能。

定义 19.3 (随机森林). 随机森林由 B 棵决策树组成，每棵树在训练时:

1. 使用自助采样（*Bootstrap Sampling*）从训练集中采样
2. 在每个节点分裂时，随机选择 k 个特征（通常 $k = \sqrt{d}$）进行考虑

预测时，对于分类问题使用投票，对于回归问题使用平均。

通俗解释: 随机森林就像一群专家投票做决策。每个专家（决策树）根据自己的经验（不同的训练数据）给出意见，最终综合所有专家的意见做出决策。这样比单个专家更可靠。

19.3.1 算法流程

Algorithm 4 随机森林训练

Require: 训练数据集 $\mathcal{D}$，树的数量 B，特征采样数 k
Ensure: 随机森林 $\{T_1, T_2, \ldots, T_B\}$
  for $b = 1$ to B **do**
    使用自助采样从 $\mathcal{D}$ 中采样得到 $\mathcal{D}_b$
    使用 $\mathcal{D}_b$ 训练决策树 T_b，在每个节点随机选择 k 个特征
  end for

19.3.2 随机性的作用

随机森林通过两种随机性提高性能：

- **样本随机性**：每棵树使用不同的训练样本（自助采样）
- **特征随机性**：每棵树在每个节点考虑不同的特征子集

这种随机性使得各棵树之间具有多样性，减少过拟合，提高泛化能力。

19.3.3 应用场景

例 19.5 (图像分类). *在 CIFAR-10 数据集上，随机森林可以作为基准模型，虽然不如深度学习模型，但训练速度快，可解释性强。*

例 19.6 (特征重要性分析). *随机森林可以计算特征重要性，帮助理解哪些特征对预测最重要。这在特征工程和模型解释中很有价值。*

19.3.4 优势与局限性

优势：

- 性能通常优于单棵决策树
- 对过拟合有较强的抵抗力
- 可以处理高维特征
- 可以计算特征重要性
- 训练可以并行化

局限性：

- 模型可解释性不如单棵决策树
- 需要更多内存和计算资源
- 对于某些问题，可能不如梯度提升方法（如 XGBoost）

19.4 支持向量机

支持向量机（Support Vector Machine, SVM）是一种强大的分类算法，基于最大间隔原理。

定义 19.4 (支持向量机). *对于线性可分的二分类问题，SVM 寻找一个超平面 $\mathbf{w}^T\mathbf{x} + b = 0$，使得两类样本之间的间隔（margin）最大。*

间隔定义为两类样本到超平面的最小距离：

$$margin = \frac{2}{\|\mathbf{w}\|}$$

SVM 的优化目标是：

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2 \quad s.t. \quad y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1, \forall i$$

通俗解释：SVM 就像在两个群体之间画一条" 最宽的路"。这条路要尽可能宽，同时要确保两边的群体都在路的正确一侧。支持向量就是那些" 站在路边" 的样本点。

19.4.1 软间隔 SVM

对于线性不可分的情况，引入松弛变量 ξ_i，允许一些样本分类错误：

$$\min_{\mathbf{w},b,\boldsymbol{\xi}} \frac{1}{2}\|\mathbf{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

约束条件：

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$$

其中 $C > 0$ 是惩罚参数，控制对误分类的惩罚程度。

19.4.2 核技巧

对于非线性问题，使用核函数将数据映射到高维空间，在高维空间中线性可分：

$$K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i)^T\phi(\mathbf{x}_j)$$

常用的核函数包括：

- **多项式核**：$K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^T\mathbf{x}_j + 1)^d$
- **径向基函数（RBF）核**：$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$
- **Sigmoid 核**：$K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\alpha\mathbf{x}_i^T\mathbf{x}_j + c)$

19.4.3 对偶形式

SVM 的对偶形式为：

$$\max_{\boldsymbol{\alpha}} \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

约束条件：

$$\sum_{i=1}^{n} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C$$

对偶形式的优势：

- 只需要计算核函数，不需要显式映射到高维空间
- 支持向量（$\alpha_i > 0$）的数量通常远小于样本数
- 更容易扩展到大规模问题

19.4.4 应用场景

例 19.7 (文本分类). *SVM 在文本分类任务中表现优异，特别是在小样本情况下。使用 TF-IDF 特征和 RBF 核，可以在新闻分类、情感分析等任务中取得良好效果。*

例 19.8 (图像分类). *在图像分类任务中，SVM 可以作为特征分类器。例如，使用 CNN 提取特征，然后用 SVM 进行分类，这在某些情况下比端到端的 CNN 更有效。*

19.4.5 优势与局限性

优势：

- 在中小规模数据集上表现优异
- 通过核函数可以处理非线性问题
- 理论基础完善（基于统计学习理论）
- 对过拟合有较好的控制
- 支持向量提供了模型的稀疏表示

局限性：

- 对大规模数据集计算成本高

- 对特征缩放敏感
- 核函数和参数选择需要经验
- 可解释性不如决策树
- 概率输出需要额外处理（Platt scaling）

20 特征工程

特征工程是机器学习中至关重要的一环，好的特征可以显著提升模型性能。特征工程包括特征选择、特征变换和特征编码等。

20.1 特征选择

特征选择是从原始特征中选择最有用的特征子集，减少维度，提高模型性能和可解释性。

20.1.1 过滤方法（Filter Methods）

过滤方法基于特征的统计特性进行选择，独立于具体的学习算法：

- **方差选择**：移除方差很小的特征（几乎不变的特征）
- **相关系数**：选择与目标变量相关性高的特征
- **互信息**：选择与目标变量互信息大的特征
- **卡方检验**：用于分类问题，检验特征与标签的独立性

20.1.2 包装方法（Wrapper Methods）

包装方法使用学习算法来评估特征子集：

- **前向选择**：从空集开始，逐步添加最有用的特征
- **后向消除**：从完整特征集开始，逐步移除最无用的特征
- **递归特征消除（RFE）**：递归地移除最不重要的特征

20.1.3 嵌入方法（Embedded Methods）

嵌入方法在模型训练过程中进行特征选择：

- **Lasso 回归**：L_1 正则化自动产生稀疏解
- **决策树**：通过特征重要性进行选择
- **随机森林**：通过特征重要性排序

20.2 特征变换

特征变换是对特征进行数学变换，改变特征的分布或关系。

20.2.1 标准化和归一化

- **标准化（Z-score）**：

$$z = \frac{x - \mu}{\sigma}$$

将特征转换为均值为 0、标准差为 1 的分布。

- **最小-最大归一化**：

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

将特征缩放到 $[0, 1]$ 区间。

- **Robust 缩放**：使用中位数和四分位距，对异常值更鲁棒。

为什么需要特征缩放：

- 许多算法（如 SVM、K-means、神经网络）对特征尺度敏感
- 梯度下降算法在特征尺度不一致时收敛慢
- 距离-based 算法（如 KNN）受特征尺度影响大

20.2.2 多项式特征

通过创建特征的多项式组合来捕捉非线性关系：

$$(x_1, x_2) \to (x_1, x_2, x_1^2, x_1 x_2, x_2^2)$$

应用场景：线性回归无法捕捉非线性关系时，可以使用多项式特征。

20.2.3 对数变换

对偏态分布进行对数变换，使其更接近正态分布：

$$x' = \log(x + 1)$$

应用场景：处理价格、收入等右偏分布的数据。

20.3 特征编码

特征编码是将类别特征转换为数值特征的过程。

20.3.1 独热编码（One-Hot Encoding）

将类别特征转换为二进制向量：

例 20.1. 颜色特征：*[”红”, ”绿”, ”蓝”]* →

- 红：$[1, 0, 0]$
- 绿：$[0, 1, 0]$
- 蓝：$[0, 0, 1]$

优势：不引入类别间的顺序关系。

局限性：类别数量多时会产生高维稀疏特征。

20.3.2 标签编码（Label Encoding）

将类别映射为整数：

例 20.2 (”低”, ”中”, ”高”)**.** → *[0, 1, 2]*

注意：只适用于有序类别，否则会引入虚假的顺序关系。

20.3.3 目标编码（Target Encoding）

使用目标变量的统计量对类别进行编码：

$$x' = \frac{\sum_{i:x_i=x} y_i}{|\{i : x_i = x\}|}$$

即用该类别的平均目标值作为编码。

优势：可以捕捉类别与目标的关系。

注意：需要防止过拟合（如使用交叉验证）。

21 模型评估与验证

模型评估是机器学习流程中的关键步骤，用于衡量模型性能、选择最佳模型和防止过拟合。

21.1 评估指标

21.1.1 分类问题指标

- **准确率（Accuracy）**：
$$\text{Accuracy} = \frac{\text{正确预测数}}{\text{总样本数}}$$
- **精确率（Precision）**：
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$
预测为正例中真正为正例的比例。
- **召回率（Recall）**：
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$
真正例中被正确预测的比例。
- **F1 分数**：
$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$
精确率和召回率的调和平均。
- **ROC 曲线和 AUC**：ROC 曲线以假正例率为横轴，真正例率为纵轴。AUC（曲线下面积）衡量分类器的整体性能。

21.1.2 回归问题指标

- **均方误差（MSE）**：
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

- **均方根误差（RMSE）**：

$$\text{RMSE} = \sqrt{\text{MSE}}$$

- **平均绝对误差（MAE）**：

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **决定系数（R^2）**：

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

  衡量模型解释的方差比例。

21.2 交叉验证

交叉验证是评估模型泛化能力的重要方法。

21.2.1 K 折交叉验证

将数据集分为 K 折，每次使用 $K-1$ 折训练，剩余 1 折测试，重复 K 次：

Algorithm 5 K 折交叉验证

Require: 数据集 $\mathcal{D}$，折数 K，学习算法 $\mathcal{A}$

Ensure: 平均性能指标

将 $\mathcal{D}$ 随机分为 K 折：$\mathcal{D}_1, \ldots, \mathcal{D}_K$

for $k = 1$ to K **do**

训练集：$\mathcal{D}_{\text{train}} = \mathcal{D} \setminus \mathcal{D}_k$

测试集：$\mathcal{D}_{\text{test}} = \mathcal{D}_k$

使用 $\mathcal{D}_{\text{train}}$ 训练模型 $M_k = \mathcal{A}(\mathcal{D}_{\text{train}})$

在 $\mathcal{D}_{\text{test}}$ 上评估性能 s_k

end for

return $\bar{s} = \frac{1}{K}\sum_{k=1}^{K} s_k$

优势：

- 充分利用数据
- 提供性能估计的方差
- 减少对数据划分的依赖

常见选择：$K = 5$ 或 $K = 10$。

21.2.2 留一法交叉验证（LOOCV）

$K = n$ 的特殊情况，每次留一个样本作为测试集。计算成本高，但无偏估计。

21.3 过拟合与欠拟合

定义 21.1 (过拟合). 过拟合（*Overfitting*）是指模型在训练集上表现很好，但在测试集上表现较差的现象。模型过度学习了训练数据的噪声和细节，导致泛化能力差。

定义 21.2 (欠拟合). 欠拟合（*Underfitting*）是指模型在训练集和测试集上都表现较差的现象。模型过于简单，无法捕捉数据中的基本模式。

通俗解释:

- **过拟合**：就像学生死记硬背了所有练习题，但遇到新题目就不会了
- **欠拟合**：就像学生只学了基础知识，连练习题都做不好

21.3.1 识别过拟合和欠拟合

- **过拟合的迹象**:
  - 训练误差很小，但验证误差很大
  - 模型复杂度高（如深度很深的决策树）
  - 训练集和验证集性能差距大
- **欠拟合的迹象**:
  - 训练误差和验证误差都很大
  - 模型复杂度低（如线性模型处理非线性问题）
  - 模型无法捕捉数据的基本模式

21.3.2 解决方法

解决过拟合:

- 增加训练数据
- 减少模型复杂度
- 正则化（L_1、L_2）

- Dropout（神经网络）
- 早停（Early Stopping）

解决欠拟合：

- 增加模型复杂度
- 增加特征
- 减少正则化
- 增加训练时间

21.4 偏差-方差权衡

定义 21.3 (偏差和方差).

- **偏差（*Bias*）**：模型的期望预测与真实值的差异，衡量模型的拟合能力
- **方差（*Variance*）**：模型在不同训练集上预测的差异，衡量模型的稳定性

总误差可以分解为：

$$Error = Bias^2 + Variance + Irreducible\ Error$$

通俗解释：

- **高偏差**：模型太简单，无法捕捉数据模式（欠拟合）
- **高方差**：模型太复杂，对训练数据的小变化敏感（过拟合）

21.4.1 偏差-方差权衡

- **简单模型**（如线性回归）：
  - 高偏差，低方差
  - 可能欠拟合
- **复杂模型**（如深度神经网络）：
  - 低偏差，高方差
  - 可能过拟合
- **理想模型**：

- 低偏差，低方差
- 需要合适的模型复杂度和正则化

22 集成学习方法

集成学习通过组合多个基学习器来提高预测性能，是机器学习中的重要技术。

22.1 Bagging

Bagging（Bootstrap Aggregating）通过训练多个模型并平均预测结果来减少方差。

定义 22.1 (Bagging)**.** *Bagging 算法*:

1. *使用自助采样从训练集中生成 B 个不同的训练集*
2. *在每个训练集上训练一个基学习器*
3. *对于分类问题使用投票，对于回归问题使用平均*

通俗解释: Bagging 就像多个专家独立给出意见，然后综合所有意见做决策。每个专家看到的数据略有不同，但综合起来更可靠。

22.1.1 随机森林

随机森林是 Bagging 的特例，基学习器是决策树，并在特征选择时引入随机性。

22.1.2 优势与局限性

优势:

- 减少方差，提高泛化能力
- 可以并行训练
- 对过拟合有抵抗力

局限性:

- 不能减少偏差
- 需要足够的计算资源

22.2 Boosting

Boosting 通过顺序训练多个弱学习器，每个学习器关注前一个学习器的错误。

定义 22.2 (Boosting)**.** *Boosting 算法*：

1. *初始化样本权重*
2. *对于* $t = 1, \ldots, T$：
   - *使用当前权重训练弱学习器* h_t
   - *计算* h_t *的误差*
   - *更新样本权重（增加错误样本的权重）*
   - *计算* h_t *的权重* α_t
3. *最终预测*：$H(\mathbf{x}) = \sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})$

通俗解释：Boosting 就像学生做错题后，老师重点讲解错题，学生反复练习直到掌握。每个弱学习器专注于前一个学习器的薄弱环节。

22.2.1 AdaBoost

AdaBoost（Adaptive Boosting）是经典的 Boosting 算法：

Algorithm 6 AdaBoost

Require: 训练集 $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$，弱学习器 $\mathcal{A}$，迭代次数 T

Ensure: 集成模型 H

初始化样本权重：$w_i^{(1)} = 1/n, \forall i$

for $t = 1$ to T **do**

使用权重 $\mathbf{w}^{(t)}$ 训练弱学习器：$h_t = \mathcal{A}(\mathcal{D}, \mathbf{w}^{(t)})$

计算加权误差：$\epsilon_t = \sum_{i: h_t(\mathbf{x}_i) \neq y_i} w_i^{(t)}$

计算学习器权重：$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$

更新样本权重：$w_i^{(t+1)} = \frac{w_i^{(t)}}{Z_t} \exp(-\alpha_t y_i h_t(\mathbf{x}_i))$

end for

return $H(\mathbf{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x})\right)$

22.2.2 梯度提升

梯度提升（Gradient Boosting）将 Boosting 视为优化问题，使用梯度下降来最小化损失函数。

核心思想：每个新学习器拟合前一个模型的负梯度（残差）。

22.2.3 XGBoost 和 LightGBM

- **XGBoost**：优化的梯度提升实现，支持并行计算、正则化、处理缺失值
- **LightGBM**：更快的梯度提升实现，使用基于直方图的算法和 Leaf-wise 树生长策略

应用场景：在 Kaggle 等数据科学竞赛中，XGBoost 和 LightGBM 经常是获胜方案的核心组件。

22.2.4 优势与局限性

优势：

- 通常比单个模型性能更好
- 可以减少偏差和方差
- 可以处理各种类型的数据

局限性：

- 训练时间较长（顺序训练）
- 对异常值敏感
- 可解释性较差

22.3 Stacking

Stacking（堆叠）使用元学习器来组合多个基学习器的预测。

定义 22.3 (Stacking)**.** *Stacking 算法*：

1. *使用交叉验证训练多个基学习器*
2. *使用基学习器的预测作为特征，训练元学习器*

3. 最终预测使用元学习器

通俗解释：Stacking 就像有一个” 超级裁判”，它不直接看原始数据，而是看各个” 专家”（基学习器）的意见，然后综合这些意见做出最终判断。

22.3.1 算法流程

Algorithm 7 Stacking

Require: 训练集 $\mathcal{D}$，基学习器 $\{\mathcal{A}_1, \ldots, \mathcal{A}_M\}$，元学习器 $\mathcal{A}_{\text{meta}}$
Ensure: 集成模型
使用 K 折交叉验证训练基学习器
for $m = 1$ to M **do**
  for 每折 k **do**
    在折 k 的训练集上训练 $h_{m,k}$
    在折 k 的验证集上生成预测 $p_{m,k}$
  end for
  基学习器 m 的完整预测：$\mathbf{p}_m = [p_{m,1}, \ldots, p_{m,K}]$
end for
使用 $\{(\mathbf{p}_1, \ldots, \mathbf{p}_M), \mathbf{y}\}$ 训练元学习器
return 集成模型（基学习器 + 元学习器）

22.3.2 优势与局限性

优势：

- 可以捕捉基学习器之间的互补性
- 通常性能优于单个模型
- 灵活性高，可以使用不同类型的基学习器

局限性：

- 计算成本高
- 需要仔细设计，避免过拟合
- 可解释性差

23 在线学习与增量学习

在线学习和增量学习使模型能够从新数据中持续学习，适应数据分布的变化。

23.1 在线学习

定义 23.1 (在线学习). *在线学习（Online Learning）是一种学习范式，模型逐个处理样本，每处理一个样本就更新模型参数，不需要存储所有历史数据。*

给定数据流 $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots\}$，在线学习算法在时刻 t：

1. *接收样本 $(\mathbf{x}_t, y_t)$*
2. *使用当前模型 f_t 进行预测*
3. *根据预测误差更新模型：$f_{t+1} = Update(f_t, (\mathbf{x}_t, y_t))$*

通俗解释：在线学习就像实时学习，每来一个新例子就立即学习，不需要等所有数据都收集完。就像学生每做一道题就立即知道答案并学习，而不是等所有题目做完再统一学习。

23.1.1 在线梯度下降

在线梯度下降是随机梯度下降的在线版本：

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \ell(f(\mathbf{x}_t; \mathbf{w}_t), y_t)$$

其中 η_t 是学习率，通常随时间衰减。

23.1.2 应用场景

例 23.1 (推荐系统). *在线推荐系统需要实时响应用户行为，根据用户的实时反馈更新推荐模型。例如，用户点击了某个商品，系统立即更新该用户的兴趣模型。*

例 23.2 (广告投放). *在线广告系统需要根据实时点击率调整广告投放策略，快速适应市场变化。*

23.1.3 优势与局限性

优势：

- 内存效率高（不需要存储所有数据）
- 可以快速适应数据分布变化
- 适合大规模数据流
- 可以实时更新模型

局限性：

- 对异常值敏感
- 可能遗忘历史信息
- 需要仔细设计学习率
- 难以评估模型性能

23.2 增量学习

定义 23.2 (增量学习). 增量学习（*Incremental Learning*）是模型在已有知识的基础上，从新数据中学习新知识，同时保留或整合旧知识的过程。

与在线学习的区别：增量学习通常处理批量新数据，并且需要处理”灾难性遗忘”问题。

通俗解释：增量学习就像人类学习新知识。学习新内容时，不会完全忘记旧知识，而是将新旧知识整合在一起。但机器学习模型容易”遗忘”旧知识，需要特殊技术来解决。

23.2.1 灾难性遗忘

灾难性遗忘（Catastrophic Forgetting）是指模型在学习新任务时，会大幅降低在旧任务上的性能。

原因：神经网络参数在训练新任务时被更新，可能破坏对旧任务的记忆。

23.2.2 解决方法

- **弹性权重巩固（EWC）**：在更新参数时，对重要参数施加惩罚，防止大幅改变
- **渐进式神经网络**：为每个任务添加新的网络分支，保留旧网络不变
- **回放机制**：存储部分旧数据，与新数据一起训练
- **知识蒸馏**：使用旧模型指导新模型学习

23.2.3 应用场景

例 23.3 (持续学习系统). 智能助手需要不断学习新技能，但不能忘记已有技能。例如，学习新语言时不能忘记已掌握的语言。

例 23.4 (个性化推荐). 推荐系统需要适应用户兴趣的变化，同时保留对用户长期偏好的理解。

24 总结

本文档系统性地介绍了机器学习的核心理论与应用，包括：

- **机器学习基础**：监督学习、无监督学习和强化学习三大范式，每种范式适用于不同类型的问题
- **经典算法**：线性回归、决策树、随机森林和支持向量机等基础算法，虽然简单但在许多场景中仍然有效
- **特征工程**：特征选择、变换和编码等技术，是提升模型性能的关键
- **模型评估**：交叉验证、过拟合/欠拟合识别、偏差-方差权衡等评估技术
- **集成学习**：Bagging、Boosting 和 Stacking 等方法，通过组合多个模型提高性能
- **在线与增量学习**：使模型能够从新数据中持续学习，适应动态环境

机器学习的核心价值：

- 能够从数据中自动学习模式，减少人工规则设计
- 可以处理高维、复杂的现实世界数据
- 通过持续学习适应环境变化
- 在多个领域取得了突破性进展

未来发展方向：

- 自动化机器学习（AutoML）：减少人工调参和特征工程
- 可解释性：提高模型的可解释性和可信度
- 持续学习：更好地处理数据分布变化和任务演化

- 小样本学习：在数据稀缺场景下仍能有效学习
- 联邦学习：在保护隐私的前提下进行分布式学习

机器学习作为人工智能的核心技术，将继续在各个领域发挥重要作用，推动技术进步和社会发展。

Part IV

第四部分：深度学习

25 引言

深度学习（Deep Learning）是机器学习的一个子领域，通过构建具有多个隐藏层的神经网络来学习数据的层次化表示。深度学习在计算机视觉、自然语言处理、语音识别、游戏 AI 等领域取得了突破性进展，成为当前人工智能领域最活跃的研究方向之一。

深度学习的核心优势：

- **自动特征提取**：无需人工设计特征，网络能够自动学习数据的层次化特征表示
- **处理复杂非线性关系**：通过多层非线性变换，能够建模高度复杂的函数关系
- **端到端学习**：直接从原始输入学习到最终输出，减少中间环节的信息损失
- **强大的表示能力**：深度网络具有强大的函数逼近能力，能够学习任意复杂的映射关系
- **迁移学习能力**：预训练模型可以迁移到相关任务，提高学习效率

深度学习面临的挑战：

- **数据需求量大**：通常需要大量标注数据才能训练出有效的模型
- **计算资源消耗**：训练深度网络需要强大的计算资源（GPU/TPU）
- **可解释性不足**：深度网络的决策过程往往缺乏可解释性，难以理解其内部机制
- **过拟合风险**：模型容量大，容易在训练数据上过拟合
- **实时性要求**：某些应用场景对推理速度有严格要求
- **数据稀疏性**：在某些领域（如医疗、金融）高质量数据稀缺

本文档系统性地介绍深度学习的核心理论、主要架构和关键技术，涵盖神经网络基础、深度网络架构、优化技术、表示学习、注意力机制以及强化学习等内容。

26 神经网络基础

26.1 感知机

感知机（Perceptron）是最简单的神经网络模型，由 Frank Rosenblatt 于 1957 年提出，是神经网络和深度学习的起点。

定义 26.1 (感知机). *感知机是一个二分类线性分类模型，其输入为特征向量* $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$，*输出为类别标签* $y \in \{-1, +1\}$。

感知机的数学表达式为：

$$f(\mathbf{x}) = sign(\mathbf{w}^T\mathbf{x} + b) = \begin{cases} +1 & if\ \mathbf{w}^T\mathbf{x} + b \geq 0 \\ -1 & if\ \mathbf{w}^T\mathbf{x} + b < 0 \end{cases} \tag{146}$$

其中， $\mathbf{w} = [w_1, w_2, \ldots, w_n]^T$ *是权重向量，* b *是偏置项。*

感知机的几何解释：感知机实际上是在特征空间中构造一个超平面 $\mathbf{w}^T\mathbf{x} + b = 0$，将数据分为两类。权重向量 $\mathbf{w}$ 垂直于超平面，指向正类区域。

感知机学习算法：使用错误驱动的学习规则，当分类错误时更新权重：

Algorithm 8 感知机学习算法

Require: 训练数据集 $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_m, y_m)\}$，学习率 η

Ensure: 权重向量 $\mathbf{w}$ 和偏置 b

初始化 $\mathbf{w} = \mathbf{0}$，$b = 0$
repeat
  for 每个样本 $(\mathbf{x}_i, y_i)$ **do**
    if $y_i(\mathbf{w}^T\mathbf{x}_i + b) \leq 0$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
      $b \leftarrow b + \eta y_i$
    end if
  end for
until 没有分类错误

感知机的局限性：感知机只能解决线性可分问题。对于线性不可分问题（如异或问题），单层感知机无法解决，这促使了多层感知机的提出。

26.2 多层感知机

多层感知机（Multi-Layer Perceptron, MLP）通过引入隐藏层和激活函数，能够解决非线性分类问题。

定义 26.2 (多层感知机). *多层感知机是由多个全连接层组成的神经网络，每层包含多个神经元。一个 L 层的 MLP 可以表示为:*

$$\mathbf{h}^{(0)} = \mathbf{x} \tag{147}$$

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}), \quad l = 1, 2, \ldots, L-1 \tag{148}$$

$$\mathbf{y} = \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)} \tag{149}$$

其中，$\mathbf{W}^{(l)}$ 是第 l 层的权重矩阵，$\mathbf{b}^{(l)}$ 是偏置向量，σ 是激活函数。

26.2.1 隐藏层详解

概念解释: 隐藏层（Hidden Layer）是人工神经网络中的核心组成部分，位于输入层和输出层之间。之所以称为" 隐藏"，是因为这些层的输出不直接暴露给外部世界（不像输入层接收原始数据、输出层给出最终预测），而是用于内部特征表示的学习。

数学表示:

从数学和结构上看:

- **输入层**: 接收原始特征，记为 $\mathbf{x} \in \mathbb{R}^d$
- **隐藏层**: 对输入进行非线性变换，提取更高层次的抽象特征。第 l 层的输出为:

$$\mathbf{h}^{(l)} = \sigma\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right) \tag{150}$$

  其中:

  - $\mathbf{W}^{(l)}$ 是权重矩阵（可学习参数）
  - $\mathbf{b}^{(l)}$ 是偏置向量
  - $\sigma(\cdot)$ 是激活函数（如 ReLU、Sigmoid、Tanh）
  - $\mathbf{h}^{(0)} = \mathbf{x}$

- **输出层**: 基于最后一层隐藏层的表示，生成最终预测

关键作用: 隐藏层通过多层非线性组合，使网络能够拟合复杂的函数（如图像识别、自然语言理解等）。没有隐藏层（即只有输入 $\rightarrow$ 输出），网络就退化为线性模型，表达能力极其有限。

配置隐藏层:

当你在代码中配置隐藏层时，实际上在配置:

- **神经元数量**：每层的神经元数量（即维度），也叫” 隐藏单元数” 或” 宽度（width）”
- **权重矩阵形状**：如 100×256 表示输入 100 维，输出 256 维
- **激活函数**：指定该隐藏层使用的激活函数（如 ReLU）
- **层数**：决定了网络的深度（depth）

示例：

```
# 配置示例：输入100维 → 隐藏层1：256个神经元 → 隐藏层2：128个神经元 → 输出1维
# 这实际上定义了：
# - 第1层：100 × 256 的权重矩阵，256个神经元
# - 第2层：256 × 128 的权重矩阵，128个神经元
# - 输出层：128 × 1 的权重矩阵，1个神经元
```

Listing 51: 隐藏层配置示例

超参数影响：

- 神经元越多、层数越深 → 模型越复杂，拟合能力越强，但也更容易过拟合
- 在深度学习中，常用 2 4 层 MLP，每层 128 512 个神经元

特征学习的层次性：

每一个” 隐藏层” 其实就是一个向量空间变换器：把低层特征映射到高层语义特征。例如：

- 第 1 层可能学到” 边缘、线条” 等低级特征
- 第 2 层可能学到” 形状、纹理” 等中级特征
- 第 3 层可能学到” 物体部件” 等高级特征
- 最终输出层预测” 完整物体” 的类别

26.2.2 输入层、隐藏层、输出层的明确区分

核心定义：

隐藏层（Hidden Layer）的定义是：既不是输入层，也不是输出层的所有中间层。

- **输入层（Input Layer）**：负责接收原始数据（如特征向量、像素值等），不是隐藏层。

- **输出层（Output Layer）**：负责产生最终预测（如分类概率、回归值等），也不是隐藏层。
- **隐藏层**：只有夹在输入层和输出层之间的层才叫隐藏层。

举例说明：

对于一个 3 层网络（输入层 → 隐藏层 → 输出层），只有中间那层是隐藏层。如果网络有更多层，如输入层 → 隐藏层 1 → 隐藏层 2 → 输出层，那么隐藏层 1 和隐藏层 2 都是隐藏层。

术语使用规范：

- **输入层**：不是隐藏层。无参数，仅传递数据。
- **中间层**：是隐藏层。如 `Linear(256,128) + ReLU`，包含可学习参数。
- **输出层**：不是隐藏层。如 `Linear(128, 1)`，用于最终映射。

重要说明：

虽然输出层通常是全连接层（用 Linear 实现，有权重和偏置），但我们不会称它为" 隐藏层"，因为它的功能是输出，不是隐藏表示。当说" 模型有 2 个隐藏层" 时，指的是中间有 2 个可学习的层，不包括输入和输出。

26.2.3 编码（Encoding）与输入层的关系

核心概念区分：

- **编码（Encoding）**：将原始数据（如文本、类别、SQL 语句、图像）转换为数值向量的过程。这是预处理步骤，发生在模型之外或模型最前端。
- **输入层（Input Layer）**：神经网络的第一层，接收已经编码好的数值向量作为输入。它本身不做编码，只是" 起点"。

数学表示：

编码过程：$\mathbf{x} = \text{encode}(\text{原始输入})$，其中 $\mathbf{x} \in \mathbb{R}^d$ 是编码后的数值向量。

输入层：$\mathbf{h}^{(0)} = \mathbf{x}$，即输入层就是编码后的向量本身，不做任何变换。

举例说明：

例 26.1 (文本分类任务)**.**

- **原始输入**：文本*"Hello World"*

- **编码过程**：
  - 分词：*["Hello", "World"]*
  - 词嵌入（*Embedding*）：将每个词转换为向量
  - 结果：$\mathbf{x} \in \mathbb{R}^d$（如 $d = 128$）
- **输入层**：接收这个 $\mathbf{x}$ 向量，不做任何变换，直接传给第一个全连接层

例 26.2 (图像分类任务).

- **原始输入**：一张 28×28 的图片（像素值 *0 255*）
- **编码过程**：
  - 归一化到 $[0, 1]$
  - 展平成 784 维向量
- **输入层**：接收这 784 维向量，直接送入第一个全连接层或卷积层

关键结论：

- **" 输入层就是做 encoding"**：不准确。输入层不执行 encoding，它只是接收已经编码好的数据。
- **"encoding 的结果作为输入层的值"**：正确。这是标准流程，编码后的向量 $\mathbf{x}$ 就是输入层的值。
- **"Embedding 层是输入层"**：不准确。`nn.Embedding` 是可学习的 encoding 模块，属于模型的第一层可学习模块，但它已经超出了" 输入层" 的范畴。

代码示例：

```
import torch
import torch.nn as nn

# 假设原始输入是词索引: [2, 5, 1]
embedding = nn.Embedding(num_embeddings=1000, embedding_dim=64)  # 这是
    encoding!
fc1 = nn.Linear(64, 128)  # 这才是真正的第一层（隐藏层）

x = torch.tensor([2, 5, 1])
emb = embedding(x)  # shape: (3, 64) ← encoded vector
out = fc1(emb)  # 输入给全连接层
```

Listing 52: 编码与输入层在代码中的体现

总结：

输入层不等于编码，但输入层的值等于编码的结果。编码是数据预处理或模型前端的转换步骤，目的是把原始信息变成神经网络能处理的数值向量。输入层只是这个向量进入网络的入口，本身不做计算（无参数）。

更准确的说法是：”我们先把数据 encode 成向量，然后喂给神经网络的输入层。”

26.2.4 输入层不做线性变换

核心结论：

输入层本身不做任何线性变换（也不做非线性变换）。输入层只是一个”数据入口”或”占位符”，它没有可学习的参数（如权重 $\mathbf{W}$ 和偏置 $\mathbf{b}$），也不执行任何计算（包括线性变换）。

什么是线性变换：

在线性代数和神经网络中，线性变换（更准确说是仿射变换）指：

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{151}$$

其中：

- $\mathbf{W}$：权重矩阵（可学习参数）
- $\mathbf{b}$：偏置向量（可学习参数）

这是全连接层（Linear/Dense Layer）的核心操作。只有包含可学习参数并执行上述运算的层，才做了线性变换。

输入层的本质：

输入层等于原始特征向量 $\mathbf{x}$ 本身。它的作用仅仅是将预处理或编码后的数据传递给网络的第一层可学习模块。

在数学上，输入层就是 $\mathbf{h}^{(0)} = \mathbf{x}$。没有 $\mathbf{W}$，没有 $\mathbf{b}$，没有运算。

类比说明：

输入层就像函数的参数，而不是函数体内的计算：

```
def neural_network(x):  # ← x 就是"输入层"
    z1 = W1 @ x + b1  # ← 第一个全连接层（才开始线性变换）
    a1 = relu(z1)
    ...
```

Listing 53: 输入层与计算层的区别

这里 **x** 本身不做任何事，只是被使用。

常见误解澄清：

- **误解：” 输入层是第一个 Linear 层”**
  澄清：错！第一个 `Linear` 层是第一个隐藏层（或输出层），不是输入层。输入层没有参数，而 `Linear` 层有可学习的权重和偏置。
- **误解：” 输入层有神经元，所以会计算”**
  澄清：错！输入” 神经元” 只是对输入维度的形象说法（如”100 维输入” 说成”100 个输入神经元”），但它们没有激活函数、没有权重，不进行计算。输入层只是数据的载体。
- **误解：”Embedding 是输入层”**
  澄清：不准确。`nn.Embedding` 是一个可学习的 encoding 层，属于模型的第一层可学习模块，但它已经超出了” 输入层” 的范畴。输入层是无参数的，而 `Embedding` 有可学习的参数。

图示说明：

$$\text{原始数据} \to [\text{Encoding}] \to \mathbf{x} \in \mathbb{R}^{100}\text{（这就是输入层，无计算）}$$
$$\downarrow$$
$$[\texttt{Linear}(\mathbf{W}_1, \mathbf{b}_1)]\text{（第一个全连接层，开始线性变换）}$$
$$\downarrow$$
$$\text{ReLU}(...)$$

特殊情况说明：

在极少数文献或框架中，有人把标准化（Normalization）或固定投影放在最前面，例如：

```
x_norm = (x - mean) / std  # 数据标准化
z = W @ x_norm + b
```

Listing 54: 标准化预处理

但请注意：标准化是预处理，不属于网络参数。即使把它写进模型，也通常视为数据 pipeline 的一部分，而非” 输入层的计算”。真正的” 输入层” 在理论和实践中都被定义为无参数、无变换的数据载体。

总结：

- **输入层会做线性变换吗?**
  不会。输入层没有权重 **W** 和偏置 **b**，不执行任何运算。

- **线性变换从哪开始?**
  从第一个全连接层（`Linear/Dense Layer`）开始。这是网络中的第一个可学习模块。
- **输入层的作用是什么?**
  接收已编码的数值向量，作为网络计算的起点。它只是数据的入口，不做任何变换。

所以可以明确地说：”输入层只是数据的入口，真正的计算从第一层全连接层才开始。”

26.3 激活函数详解

激活函数（Activation Function）是神经网络中的关键组件，它引入非线性变换，使网络能够学习复杂的非线性函数关系。没有激活函数，多层神经网络就退化为单层线性模型。

26.3.1 常用激活函数

1. Sigmoid 函数

数学公式:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{152}$$

导数:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{153}$$

输出范围: $(0, 1)$

用处:

- 二分类任务的输出层，输出概率
- 需要将输出映射到 $(0, 1)$ 区间的场景
- 门控机制（如 LSTM、GRU）

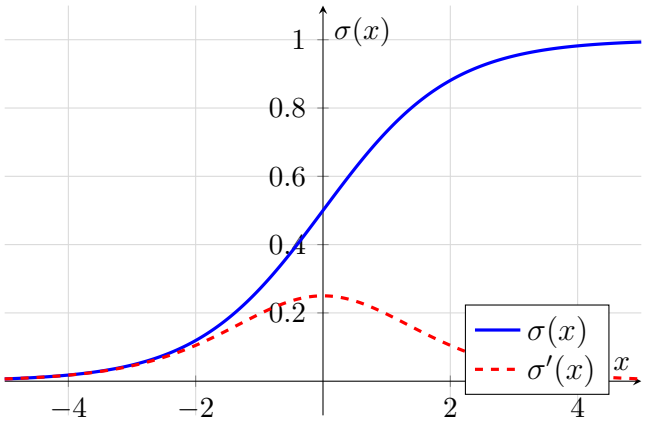
为什么用 Sigmoid:

- **概率解释**: 输出可以解释为概率，便于理解
- **平滑可导**: 处处可导，梯度计算方便
- **历史原因**: 早期神经网络常用，与生物神经元激活模式相似

缺点:

- **梯度消失**：当输入很大或很小时，梯度接近 0，导致深层网络难以训练
- **非零中心**：输出均值不为 0，导致梯度更新效率低
- **计算成本**：涉及指数运算，计算较慢

函数图像：



2. Tanh 函数（双曲正切）

数学公式：

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 \tag{154}$$

导数：

$$\tanh'(x) = 1 - \tanh^2(x) \tag{155}$$

输出范围：$(-1, 1)$

用处：

- RNN 中的隐藏层激活函数
- 需要零中心化输出的场景
- 数据归一化到 $(-1, 1)$ 区间

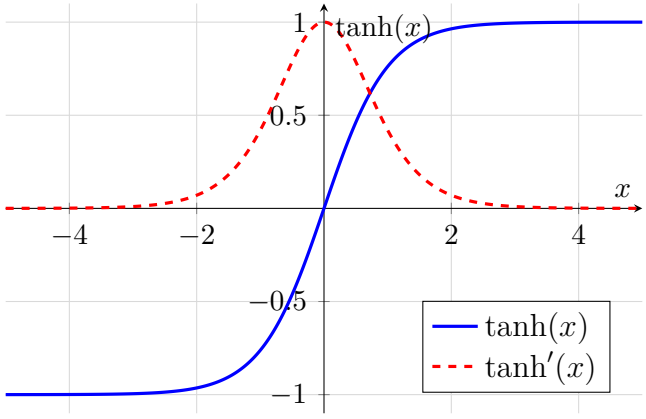
为什么用 Tanh：

- **零中心化**：输出均值为 0，梯度更新更高效
- **比 Sigmoid 更好**：在大多数情况下，tanh 比 sigmoid 表现更好
- **对称性**：关于原点对称，数学性质更好

缺点：

- **梯度消失**：与 Sigmoid 类似，在饱和区域梯度接近 0
- **计算成本**：涉及指数运算

函数图像：



3. ReLU 函数（Rectified Linear Unit）

数学公式：

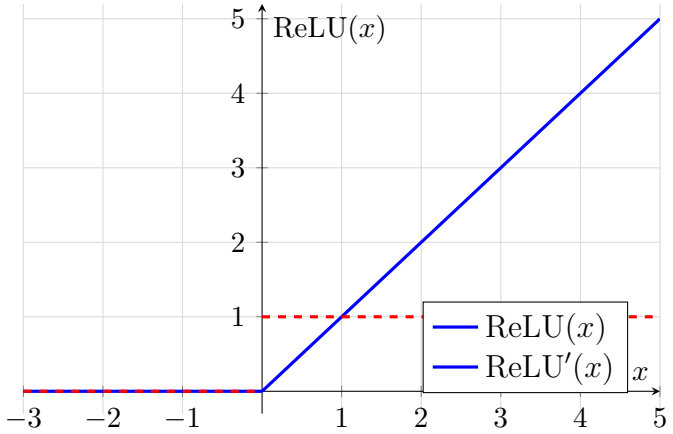
$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{156}$$

导数：

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{157}$$

输出范围：$[0, +\infty)$

函数图像：



用处：

- 深度神经网络隐藏层的首选激活函数
- CNN、MLP 等大多数现代神经网络架构
- 需要稀疏激活的场景

为什么用 ReLU：

- **缓解梯度消失**：在正区间梯度恒为 1，不会衰减
- **计算高效**：只需比较和选择操作，计算速度快
- **稀疏激活**：约 50% 的神经元会被激活，提高计算效率
- **生物学合理性**：模拟生物神经元的单侧抑制特性

缺点：

- **死亡 ReLU 问题**：如果神经元输出始终为负，梯度为 0，无法更新
- **非零中心**：输出均值不为 0
- **无界**：输出可以无限大，可能导致数值不稳定

4. Leaky ReLU

数学公式：

$$\text{LeakyReLU}(x) = \max(\alpha x, x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \tag{158}$$

其中 α 是小的正数（通常取 0.01）。

导数：

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{if } x \leq 0 \end{cases} \tag{159}$$

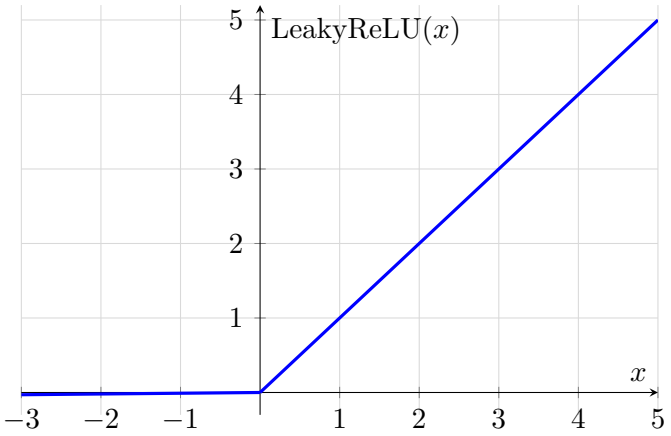
用处：

- 解决 ReLU 的死亡神经元问题
- 需要负值输出的场景
- 对梯度消失敏感的网络

为什么用 Leaky ReLU：

- **避免死亡神经元**：负区间有小的梯度，神经元不会完全” 死亡”
- **保持 ReLU 优点**：正区间仍保持线性，计算高效
- **更好的梯度流**：负区间也有梯度，信息可以反向传播

函数图像（$\alpha = 0.01$）：



5. ELU（Exponential Linear Unit）

数学公式：

$$\text{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{160}$$

其中 α 通常取 1.0。

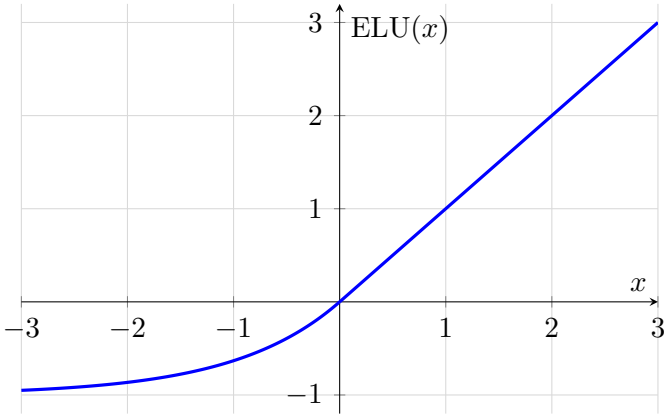
用处：

- 需要平滑负值输出的场景
- 对噪声敏感的任务
- 需要零均值输出的网络

为什么用 ELU：

- **平滑性**：负区间平滑，避免 ReLU 的不连续性
- **零均值**：输出均值接近 0，有助于训练
- **负值处理**：负区间有梯度，避免死亡神经元

函数图像（$\alpha = 1.0$）：



6. Softmax 函数

数学公式：

$$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{C} e^{x_j}} \tag{161}$$

其中 C 是类别数，$\mathbf{x} \in \mathbb{R}^C$。

输出范围：$(0, 1)$，且 $\sum_{i=1}^{C} \text{softmax}(\mathbf{x})_i = 1$

用处：

- 多分类任务的输出层
- 需要输出概率分布的场景
- 注意力机制中的注意力权重计算

为什么用 Softmax：

- **概率分布**：输出是有效的概率分布，总和为 1
- **与交叉熵配合**：与交叉熵损失函数配合使用，梯度形式简单
- **可解释性**：输出可以解释为各类别的概率

26.3.2 激活函数选择指南

激活函数选择指南：

- **Sigmoid**：

  - 适用场景：二分类输出层、门控机制
  - 选择原因：输出概率、历史原因

- **Tanh**：
  - 适用场景：RNN 隐藏层、需要零中心化
  - 选择原因：零均值、比 Sigmoid 更好

- **ReLU**：
  - 适用场景：深度网络隐藏层（首选）
  - 选择原因：缓解梯度消失、计算高效

- **Leaky ReLU**：
  - 适用场景：解决死亡神经元问题
  - 选择原因：负区间有梯度

- **ELU**：
  - 适用场景：需要平滑负值输出
  - 选择原因：平滑、零均值

- **Softmax**：
  - 适用场景：多分类输出层
  - 选择原因：概率分布、与交叉熵配合

应用场景：

例 26.3 (手写数字识别). *MNIST 数据集是经典的图像分类任务，使用 MLP 可以达到较高的准确率。输入是 $28 \times 28 = 784$ 维的像素向量，输出是 10 个类别的概率分布。*

26.4 前向传播

前向传播（Forward Propagation）是神经网络从输入到输出的计算过程。

计算复杂度：对于 L 层网络，每层有 n_l 个神经元，前向传播的时间复杂度为 $O(\sum_{l=1}^{L} n_{l-1} n_l)$。

Algorithm 9 前向传播算法

Require: 输入 $\mathbf{x}$，网络参数 $\{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L}$

Ensure: 输出 $\mathbf{y}$

$\mathbf{h}^{(0)} \leftarrow \mathbf{x}$

for $l = 1$ to $L-1$ **do**

  $\mathbf{z}^{(l)} \leftarrow \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$

  $\mathbf{h}^{(l)} \leftarrow \sigma(\mathbf{z}^{(l)})$

end for

$\mathbf{z}^{(L)} \leftarrow \mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$

$\mathbf{y} \leftarrow \mathbf{z}^{(L)}$（或应用 softmax 等函数）

26.5 反向传播

反向传播（Backpropagation）是训练神经网络的核心算法，通过链式法则计算损失函数对网络参数的梯度。

定义 26.3 (反向传播). 给定损失函数 $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$，反向传播算法计算梯度 $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}$ 和 $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$。

对于输出层：

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \tag{162}$$

对于隐藏层 $l = L-1, L-2, \ldots, 1$：

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(l)}} = (\mathbf{W}^{(l+1)})^T \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l+1)}} \tag{163}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}^{(l)}} \odot \sigma'(\mathbf{z}^{(l)}) \tag{164}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} (\mathbf{h}^{(l-1)})^T \tag{165}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \tag{166}$$

其中，$\odot$ 表示逐元素相乘（*Hadamard* 积），σ' 是激活函数的导数。

梯度消失问题：在深层网络中，梯度在反向传播过程中可能指数级衰减，导致底层参数难以更新。使用 ReLU 激活函数、残差连接、批量归一化等技术可以缓解这一问题。

26.6 损失函数

损失函数（Loss Function）用于衡量模型预测值与真实值之间的差异，是训练神经网络的核心目标函数。选择合适的损失函数对模型性能至关重要。

Algorithm 10 反向传播算法

Require: 前向传播的输出 $\mathbf{y}$，真实标签 $\hat{\mathbf{y}}$，损失函数 $\mathcal{L}$

Ensure: 梯度 $\{\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}\}_{l=1}^{L}$

计算输出层梯度：$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}}$

for $l = L-1$ down to 1 **do**

  $\delta^{(l)} = (\mathbf{W}^{(l+1)})^T \delta^{(l+1)} \odot \sigma'(\mathbf{z}^{(l)})$

  $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \delta^{(l)} (\mathbf{h}^{(l-1)})^T$

  $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$

end for

26.6.1 回归任务的损失函数

均方误差（Mean Squared Error, MSE）：

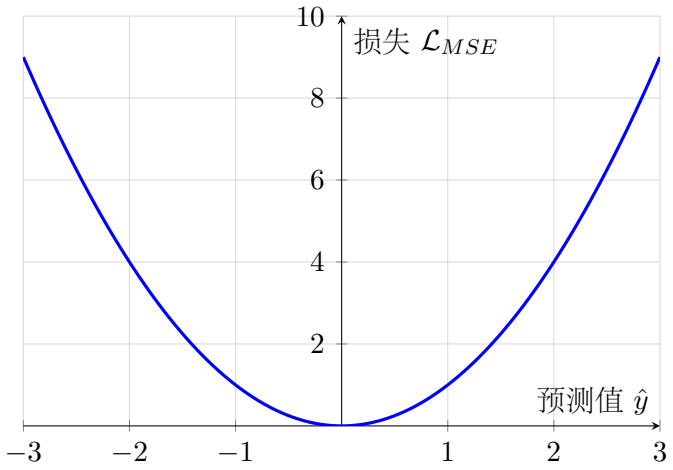
$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{167}$$

其中，y_i 是真实值，$\hat{y}_i$ 是预测值，n 是样本数量。

特点：

- 对大误差敏感，惩罚较大
- 梯度与误差成正比，训练稳定
- 适用于回归问题，假设误差服从高斯分布

函数图像（假设真实值 $y = 0$）：



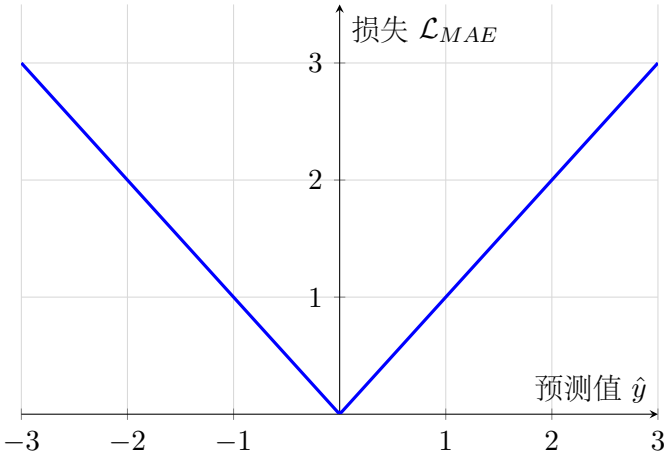
平均绝对误差（Mean Absolute Error, MAE）：

$$\mathcal{L}_{MAE} = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i| \tag{168}$$

特点：

- 对大误差不敏感，对异常值更鲁棒
- 梯度为常数（± 1），训练可能不稳定
- 适用于需要鲁棒性的回归问题

函数图像（假设真实值 $y = 0$）：

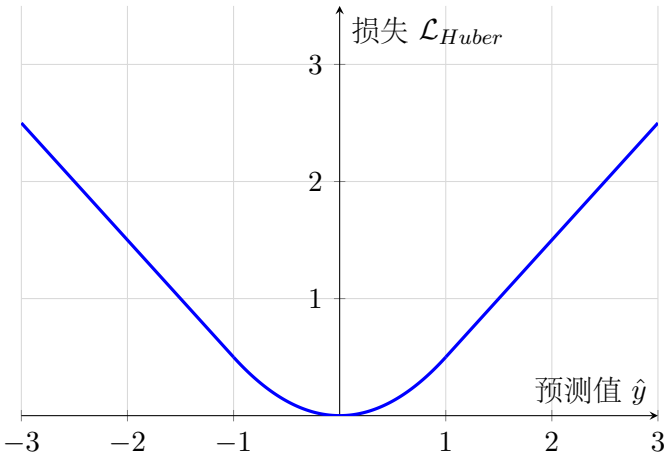


Huber Loss：结合 MSE 和 MAE 的优点：

$$\mathcal{L}_{Huber}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta |y - \hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases} \tag{169}$$

其中，δ 是超参数（通常取 1.0）。当误差小于 δ 时，使用 MSE；否则使用 MAE。

函数图像（假设真实值 $y = 0$，$\delta = 1.0$）：



26.6.2 分类任务的损失函数

交叉熵损失（Cross-Entropy Loss）：对于二分类问题：

$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{170}$$

对于多分类问题（C 个类别）：

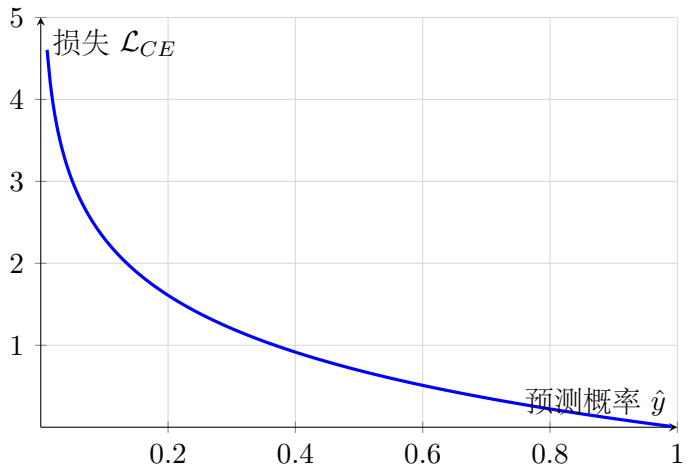
$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c}) \tag{171}$$

其中，$y_{i,c}$ 是真实标签的 one-hot 编码，$\hat{y}_{i,c}$ 是模型预测的概率。

特点：

- 与 softmax 激活函数配合使用效果最佳
- 梯度形式简单：$\frac{\partial \mathcal{L}_{CE}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$
- 适用于分类问题，假设输出服从多项分布

函数图像（二分类，真实标签 $y=1$）：



Focal Loss：解决类别不平衡问题：

$$\mathcal{L}_{Focal} = -\alpha_t(1-p_t)^{\gamma}\log(p_t) \tag{172}$$

其中，p_t 是模型预测的正确类别的概率，α_t 是平衡因子，γ 是聚焦参数（通常 $\gamma=2$）。

特点：

- $(1-p_t)^{\gamma}$ 项降低易分类样本的权重
- 专注于难分类样本，提高模型性能
- 在目标检测任务中广泛应用（如 RetinaNet）

26.6.3 其他损失函数

对比损失（Contrastive Loss）：用于学习相似性：

$$\mathcal{L}_{Contrastive} = \begin{cases} \frac{1}{2}d^2 & \text{if } y = 1 \text{ (相似)} \\ \frac{1}{2}\max(0, m-d)^2 & \text{if } y = 0 \text{ (不相似)} \end{cases} \tag{173}$$

其中，d 是两个样本的欧氏距离，m 是边界参数（margin）。

三元组损失（Triplet Loss）：用于度量学习：

$$\mathcal{L}_{Triplet} = \max(0, d(a,p) - d(a,n) + m) \tag{174}$$

其中，a 是锚样本（anchor），p 是正样本（positive），n 是负样本（negative），$d(\cdot,\cdot)$ 是距离函数，m 是边界。

应用场景：

- 人脸识别：FaceNet 使用三元组损失
- 图像检索：学习图像的相似性表示
- 推荐系统：学习用户和物品的嵌入

感知损失（Perceptual Loss）：用于图像生成任务：

$$\mathcal{L}_{Perceptual} = \sum_{l} \lambda_l \|\phi_l(\hat{I}) - \phi_l(I)\|_2^2 \tag{175}$$

其中，ϕ_l 是预训练网络（如 VGG）的第 l 层特征，λ_l 是权重。

应用场景：

- 图像超分辨率：SRGAN
- 风格迁移：保持内容的同时改变风格
- 图像修复：生成缺失区域

26.6.4 损失函数的选择原则

1. **任务类型**：
    - 回归任务：MSE、MAE、Huber Loss
    - 二分类：二元交叉熵

- 多分类：多元交叉熵
- 多标签分类：二元交叉熵（每个类别独立）

2. **数据分布**：

- 类别不平衡：Focal Loss、加权交叉熵
- 异常值多：MAE、Huber Loss
- 高斯噪声：MSE

3. **训练稳定性**：

- 梯度爆炸风险：使用梯度裁剪
- 梯度消失：选择合适的激活函数和损失函数组合

4. **应用需求**：

- 需要可解释性：使用简单的损失函数
- 需要特定属性：使用定制损失函数（如感知损失）

注 26.1. *在实际应用中，可以组合多个损失函数：*

$$\mathcal{L}_{Total} = \lambda_1 \mathcal{L}_1 + \lambda_2 \mathcal{L}_2 + \cdots + \lambda_k \mathcal{L}_k \tag{176}$$

其中，λ_i 是各损失函数的权重，需要根据任务需求调整。

26.6.5 损失函数使用原因总结

常见损失函数的使用原因：

- **MSE（均方误差）**：
  - 主要用途：回归任务
  - 为什么用它：假设误差服从高斯分布，对大误差敏感，梯度稳定
- **MAE（平均绝对误差）**：
  - 主要用途：回归任务（异常值多）
  - 为什么用它：对异常值鲁棒，梯度为常数
- **Huber Loss**：
  - 主要用途：回归任务（平衡鲁棒性和稳定性）

  - 为什么用它：结合 MSE 和 MAE 优点，小误差用 MSE，大误差用 MAE

- **交叉熵（Cross-Entropy）**：
  - 主要用途：分类任务
  - 为什么用它：与 softmax 配合，梯度形式简单，假设输出为多项分布

- **Focal Loss**：
  - 主要用途：类别不平衡的分类任务
  - 为什么用它：降低易分类样本权重，专注于难样本

- **对比损失（Contrastive Loss）**：
  - 主要用途：相似性学习
  - 为什么用它：学习样本间的相似性关系

- **三元组损失（Triplet Loss）**：
  - 主要用途：度量学习
  - 为什么用它：学习样本间的相对距离关系

- **感知损失（Perceptual Loss）**：
  - 主要用途：图像生成
  - 为什么用它：在特征空间计算损失，更符合人类感知

26.7 评估函数（评估指标）

评估函数（Evaluation Metrics）用于衡量模型在测试集上的性能，与损失函数不同，评估函数通常更关注实际应用中的性能表现。

26.7.1 分类任务的评估指标

1. 准确率（Accuracy）

数学公式：

$$\text{Accuracy} = \frac{\text{正确预测数}}{\text{总样本数}} = \frac{TP + TN}{TP + TN + FP + FN} \tag{177}$$

其中，TP（True Positive）是真阳性，TN（True Negative）是真阴性，FP（False Positive）是假阳性，FN（False Negative）是假阴性。

用处：

- 类别平衡的分类任务
- 需要整体性能评估的场景
- 快速了解模型整体表现

为什么用准确率:

- **直观易懂**：最直观的性能指标，容易理解
- **计算简单**：计算成本低，适合大规模评估
- **通用性强**：适用于所有分类任务

缺点:

- **类别不平衡时失效**：如果 99% 的样本是正类，预测全为正类也能达到 99% 准确率
- **忽略错误类型**：不区分假阳性 and 假阴性

2. 精确率（Precision）

数学公式:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{178}$$

用处:

- 需要减少假阳性的场景（如垃圾邮件检测）
- 关注” 预测为正类的样本中，有多少是真的正类”

为什么用精确率:

- **减少误报**：当假阳性代价高时，精确率更重要
- **资源有限**：当处理正类样本需要消耗资源时，精确率很重要

3. 召回率（Recall）

数学公式:

$$\text{Recall} = \frac{TP}{TP + FN} = \text{Sensitivity} \tag{179}$$

用处:

- 需要减少假阴性的场景（如疾病诊断）

- 关注” 所有正类样本中，有多少被正确识别”

为什么用召回率：

- **减少漏报**：当假阴性代价高时，召回率更重要
- **全面覆盖**：需要尽可能找到所有正类样本

4. F1 分数（F1-Score）

数学公式：

$$F_1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN} \tag{180}$$

用处：

- 需要平衡精确率和召回率的场景
- 类别不平衡的分类任务
- 单一指标评估模型性能

为什么用 F1 分数：

- **平衡指标**：综合考虑精确率和召回率
- **调和平均数**：使用调和平均数，对低值更敏感
- **单一指标**：用一个数字概括模型性能，便于比较

5. ROC 曲线和 AUC

ROC 曲线：以假阳性率（FPR）为横轴，真阳性率（TPR，即召回率）为纵轴的曲线。

AUC（Area Under Curve）：ROC 曲线下的面积。

数学公式：

$$\text{FPR} = \frac{FP}{FP + TN} \tag{181}$$

$$\text{TPR} = \frac{TP}{TP + FN} = \text{Recall} \tag{182}$$

用处：

- 二分类任务的性能评估
- 需要比较不同模型的整体性能

- 类别不平衡的任务

为什么用 AUC：

- **阈值无关**：不依赖于分类阈值，评估模型整体性能
- **类别不平衡鲁棒**：对类别不平衡相对鲁棒
- **概率解释**：AUC 可以解释为” 随机选择一个正样本和一个负样本，模型对正样本的预测概率大于负样本的概率”

26.7.2 回归任务的评估指标

1. 均方误差（MSE）

数学公式：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{183}$$

用处：

- 回归任务的标准评估指标
- 需要惩罚大误差的场景

为什么用 MSE：

- **标准指标**：最常用的回归评估指标
- **可导性**：可导，便于优化
- **大误差敏感**：对大误差敏感，符合实际需求

2. 均方根误差（RMSE）

数学公式：

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{184}$$

用处：

- 需要与原数据相同量纲的评估指标
- 更直观的误差表示

为什么用 RMSE：

- **量纲一致**：与原数据量纲相同，更直观
- **可解释性**：可以直接理解为” 平均误差”

3. 平均绝对误差（MAE）

数学公式：

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{185}$$

用处：

- 需要鲁棒性评估的场景
- 异常值较多的数据

为什么用 MAE：

- **鲁棒性**：对异常值不敏感
- **直观性**：直接表示平均误差，易于理解

4. 决定系数（R^2 Score）

数学公式：

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = 1 - \frac{\text{SS}_{res}}{\text{SS}_{tot}} \tag{186}$$

其中 $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ 是真实值的均值。

输出范围：$(-\infty, 1]$，越接近 1 越好。

用处：

- 需要相对性能评估的场景
- 比较不同模型的解释能力

为什么用 R^2：

- **相对性能**：表示模型相对于简单均值预测的改进程度
- **标准化**：值域固定，便于比较不同数据集上的模型
- **可解释性**：可以解释为” 模型解释了多少方差”

26.7.3 评估指标选择指南

评估指标选择指南：

- **准确率（Accuracy）**：
  - 适用任务：类别平衡的分类任务
  - 选择原因：直观易懂，计算简单
- **精确率（Precision）**：
  - 适用任务：需要减少假阳性
  - 选择原因：关注预测正类的准确性
- **召回率（Recall）**：
  - 适用任务：需要减少假阴性
  - 选择原因：关注找到所有正类
- **F1 分数**：
  - 适用任务：需要平衡精确率和召回率
  - 选择原因：单一指标，平衡两者
- **AUC（ROC 曲线下面积）**：
  - 适用任务：二分类任务，类别不平衡
  - 选择原因：阈值无关，整体性能
- **MSE/RMSE（均方误差/均方根误差）**：
  - 适用任务：回归任务
  - 选择原因：标准指标，大误差敏感
- **MAE（平均绝对误差）**：
  - 适用任务：回归任务（异常值多）
  - 选择原因：鲁棒性强
- **R^2（决定系数）**：
  - 适用任务：回归任务（相对性能）
  - 选择原因：标准化，可解释性强

27 简易神经网络完整示例

本节通过一个完整的简易神经网络示例，展示深度学习的核心概念，包括编码（Encode）、解码（Decode）、训练和推理的全过程。

27.1 网络架构设计

任务：使用一个简单的三层神经网络进行二分类任务（如判断邮件是否为垃圾邮件）。

网络结构：

- **输入层**：$d_{in} = 4$ 个特征（如邮件的关键词频率）
- **隐藏层 1**：$h_1 = 8$ 个神经元，使用 ReLU 激活函数
- **隐藏层 2**：$h_2 = 4$ 个神经元，使用 ReLU 激活函数
- **输出层**：$d_{out} = 1$ 个神经元，使用 Sigmoid 激活函数（输出概率）

数学表示：

对于输入 $\mathbf{x} \in \mathbb{R}^4$，网络的前向传播为：

$$\mathbf{h}^{(1)} = \text{ReLU}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \quad \mathbf{W}^{(1)} \in \mathbb{R}^{8\times 4}, \mathbf{b}^{(1)} \in \mathbb{R}^8 \tag{187}$$

$$\mathbf{h}^{(2)} = \text{ReLU}(\mathbf{W}^{(2)}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}) \quad \mathbf{W}^{(2)} \in \mathbb{R}^{4\times 8}, \mathbf{b}^{(2)} \in \mathbb{R}^4 \tag{188}$$

$$\hat{y} = \sigma(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}) \quad \mathbf{W}^{(3)} \in \mathbb{R}^{1\times 4}, \mathbf{b}^{(3)} \in \mathbb{R}^1 \tag{189}$$

其中 σ 是 Sigmoid 函数：$\sigma(x) = \frac{1}{1+e^{-x}}$。

27.2 从零实现神经网络

完整代码实现：

```
import numpy as np
import pandas as pd

class SimpleNeuralNetwork:
    """简易三层神经网络"""

    def __init__(self, input_size=4, hidden1_size=8, hidden2_size=4,
   output_size=1):
```

```
        """
        初始化神经网络

        参数:
            input_size: 输入特征维度
            hidden1_size: 第一个隐藏层神经元数量
            hidden2_size: 第二个隐藏层神经元数量
            output_size: 输出维度
        """
        # 初始化权重矩阵（使用 Xavier 初始化）
        self.W1 = np.random.randn(hidden1_size, input_size) * np.sqrt(2.0
 / input_size)
        self.b1 = np.zeros((hidden1_size, 1))

        self.W2 = np.random.randn(hidden2_size, hidden1_size) * np.sqrt
(2.0 / hidden1_size)
        self.b2 = np.zeros((hidden2_size, 1))

        self.W3 = np.random.randn(output_size, hidden2_size) * np.sqrt
(2.0 / hidden2_size)
        self.b3 = np.zeros((output_size, 1))

    def relu(self, x):
        """ReLU 激活函数"""
        return np.maximum(0, x)

    def relu_derivative(self, x):
        """ReLU 的导数"""
        return (x > 0).astype(float)

    def sigmoid(self, x):
        """Sigmoid 激活函数"""
        # 防止溢出
        x = np.clip(x, -500, 500)
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        """Sigmoid 的导数"""
        s = self.sigmoid(x)
        return s * (1 - s)
```

```
    def forward(self, X):
        """
        前向传播（Encode）

        参数:
            X: 输入数据 (batch_size, input_size)

        返回:
            y_pred: 预测输出 (batch_size, output_size)
            cache: 中间结果（用于反向传播）
        """
        # 输入层 → 隐藏层1
        self.z1 = X.dot(self.W1.T) + self.b1.T  # (batch_size,
hidden1_size)
        self.a1 = self.relu(self.z1)  # 激活

        # 隐藏层1 → 隐藏层2
        self.z2 = self.a1.dot(self.W2.T) + self.b2.T  # (batch_size,
hidden2_size)
        self.a2 = self.relu(self.z2)  # 激活

        # 隐藏层2 → 输出层
        self.z3 = self.a2.dot(self.W3.T) + self.b3.T  # (batch_size,
output_size)
        self.a3 = self.sigmoid(self.z3)  # 激活（输出概率）

        return self.a3

    def backward(self, X, y, y_pred):
        """
        反向传播（计算梯度）

        参数:
            X: 输入数据 (batch_size, input_size)
            y: 真实标签 (batch_size, 1)
            y_pred: 预测输出 (batch_size, output_size)
        """
        m = X.shape[0]  # 批次大小
```

```
        # 输出层误差
        dz3 = y_pred - y  # (batch_size, output_size)
        dW3 = (1/m) * dz3.T.dot(self.a2)  # (output_size, hidden2_size)
        db3 = (1/m) * np.sum(dz3, axis=0, keepdims=True).T  # (
    output_size, 1)

        # 隐藏层2误差
        da2 = dz3.dot(self.W3)  # (batch_size, hidden2_size)
        dz2 = da2 * self.relu_derivative(self.z2)  # 链式法则
        dW2 = (1/m) * dz2.T.dot(self.a1)  # (hidden2_size, hidden1_size)
        db2 = (1/m) * np.sum(dz2, axis=0, keepdims=True).T  # (
    hidden2_size, 1)

        # 隐藏层1误差
        da1 = dz2.dot(self.W2)  # (batch_size, hidden1_size)
        dz1 = da1 * self.relu_derivative(self.z1)  # 链式法则
        dW1 = (1/m) * dz1.T.dot(X)  # (hidden1_size, input_size)
        db1 = (1/m) * np.sum(dz1, axis=0, keepdims=True).T  # (
    hidden1_size, 1)

        return dW1, db1, dW2, db2, dW3, db3

    def update_parameters(self, dW1, db1, dW2, db2, dW3, db3,
    learning_rate):
        """更新参数（梯度下降）"""
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W3 -= learning_rate * dW3
        self.b3 -= learning_rate * db3

    def compute_loss(self, y_pred, y):
        """
        计算损失函数（二元交叉熵）

        数学公式:
        L = -\frac{1}{m}\sum_{i=1}^m [y_i \log(\hat{y}_i) + (1-y_i)\log
    (1-\hat{y}_i)]
        """
```

```
        m = y.shape[0]
        # 防止 log(0)
        epsilon = 1e-15
        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
        loss = -(1/m) * np.sum(y * np.log(y_pred) + (1 - y) * np.log(1 -
y_pred))
        return loss

    def train(self, X_train, y_train, epochs=1000, learning_rate=0.01,
verbose=True):
        """
        训练神经网络

        参数:
            X_train: 训练数据 (n_samples, input_size)
            y_train: 训练标签 (n_samples, 1)
            epochs: 训练轮数
            learning_rate: 学习率
            verbose: 是否打印训练过程
        """
        losses = []

        for epoch in range(epochs):
            # 前向传播
            y_pred = self.forward(X_train)

            # 计算损失
            loss = self.compute_loss(y_pred, y_train)
            losses.append(loss)

            # 反向传播
            dW1, db1, dW2, db2, dW3, db3 = self.backward(X_train, y_train
, y_pred)

            # 更新参数
            self.update_parameters(dW1, db1, dW2, db2, dW3, db3,
learning_rate)

            # 打印训练进度
            if verbose and (epoch + 1) % 100 == 0:
```

```
                print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}")

        return losses

    def predict(self, X):
        """
        预测（Decode/推理）

        参数:
            X: 输入数据 (n_samples, input_size)

        返回:
            predictions: 预测结果 (n_samples, 1)，值在 [0, 1] 之间
        """
        y_pred = self.forward(X)
        return y_pred

    def predict_class(self, X, threshold=0.5):
        """
        预测类别（二分类）

        参数:
            X: 输入数据
            threshold: 分类阈值

        返回:
            classes: 预测类别 (0 或 1)
        """
        y_pred = self.predict(X)
        return (y_pred > threshold).astype(int)

# 使用示例
# 1. 准备数据（编码：将原始数据转换为数值特征）
np.random.seed(42)
n_samples = 1000
X_train = np.random.randn(n_samples, 4)  # 4个特征
y_train = (X_train.sum(axis=1) > 0).astype(float).reshape(-1, 1)  # 简单
    二分类任务

# 2. 创建和训练模型
```

```
model = SimpleNeuralNetwork(input_size=4, hidden1_size=8, hidden2_size=4,
    output_size=1)
losses = model.train(X_train, y_train, epochs=1000, learning_rate=0.01)

# 3. 预测（解码：将网络输出转换为类别）
X_test = np.random.randn(10, 4)
predictions = model.predict(X_test)
classes = model.predict_class(X_test)

print("\n预测结果:")
print(f"预测概率: {predictions.flatten()}")
print(f"预测类别: {classes.flatten()}")
```

Listing 55: 简易神经网络完整实现（使用 NumPy）

27.3 概念对应关系

通过这个简易神经网络示例，我们可以清楚地看到深度学习各个概念的对应关系：

深度学习概念在代码中的对应：

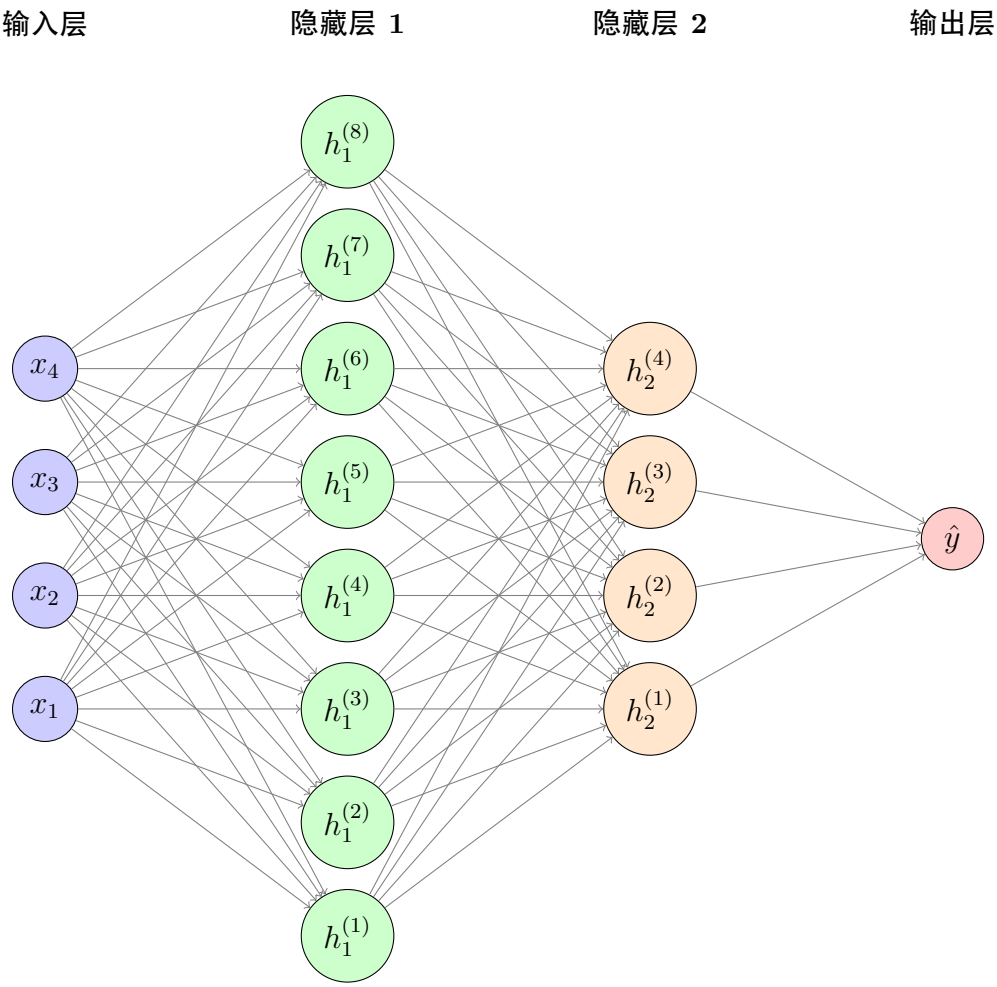
- **输入层**：`X`（输入数据矩阵）
- **隐藏层 1**：`self.a1`（8 个神经元）
- **隐藏层 2**：`self.a2`（4 个神经元）
- **输出层**：`self.a3`（1 个神经元，输出概率）
- **权重矩阵**：`self.W1`, `self.W2`, `self.W3`
- **偏置向量**：`self.b1`, `self.b2`, `self.b3`
- **激活函数**：`relu()`, `sigmoid()`
- **前向传播（Encode）**：`forward()` 方法
- **反向传播**：`backward()` 方法
- **损失函数**：`compute_loss()`（交叉熵）
- **梯度下降**：`update_parameters()`
- **训练**：`train()` 方法
- **推理（Decode）**：`predict()` 和 `predict_class()`

27.4 网络结构图示

网络架构图：

输入层 (4 个神经元) → 隐藏层 1 (8 个神经元, ReLU) → 隐藏层 2 (4 个神经元, ReLU) → 输出层 (1 个神经元, Sigmoid)

详细结构图示：



连接说明：

- 输入层的每个神经元（x_1, x_2, x_3, x_4）与隐藏层 1 的所有 8 个神经元全连接
- 隐藏层 1 的每个神经元与隐藏层 2 的所有 4 个神经元全连接
- 隐藏层 2 的每个神经元与输出层的 1 个神经元全连接

- 每个连接都有一个权重参数，每个神经元都有一个偏置参数

图示说明：

- 每个圆圈代表一个**神经元**（Neuron）
- 每条箭头线代表一个**权重**（Weight）连接
- **输入层**（蓝色）：4 个神经元，接收 4 个特征 $\mathbf{x} = [x_1, x_2, x_3, x_4]^T$
- **隐藏层 1**（绿色）：8 个神经元，每个神经元接收 4 个输入，使用 ReLU 激活函数
- **隐藏层 2**（橙色）：4 个神经元，每个神经元接收 8 个输入，使用 ReLU 激活函数
- **输出层**（红色）：1 个神经元，接收 4 个输入，使用 Sigmoid 激活函数，输出概率 $\hat{y} \in [0, 1]$

连接关系：

- 输入层 → 隐藏层 1：$4 \times 8 = 32$ 个权重连接
- 隐藏层 1 → 隐藏层 2：$8 \times 4 = 32$ 个权重连接
- 隐藏层 2 → 输出层：$4 \times 1 = 4$ 个权重连接
- 总参数量：$(4 \times 8 + 8) + (8 \times 4 + 4) + (4 \times 1 + 1) = 40 + 36 + 5 = 81$ 个参数

27.5 训练过程详解

训练步骤：

1. **前向传播（Encode）**：
   - 输入数据 $\mathbf{x}$ 经过网络，逐层计算
   - 每层计算：$\mathbf{z} = \mathbf{W}\mathbf{h} + \mathbf{b}$，然后应用激活函数
   - 最终得到预测输出 $\hat{y}$
2. **计算损失**：
   - 比较预测值 $\hat{y}$ 和真实值 y
   - 使用交叉熵损失函数：$L = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$
3. **反向传播**：
   - 从输出层开始，逐层向前计算梯度

- 使用链式法则：$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \mathbf{h}^{(l)}} \frac{\partial \mathbf{h}^{(l)}}{\partial W^{(l)}}$
- 计算所有权重和偏置的梯度

4. **参数更新**：
   - 使用梯度下降更新参数：$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}$
   - 其中 η 是学习率
5. **重复迭代**：重复步骤 1-4，直到损失收敛

27.6 推理过程（Decode）

推理步骤：

1. **输入编码**：将原始输入（如文本、图像特征）编码为数值向量 $\mathbf{x}$
2. **前向传播**：数据通过网络，得到输出 $\hat{y} \in [0, 1]$
3. **输出解码**：
   - 对于二分类：如果 $\hat{y} > 0.5$，预测为正类（1），否则为负类（0）
   - 对于多分类：使用 softmax 将输出转换为概率分布，选择概率最大的类别

数学表示：

编码过程：$\mathbf{x} = \text{encode}(\text{原始输入})$

前向传播：$\hat{y} = f(\mathbf{x}; \theta)$，其中 f 是神经网络，θ 是参数

解码过程：预测类别 $= \text{decode}(\hat{y}) = \begin{cases} 1 & \text{if } \hat{y} > 0.5 \\ 0 & \text{otherwise} \end{cases}$

28 深度网络架构

28.1 全连接层（Fully Connected Layer）

概念解释：全连接层（Fully Connected Layer），也称为密集层（Dense Layer）或线性层（Linear Layer），是神经网络中最基础的层类型。在全连接层中，每个神经元都与前一层的所有神经元相连。

数学表示：

对于输入 $\mathbf{x} \in \mathbb{R}^{d_{in}}$，全连接层的输出为：

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{190}$$

其中：

- $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$：权重矩阵
- $\mathbf{b} \in \mathbb{R}^{d_{out}}$：偏置向量
- d_{in}：输入维度
- d_{out}：输出维度（神经元数量）

参数量：对于输入维度 d_{in}、输出维度 d_{out} 的全连接层，参数量为：

$$\text{参数量} = d_{in} \times d_{out} + d_{out} = d_{out}(d_{in} + 1) \tag{191}$$

通俗解释：全连接层就像一个”完全连接的转换器”。想象一个房间里有 d_{in} 个人（输入神经元），另一个房间里有 d_{out} 个人（输出神经元）。每个人都要和另一个房间的每个人握手（连接），每次握手都有一个”权重”（连接强度）。最后，输出房间的每个人计算自己收到的所有”握手强度”的总和，再加上自己的”基础值”（偏置），就得到了输出。

应用场景：

- MLP 中的主要层类型
- CNN 中的分类层（在卷积和池化之后）
- Transformer 中的前馈网络（Feed-Forward Network）

优势与局限：

优势：

- 表达能力强大，可以学习任意复杂的映射关系
- 实现简单，计算高效（矩阵乘法）

局限：

- 参数量大，容易过拟合
- 忽略了输入的空间结构（如图像的局部相关性）
- 对输入尺寸敏感，需要固定输入维度

28.1.1 输入层和输出层与全连接层的关系

核心问题：输入层和输出层是” 全连接层” 吗?

这个问题需要分两部分回答：

(1) 输入层本身通常不被视为” 全连接层”：

输入层只是一个数据接口，它没有可学习的权重或偏置。它只是把原始数据 $\mathbf{x} \in \mathbb{R}^d$ 传递给下一层。

全连接层必须包含可学习的参数（$\mathbf{W}, \mathbf{b}$）并执行 $\mathbf{W}\mathbf{x} + \mathbf{b}$ 运算，而输入层不做任何计算。

结论：输入层不等于全连接层（它甚至不算一个” 层” 在参数意义上）。

(2) 输出层通常是全连接层：

在绝大多数神经网络中（包括分类任务、回归任务），输出层是一个全连接层。

例如：

```
nn.Linear(128, 1)  # 从128维隐藏表示映射到1个输出（如回归值）
nn.Linear(128, 10)  # 从128维隐藏表示映射到10个输出（如10分类）
```

Listing 56: 输出层作为全连接层

这就是标准的全连接层（带权重和偏置）。

结论：输出层通常是全连接层（但它的角色是” 输出”，不是” 隐藏”）。

术语使用规范：

层类型与全连接层的关系：

- **输入层**：
  - 是隐藏层? 否
  - 是全连接层? 通常不算（无参数，仅传递数据）
- **中间层**：
  - 是隐藏层? 是
  - 是全连接层? 常是（如 MLP 中的 `Linear(256,128) + ReLU`）
- **输出层**：
  - 是隐藏层? 否
  - 是全连接层? 通常是（如 `Linear(128, 1)`，用于最终映射）

重要说明:

虽然输出层是全连接层，但我们不会称它为”隐藏层”，因为它的功能是输出，不是隐藏表示。当配置 `nn.Linear(in, out)` 时，无论它在中间还是最后，都是全连接层，但只有中间的才是隐藏层。

总结:

- **第一层（输入层）和最后一层（输出层）是不是隐藏层?**
  都不是。隐藏层仅指中间层，即夹在输入层和输出层之间的所有层。
- **它们是不是全连接层?**
  - **输入层**：不是。输入层无参数，不算计算层，只是数据的载体。
  - **输出层**：通常是。输出层用 `Linear` 实现，有 **W** 和 **b** 参数，执行线性变换。

代码示例:

```
import torch.nn as nn

model = nn.Sequential(
    # 输入层: 只是数据入口, 不是全连接层
    # x 直接传入下一层

    # 隐藏层1: 全连接层 + 激活函数
    nn.Linear(100, 256),  # ← 这是全连接层, 也是隐藏层
    nn.ReLU(),

    # 隐藏层2: 全连接层 + 激活函数
    nn.Linear(256, 128),  # ← 这是全连接层, 也是隐藏层
    nn.ReLU(),

    # 输出层: 全连接层 (但不是隐藏层)
    nn.Linear(128, 1)  # ← 这是全连接层, 但不是隐藏层
)
```

Listing 57: 层类型的代码示例

小贴士:

- 当说”模型有 2 个隐藏层”时，指的是中间有 2 个可学习的层，不包括输入和输出

- 当配置 `nn.Linear(in, out)` 时，无论它在中间还是最后，都是全连接层，但只有中间的才是隐藏层
- 输入层没有参数，输出层有参数（$\mathbf{W}$ 和 $\mathbf{b}$），但输出层不是隐藏层

28.2 卷积层（Convolutional Layer）

概念解释：卷积层是卷积神经网络的核心组件，通过卷积操作提取局部特征。卷积操作利用局部连接和权重共享，大幅减少参数量。

数学表示：

对于二维卷积，给定输入特征图 $\mathbf{X} \in \mathbb{R}^{H \times W \times C}$ 和卷积核 $\mathbf{K} \in \mathbb{R}^{k_h \times k_w \times C \times C'}$，卷积操作定义为：

$$(\mathbf{X} * \mathbf{K})_{i,j,c'} = \sum_{c=1}^{C} \sum_{u=0}^{k_h-1} \sum_{v=0}^{k_w-1} \mathbf{X}_{i+u,j+v,c} \cdot \mathbf{K}_{u,v,c,c'} \tag{192}$$

其中，$*$ 表示卷积操作。

参数量：对于卷积核大小为 $k_h \times k_w$，输入通道数 C，输出通道数 C' 的卷积层：

$$\text{参数量} = k_h \times k_w \times C \times C' + C' \tag{193}$$

通俗解释：卷积层就像一个”滑动窗口特征提取器”。想象你拿着一块小模板（卷积核）在图像上滑动，每次滑动时，模板会”扫描”一小块区域，提取这块区域的特征。不同的模板提取不同的特征（如边缘、纹理、形状等）。通过多个模板（多个卷积核），可以同时提取多种特征。

核心特性：

- **局部连接**：每个神经元只连接输入的一个局部区域
- **权重共享**：同一卷积核在整个输入上滑动，共享参数
- **平移不变性**：对输入的位置变化具有鲁棒性

应用场景：

- 图像分类：提取图像的层次化特征
- 目标检测：定位和识别图像中的物体
- 语义分割：对图像的每个像素进行分类

28.3 池化层（Pooling Layer）

概念解释：池化层用于降低特征图的空间维度，减少参数量和计算量，同时提供一定的平移不变性。

数学表示：

最大池化（Max Pooling）：

$$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{u,v \in \text{window}} \mathbf{X}_{i+u,j+v} \tag{194}$$

平均池化（Average Pooling）：

$$\text{AvgPool}(\mathbf{X})_{i,j} = \frac{1}{|\text{window}|} \sum_{u,v \in \text{window}} \mathbf{X}_{i+u,j+v} \tag{195}$$

其中，window 是池化窗口（如 2×2）。

参数量：池化层没有可学习参数，只有超参数（窗口大小、步长）。

通俗解释：池化层就像一个” 信息压缩器”。想象你把一张大照片分成很多 2×2 的小块，然后对每块做以下操作：

- **最大池化**：保留每块中最亮的像素（最显著的特征）
- **平均池化**：计算每块的平均亮度（平滑的特征）

这样，照片的尺寸就缩小了一半，但保留了最重要的信息。

作用：

- **降维**：减少特征图的空间尺寸，降低计算量
- **特征不变性**：对小的平移和变形具有鲁棒性
- **防止过拟合**：减少参数量，降低模型复杂度

比较：

最大池化 vs 平均池化：

- **保留信息**：
  - 最大池化：保留最显著特征
  - 平均池化：保留整体特征

- **对噪声**：
  - 最大池化：更鲁棒
  - 平均池化：更敏感
- **适用场景**：
  - 最大池化：特征检测
  - 平均池化：平滑特征

28.4 全连接网络

全连接网络（Fully Connected Network, FCN）是由多个全连接层组成的深度网络架构。

架构特点：

- 参数量大：对于 L 层网络，参数量为 $\sum_{l=1}^{L}(n_{l-1}+1)n_l$
- 计算密集：需要大量的矩阵乘法运算
- 适合处理向量化输入：如图像展平后的向量、特征向量等

应用场景：

例 28.1 (图像分类). *在 CIFAR-10 数据集中，可以将 $32\times32\times3$ 的图像展平为 3072 维向量，然后通过多层全连接网络进行分类。*

28.5 卷积神经网络

卷积神经网络（Convolutional Neural Network, CNN）是专门设计用于处理具有网格结构数据（如图像）的深度学习架构。

定义 28.1 (卷积操作). *对于二维卷积，给定输入特征图 $\mathbf{X}\in\mathbb{R}^{H\times W\times C}$ 和卷积核 $\mathbf{K}\in\mathbb{R}^{k_h\times k_w\times C\times C'}$，卷积操作定义为：*

$$(\mathbf{X}*\mathbf{K})_{i,j,c'}=\sum_{c=1}^{C}\sum_{u=0}^{k_h-1}\sum_{v=0}^{k_w-1}\mathbf{X}_{i+u,j+v,c}\cdot\mathbf{K}_{u,v,c,c'} \tag{196}$$

其中，$*$ 表示卷积操作。

CNN 的核心组件：

1. **卷积层（Convolutional Layer）**：通过卷积操作提取局部特征
   - 局部连接：每个神经元只连接输入的一个局部区域
   - 权重共享：同一卷积核在整个输入上滑动，共享参数
   - 平移不变性：对输入的位置变化具有鲁棒性
2. **池化层（Pooling Layer）**：降低特征图的空间维度，减少参数量和计算量
   - 最大池化：$\text{MaxPool}(\mathbf{X})_{i,j} = \max_{u,v \in \text{window}} \mathbf{X}_{i+u,j+v}$
   - 平均池化：$\text{AvgPool}(\mathbf{X})_{i,j} = \frac{1}{|\text{window}|} \sum_{u,v \in \text{window}} \mathbf{X}_{i+u,j+v}$
3. **全连接层**：在卷积和池化之后，用于最终的分类或回归

经典 CNN 架构：

例 28.2 (LeNet-5). *LeNet-5 是 Yann LeCun 在 1998 年提出的用于手写数字识别的 CNN 架构，包含两个卷积层、两个池化层和三个全连接层。*

例 28.3 (AlexNet). *AlexNet 在 2012 年 ImageNet 竞赛中取得突破性成果，包含 5 个卷积层和 3 个全连接层，首次使用 ReLU 激活函数和 Dropout 技术。*

例 28.4 (VGGNet). *VGGNet 使用更深的网络（16-19 层）和更小的卷积核（3×3），证明了网络深度的重要性。*

例 28.5 (ResNet). *ResNet 引入残差连接，解决了深层网络的退化问题，可以训练超过 100 层的网络。*

CNN 的优势：

- **参数效率**：通过局部连接和权重共享，大幅减少参数量
- **平移不变性**：对输入的位置变化具有鲁棒性
- **层次化特征学习**：底层学习边缘、纹理等低级特征，高层学习语义、对象等高级特征
- **广泛应用**：在图像分类、目标检测、语义分割等任务中表现优异

应用场景：

- **图像分类**：ImageNet 图像分类挑战
- **目标检测**：YOLO、R-CNN 系列
- **语义分割**：FCN、U-Net
- **人脸识别**：FaceNet、DeepFace
- **医学影像**：CT、MRI 图像分析

28.6 循环神经网络

循环神经网络（Recurrent Neural Network, RNN）是专门设计用于处理序列数据的神经网络架构。

定义 28.2 (RNN). *RNN 通过维护隐藏状态来记忆历史信息。对于输入序列 $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$，RNN 的计算过程为：*

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}_h) \tag{197}$$

$$\mathbf{y}_t = \mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y \tag{198}$$

其中，$\mathbf{h}_t$ 是时刻 t 的隐藏状态，$\mathbf{W}_h$、$\mathbf{W}_x$、$\mathbf{W}_y$ 是权重矩阵。

RNN 的特点：

- **参数共享**：所有时间步共享相同的参数
- **记忆能力**：通过隐藏状态传递历史信息
- **变长输入**：可以处理不同长度的序列

梯度消失和梯度爆炸：在长序列中，RNN 容易出现梯度消失或梯度爆炸问题，导致难以学习长期依赖关系。

LSTM（Long Short-Term Memory）：LSTM 通过引入门控机制解决长期依赖问题。

定义 28.3 (LSTM). *LSTM 单元包含三个门：遗忘门、输入门和输出门。*

$$\mathbf{f}_t = \sigma(\mathbf{W}_f [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \text{（遗忘门）} \tag{199}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_i [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \text{（输入门）} \tag{200}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{W}_C [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_C) \tag{201}$$

$$\mathbf{C}_t = \mathbf{f}_t \odot \mathbf{C}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{C}}_t \quad \text{（细胞状态）} \tag{202}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \text{（输出门）} \tag{203}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \tag{204}$$

其中，$\mathbf{C}_t$ 是细胞状态，$\odot$ 表示逐元素相乘。

GRU（Gated Recurrent Unit）：GRU 是 LSTM 的简化版本，只有两个门（更新门和重置门），计算效率更高。

应用场景：

- **自然语言处理**：机器翻译、文本生成、情感分析
- **语音识别**：语音转文字
- **时间序列预测**：股票价格预测、天气预测
- **序列标注**：命名实体识别、词性标注

例 28.6 (机器翻译). *Google 的神经机器翻译系统使用编码器-解码器架构，编码器将源语言序列编码为固定维度的向量，解码器生成目标语言序列。*

29 深度学习优化技术

29.1 梯度下降变体

梯度下降及其变体是训练深度神经网络的核心优化算法。

批量梯度下降（Batch Gradient Descent）：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}(\theta_t) \tag{205}$$

其中，$\nabla_\theta \mathcal{L}(\theta_t) = \frac{1}{m} \sum_{i=1}^{m} \nabla_\theta \ell(f(\mathbf{x}_i; \theta_t), y_i)$ 是损失函数在整个训练集上的梯度。

随机梯度下降（Stochastic Gradient Descent, SGD）：

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \ell(f(\mathbf{x}_i; \theta_t), y_i) \tag{206}$$

每次只使用一个样本更新参数，计算速度快但梯度估计方差大。

小批量梯度下降（Mini-batch Gradient Descent）：

$$\theta_{t+1} = \theta_t - \eta \frac{1}{B} \sum_{i \in \mathcal{B}_t} \nabla_\theta \ell(f(\mathbf{x}_i; \theta_t), y_i) \tag{207}$$

使用小批量样本（通常 $B = 32, 64, 128$），在计算效率和梯度稳定性之间取得平衡。

动量法（Momentum）：

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1 - \beta) \nabla_\theta \mathcal{L}(\theta_t) \tag{208}$$

$$\theta_{t+1} = \theta_t - \eta \mathbf{v}_t \tag{209}$$

其中，$\beta \in [0, 1)$ 是动量系数（通常取 0.9）。动量法可以加速收敛并减少震荡。

Adam（Adaptive Moment Estimation）：Adam 结合了动量和自适应学习率的思想：

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t) \quad \text{（一阶矩估计）} \tag{210}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\nabla_\theta \mathcal{L}(\theta_t))^2 \quad \text{（二阶矩估计）} \tag{211}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \quad \text{（偏差修正）} \tag{212}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{213}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t \tag{214}$$

其中，$\beta_1 = 0.9$，$\beta_2 = 0.999$，$\epsilon = 10^{-8}$。Adam 是目前最常用的优化算法之一。

29.2 批量归一化

批量归一化（Batch Normalization, BN）通过归一化每层的输入分布来加速训练并提高模型稳定性。

定义 29.1 (批量归一化). 对于小批量 $\mathcal{B} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_B\}$，批量归一化计算：

$$\mu_{\mathcal{B}} = \frac{1}{B}\sum_{i=1}^{B} \mathbf{x}_i \quad \text{（批量均值）} \tag{215}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{B}\sum_{i=1}^{B} (\mathbf{x}_i - \mu_{\mathcal{B}})^2 \quad \text{（批量方差）} \tag{216}$$

$$\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad \text{（归一化）} \tag{217}$$

$$\mathbf{y}_i = \gamma \hat{\mathbf{x}}_i + \beta \quad \text{（缩放和平移）} \tag{218}$$

其中，γ 和 β 是可学习的参数，ϵ 是小的常数（如 10^{-5}）防止除零。

批量归一化的优势：

- **加速训练**：允许使用更大的学习率
- **减少内部协变量偏移**：稳定每层的输入分布
- **正则化效果**：减少对 Dropout 的依赖
- **缓解梯度消失**：使激活值分布在合适的范围内

应用位置：通常放在卷积层或全连接层之后、激活函数之前（或之后，取决于具体实现）。

29.3 Dropout

Dropout 是一种正则化技术，通过在训练过程中随机丢弃部分神经元来防止过拟合。

定义 29.2 (Dropout)**.** *在训练阶段，对于每个神经元，以概率 p（通常 $p = 0.5$）将其输出置为零：*

$$\mathbf{h}_i^{(l)} = \begin{cases} 0 & \text{以概率} p \\ \frac{\mathbf{h}_i^{(l)}}{1-p} & \text{以概率} 1-p \end{cases} \tag{219}$$

在测试阶段，所有神经元都参与计算，但输出需要乘以 $(1-p)$ 以保持期望值不变。

Dropout 的机制：

- 防止神经元之间的共适应，迫使网络学习更鲁棒的特征
- 相当于训练多个不同的子网络，测试时进行集成
- 减少过拟合，提高泛化能力

应用场景：通常应用在全连接层，卷积层也可以使用（Spatial Dropout）。

29.4 残差连接

残差连接（Residual Connection）通过跳跃连接将输入直接传递到输出，解决了深层网络的退化问题。

定义 29.3 (残差块)**.** *残差块（Residual Block）的计算为：*

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x} \tag{220}$$

其中，$\mathcal{F}(\mathbf{x})$ 是残差函数（通常是几个卷积层），$\mathbf{x}$ 是输入（通过跳跃连接直接传递）。

残差连接的优势：

- **解决退化问题**：即使残差函数学习到零映射，网络也能保持恒等映射
- **缓解梯度消失**：梯度可以直接通过跳跃连接反向传播
- **允许训练更深的网络**：ResNet 可以训练超过 1000 层的网络

架构变体：

- **ResNet**：基本的残差网络
- **DenseNet**：密集连接，每层都连接到所有后续层
- **Highway Networks**：使用门控机制控制信息流

例 29.1 (ResNet-50). *ResNet-50 包含 50 个卷积层，在 ImageNet 上取得了优异的性能，成为计算机视觉领域的标准架构之一。*

30 表示学习与嵌入

表示学习（Representation Learning）旨在学习数据的有效表示，使得学习任务更容易完成。嵌入（Embedding）是将离散对象（如词、节点）映射到连续向量空间的技术。

30.1 词嵌入

词嵌入（Word Embedding）将词汇表中的词映射到低维连续向量空间，使得语义相似的词在向量空间中距离较近。

Word2Vec：Word2Vec 是 Google 提出的词嵌入方法，包含两种模型：

1. **Skip-gram**：给定中心词，预测上下文词
2. **CBOW（Continuous Bag of Words）**：给定上下文词，预测中心词

Skip-gram 模型：

$$P(w_{t+j}|w_t) = \frac{\exp(\mathbf{v}_{w_{t+j}}^T \mathbf{u}_{w_t})}{\sum_{w \in V} \exp(\mathbf{v}_w^T \mathbf{u}_{w_t})} \tag{221}$$

其中，$\mathbf{u}_w$ 是词 w 作为中心词的向量，$\mathbf{v}_w$ 是词 w 作为上下文词的向量，V 是词汇表。

负采样：为了加速训练，使用负采样近似 softmax：

$$\log \sigma(\mathbf{v}_{w_O}^T \mathbf{u}_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} [\log \sigma(-\mathbf{v}_{w_i}^T \mathbf{u}_{w_I})] \tag{222}$$

其中，w_O 是正样本（真实上下文词），w_i 是负样本（从噪声分布 $P_n(w)$ 中采样）。

GloVe（Global Vectors）：GloVe 结合了全局统计信息和局部上下文窗口：

$$\mathcal{L} = \sum_{i,j=1}^{V} f(X_{ij})(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{223}$$

其中，X_{ij} 是词 i 和词 j 的共现次数，$f(X_{ij})$ 是权重函数。

应用场景：

- **文本分类**：将词嵌入作为特征输入分类器
- **机器翻译**：作为编码器的输入
- **情感分析**：捕捉词的语义信息
- **推荐系统**：将物品描述转换为向量

30.2 图嵌入

图嵌入（Graph Embedding）将图中的节点映射到低维向量空间，保持图的结构和属性信息。

DeepWalk：DeepWalk 使用随机游走生成节点序列，然后使用 Word2Vec 学习节点嵌入：

Algorithm 11 DeepWalk 算法

Require: 图 $G(V, E)$，窗口大小 w，游走长度 t，嵌入维度 d

Ensure: 节点嵌入 $\Phi : V \to \mathbb{R}^d$

初始化随机游走序列集合 $\mathcal{S} = \emptyset$
for 每个节点 $v_i \in V$ **do**
  for $r = 1$ to γ **do**
    从 v_i 开始执行长度为 t 的随机游走，得到序列 s_i
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{s_i\}$
  end for
end for
使用 Skip-gram 在序列集合 $\mathcal{S}$ 上学习节点嵌入

Node2Vec：Node2Vec 使用有偏随机游走，平衡广度优先搜索（BFS）和深度优先搜索（DFS）：

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases} \tag{224}$$

其中，π_{vx} 是未归一化的转移概率，Z 是归一化常数。

应用场景：

- **节点分类**：预测节点的类别
- **链接预测**：预测节点之间是否存在边
- **社区检测**：发现图中的社区结构
- **推荐系统**：用户-物品二部图嵌入

31 注意力机制与 Transformer 架构

31.1 注意力机制

注意力机制（Attention Mechanism）允许模型在处理序列时动态地关注不同位置的信息。

定义 31.1 (注意力机制). 给定查询（*Query*）$\mathbf{Q}$、键（*Key*）$\mathbf{K}$ 和值（*Value*）$\mathbf{V}$，注意力机制计算：

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{225}$$

其中，d_k 是键的维度，$\sqrt{d_k}$ 是缩放因子，防止点积过大导致 *softmax* 梯度消失。

注意力机制的直观理解：

- **查询（Query）**：表示当前需要关注什么信息
- **键（Key）**：表示每个位置提供什么信息
- **值（Value）**：表示每个位置的实际内容
- 注意力权重：通过查询和键的相似度计算，决定关注哪些位置

自注意力（Self-Attention）：当查询、键和值都来自同一输入序列时，称为自注意力：

$$\mathbf{Z} = \text{Attention}(\mathbf{X}\mathbf{W}_Q, \mathbf{X}\mathbf{W}_K, \mathbf{X}\mathbf{W}_V) \tag{226}$$

其中，$\mathbf{X}$ 是输入序列，$\mathbf{W}_Q$、$\mathbf{W}_K$、$\mathbf{W}_V$ 是可学习的权重矩阵。

多头注意力（Multi-Head Attention）：使用多个注意力头并行计算，然后拼接：

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \tag{227}$$

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{228}$$

其中，h 是注意力头的数量（通常 $h = 8$）。

31.2 Transformer 架构

Transformer 是 Vaswani 等人在 2017 年提出的完全基于注意力机制的架构，成为现代 NLP 的基础。

Transformer 的整体架构：

- **编码器（Encoder）**：N 个相同的层堆叠，每层包含：
  - 多头自注意力子层
  - 前馈神经网络子层
  - 残差连接和层归一化
- **解码器（Decoder）**：N 个相同的层堆叠，每层包含：
  - 掩码多头自注意力子层（防止看到未来信息）
  - 编码器-解码器注意力子层
  - 前馈神经网络子层
  - 残差连接和层归一化

位置编码（Positional Encoding）：由于 Transformer 没有循环结构，需要显式编码位置信息：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{229}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{230}$$

其中，pos 是位置，i 是维度索引，d_{model} 是模型维度。

前馈神经网络：

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{231}$$

通常使用两层全连接网络，中间使用 ReLU 激活函数。

Transformer 的优势：

- **并行计算**：不像 RNN 需要顺序计算，可以并行处理所有位置
- **长距离依赖**：注意力机制可以直接建模任意距离的依赖关系
- **可解释性**：注意力权重可以可视化，了解模型关注的位置

应用场景：

- **机器翻译**：Transformer 在 WMT 2014 数据集上取得了最先进的性能
- **文本生成**：GPT 系列模型基于 Transformer 解码器
- **文本理解**：BERT 基于 Transformer 编码器
- **代码生成**：GitHub Copilot 使用基于 Transformer 的模型

例 31.1 (BERT). *BERT（Bidirectional Encoder Representations from Transformers）使用 Transformer 编码器和掩码语言模型预训练，在多个 NLP 任务上取得了突破性成果。*

32 强化学习及其应用

强化学习（Reinforcement Learning, RL）是机器学习的一个分支，通过智能体（Agent）与环境（Environment）交互来学习最优策略。

32.1 强化学习基础

定义 32.1 (马尔可夫决策过程). *马尔可夫决策过程（Markov Decision Process, MDP）由五元组 $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ 定义：*

- $\mathcal{S}$：*状态空间*
- $\mathcal{A}$：*动作空间*
- $\mathcal{P}$：*状态转移概率，$P(s'|s,a)$*
- $\mathcal{R}$：*奖励函数，$R(s,a,s')$*
- $\gamma \in [0,1]$：*折扣因子*

强化学习的目标：学习策略 $\pi(a|s)$，使得累积奖励的期望最大：

$$J(\pi) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{232}$$

其中，$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$ 是轨迹。

价值函数：

- **状态价值函数**：$V^\pi(s) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$
- **动作价值函数**：$Q^\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a]$

32.2 深度强化学习

深度强化学习（Deep Reinforcement Learning）将深度学习与强化学习结合，使用神经网络近似价值函数或策略。

Deep Q-Network (DQN)：DQN 使用深度神经网络近似 Q 函数：

$$Q(s, a; \theta) \approx Q^*(s, a) \tag{233}$$

DQN 的关键技术：

- **经验回放（Experience Replay）**：存储经验 (s_t, a_t, r_t, s_{t+1})，随机采样进行训练
- **目标网络（Target Network）**：使用独立的网络计算目标 Q 值，提高训练稳定性

策略梯度方法：直接优化策略参数：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R_t \right] \tag{234}$$

其中，$R_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$ 是回报。

Actor-Critic 方法：结合策略梯度（Actor）和价值函数（Critic）：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) A_t \right] \tag{235}$$

$$A_t = Q(s_t, a_t) - V(s_t) \quad \text{（优势函数）} \tag{236}$$

应用场景：

- **游戏 AI**：AlphaGo、AlphaStar、OpenAI Five
- **机器人控制**：机器人导航、操作任务
- **自动驾驶**：决策和控制
- **推荐系统**：动态推荐策略
- **资源调度**：云计算资源分配

例 32.1 (AlphaGo). *AlphaGo 使用深度强化学习（结合蒙特卡洛树搜索）击败了世界围棋冠军，展示了深度强化学习在复杂决策问题中的强大能力。*

例 32.2 (OpenAI Five). *OpenAI Five 在 Dota 2 游戏中击败了世界冠军团队，展示了多智能体强化学习的潜力。*

33 深度学习研究案例

33.1 计算机视觉

例 33.1 (ImageNet 图像分类). *ImageNet* 大规模视觉识别挑战(*ILSVRC*)推动了深度学习在计算机视觉领域的发展。*AlexNet*(*2012*) 首次使用深度卷积神经网络取得突破，随后的 *VGGNet*、*ResNet*、*DenseNet* 等不断刷新记录，错误率从 *25.8%* 降低到 *2.25%*。

例 33.2 (目标检测). *R-CNN* 系列 (*R-CNN*、*Fast R-CNN*、*Faster R-CNN*) 和 *YOLO* 系列推动了目标检测技术的发展，在 *COCO* 数据集上取得了优异的性能，广泛应用于自动驾驶、安防监控等领域。

33.2 自然语言处理

例 33.3 (机器翻译). *Google* 神经机器翻译 (*GNMT*) 使用编码器-解码器架构和注意力机制，在多个语言对上取得了接近人类水平的翻译质量。

例 33.4 (预训练语言模型). *BERT*、*GPT*、*T5* 等预训练语言模型通过大规模无监督预训练和任务特定微调，在多个 *NLP* 任务上取得了显著提升，推动了 *NLP* 领域的范式转变。

33.3 语音识别

例 33.5 (语音转文字). *Deep Speech*、*Wav2Vec* 等模型使用深度神经网络进行语音识别，在多个数据集上达到了接近人类水平的准确率，广泛应用于语音助手、实时字幕等场景。

33.4 多模态学习

例 33.6 (图像描述生成). *Show and Tell*、*Bottom-Up and Top-Down Attention* 等模型结合 *CNN* 和 *RNN/Transformer*，能够生成图像的文本描述，在 *COCO Captions* 数据集上取得了优异的性能。

34 总结与展望

深度学习作为人工智能领域的重要分支，通过多层神经网络学习数据的层次化表示，在多个领域取得了突破性进展。从感知机到 Transformer，从图像分类到自然语言处理，深度学习不断推动着人工智能的发展。

未来发展方向:

- **更高效的架构**：减少参数量和计算量，提高推理速度
- **更好的可解释性**：理解模型的决策过程，提高可信度
- **少样本学习**：减少对大规模标注数据的依赖
- **多模态融合**：整合文本、图像、语音等多种模态的信息
- **自监督学习**：从无标注数据中学习有效表示
- **神经符号结合**：结合神经网络的表示能力和符号推理的逻辑能力

深度学习将继续在科学研究、工业应用和社会生活中发挥重要作用，推动人工智能技术的进一步发展。

35 作业与练习

35.1 概念题

1. **梯度消失问题**：
   - 解释什么是梯度消失问题，为什么会出现?
   - 列举至少三种缓解梯度消失问题的方法，并说明其原理。
   - 为什么 ReLU 激活函数能够缓解梯度消失问题?
2. **卷积神经网络**：
   - 解释卷积操作中的局部连接和权重共享如何减少参数量。
   - 比较最大池化和平均池化的优缺点。
   - 为什么 CNN 适合处理图像数据?
3. **注意力机制**：
   - 解释自注意力和交叉注意力的区别。
   - 为什么 Transformer 需要位置编码?
   - 多头注意力的优势是什么?
4. **批量归一化**：
   - 解释批量归一化为什么能够加速训练。
   - 批量归一化在训练和测试阶段的区别是什么?

- 为什么批量归一化具有正则化效果?

5. **强化学习**:

- 解释强化学习与监督学习的区别。
- 什么是探索-利用权衡（Exploration-Exploitation Trade-off)?
- DQN 中的经验回放和目标网络的作用是什么?

6. **损失函数**:

- 比较均方误差（MSE）和平均绝对误差（MAE）的优缺点，各适用于什么场景?
- 为什么交叉熵损失函数与 softmax 激活函数配合使用效果最佳?
- 解释 Focal Loss 如何解决类别不平衡问题，其核心思想是什么?
- 在什么情况下应该使用 Huber Loss 而不是 MSE 或 MAE?
- 三元组损失函数在度量学习中的作用是什么？如何选择正负样本?

35.2 编程题

1. **实现多层感知机**:

- 使用 PyTorch 或 TensorFlow 实现一个三层 MLP（输入层、隐藏层、输出层）
- 在 MNIST 数据集上训练模型
- 实现前向传播和反向传播（可以调用框架的自动微分功能）
- 尝试不同的激活函数（Sigmoid、tanh、ReLU）并比较效果

2. **实现简单的 CNN**:

- 实现一个包含卷积层、池化层和全连接层的 CNN
- 在 CIFAR-10 数据集上训练模型
- 可视化卷积层的特征图，观察不同层学习到的特征

3. **实现 LSTM**:

- 使用 PyTorch 或 TensorFlow 实现 LSTM 单元
- 在文本分类任务（如情感分析）上训练模型
- 比较 LSTM 和简单 RNN 的性能差异

4. **实现注意力机制**:

- 实现自注意力机制
- 实现多头注意力机制
- 在序列到序列任务（如机器翻译）中应用注意力机制

5. **实现批量归一化**：

- 实现批量归一化层（包括训练和测试模式）
- 在深度网络中应用批量归一化，观察训练速度和最终性能的变化
- 比较使用和不使用批量归一化的训练曲线

6. **实现 Dropout**：

- 实现 Dropout 层（包括训练和测试模式）
- 在过拟合的模型上应用 Dropout，观察正则化效果
- 可视化 Dropout 对模型权重分布的影响

7. **实现不同的损失函数**：

- 实现 MSE、MAE 和 Huber Loss，比较它们在回归任务上的表现
- 实现交叉熵损失函数，在分类任务上验证其效果
- 实现 Focal Loss，在类别不平衡的数据集上比较其与标准交叉熵的差异
- 实现三元组损失函数，在图像检索任务中应用
- 尝试组合多个损失函数（如内容损失 + 风格损失），观察效果

35.3 综合项目

项目：图像分类系统

- 使用深度学习框架（PyTorch 或 TensorFlow）构建一个完整的图像分类系统
- 实现数据加载、预处理、模型定义、训练和评估的完整流程
- 尝试不同的网络架构（MLP、CNN、ResNet）
- 应用不同的优化技术（批量归一化、Dropout、学习率调度）
- 使用数据增强技术提高模型泛化能力
- 可视化训练过程（损失曲线、准确率曲线）
- 分析模型的错误案例，提出改进方案

项目：文本生成系统

- 使用 RNN/LSTM/GRU 或 Transformer 构建文本生成模型
- 在文本数据集（如小说、新闻）上训练模型
- 实现不同的采样策略（贪婪搜索、随机采样、束搜索）
- 评估生成文本的质量（困惑度、BLEU 分数等）
- 尝试不同的超参数设置，观察对生成质量的影响

项目：强化学习智能体

- 使用深度强化学习（DQN 或策略梯度方法）训练游戏智能体
- 在 OpenAI Gym 环境（如 CartPole、Atari 游戏）中训练
- 实现经验回放和目标网络（对于 DQN）
- 可视化训练过程和智能体的行为
- 分析不同超参数（学习率、折扣因子等）对性能的影响

35.4 研究性作业

1. **文献阅读**：阅读以下经典论文，总结其主要贡献：
   - LeCun et al. (1998). "Gradient-based learning applied to document recognition"
   - Krizhevsky et al. (2012). "ImageNet classification with deep convolutional neural networks"
   - He et al. (2016). "Deep residual learning for image recognition"
   - Vaswani et al. (2017). "Attention is all you need"
   - Devlin et al. (2018). "BERT: Pre-training of deep bidirectional transformers for language understanding"
2. **技术调研**：选择一个深度学习应用领域（如医疗影像、自动驾驶、金融风控），调研该领域的最新进展，包括：
   - 主要技术方法
   - 面临的挑战
   - 最新研究成果

- 实际应用案例

3. **实验设计**：设计一个实验来验证某个深度学习技术（如批量归一化、残差连接）的有效性：

   - 明确研究问题
   - 设计对比实验
   - 确定评估指标
   - 分析实验结果

通过完成以上作业和练习，读者可以深入理解深度学习的核心概念和技术，掌握实际应用的能力，为进一步的研究和应用打下坚实的基础。

Part V

第五部分：大语言模型基础

36 PyTorch 基础框架

PyTorch 是 Facebook 开发的深度学习框架，以其动态计算图和易用性成为大语言模型开发的首选工具。

36.1 PyTorch 核心概念

张量（Tensor）：PyTorch 的基本数据结构，类似于 NumPy 数组，但支持 GPU 加速。

```
import torch

# 创建张量
x = torch.tensor([1, 2, 3])  # 一维张量
y = torch.randn(3, 4)        # 3x4 随机张量
z = torch.zeros(2, 3)        # 2x3 零张量

# 张量运算
a = torch.tensor([[1, 2], [3, 4]])
b = torch.tensor([[5, 6], [7, 8]])
c = torch.matmul(a, b)       # 矩阵乘法
d = a + b                    # 逐元素加法

# GPU 支持
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
x_gpu = x.to(device)
```

Listing 58: PyTorch 张量基础操作

自动微分（Autograd）：PyTorch 的自动微分系统，用于计算梯度。

```
import torch

x = torch.tensor(2.0, requires_grad=True)
y = x ** 2 + 3 * x + 1
y.backward()  # 反向传播
```

```
print(x.grad)  # 输出: tensor(7.) = 2*x + 3 (在 x=2 处)
```

Listing 59: 自动微分示例

神经网络模块（nn.Module）：PyTorch 中定义神经网络的基础类。

```
import torch
import torch.nn as nn

class SimpleNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleNet, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

# 使用模型
model = SimpleNet(784, 128, 10)
x = torch.randn(32, 784)  # 批量大小 32
output = model(x)
```

Listing 60: 简单的神经网络定义

36.2 PyTorch 常用 API

nn.Linear：全连接层

```
linear = nn.Linear(in_features=768, out_features=3072)
# 输入: (batch_size, 768)
# 输出: (batch_size, 3072)
```

nn.Embedding：词嵌入层

```
embedding = nn.Embedding(num_embeddings=50000, embedding_dim=768)
```

```
# 输入: (batch_size, seq_len) - 整数索引
# 输出: (batch_size, seq_len, 768) - 嵌入向量
```

nn.LayerNorm：层归一化

```
layer_norm = nn.LayerNorm(normalized_shape=768)
# 输入: (batch_size, seq_len, 768)
# 输出: (batch_size, seq_len, 768)
```

nn.Dropout：Dropout 层

```
dropout = nn.Dropout(p=0.1)
# 训练时随机丢弃 10% 的神经元
# 测试时所有神经元都参与计算
```

nn.MultiheadAttention：多头注意力（PyTorch 内置实现）

```
attention = nn.MultiheadAttention(
    embed_dim=768,
    num_heads=12,
    dropout=0.1,
    batch_first=True
)
# query, key, value: (batch_size, seq_len, 768)
# output: (batch_size, seq_len, 768)
# attn_weights: (batch_size, seq_len, seq_len)
```

37 Transformer 架构基础

Transformer 是 Vaswani 等人在 2017 年提出的完全基于注意力机制的架构，成为现代大语言模型的基础。

37.1 Transformer 整体架构

Transformer 由编码器（Encoder）和解码器（Decoder）组成：

编码器：N 个相同的层堆叠，每层包含：

- 多头自注意力子层
- 前馈神经网络子层
- 残差连接和层归一化

解码器：N 个相同的层堆叠，每层包含：

- 掩码多头自注意力子层（防止看到未来信息）
- 编码器-解码器注意力子层
- 前馈神经网络子层
- 残差连接和层归一化

```
import torch
import torch.nn as nn
import math

class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward, dropout=0.1):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(
            d_model, nhead, dropout=dropout, batch_first=True
        )
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, src, src_mask=None):
        # 自注意力
        src2 = self.self_attn(src, src, src, attn_mask=src_mask)[0]
        src = src + self.dropout1(src2)
        src = self.norm1(src)

        # 前馈网络
        src2 = self.linear2(self.dropout(torch.relu(self.linear1(src))))
```

```
        src = src + self.dropout2(src2)
        src = self.norm2(src)

        return src
```

Listing 61: Transformer 编码器层实现

37.2 位置编码

由于 Transformer 没有循环结构，需要显式编码位置信息。

定义 37.1 (位置编码). *对于位置 pos 和维度 i，位置编码定义为：*

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{237}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{238}$$

其中，d_{model} 是模型维度。

```
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()

        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze
   (1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: (batch_size, seq_len, d_model)
        x = x + self.pe[:, :x.size(1), :]
```

```
        return x
```

Listing 62: 位置编码实现

38 注意力机制详解

注意力机制（Attention Mechanism）是 Transformer 的核心创新，它允许模型在处理序列时动态关注不同位置的信息，解决了传统 RNN 无法并行计算和长距离依赖建模困难的问题。

38.1 注意力机制的基本原理

38.1.1 为什么需要注意力机制?

传统序列模型的局限性：

- **RNN 的问题**：需要顺序计算，无法并行；长距离依赖建模困难；梯度消失/爆炸
- **CNN 的问题**：感受野有限，需要多层才能捕获长距离依赖
- **解决方案**：注意力机制可以直接建模任意距离的依赖关系，且支持并行计算

38.1.2 注意力机制的直观理解

类比：信息检索系统

想象你在图书馆查找资料：

- **查询（Query, Q）**：你的问题或需求（” 我想找什么信息”）
- **键（Key, K）**：每本书的索引标签（” 我提供什么信息”）
- **值（Value, V）**：书中的实际内容（” 实际的信息内容”）
- **注意力权重**：通过比较你的问题（Q）和书的索引（K），决定哪些书最相关，然后从这些书（V）中提取信息

在神经网络中的应用：

- 每个位置都可以” 查询” 其他位置的信息
- 通过计算查询和键的相似度，得到注意力权重
- 使用注意力权重对值进行加权求和，得到该位置的输出

38.1.3 缩放点积注意力的数学原理

定义 38.1 (缩放点积注意力（Scaled Dot-Product Attention）). 给定查询矩阵 $\mathbf{Q} \in \mathbb{R}^{n \times d_k}$、键矩阵 $\mathbf{K} \in \mathbb{R}^{m \times d_k}$ 和值矩阵 $\mathbf{V} \in \mathbb{R}^{m \times d_v}$，缩放点积注意力计算为：

$$Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = softmax\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{239}$$

其中，n 是查询序列长度，m 是键值序列长度，d_k 是键的维度，$\sqrt{d_k}$ 是缩放因子。

计算步骤详解：

步骤 1：计算注意力分数（Attention Scores）

对于每个查询向量 $\mathbf{q}_i$ 和键向量 $\mathbf{k}_j$，计算点积：

$$s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j = \mathbf{q}_i^T \mathbf{k}_j \tag{240}$$

矩阵形式：

$$\mathbf{S} = \mathbf{Q}\mathbf{K}^T \in \mathbb{R}^{n \times m} \tag{241}$$

其中，S_{ij} 表示第 i 个查询对第 j 个键的注意力分数。

步骤 2：缩放（Scaling）

为了防止点积值过大导致 softmax 梯度消失，除以 $\sqrt{d_k}$：

$$\mathbf{S}_{scaled} = \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}} \tag{242}$$

为什么需要缩放?

当 d_k 很大时，点积的值会很大，导致 softmax 函数进入饱和区域（梯度接近 0）。缩放因子 $\sqrt{d_k}$ 使得点积的方差保持为 1，假设 $\mathbf{Q}$ 和 $\mathbf{K}$ 的元素独立且均值为 0、方差为 1。

步骤 3：Softmax 归一化

将注意力分数转换为概率分布：

$$\alpha_{ij} = \frac{\exp(s_{ij}/\sqrt{d_k})}{\sum_{k=1}^{m} \exp(s_{ik}/\sqrt{d_k})} \tag{243}$$

矩阵形式：

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \in \mathbb{R}^{n \times m} \tag{244}$$

其中，α_{ij} 表示第 i 个查询对第 j 个键的注意力权重，满足 $\sum_{j=1}^{m} \alpha_{ij} = 1$。

步骤 4：加权求和

使用注意力权重对值进行加权求和：

$$\mathbf{z}_i = \sum_{j=1}^{m} \alpha_{ij} \mathbf{v}_j \tag{245}$$

矩阵形式：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{A}\mathbf{V} \in \mathbb{R}^{n \times d_v} \tag{246}$$

完整的数学表达式：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{247}$$

$$= \left[\frac{\exp(\mathbf{Q}\mathbf{K}^T/\sqrt{d_k})}{\mathbf{1}_m^T \exp(\mathbf{Q}\mathbf{K}^T/\sqrt{d_k})}\right]\mathbf{V} \tag{248}$$

其中，$\mathbf{1}_m$ 是全 1 向量，分母是逐行的归一化因子。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    缩放点积注意力实现

    参数:
        Q: 查询矩阵 (batch_size, seq_len_q, d_k)
        K: 键矩阵 (batch_size, seq_len_k, d_k)
        V: 值矩阵 (batch_size, seq_len_v, d_v)
        mask: 掩码矩阵 (batch_size, seq_len_q, seq_len_k) 或 None
              - mask[i,j] = 1 表示允许注意力
              - mask[i,j] = 0 表示禁止注意力（会被设为 -inf）

    返回:
        output: 注意力输出 (batch_size, seq_len_q, d_v)
        attn_weights: 注意力权重 (batch_size, seq_len_q, seq_len_k)
    """
    d_k = Q.size(-1)  # 获取键的维度

    # 步骤1: 计算注意力分数 QK^T
```

```
    # Q: (batch_size, seq_len_q, d_k)
    # K: (batch_size, seq_len_k, d_k)
    # scores: (batch_size, seq_len_q, seq_len_k)
    scores = torch.matmul(Q, K.transpose(-2, -1))

    # 步骤2: 缩放, 除以 sqrt(d_k)
    # 防止点积值过大, 避免 softmax 梯度消失
    scores = scores / math.sqrt(d_k)

    # 步骤3: 应用掩码 (如果有)
    # 将掩码为0的位置设为很大的负数, softmax后接近0
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # 步骤4: Softmax 归一化, 得到注意力权重
    # attn_weights[i,j] 表示第i个查询对第j个键的注意力权重
    attn_weights = F.softmax(scores, dim=-1)

    # 步骤5: 加权求和, 使用注意力权重对值进行加权
    # attn_weights: (batch_size, seq_len_q, seq_len_k)
    # V: (batch_size, seq_len_k, d_v)
    # output: (batch_size, seq_len_q, d_v)
    output = torch.matmul(attn_weights, V)

    return output, attn_weights

# 使用示例
batch_size = 2
seq_len_q = 5  # 查询序列长度
seq_len_k = 5  # 键值序列长度
d_k = 64       # 键的维度
d_v = 64       # 值的维度

Q = torch.randn(batch_size, seq_len_q, d_k)
K = torch.randn(batch_size, seq_len_k, d_k)
V = torch.randn(batch_size, seq_len_k, d_v)

# 计算注意力
output, attn_weights = scaled_dot_product_attention(Q, K, V)
```

```
print(f"输入 Q 形状: {Q.shape}")
print(f"输入 K 形状: {K.shape}")
print(f"输入 V 形状: {V.shape}")
print(f"输出形状: {output.shape}")
print(f"注意力权重形状: {attn_weights.shape}")
print(f"注意力权重每行和: {attn_weights.sum(dim=-1)}")  # 应该接近1
```

Listing 63: 缩放点积注意力详细实现（带注释）

时间复杂度分析:

对于序列长度 n 和维度 d:

- **计算 $\mathbf{QK}^T$**: $O(n^2 \cdot d)$
- **Softmax**: $O(n^2)$
- **加权求和 $\mathbf{AV}$**: $O(n^2 \cdot d)$
- **总复杂度**: $O(n^2 \cdot d)$

注意：注意力机制的复杂度是序列长度的平方，这是 Transformer 的主要计算瓶颈。

38.2 自注意力机制（Self-Attention）

38.2.1 自注意力的定义

自注意力（Self-Attention）是查询、键和值都来自同一输入序列的注意力机制。也就是说，序列中的每个位置都可以”查询”序列中所有位置（包括自己）的信息。

定义 38.2 (自注意力). *对于输入序列 $\mathbf{X} \in \mathbb{R}^{n \times d}$，自注意力计算为:*

$$\mathbf{Z} = Attention(\mathbf{XW}_Q, \mathbf{XW}_K, \mathbf{XW}_V) \tag{249}$$

其中:

- $\mathbf{W}_Q \in \mathbb{R}^{d \times d_k}$: *查询投影矩阵*
- $\mathbf{W}_K \in \mathbb{R}^{d \times d_k}$: *键投影矩阵*
- $\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$: *值投影矩阵*
- $\mathbf{Z} \in \mathbb{R}^{n \times d_v}$: *自注意力输出*

38.2.2 自注意力的计算过程

步骤详解：

步骤 1：生成 Q、K、V

对于输入序列 $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]^T$：

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q = [\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n]^T \tag{250}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K = [\mathbf{k}_1, \mathbf{k}_2, \ldots, \mathbf{k}_n]^T \tag{251}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n]^T \tag{252}$$

步骤 2：计算注意力权重

对于位置 i，计算它对所有位置的注意力权重：

$$\alpha_{ij} = \frac{\exp(\mathbf{q}_i^T \mathbf{k}_j / \sqrt{d_k})}{\sum_{k=1}^{n} \exp(\mathbf{q}_i^T \mathbf{k}_k / \sqrt{d_k})} \tag{253}$$

步骤 3：加权求和

位置 i 的输出是所有位置的值的加权和：

$$\mathbf{z}_i = \sum_{j=1}^{n} \alpha_{ij} \mathbf{v}_j \tag{254}$$

矩阵形式：

$$\mathbf{Z} = \text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_Q(\mathbf{X}\mathbf{W}_K)^T}{\sqrt{d_k}}\right)\mathbf{X}\mathbf{W}_V \tag{255}$$

38.2.3 自注意力的优势

1. 并行计算：

- 所有位置的 Q、K、V 可以并行计算
- 所有位置的注意力输出可以并行计算
- 时间复杂度：$O(n^2 \cdot d)$，但可以完全并行
- 相比 RNN 的 $O(n \cdot d^2)$ 顺序计算，虽然复杂度更高，但并行度更高

2. 长距离依赖：

- 可以直接建模任意距离的依赖关系

- 不需要像 RNN 那样通过多步传播
- 不需要像 CNN 那样通过多层堆叠

3. 可解释性:

- 注意力权重 α_{ij} 可以可视化
- 可以看到模型在关注哪些位置
- 有助于理解模型的决策过程

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class SelfAttention(nn.Module):
    """
    自注意力模块

    参数:
        d_model: 输入和输出的维度
        d_k: 键和查询的维度（默认等于 d_model）
        d_v: 值的维度（默认等于 d_model）
    """
    def __init__(self, d_model, d_k=None, d_v=None):
        super().__init__()
        if d_k is None:
            d_k = d_model
        if d_v is None:
            d_v = d_model

        # 定义三个线性投影层
        # 将输入 x 投影到 Q、K、V 空间
        self.W_q = nn.Linear(d_model, d_k)  # 查询投影
        self.W_k = nn.Linear(d_model, d_k)  # 键投影
        self.W_v = nn.Linear(d_model, d_v)  # 值投影
        self.d_k = d_k

    def forward(self, x, mask=None):
```

```
        """
        前向传播

        参数:
            x: 输入序列 (batch_size, seq_len, d_model)
            mask: 掩码矩阵 (batch_size, seq_len, seq_len) 或 None

        返回:
            output: 自注意力输出 (batch_size, seq_len, d_v)
            attn_weights: 注意力权重 (batch_size, seq_len, seq_len)
        """
        batch_size, seq_len, d_model = x.size()

        # 步骤1: 通过线性投影生成 Q、K、V
        # 所有位置共享相同的投影矩阵
        Q = self.W_q(x)  # (batch_size, seq_len, d_k)
        K = self.W_k(x)  # (batch_size, seq_len, d_k)
        V = self.W_v(x)  # (batch_size, seq_len, d_v)

        # 步骤2: 计算缩放点积注意力
        output, attn_weights = scaled_dot_product_attention(Q, K, V, mask
    )

        return output, attn_weights

# 使用示例
d_model = 512
seq_len = 10
batch_size = 2

# 创建自注意力模块
self_attn = SelfAttention(d_model=d_model)

# 创建输入（随机初始化）
x = torch.randn(batch_size, seq_len, d_model)

# 前向传播
output, attn_weights = self_attn(x)

print(f"输入形状: {x.shape}")
```

```
print(f"输出形状: {output.shape}")
print(f"注意力权重形状: {attn_weights.shape}")
print(f"注意力权重示例（第一个样本，第一个位置）:")
print(attn_weights[0, 0, :])  # 第一个样本，第一个位置对所有位置的注意力
print(f"注意力权重和: {attn_weights[0, 0, :].sum()}")  # 应该接近1
```

Listing 64: 自注意力详细实现（带注释）

38.3 多头注意力机制（Multi-Head Attention）

38.3.1 为什么需要多头注意力?

单头注意力的局限性:

- 只有一个注意力头，只能学习一种类型的依赖关系
- 表达能力有限，难以同时捕获多种语义关系
- 例如：语法关系、语义关系、位置关系等

多头注意力的优势:

- **多视角学习**：不同头可以关注不同的关系类型
- **表达能力增强**：通过多个子空间学习，增强模型表达能力
- **并行计算**：多个头可以并行计算，提高效率
- **专业化**：不同头可以学习不同的模式（如语法、语义、位置等）

38.3.2 多头注意力的数学原理

定义 38.3 (多头注意力). 多头注意力使用 h 个注意力头并行计算，每个头在不同的表示子空间中学习:

$$MultiHead(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Concat(head_1, \ldots, head_h)\mathbf{W}^O \tag{256}$$

$$head_i = Attention(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{257}$$

其中:

- h 是注意力头的数量（通常取 *8*、*12*、*16* 等）
- $\mathbf{W}_i^Q \in \mathbb{R}^{d_{model} \times d_k}$：第 i 个头的查询投影矩阵
- $\mathbf{W}_i^K \in \mathbb{R}^{d_{model} \times d_k}$：第 i 个头的键投影矩阵
- $\mathbf{W}_i^V \in \mathbb{R}^{d_{model} \times d_v}$：第 i 个头的值投影矩阵
- $\mathbf{W}^O \in \mathbb{R}^{h \cdot d_v \times d_{model}}$：输出投影矩阵
- $d_k = d_v = d_{model}/h$：每个头的维度

维度关系：

假设 $d_{model} = 512$，$h = 8$：

- 每个头的维度：$d_k = d_v = 512/8 = 64$
- 所有头拼接后：$h \cdot d_v = 8 \times 64 = 512$
- 输出投影后：512×512，保持维度不变

计算过程详解：

步骤 1：为每个头生成 Q、K、V

对于输入 $\mathbf{X} \in \mathbb{R}^{n \times d_{model}}$，首先通过共享的线性投影：

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \in \mathbb{R}^{n \times d_{model}} \tag{258}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K \in \mathbb{R}^{n \times d_{model}} \tag{259}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V \in \mathbb{R}^{n \times d_{model}} \tag{260}$$

然后将 $\mathbf{Q}$、$\mathbf{K}$、$\mathbf{V}$ 重塑并分割为 h 个头：

$$\mathbf{Q} \to [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_h], \quad \mathbf{Q}_i \in \mathbb{R}^{n \times d_k} \tag{261}$$

$$\mathbf{K} \to [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_h], \quad \mathbf{K}_i \in \mathbb{R}^{n \times d_k} \tag{262}$$

$$\mathbf{V} \to [\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_h], \quad \mathbf{V}_i \in \mathbb{R}^{n \times d_v} \tag{263}$$

步骤 2：每个头独立计算注意力

对于第 i 个头：

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \in \mathbb{R}^{n \times d_v} \tag{264}$$

步骤 3：拼接所有头

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_h) \in \mathbb{R}^{n \times (h \cdot d_v)} = \mathbb{R}^{n \times d_{model}} \tag{265}$$

步骤 4：输出投影

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}^O \in \mathbb{R}^{n \times d_{model}} \tag{266}$$

38.3.3 如何实现多头注意力?

实现策略：

有两种实现多头注意力的方式：

方式 1：为每个头单独定义投影矩阵（理论方式，不常用）

- 为每个头定义独立的 $\mathbf{W}_i^Q$、$\mathbf{W}_i^K$、$\mathbf{W}_i^V$
- 参数量：$h \times (d_{model} \times d_k + d_{model} \times d_k + d_{model} \times d_v)$
- 优点：每个头完全独立
- 缺点：参数量大，实现复杂

方式 2：共享投影矩阵，然后分割（实际使用的方式）

- 使用共享的 $\mathbf{W}_Q$、$\mathbf{W}_K$、$\mathbf{W}_V$ 投影到 d_{model} 维
- 然后将结果重塑并分割为 h 个头
- 参数量：$d_{model} \times d_{model} \times 3 + d_{model} \times d_{model}$（更少）
- 优点：参数量少，实现简单，效果相同

维度变换详解：

假设 $d_{model} = 512$，$h = 8$，$d_k = d_v = 64$，序列长度 $n = 10$：

步骤 1：共享线性投影

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \in \mathbb{R}^{10 \times 512} \tag{267}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K \in \mathbb{R}^{10 \times 512} \tag{268}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V \in \mathbb{R}^{10 \times 512} \tag{269}$$

步骤 2：重塑为多头形式

将 $\mathbf{Q}$ 重塑为 (n, h, d_k)：

$$\mathbf{Q} \in \mathbb{R}^{10\times512} \to \mathbf{Q}' \in \mathbb{R}^{10\times8\times64} \tag{270}$$

然后转置为 (h, n, d_k) 以便并行计算：

$$\mathbf{Q}' \in \mathbb{R}^{10\times8\times64} \to \mathbf{Q}'' \in \mathbb{R}^{8\times10\times64} \tag{271}$$

这样，$\mathbf{Q}''[i]$ 就是第 i 个头的查询矩阵 $\mathbf{Q}_i \in \mathbb{R}^{10\times64}$。

步骤 3：并行计算所有头的注意力

对于每个头 i：

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) \in \mathbb{R}^{10\times64} \tag{272}$$

所有头的结果：$\text{head}_1, \text{head}_2, \ldots, \text{head}_8$，每个都是 $\mathbb{R}^{10\times64}$。

步骤 4：拼接所有头

将 h 个头的结果拼接：

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_8) \in \mathbb{R}^{10\times512} \tag{273}$$

步骤 5：输出投影

$$\text{output} = \text{Concat}(\text{head}_1, \ldots, \text{head}_8)\mathbf{W}^O \in \mathbb{R}^{10\times512} \tag{274}$$

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class MultiHeadAttention(nn.Module):
    """
    多头注意力模块

    参数:
        d_model: 模型维度（输入和输出的维度）
        num_heads: 注意力头的数量
    """
    def __init__(self, d_model, num_heads):
        super().__init__()
        # 确保 d_model 可以被 num_heads 整除
```

```
        assert d_model % num_heads == 0, "d_model 必须能被 num_heads 整除
"

        self.d_model = d_model      # 模型维度，如 512
        self.num_heads = num_heads  # 注意力头数量，如 8
        self.d_k = d_model // num_heads  # 每个头的维度，如 512 // 8 = 64

        # 步骤1: 定义共享的线性投影层
        # 这些投影层将输入投影到 d_model 维，然后会被分割为多个头
        self.W_q = nn.Linear(d_model, d_model)  # 查询投影
        self.W_k = nn.Linear(d_model, d_model)  # 键投影
        self.W_v = nn.Linear(d_model, d_model)  # 值投影

        # 输出投影层，将拼接后的多头结果投影回 d_model 维
        self.W_o = nn.Linear(d_model, d_model)

    def forward(self, Q, K, V, mask=None):
        """
        前向传播

        参数:
            Q: 查询矩阵 (batch_size, seq_len_q, d_model)
            K: 键矩阵 (batch_size, seq_len_k, d_model)
            V: 值矩阵 (batch_size, seq_len_v, d_model)
            mask: 掩码矩阵 (batch_size, seq_len_q, seq_len_k) 或 None

        返回:
            output: 多头注意力输出 (batch_size, seq_len_q, d_model)
            attn_weights: 注意力权重 (batch_size, num_heads, seq_len_q,
seq_len_k)
        """
        batch_size = Q.size(0)
        seq_len_q = Q.size(1)
        seq_len_k = K.size(1)

        # ========== 步骤1: 共享线性投影 ==========
        # 将 Q、K、V 投影到 d_model 维
        # Q_proj: (batch_size, seq_len_q, d_model)
        # K_proj: (batch_size, seq_len_k, d_model)
        # V_proj: (batch_size, seq_len_v, d_model)
```

```
        Q_proj = self.W_q(Q)
        K_proj = self.W_k(K)
        V_proj = self.W_v(V)

        # ========== 步骤2: 重塑为多头形式 ==========
        # 将 d_model 维分割为 num_heads 个头, 每个头 d_k 维
        #
        # 例如: d_model=512, num_heads=8, d_k=64
        # Q_proj: (batch_size, seq_len_q, 512)
        #   -> view: (batch_size, seq_len_q, 8, 64)
        #   -> transpose(1,2): (batch_size, 8, seq_len_q, 64)
        #
        # 这样 Q_multi[i] 就是第 i 个头的查询矩阵
        Q_multi = Q_proj.view(batch_size, seq_len_q, self.num_heads, self
    .d_k).transpose(1, 2)
        K_multi = K_proj.view(batch_size, seq_len_k, self.num_heads, self
    .d_k).transpose(1, 2)
        V_multi = V_proj.view(batch_size, seq_len_k, self.num_heads, self
    .d_k).transpose(1, 2)
        # Q_multi: (batch_size, num_heads, seq_len_q, d_k)
        # K_multi: (batch_size, num_heads, seq_len_k, d_k)
        # V_multi: (batch_size, num_heads, seq_len_k, d_k)

        # ========== 步骤3: 并行计算所有头的注意力 ==========
        # 使用批量矩阵乘法, 同时计算所有头的注意力
        #
        # 计算注意力分数: QK^T
        # Q_multi: (batch_size, num_heads, seq_len_q, d_k)
        # K_multi: (batch_size, num_heads, seq_len_k, d_k)
        # scores: (batch_size, num_heads, seq_len_q, seq_len_k)
        scores = torch.matmul(Q_multi, K_multi.transpose(-2, -1)) / math.
    sqrt(self.d_k)

        # 应用掩码 (如果有)
        if mask is not None:
            # mask: (batch_size, seq_len_q, seq_len_k)
            # 需要扩展维度以匹配 scores 的形状
            mask = mask.unsqueeze(1)  # (batch_size, 1, seq_len_q,
    seq_len_k)
            scores = scores.masked_fill(mask == 0, -1e9)
```

```
        # Softmax 归一化，得到注意力权重
        attn_weights = F.softmax(scores, dim=-1)
        # attn_weights: (batch_size, num_heads, seq_len_q, seq_len_k)

        # 加权求和：注意力权重 × 值
        # attn_weights: (batch_size, num_heads, seq_len_q, seq_len_k)
        # V_multi: (batch_size, num_heads, seq_len_k, d_k)
        # attn_output: (batch_size, num_heads, seq_len_q, d_k)
        attn_output = torch.matmul(attn_weights, V_multi)

        # ========== 步骤4：拼接所有头 ==========
        # 将多个头的结果拼接在一起
        #
        # attn_output: (batch_size, num_heads, seq_len_q, d_k)
        #   -> transpose(1,2): (batch_size, seq_len_q, num_heads, d_k)
        #   -> contiguous().view: (batch_size, seq_len_q, num_heads * d_k
    )
        #   = (batch_size, seq_len_q, d_model)
        attn_output = attn_output.transpose(1, 2).contiguous().view(
            batch_size, seq_len_q, self.d_model
        )
        # attn_output: (batch_size, seq_len_q, d_model)

        # ========== 步骤5：输出投影 ==========
        # 将拼接后的结果投影回 d_model 维
        output = self.W_o(attn_output)
        # output: (batch_size, seq_len_q, d_model)

        return output, attn_weights

# ========== 使用示例 ==========
d_model = 512
num_heads = 8
seq_len = 10
batch_size = 2

# 创建多头注意力模块
multi_head_attn = MultiHeadAttention(d_model=d_model, num_heads=num_heads
    )
```

```
# 创建输入（Q、K、V 可以相同，也可以不同）
Q = torch.randn(batch_size, seq_len, d_model)
K = torch.randn(batch_size, seq_len, d_model)
V = torch.randn(batch_size, seq_len, d_model)

# 前向传播
output, attn_weights = multi_head_attn(Q, K, V)

print(f"输入 Q 形状: {Q.shape}")
print(f"输入 K 形状: {K.shape}")
print(f"输入 V 形状: {V.shape}")
print(f"输出形状: {output.shape}")
print(f"注意力权重形状: {attn_weights.shape}")  # (batch_size, num_heads,
    seq_len, seq_len)
print(f"每个头的维度 d_k: {multi_head_attn.d_k}")
print(f"注意力权重示例（第一个样本，第一个头，第一个位置）:")
print(attn_weights[0, 0, 0, :])  # 第一个样本，第一个头，第一个位置对所有
   位置的注意力
print(f"注意力权重和: {attn_weights[0, 0, 0, :].sum()}")  # 应该接近1
```

Listing 65: 多头注意力详细实现（带完整注释）

38.3.4 多头注意力的维度变换可视化

维度变换流程（以 $d_{model} = 512$，$h = 8$，$d_k = 64$ 为例）：

输入 $\mathbf{X}$: $(batch, seq, 512)$

↓ 线性投影

$\mathbf{Q}$: $(batch, seq, 512)$

↓ 重塑

$\mathbf{Q}'$: $(batch, seq, 8, 64)$

↓ 转置

$\mathbf{Q}''$: $(batch, 8, seq, 64)$

↓ 并行计算

$\text{head}_1, \ldots, \text{head}_8$: 每个 $(batch, seq, 64)$

↓ 拼接

Concat: $(batch, seq, 512)$

↓ 输出投影

输出: $(batch, seq, 512)$

关键点:

- **共享投影**: 所有头共享 $\mathbf{W}_Q$、$\mathbf{W}_K$、$\mathbf{W}_V$，参数量更少
- **分割策略**: 通过 reshape 和 transpose 将 d_{model} 维分割为 h 个 d_k 维的头
- **并行计算**: 所有头的注意力可以并行计算，提高效率
- **拼接和投影**: 拼接所有头的结果，然后通过 $\mathbf{W}^O$ 投影回原始维度

38.3.5 多头注意力的优势和应用

优势:

1. 多视角学习:

- 不同头可以关注不同的关系类型
- 例如：头 1 关注语法关系，头 2 关注语义关系，头 3 关注位置关系等
- 通过可视化注意力权重可以观察到这种现象

2. 表达能力增强:

- 通过多个子空间学习，增强模型表达能力
- 相当于从多个角度理解输入序列
- 比单头注意力有更强的拟合能力

3. 并行计算:

- 多个头可以完全并行计算
- 时间复杂度：$O(n^2 \cdot d)$（与单头相同）
- 但可以充分利用 GPU 的并行计算能力

4. 参数量优化:

- 使用共享投影矩阵，参数量为 $O(d_{model}^2)$
- 如果为每个头单独定义投影矩阵，参数量为 $O(h \cdot d_{model}^2)$

- 共享投影既减少了参数量，又保持了表达能力

实际应用中的观察：

研究表明，在训练好的 Transformer 模型中：

- 不同头确实学习到了不同的模式
- 有些头关注局部依赖（相邻位置）
- 有些头关注长距离依赖（远距离位置）
- 有些头关注特定类型的语法或语义关系

头数量的选择：

- **常见配置**：$h = 8$（BERT-base）、$h = 12$（BERT-large）、$h = 16$（GPT-3）
- **原则**：d_{model} 必须能被 h 整除
- **权衡**：头数越多，表达能力越强，但计算量也越大
- **经验**：通常 $h = 8$ 或 $h = 16$ 效果较好

39 完整的 Transformer 实现

本节提供一个简化的 Transformer 编码器实现，展示如何将各个组件组合起来。

```
import torch
import torch.nn as nn
import math

class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, d_model=512, nhead=8,
                 num_layers=6, dim_feedforward=2048,
                 max_seq_length=5000, dropout=0.1):
        super().__init__()

        self.d_model = d_model
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoding = PositionalEncoding(d_model, max_seq_length)
```

```
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
            batch_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer, num_layers=num_layers
        )

        self.dropout = nn.Dropout(dropout)

    def forward(self, src, src_mask=None, src_key_padding_mask=None):
        # src: (batch_size, seq_len)
        # 词嵌入
        src = self.embedding(src) * math.sqrt(self.d_model)
        # 位置编码
        src = self.pos_encoding(src)
        src = self.dropout(src)

        # Transformer 编码
        output = self.transformer_encoder(
            src,
            mask=src_mask,
            src_key_padding_mask=src_key_padding_mask
        )

        return output

# 使用示例
model = TransformerEncoder(
    vocab_size=10000,
    d_model=512,
    nhead=8,
    num_layers=6
)

# 输入: (batch_size=32, seq_len=128)
input_ids = torch.randint(0, 10000, (32, 128))
```

```
output = model(input_ids)
# 输出: (32, 128, 512)
```

Listing 66: 完整的 Transformer 编码器

40 训练与推理基础

40.1 训练流程

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

# 定义模型
model = TransformerEncoder(vocab_size=10000, d_model=512)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)

# 训练循环
def train_epoch(model, dataloader, criterion, optimizer, device):
    model.train()
    total_loss = 0

    for batch in dataloader:
        input_ids, labels = batch
        input_ids = input_ids.to(device)
        labels = labels.to(device)

        # 前向传播
        outputs = model(input_ids)

        # 计算损失（假设是语言建模任务）
        # outputs: (batch_size, seq_len, vocab_size)
        # labels: (batch_size, seq_len)
        logits = outputs.view(-1, outputs.size(-1))
        labels_flat = labels.view(-1)
        loss = criterion(logits, labels_flat)
```

```
        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)
```

Listing 67: Transformer 训练示例

40.2 推理流程

```
def generate_text(model, tokenizer, prompt, max_length=100, temperature
    =1.0):
    model.eval()
    tokens = tokenizer.encode(prompt)

    with torch.no_grad():
        for _ in range(max_length):
            # 准备输入
            input_ids = torch.tensor([tokens]).to(device)

            # 前向传播
            outputs = model(input_ids)

            # 获取最后一个位置的 logits
            logits = outputs[0, -1, :] / temperature

            # 采样下一个 token
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1).item()

            tokens.append(next_token)

            # 如果生成了结束符, 停止
            if next_token == tokenizer.eos_token_id:
                break
```

```
    return tokenizer.decode(tokens)
```

Listing 68: Transformer 推理示例

41 总结

第一部分介绍了大语言模型的基础概念和框架：

- **PyTorch 框架**：深度学习开发的基础工具
- **Transformer 架构**：现代 LLM 的核心架构
- **注意力机制**：包括缩放点积注意力、自注意力和多头注意力
- **位置编码**：为序列添加位置信息
- **训练与推理**：基本的模型训练和文本生成流程

这些基础内容是理解和使用大语言模型的前提，与深度学习和机器学习紧密相关。在第二部分中，我们将介绍更高级的技术和应用。

42 预训练与微调

预训练是大语言模型的核心，通过在大量无标注文本上学习语言表示，然后通过微调适应特定任务。

42.1 预训练目标

自回归语言建模（Autoregressive Language Modeling）：

$$\mathcal{L}_{LM} = -\sum_{t=1}^{T} \log P(x_t | x_{<t}) \tag{275}$$

模型从左到右预测下一个词，GPT 系列使用此目标。

掩码语言建模（Masked Language Modeling）：

$$\mathcal{L}_{MLM} = -\sum_{i \in M} \log P(x_i | \mathbf{x}_{\setminus M}) \tag{276}$$

随机掩码部分词，预测被掩码的词，BERT 使用此目标。

下一句预测（Next Sentence Prediction）：判断两个句子是否连续，BERT 使用此辅助任务。

```
import torch
import torch.nn as nn
import random

def create_masked_lm_predictions(tokens, tokenizer, mlm_probability=0.15):
    """创建掩码语言建模的标签"""
    labels = tokens.clone()
    probability_matrix = torch.full(labels.shape, mlm_probability)

    # 不掩码特殊 token
    special_tokens_mask = torch.tensor([
        tokenizer.get_special_tokens_mask(val, already_has_special_tokens=True)
        for val in labels.tolist()
    ], dtype=torch.bool)
    probability_matrix.masked_fill_(special_tokens_mask, value=0.0)

    # 80% 的时间用 [MASK] 替换
    # 10% 的时间用随机词替换
    # 10% 的时间保持原词
    masked_indices = torch.bernoulli(probability_matrix).bool()
    labels[~masked_indices] = -100  # 只计算被掩码位置的损失

    # 80% 替换为 [MASK]
    indices_replaced = torch.bernoulli(
        torch.full(labels.shape, 0.8)
    ).bool() & masked_indices
    tokens[indices_replaced] = tokenizer.convert_tokens_to_ids('[MASK]')

    # 10% 替换为随机词
    indices_random = torch.bernoulli(
        torch.full(labels.shape, 0.5)
    ).bool() & masked_indices & ~indices_replaced
    random_words = torch.randint(len(tokenizer), labels.shape, dtype=torch.long)
```

```
    tokens[indices_random] = random_words[indices_random]

    return tokens, labels
```

Listing 69: 掩码语言建模实现

42.2 Few-shot Learning

Few-shot Learning 是指模型只需少量示例就能适应新任务。

In-context Learning：通过在提示中包含示例，让模型学习任务模式，无需更新参数。

例 42.1 (Few-shot 示例)**. 任务**：情感分析

提示：

```
评论：这部电影太棒了！
情感：正面

评论：服务很差，不推荐。
情感：负面

评论：还可以，一般般。
情感：
```

模型应该输出”中性”。

42.3 微调策略

全参数微调：更新所有模型参数。

参数高效微调（Parameter-Efficient Fine-tuning）：

- **LoRA（Low-Rank Adaptation）**：只训练低秩矩阵
- **Adapter**：在 Transformer 层中插入小的适配器模块
- **Prefix Tuning**：学习可训练的前缀

```
import torch
import torch.nn as nn
```

```python
class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=8, alpha=16):
        super().__init__()
        self.rank = rank
        self.alpha = alpha

        # LoRA 矩阵 A 和 B
        self.lora_A = nn.Parameter(torch.randn(in_features, rank))
        self.lora_B = nn.Parameter(torch.zeros(rank, out_features))

        # 原始权重（冻结）
        self.weight = nn.Parameter(torch.randn(in_features, out_features)
    )
        self.weight.requires_grad = False

    def forward(self, x):
        # 原始输出 + LoRA 调整
        return F.linear(x, self.weight + (self.alpha / self.rank) *
                        torch.matmul(self.lora_B, self.lora_A))

# 应用 LoRA 到线性层
class LoRALinear(nn.Module):
    def __init__(self, linear_layer, rank=8, alpha=16):
        super().__init__()
        self.original = linear_layer
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank,
            alpha
        )

    def forward(self, x):
        return self.original(x) + self.lora(x)
```

Listing 70: LoRA 实现

43 提示工程

提示工程（Prompt Engineering）是通过设计有效的提示来引导模型生成期望的输出。

43.1 提示设计原则

1. **明确任务**：清楚说明要完成的任务
2. **提供示例**：Few-shot 示例帮助模型理解任务
3. **指定格式**：明确输出格式要求
4. **分解任务**：将复杂任务分解为子任务

43.2 Chain-of-Thought（CoT）

Chain-of-Thought 通过引导模型逐步推理，提高复杂问题的解决能力。

例 43.1 (Chain-of-Thought 示例)**. 问题**：一个商店有 *23* 个苹果。如果他们卖了 *20* 个，又买了 *6* 个，现在有多少个?

标准提示:

```
Q: 一个商店有 23 个苹果。如果他们卖了 20 个，又买了 6 个，现在有多少个?
A:
```

***Chain-of-Thought* 提示**:

```
Q: 一个商店有 23 个苹果。如果他们卖了 20 个，又买了 6 个，现在有多少个?
A: 让我们一步步思考。
商店开始有 23 个苹果。
卖了 20 个后，剩下 23 - 20 = 3 个。
又买了 6 个，现在有 3 + 6 = 9 个。
所以答案是 9。
```

```python
def generate_cot_prompt(question, examples=None):
    """生成 Chain-of-Thought 提示"""
    prompt = "让我们一步步思考并解决这个问题。\n\n"

    if examples:
        for ex_q, ex_a in examples:
```

```
            prompt += f"问题: {ex_q}\n"
            prompt += f"解答: {ex_a}\n\n"

    prompt += f"问题: {question}\n"
    prompt += "解答: "
    return prompt

# 使用示例
question = "如果一本书有 300 页，每天读 25 页，需要多少天读完？"
cot_prompt = generate_cot_prompt(question)
# 模型会生成逐步推理过程
```

Listing 71: Chain-of-Thought 实现示例

43.3 Few-shot Prompting

Few-shot Prompting 通过在提示中包含多个示例，让模型学习任务模式。

```
def create_few_shot_prompt(task_description, examples, query):
    """创建 Few-shot 提示"""
    prompt = f"{task_description}\n\n"

    for i, (input_ex, output_ex) in enumerate(examples, 1):
        prompt += f"示例 {i}:\n"
        prompt += f"输入: {input_ex}\n"
        prompt += f"输出: {output_ex}\n\n"

    prompt += f"输入: {query}\n"
    prompt += "输出: "
    return prompt

# 使用示例: 情感分析
task = "对以下评论进行情感分析，输出'正面'、'负面'或'中性'。"
examples = [
    ("这部电影很棒！", "正面"),
    ("服务很差。", "负面"),
    ("还可以吧。", "中性")
]
query = "这个产品一般般。"
```

```
prompt = create_few_shot_prompt(task, examples, query)
```

Listing 72: Few-shot Prompting 实现

44 检索增强生成（RAG）

RAG（Retrieval-Augmented Generation）通过检索外部知识库来增强生成，提高准确性和可追溯性。

44.1 RAG 架构

RAG 包含两个主要组件：

1. **检索器（Retriever）**：从知识库中检索相关文档
2. **生成器（Generator）**：基于检索到的文档生成答案

```
from transformers import AutoTokenizer, AutoModel
import faiss
import numpy as np

class RAGSystem:
    def __init__(self, generator_model, retriever_model, knowledge_base):
        self.generator_tokenizer = AutoTokenizer.from_pretrained(
   generator_model)
        self.generator_model = AutoModel.from_pretrained(generator_model)
        self.retriever_tokenizer = AutoTokenizer.from_pretrained(
   retriever_model)
        self.retriever_model = AutoModel.from_pretrained(retriever_model)
        self.knowledge_base = knowledge_base

        # 构建向量索引
        self.index = self._build_index()

    def _build_index(self):
        """构建知识库的向量索引"""
        embeddings = []
        for doc in self.knowledge_base:
```

```
            inputs = self.retriever_tokenizer(
                doc, return_tensors='pt', padding=True, truncation=True
            )
            with torch.no_grad():
                embedding = self.retriever_model(**inputs).
last_hidden_state.mean(dim=1)
            embeddings.append(embedding.numpy())

        embeddings = np.vstack(embeddings)
        dimension = embeddings.shape[1]
        index = faiss.IndexFlatL2(dimension)
        index.add(embeddings.astype('float32'))
        return index

    def retrieve(self, query, top_k=5):
        """检索相关文档"""
        inputs = self.retriever_tokenizer(
            query, return_tensors='pt', padding=True, truncation=True
        )
        with torch.no_grad():
            query_embedding = self.retriever_model(**inputs).
last_hidden_state.mean(dim=1)

        query_embedding = query_embedding.numpy().astype('float32')
        distances, indices = self.index.search(query_embedding, top_k)

        retrieved_docs = [self.knowledge_base[i] for i in indices[0]]
        return retrieved_docs

    def generate(self, query, retrieved_docs):
        """基于检索到的文档生成答案"""
        context = "\n\n".join(retrieved_docs)
        prompt = f"基于以下上下文回答问题：\n\n{context}\n\n问题：{query
}\n答案："

        inputs = self.generator_tokenizer(
            prompt, return_tensors='pt', padding=True, truncation=True,
max_length=512
        )
```

```
        with torch.no_grad():
            outputs = self.generator_model.generate(
                **inputs,
                max_length=512,
                num_beams=4,
                early_stopping=True
            )

        answer = self.generator_tokenizer.decode(outputs[0],
    skip_special_tokens=True)
        return answer
```

Listing 73: RAG 实现示例

44.2 RAG 的优势

- **准确性**：基于真实文档生成，减少幻觉
- **可追溯性**：可以追溯到源文档
- **知识更新**：通过更新知识库，无需重新训练模型
- **领域适应**：可以针对特定领域构建知识库

45 代码生成与程序理解

大语言模型在代码生成和理解方面展现出强大能力，推动了 AI 辅助编程的发展。

45.1 代码生成

```
def generate_code(prompt, model, tokenizer, max_length=512):
    """根据自然语言描述生成代码"""
    full_prompt = f"""根据以下描述生成 Python 代码：

描述：{prompt}

代码："""
```

```
    inputs = tokenizer(full_prompt, return_tensors='pt')
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_length=max_length,
            temperature=0.7,
            do_sample=True,
            pad_token_id=tokenizer.eos_token_id
        )

    generated_code = tokenizer.decode(outputs[0], skip_special_tokens=
   True)
    # 提取代码部分
    code = generated_code.split("代码: ")[-1].strip()
    return code

# 使用示例
description = "写一个函数计算斐波那契数列的第 n 项"
code = generate_code(description, model, tokenizer)
print(code)
```

Listing 74: 代码生成示例

45.2 程序理解

程序理解包括代码摘要、代码搜索、bug 检测等任务。

```
def summarize_code(code, model, tokenizer):
    """生成代码摘要"""
    prompt = f"""为以下代码生成简洁的摘要:

代码:
{code}

摘要: """

    inputs = tokenizer(prompt, return_tensors='pt', max_length=1024,
   truncation=True)
    with torch.no_grad():
```

```
        outputs = model.generate(
            **inputs,
            max_length=200,
            num_beams=4,
            early_stopping=True
        )

    summary = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return summary.split("摘要：")[-1].strip()
```

Listing 75: 代码摘要生成

46 常用大模型介绍

46.1 GPT 系列

GPT-1（2018）：1.17 亿参数，首次展示预训练 + 微调的有效性。

GPT-2（2019）：15 亿参数，展示了大模型的语言生成能力。

GPT-3（2020）：1750 亿参数，展示了强大的 few-shot 学习能力。

GPT-4（2023）：多模态模型，支持图像和文本输入。

46.2 BERT 系列

BERT（2018）：双向编码器，在多个 NLP 任务上取得突破。

RoBERTa：优化了 BERT 的训练策略。

ALBERT：通过参数共享减少参数量。

46.3 其他重要模型

T5：将所有 NLP 任务统一为文本到文本的生成任务。

PaLM：Google 的 Pathways 语言模型，5400 亿参数。

LLaMA：Meta 的开源大模型，参数效率高。

47 模型评估与测试

47.1 评估指标

困惑度（Perplexity）：

$$\text{PPL} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(x_i|x_{<i})\right) \tag{277}$$

BLEU：用于机器翻译和文本生成。

ROUGE：用于文本摘要。

准确率：用于分类任务。

```python
def evaluate_model(model, tokenizer, test_dataset):
    """评估模型性能"""
    model.eval()
    total_loss = 0
    total_tokens = 0

    with torch.no_grad():
        for batch in test_dataset:
            input_ids, labels = batch
            outputs = model(input_ids, labels=labels)
            loss = outputs.loss
            total_loss += loss.item() * input_ids.size(0)
            total_tokens += input_ids.size(0)

    avg_loss = total_loss / total_tokens
    perplexity = torch.exp(torch.tensor(avg_loss))

    return {
        'loss': avg_loss,
        'perplexity': perplexity.item()
    }
```

Listing 76: 模型评估示例

48 训练到部署全流程

48.1 数据准备

```python
from datasets import load_dataset
from transformers import AutoTokenizer

def prepare_dataset(dataset_name, tokenizer_name, max_length=512):
    """准备训练数据集"""
    # 加载数据集
    dataset = load_dataset(dataset_name)

    # 加载分词器
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)

    def tokenize_function(examples):
        return tokenizer(
            examples['text'],
            truncation=True,
            padding='max_length',
            max_length=max_length
        )

    # 分词
    tokenized_dataset = dataset.map(tokenize_function, batched=True)

    return tokenized_dataset
```

Listing 77: 数据预处理

48.2 模型训练

```python
from transformers import Trainer, TrainingArguments
from transformers import AutoModelForCausalLM

def train_model(model_name, dataset, output_dir):
    """训练模型"""
    # 加载模型
```

```
    model = AutoModelForCausalLM.from_pretrained(model_name)
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # 训练参数
    training_args = TrainingArguments(
        output_dir=output_dir,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        gradient_accumulation_steps=4,
        warmup_steps=500,
        logging_steps=100,
        save_steps=1000,
        evaluation_strategy="steps",
        eval_steps=500,
        save_total_limit=3,
        fp16=True,  # 混合精度训练
        dataloader_pin_memory=False,
    )

    # 训练器
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=dataset['train'],
        eval_dataset=dataset['validation'],
    )

    # 开始训练
    trainer.train()

    # 保存模型
    trainer.save_model()
    tokenizer.save_pretrained(output_dir)
```

Listing 78: 完整训练流程

48.3 模型部署

```
from transformers import pipeline
```

```python
import torch

class ModelServer:
    def __init__(self, model_path):
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        self.generator = pipeline(
            'text-generation',
            model=model_path,
            device=0 if self.device == 'cuda' else -1
        )

    def generate(self, prompt, max_length=100, temperature=0.7):
        """生成文本"""
        results = self.generator(
            prompt,
            max_length=max_length,
            temperature=temperature,
            num_return_sequences=1,
            do_sample=True
        )
        return results[0]['generated_text']

    def batch_generate(self, prompts, **kwargs):
        """批量生成"""
        return [self.generate(p, **kwargs) for p in prompts]

# 使用示例
server = ModelServer('path/to/model')
response = server.generate("今天天气怎么样？")
print(response)
```

Listing 79: 模型部署示例

49 应用场景

49.1 文本生成

- **内容创作**：文章、故事、诗歌生成

- **对话系统**：智能客服、聊天机器人
- **文本摘要**：自动生成文档摘要

49.2 知识问答

- **开放域问答**：回答各种知识性问题
- **阅读理解**：基于文档回答问题
- **多轮对话**：理解上下文进行连续对话

49.3 代码助手

- **代码生成**：根据描述生成代码
- **代码补全**：IDE 中的智能补全
- **代码审查**：检测代码问题和改进建议

50 未来发展方向

50.1 多模态交互

- **视觉-语言模型**：理解图像和文本的联合表示
- **音频-语言模型**：语音识别和生成
- **视频理解**：视频内容理解和生成

50.2 智能助手

- **个人助理**：日程管理、信息检索、任务执行
- **专业助手**：医疗、法律、教育等领域的专业助手
- **创作助手**：协助写作、设计、编程等创作任务

50.3 Agent 应用

- **自主 Agent**：能够自主规划和执行任务的智能体
- **工具使用**：调用外部 API 和工具完成任务
- **多 Agent 协作**：多个 Agent 协作解决复杂问题

51 作业与练习

51.1 概念题

1. **Transformer 架构**：
    - 解释自注意力和交叉注意力的区别。
    - 为什么 Transformer 需要位置编码?
    - 多头注意力的优势是什么?
2. **预训练与微调**：
    - 比较自回归语言建模和掩码语言建模的优缺点。
    - 解释 Few-shot Learning 和 In-context Learning 的区别。
    - LoRA 如何实现参数高效微调?
3. **提示工程**：
    - Chain-of-Thought 如何提高模型的推理能力?
    - Few-shot Prompting 的工作原理是什么?
    - 如何设计有效的提示?
4. **RAG**：
    - RAG 相比直接生成有什么优势?
    - 如何构建高效的检索系统?
    - RAG 如何解决大模型的幻觉问题?

51.2 编程题

1. **实现 Transformer 编码器**：
   - 使用 PyTorch 实现完整的 Transformer 编码器
   - 包含位置编码、多头注意力、前馈网络
   - 在文本分类任务上测试
2. **实现注意力机制**：
   - 实现缩放点积注意力
   - 实现多头注意力
   - 可视化注意力权重
3. **实现 RAG 系统**：
   - 构建文档向量索引
   - 实现检索功能
   - 基于检索结果生成答案
4. **提示工程实践**：
   - 设计 Few-shot Prompting 提示
   - 实现 Chain-of-Thought 提示
   - 比较不同提示策略的效果
5. **模型微调**：
   - 使用 Hugging Face Transformers 微调预训练模型
   - 实现 LoRA 微调
   - 比较全参数微调和 LoRA 的效果

51.3 综合项目

项目：构建智能问答系统

- 使用预训练的大语言模型
- 实现 RAG 增强生成
- 构建知识库和检索系统

- 设计用户界面
- 评估系统性能

项目：代码生成助手

- 在代码数据集上微调模型
- 实现代码生成功能
- 添加代码补全功能
- 实现代码审查功能

通过完成以上作业和练习，读者可以深入理解大语言模型的核心技术，掌握从基础架构到高级应用的完整知识体系，为进一步的研究和应用打下坚实的基础。

Part VI

第六部分：大语言模型先进技术

52 参数高效微调技术

传统的全参数微调需要更新模型的所有参数，对于大模型来说成本极高。参数高效微调（Parameter-Efficient Fine-Tuning, PEFT）技术通过只更新少量参数就能达到接近全参数微调的效果，大幅降低了微调成本。

52.1 LoRA (Low-Rank Adaptation)

概念解释：LoRA 是 Microsoft 在 2021 年提出的参数高效微调方法。其核心思想是：对于预训练模型的权重矩阵 $\mathbf{W}$，不直接更新它，而是学习一个低秩分解的增量 $\Delta\mathbf{W}$，使得 $\mathbf{W}+\Delta\mathbf{W}$ 能够适应新任务。

数学公式：

对于原始权重矩阵 $\mathbf{W} \in \mathbb{R}^{d\times k}$，LoRA 将其更新分解为：

$$\mathbf{W}' = \mathbf{W} + \Delta\mathbf{W} = \mathbf{W} + \mathbf{BA} \tag{278}$$

其中：

- $\mathbf{A} \in \mathbb{R}^{r\times k}$：低秩矩阵，随机初始化
- $\mathbf{B} \in \mathbb{R}^{d\times r}$：低秩矩阵，初始化为零
- $r \ll \min(d, k)$：秩（rank），通常 $r \in \{4, 8, 16, 32, 64\}$

在前向传播时：

$$\mathbf{h} = \mathbf{W}'\mathbf{x} = (\mathbf{W} + \mathbf{BA})\mathbf{x} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x}) \tag{279}$$

参数量对比：

- 全参数微调：$d \times k$ 个参数
- LoRA：$r \times (d + k)$ 个参数
- 参数减少比例：$\frac{r(d+k)}{dk} = r\left(\frac{1}{k} + \frac{1}{d}\right)$

当 $r = 8$，$d = 4096$，$k = 4096$ 时，参数量从 $16,777,216$ 减少到 $65,536$，减少了约 256 倍。

算法原理：

Algorithm 12 LoRA 微调算法

Require: 预训练模型权重 $\mathbf{W}$，训练数据 $\mathcal{D}$，秩 r，学习率 η

Ensure: LoRA 权重 $\mathbf{A}$ 和 $\mathbf{B}$

初始化 $\mathbf{A}$ 为随机小值，$\mathbf{B}$ 为零矩阵
冻结原始权重 $\mathbf{W}$
repeat
  for 每个批次 $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
    前向传播：$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x})$
    计算损失：$\mathcal{L} = \text{loss}(\mathbf{h}, \mathbf{y})$
    反向传播，只更新 $\mathbf{A}$ 和 $\mathbf{B}$
    $\mathbf{A} \leftarrow \mathbf{A} - \eta \nabla_{\mathbf{A}} \mathcal{L}$
    $\mathbf{B} \leftarrow \mathbf{B} - \eta \nabla_{\mathbf{B}} \mathcal{L}$
  end for
until 收敛

代码实现：

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class LoRALayer(nn.Module):
    """LoRA 层实现"""

    def __init__(self, in_features, out_features, rank=8, alpha=16,
   dropout=0.0):
        """
        参数:
            in_features: 输入特征维度
            out_features: 输出特征维度
            rank: LoRA 的秩
            alpha: 缩放因子，通常等于 rank
            dropout: Dropout 概率
        """
        super().__init__()
```

```
        self.rank = rank
        self.alpha = alpha
        self.scaling = alpha / rank

        # LoRA 矩阵 A 和 B
        self.lora_A = nn.Parameter(torch.randn(rank, in_features) * 0.02)
        self.lora_B = nn.Parameter(torch.zeros(out_features, rank))

        # Dropout
        self.dropout = nn.Dropout(dropout) if dropout > 0 else nn.
    Identity()

        # 原始权重（冻结）
        self.weight = None  # 将在外部设置

    def forward(self, x, weight):
        """
        前向传播

        参数:
            x: 输入张量 (batch_size, ..., in_features)
            weight: 原始权重矩阵 (out_features, in_features)
        """
        # 原始输出
        original_output = F.linear(x, weight)

        # LoRA 输出: B @ (A @ x)
        x_dropout = self.dropout(x)
        lora_output = F.linear(
            F.linear(x_dropout, self.lora_A),
            self.lora_B
        )

        # 缩放并相加
        return original_output + self.scaling * lora_output

class LoRALinear(nn.Module):
    """包装的 LoRA 线性层"""

    def __init__(self, linear_layer, rank=8, alpha=16, dropout=0.0):
```

```
        super().__init__()
        self.original = linear_layer
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank,
            alpha,
            dropout
        )
        # 冻结原始权重
        for param in self.original.parameters():
            param.requires_grad = False

    def forward(self, x):
        return self.lora(x, self.original.weight)

# 使用示例
# 假设我们有一个预训练的线性层
pretrained_linear = nn.Linear(768, 3072)

# 包装为 LoRA 层
lora_linear = LoRALinear(pretrained_linear, rank=8, alpha=16)

# 前向传播
x = torch.randn(32, 768)  # batch_size=32
output = lora_linear(x)
print(f"输入形状: {x.shape}")
print(f"输出形状: {output.shape}")

# 检查可训练参数
total_params = sum(p.numel() for p in lora_linear.parameters() if p.
    requires_grad)
print(f"可训练参数数量: {total_params}")  # 8 * (768 + 3072) = 30720
print(f"原始参数数量: {pretrained_linear.weight.numel()}")  # 768 * 3072
    = 2359296
print(f"参数减少比例: {pretrained_linear.weight.numel() / total_params:.2
    f}x")
```

Listing 80: LoRA 从零实现

使用 PEFT 库实现：

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig, get_peft_model, TaskType

# 加载预训练模型
model_name = "meta-llama/Llama-2-7b-hf"  # 示例模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# 配置 LoRA
lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=8,  # rank
    lora_alpha=16,  # alpha
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"],  # 目标模块
    bias="none"
)

# 应用 LoRA
model = get_peft_model(model, lora_config)

# 打印可训练参数
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,738,415,616 || trainable
   \%: 0.06

# 训练（只更新 LoRA 参数）
# ... 训练代码 ...
```

Listing 81: 使用 Hugging Face PEFT 库

适用场景：

- 资源受限环境下的模型微调

- 需要快速适应多个任务的场景
- 模型服务化部署前的快速迭代

优势与局限：

优势：

- 参数量大幅减少（通常减少 100-1000 倍）
- 训练速度快，内存占用低
- 可以保存多个 LoRA 适配器，快速切换任务
- 效果接近全参数微调

局限：

- 在某些复杂任务上可能不如全参数微调
- 需要选择合适的 rank 和目标模块
- 多个 LoRA 适配器组合时可能产生冲突

52.2 QLoRA (Quantized LoRA)

概念解释：QLoRA 是 LoRA 的量化版本，通过 4-bit 量化进一步降低内存占用，使得在消费级 GPU 上微调大模型成为可能。

量化原理：

QLoRA 使用 4-bit NormalFloat (NF4) 量化，将 32-bit 浮点数映射到 4-bit 整数：

$$Q(x) = \text{round}\left(\frac{x - \text{offset}}{\text{scale}}\right) \times \text{scale} + \text{offset} \tag{280}$$

其中：

- offset：量化偏移量
- scale：量化缩放因子
- $Q(x)$：量化后的值

与 LoRA 的区别：

- LoRA：原始权重保持 FP16/BF16，只量化激活值
- QLoRA：原始权重量化为 4-bit，激活值量化为 8-bit，LoRA 权重保持 16-bit
- 内存节省：QLoRA 可以节省约 75% 的内存

数学表达式：

对于量化权重 $\mathbf{W}_Q$ 和 LoRA 增量：

$$\mathbf{W}' = \mathbf{W}_Q + \frac{\alpha}{r}\mathbf{BA} \tag{281}$$

其中 $\mathbf{W}_Q$ 是量化后的权重，在推理时动态反量化。

代码实现：

```
from transformers import AutoModelForCausalLM, AutoTokenizer,
    BitsAndBytesConfig
from peft import LoraConfig, get_peft_model,
    prepare_model_for_kbit_training
import torch

# 4-bit 量化配置
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,  # 嵌套量化
)

# 加载模型（自动量化）
model_name = "meta-llama/Llama-2-7b-hf"
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# 准备模型进行 k-bit 训练
model = prepare_model_for_kbit_training(model)

# LoRA 配置
```

```
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM"
)

# 应用 LoRA
model = get_peft_model(model, lora_config)

# 打印内存使用
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,738,415,616 || trainable
   \%: 0.06

# 内存占用对比:
# FP16 LoRA: ~14GB
# QLoRA: ~6GB (节省约 57\%)
```

Listing 82: 使用 bitsandbytes 和 PEFT 实现 QLoRA

内存优化效果:

对于 7B 参数的模型:

- 全参数微调（FP16）：约 28GB
- LoRA（FP16）：约 14GB
- QLoRA（4-bit）：约 6GB

优势与局限:

优势:

- 内存占用极低，可在消费级 GPU 上运行
- 训练速度与 LoRA 相当
- 效果损失很小（通常 $< 1\%$）

局限:

- 量化可能引入轻微的性能损失
- 需要支持 4-bit 量化的硬件
- 某些操作可能不支持量化

52.3 PEFT 框架

概念解释：PEFT（Parameter-Efficient Fine-Tuning）是 Hugging Face 提供的统一框架，支持多种参数高效微调方法。

支持的方法：

- LoRA
- Prefix Tuning
- P-Tuning v2
- Prompt Tuning
- AdaLoRA
- 自定义方法

统一接口：

```
from peft import (
    LoraConfig,
    PrefixTuningConfig,
    PromptTuningConfig,
    get_peft_model
)

# LoRA 配置
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"]
)

# Prefix Tuning 配置
prefix_config = PrefixTuningConfig(
```

```
    task_type="CAUSAL_LM",
    num_virtual_tokens=20
)

# Prompt Tuning 配置
prompt_config = PromptTuningConfig(
    task_type="CAUSAL_LM",
    num_virtual_tokens=20
)

# 统一接口应用
model = get_peft_model(model, lora_config)  # 或 prefix_config,
   prompt_config
```

Listing 83: PEFT 框架使用示例

52.4 LoRA 变体方法

AdaLoRA：自适应 LoRA，动态调整 rank。

数学公式：

$$\mathbf{W}' = \mathbf{W} + \sum_{i=1}^{r} s_i \mathbf{b}_i \mathbf{a}_i^T \tag{282}$$

其中 s_i 是重要性分数，用于剪枝不重要的 LoRA 模块。

DoRA (Weight-Decomposed Low-Rank Adaptation)：将权重分解为幅度和方向。

数学公式：

$$\mathbf{W}' = \frac{m}{||\mathbf{W} + \Delta\mathbf{W}||_c}(\mathbf{W} + \Delta\mathbf{W}) \tag{283}$$

其中 m 是可学习的幅度参数，$||\cdot||_c$ 是列范数。

52.5 参数效率对比

53 监督微调技术

53.1 SFT (Supervised Fine-Tuning)

概念解释：SFT 是在有标签数据上对预训练模型进行微调，使模型适应特定任务。

表 1: 不同 PEFT 方法的参数效率对比

方法	参数量比例	内存占用	训练速度	效果
全参数微调	100%	高	慢	最佳
LoRA	0.1-1%	中	快	接近最佳
QLoRA	0.1-1%	低	快	接近最佳
Prefix Tuning	0.1-0.5%	低	快	良好
P-Tuning v2	0.1-1%	中	中	良好
AdaLoRA	0.1-1%	中	中	接近最佳

数据格式:

```
# JSON 格式
{
    "instruction": "将以下文本翻译成英文",
    "input": "你好，世界",
    "output": "Hello, world"
}

# 对话格式
{
    "messages": [
        {"role": "user", "content": "什么是机器学习？"},
        {"role": "assistant", "content": "机器学习是..."}
    ]
}
```

Listing 84: SFT 数据格式示例

训练流程:

```
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
from datasets import load_dataset
import torch
```

```
# 加载模型和分词器
model_name = "meta-llama/Llama-2-7b-hf"
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token

# 准备数据
def format_prompt(example):
    prompt = f"### Instruction:\n{example['instruction']}\n\n### Input:\n
   {example['input']}\n\n### Response:\n{example['output']}"
    return {"text": prompt}

dataset = load_dataset("json", data_files="train.json")
dataset = dataset.map(format_prompt)

def tokenize_function(examples):
    return tokenizer(
        examples["text"],
        truncation=True,
        max_length=512,
        padding="max_length"
    )

tokenized_dataset = dataset.map(tokenize_function, batched=True)

# 训练参数
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-5,
    fp16=True,
    logging_steps=100,
    save_steps=500,
)

# 数据整理器
data_collator = DataCollatorForLanguageModeling(
```

```
    tokenizer=tokenizer,
    mlm=False  # 因果语言建模
)

# 训练器
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    data_collator=data_collator,
)

# 开始训练
trainer.train()
```

Listing 85: SFT 训练示例

53.2 指令微调（Instruction Tuning）

概念解释：指令微调通过大量指令-输出对训练模型，使模型能够理解和遵循指令。

数据构建：

```
# 指令数据格式
instruction_data = [
    {
        "instruction": "解释以下概念",
        "input": "量子计算",
        "output": "量子计算是利用量子力学原理..."
    },
    {
        "instruction": "总结以下文本",
        "input": "长文本...",
        "output": "总结内容..."
    }
]

# 使用 Alpaca 格式
def format_alpaca(example):
    return {
```

```
        "text": f"""Below is an instruction that describes a task. Write
    a response that appropriately completes the request.

### Instruction:
{example['instruction']}

### Input:
{example['input']}

### Response:
{example['output']}"""
    }
```

Listing 86: 指令数据构建示例

效果评估:

- 指令遵循准确率
- 输出质量评分
- 任务完成率

53.3 对话微调（Chat Fine-Tuning）

概念解释：对话微调专门针对多轮对话场景，训练模型进行自然对话。

多轮对话数据处理:

```
def format_chat(messages):
    """格式化多轮对话"""
    formatted = ""
    for msg in messages:
        role = msg["role"]
        content = msg["content"]
        if role == "user":
            formatted += f"User: {content}\n"
        elif role == "assistant":
            formatted += f"Assistant: {content}\n"
    return formatted
```

```
# 对话数据示例
chat_data = {
    "messages": [
        {"role": "user", "content": "你好"},
        {"role": "assistant", "content": "你好！有什么可以帮助你的吗？"},
        {"role": "user", "content": "介绍一下Python"},
        {"role": "assistant", "content": "Python是一种高级编程语言..."}
    ]
}
```

Listing 87: 对话数据处理示例

54 推理加速技术

54.1 vLLM (Very Large Language Model)

概念解释：vLLM 是专门为大模型推理优化的服务框架，通过 PagedAttention 等技术大幅提升吞吐量。

PagedAttention 原理：

传统 Attention 的 KV Cache 是连续的，导致内存碎片化。PagedAttention 将 KV Cache 分页管理：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{284}$$

PagedAttention 将 $\mathbf{K}$ 和 $\mathbf{V}$ 存储在非连续的页面中，按需分配。

KV Cache 优化：

```
from vllm import LLM, SamplingParams

# 初始化模型
llm = LLM(
    model="meta-llama/Llama-2-7b-hf",
    tensor_parallel_size=1,
    gpu_memory_utilization=0.9
)

# 采样参数
```

```
sampling_params = SamplingParams(
    temperature=0.7,
    top_p=0.9,
    max_tokens=512
)

# 批量推理
prompts = [
    "什么是机器学习？",
    "解释一下深度学习",
    "Python 的特点是什么？"
]

outputs = llm.generate(prompts, sampling_params)

for output in outputs:
    print(f"Prompt: {output.prompt}")
    print(f"Generated: {output.outputs[0].text}\n")
```

Listing 88: vLLM 使用示例

性能优势:

- 吞吐量提升 2-4 倍
- 内存利用率提高 50-80%
- 支持连续批处理

54.2 Flash Attention

概念解释: Flash Attention 通过分块计算和在线 softmax 优化注意力机制，减少内存占用。

数学原理:

标准 Attention:

$$\mathbf{O} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{285}$$

Flash Attention 分块计算：

$$\mathbf{O}_i = \sum_{j=1}^{N} \frac{\exp(\mathbf{s}_{ij} - m_i)}{\sum_{k=1}^{N} \exp(\mathbf{s}_{ik} - m_i)} \mathbf{V}_j \tag{286}$$

$$m_i = \max_j \mathbf{s}_{ij}, \quad \mathbf{s}_{ij} = \frac{\mathbf{Q}_i \mathbf{K}_j^T}{\sqrt{d_k}} \tag{287}$$

内存优化：

- 标准 Attention：$O(N^2)$ 内存
- Flash Attention：$O(N)$ 内存

```python
import torch
from flash_attn import flash_attn_func

# 输入
q = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")
k = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")
v = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")

# Flash Attention
output = flash_attn_func(q, k, v, dropout_p=0.0, softmax_scale=1.0/sqrt
    (64))

print(f"输出形状：{output.shape}")  # (32, 128, 8, 64)
```

Listing 89: Flash Attention 使用示例

54.3 量化推理

概念解释：量化推理通过降低模型精度减少内存占用和加速推理。

INT8 量化：

$$Q(x) = \text{round}\left(\frac{x}{\text{scale}}\right) \times \text{scale} \tag{288}$$

GPTQ 量化：

GPTQ（GPT Quantization）是一种后训练量化方法：

$$\arg\min_{\hat{\mathbf{W}}} ||\mathbf{W}\mathbf{X} - \hat{\mathbf{W}}\mathbf{X}||_2^2 \tag{289}$$

```python
from transformers import AutoModelForCausalLM, AutoTokenizer
from auto_gptq import AutoGPTQForCausalLM

# 加载模型
model_name = "meta-llama/Llama-2-7b-hf"

# GPTQ 量化
model = AutoGPTQForCausalLM.from_quantized(
    model_name,
    use_safetensors=True,
    device="cuda:0"
)

tokenizer = AutoTokenizer.from_pretrained(model_name)

# 推理
inputs = tokenizer("Hello, how are you?", return_tensors="pt").to("cuda")
outputs = model.generate(**inputs, max_length=50)
print(tokenizer.decode(outputs[0]))
```

Listing 90: GPTQ 量化示例

54.4 模型并行与张量并行

概念解释：当模型太大无法放入单卡时，需要将模型分布到多个 GPU。

张量并行：将矩阵乘法分块：

$$\mathbf{Y} = \mathbf{X}\mathbf{W} = \mathbf{X}[\mathbf{W}_1|\mathbf{W}_2] = [\mathbf{X}\mathbf{W}_1|\mathbf{X}\mathbf{W}_2] \tag{290}$$

流水线并行：将模型按层分割到不同 GPU。

55 模型部署技术

55.1 模型压缩

知识蒸馏：用大模型（教师）指导小模型（学生）学习。

数学公式：

$$\mathcal{L} = \alpha \mathcal{L}_{CE}(\mathbf{y}, \mathbf{p}_s) + (1-\alpha)\mathcal{L}_{KL}(\mathbf{p}_t, \mathbf{p}_s) \tag{291}$$

其中 $\mathbf{p}_t$ 是教师模型的输出，$\mathbf{p}_s$ 是学生模型的输出。

55.2 服务化部署

API 服务：使用 FastAPI、Flask 等框架部署模型服务。

```python
from fastapi import FastAPI
from transformers import pipeline
import torch

app = FastAPI()

# 加载模型
model = pipeline(
    "text-generation",
    model="meta-llama/Llama-2-7b-hf",
    device=0 if torch.cuda.is_available() else -1
)

@app.post("/generate")
async def generate_text(prompt: str, max_length: int = 100):
    result = model(prompt, max_length=max_length)
    return {"generated_text": result[0]["generated_text"]}

# 运行: uvicorn app:app --host 0.0.0.0 --port 8000
```

Listing 91: 模型服务化部署示例

55.3 边缘部署

模型量化：INT8/INT4 量化减少模型大小。

模型剪枝：移除不重要的权重。

专用硬件：使用 NPU、TPU 等专用芯片加速。

56 推理优化技术

56.1 KV Cache

概念解释：KV Cache 缓存之前计算的 Key 和 Value，避免重复计算。

优化效果：

- 减少计算量：从 $O(n^2)$ 到 $O(n)$
- 提升速度：2-10 倍加速

56.2 连续批处理（Continuous Batching）

概念解释：动态管理批次，新请求可以立即加入，完成的请求可以立即释放。

优势：

- 提高 GPU 利用率
- 降低延迟
- 支持动态负载

56.3 动态批处理

概念解释：根据请求长度动态调整批次大小。

策略：

- 短请求优先
- 长度相似请求分组
- 最大批次大小限制

57 总结

第一部分介绍了大语言模型的先进技术：

参数高效微调：

- LoRA：低秩适应，大幅减少参数量
- QLoRA：量化 + LoRA，进一步降低内存
- PEFT：统一框架，支持多种方法

监督微调：

- SFT：有监督微调
- 指令微调：提升指令遵循能力
- 对话微调：优化多轮对话

推理加速：

- vLLM：PagedAttention 优化
- Flash Attention：内存高效注意力
- 量化推理：INT8/INT4 量化

部署与优化：

- 模型压缩：知识蒸馏、剪枝
- 服务化部署：API 服务
- 推理优化：KV Cache、连续批处理

58 评估指标与方法

评估指标是衡量大语言模型性能的重要工具。不同的任务需要不同的评估指标，本节详细介绍各种评估指标的定义、计算方法和实现。

58.1 困惑度（Perplexity）

概念解释：困惑度（Perplexity, PPL）是语言模型评估中最常用的指标，衡量模型对测试数据的预测不确定性。困惑度越低，模型性能越好。

数学定义：

对于测试序列 $\mathbf{w} = w_1, w_2, \ldots, w_N$，困惑度定义为：

$$\text{PPL}(\mathbf{w}) = P(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}} \tag{292}$$

使用链式法则展开：

$$\text{PPL}(\mathbf{w}) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1, \ldots, w_{i-1})}} \tag{293}$$

使用对数形式（数值稳定）：

$$\text{PPL}(\mathbf{w}) = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log P(w_i | w_1, \ldots, w_{i-1})\right) \tag{294}$$

符号说明：

- N：序列长度（词数）
- w_i：第 i 个词
- $P(w_i | w_1, \ldots, w_{i-1})$：给定前文的条件概率
- log：自然对数

从零实现：

```
import torch
import torch.nn.functional as F
import math

def calculate_perplexity(model, tokenizer, text, device="cuda"):
    """
    计算文本的困惑度

    参数:
        model: 语言模型
        tokenizer: 分词器
```

```
        text: 输入文本
        device: 设备
    """
    # 分词
    tokens = tokenizer.encode(text, return_tensors="pt").to(device)
    input_ids = tokens[:, :-1]  # 输入（除了最后一个token）
    target_ids = tokens[:, 1:]   # 目标（除了第一个token）

    model.eval()
    with torch.no_grad():
        # 前向传播
        outputs = model(input_ids)
        logits = outputs.logits  # (batch_size, seq_len, vocab_size)

        # 计算每个位置的对数概率
        log_probs = F.log_softmax(logits, dim=-1)

        # 获取目标token的对数概率
        # log_probs: (batch_size, seq_len, vocab_size)
        # target_ids: (batch_size, seq_len)
        # 需要选择每个位置对应target的对数概率
        selected_log_probs = log_probs.gather(
            dim=2,
            index=target_ids.unsqueeze(2)
        ).squeeze(2)  # (batch_size, seq_len)

        # 计算平均负对数似然
        nll = -selected_log_probs.mean().item()

        # 困惑度
        perplexity = math.exp(nll)

    return perplexity, nll

# 使用示例
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "gpt2"  # 示例模型
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
text = "The quick brown fox jumps over the lazy dog."
ppl, nll = calculate_perplexity(model, tokenizer, text)
print(f"文本: {text}")
print(f"负对数似然: {nll:.4f}")
print(f"困惑度: {ppl:.4f}")
```

Listing 92: 困惑度从零实现

使用库实现:

```
from evaluate import load

perplexity = load("perplexity", module_type="metric")

# 计算困惑度
results = perplexity.compute(
    model_id="gpt2",
    add_start_token=False,
    predictions=["The quick brown fox jumps over the lazy dog."]
)

print(f"困惑度: {results['mean_perplexity']:.4f}")
```

Listing 93: 使用 evaluate 库计算困惑度

计算案例:

案例 1: 给定序列"the cat sat"，假设模型预测的概率为:

- $P(\text{cat}|\text{the}) = 0.3$
- $P(\text{sat}|\text{the cat}) = 0.5$

逐步计算：

$$\text{PPL} = \exp\left(-\frac{1}{2}[\log(0.3) + \log(0.5)]\right) \tag{295}$$

$$= \exp\left(-\frac{1}{2}[-1.204 + (-0.693)]\right) \tag{296}$$

$$= \exp\left(-\frac{1}{2} \times (-1.897)\right) \tag{297}$$

$$= \exp(0.9485) \tag{298}$$

$$= 2.582 \tag{299}$$

代码验证：

```python
import math

# 给定概率
probs = [0.3, 0.5]

# 计算困惑度
log_probs = [math.log(p) for p in probs]
nll = -sum(log_probs) / len(probs)
ppl = math.exp(nll)

print(f"概率：{probs}")
print(f"对数概率：{log_probs}")
print(f"平均负对数似然：{nll:.4f}")
print(f"困惑度：{ppl:.4f}")  # 2.582
```

Listing 94: 案例 1 代码验证

案例 2：更长的序列

假设序列长度为 10，平均对数概率为 -2.0：

$$\text{PPL} = \exp\left(-\frac{1}{10} \times (-2.0) \times 10\right) \tag{300}$$

$$= \exp(2.0) \tag{301}$$

$$= 7.389 \tag{302}$$

58.2 BLEU 分数

概念解释：BLEU（Bilingual Evaluation Understudy）是机器翻译和文本生成任务中最常用的评估指标，通过比较 n-gram 匹配来衡量生成文本与参考文本的相似度。

数学定义：

n-gram 精确度：

$$P_n = \frac{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}(\text{n-gram})} \tag{303}$$

其中 $\text{Count}_{\text{clip}}$ 是截断计数，不超过参考文本中该 n-gram 的最大出现次数。

BLEU 分数：

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log P_n\right) \tag{304}$$

其中：

- BP：简短惩罚（Brevity Penalty）
- w_n：n-gram 权重，通常 $w_n = 1/N$
- N：最大 n-gram 阶数，通常 $N = 4$

简短惩罚：

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{305}$$

其中 c 是候选文本长度，r 是参考文本长度。

从零实现：

```
from collections import Counter
import math

def get_ngrams(tokens, n):
    """获取 n-gram"""
    return [tuple(tokens[i:i+n]) for i in range(len(tokens)-n+1)]

def calculate_bleu(reference, candidate, max_n=4):
    """
    计算 BLEU 分数

    参数:
```

```
        reference: 参考文本（列表，每个元素是一个参考）
        candidate: 候选文本（token列表）
        max_n: 最大 n-gram 阶数
    """
    # 如果 reference 是字符串，转换为列表
    if isinstance(reference[0], str):
        reference = [ref.split() for ref in reference]
    if isinstance(candidate, str):
        candidate = candidate.split()

    # 计算简短惩罚
    c = len(candidate)
    r = min(len(ref) for ref in reference)  # 最接近的参考长度
    bp = 1.0 if c > r else math.exp(1 - r / c)

    # 计算各阶 n-gram 精确度
    precisions = []

    for n in range(1, max_n + 1):
        # 候选文本的 n-gram
        candidate_ngrams = get_ngrams(candidate, n)
        candidate_counts = Counter(candidate_ngrams)

        # 参考文本的 n-gram（所有参考）
        reference_counts_list = []
        for ref in reference:
            ref_ngrams = get_ngrams(ref, n)
            reference_counts_list.append(Counter(ref_ngrams))

        # 计算截断计数
        clipped_count = 0
        total_count = sum(candidate_counts.values())

        for ngram, count in candidate_counts.items():
            # 在所有参考中找到该 n-gram 的最大计数
            max_ref_count = max(
                ref_counts.get(ngram, 0) for ref_counts in
reference_counts_list
            )
            clipped_count += min(count, max_ref_count)
```

```
        # n-gram 精确度
        precision = clipped_count / total_count if total_count > 0 else 0
        precisions.append(precision)

    # 计算几何平均
    if any(p == 0 for p in precisions):
        return 0.0

    log_precision_sum = sum(math.log(p) for p in precisions)
    bleu = bp * math.exp(log_precision_sum / len(precisions))

    return bleu, precisions, bp

# 测试案例
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"参考: {reference}")
print(f"候选: {candidate}")
print(f"1-gram 精确度: {precisions[0]:.4f}")
print(f"2-gram 精确度: {precisions[1]:.4f}")
print(f"3-gram 精确度: {precisions[2]:.4f}")
print(f"4-gram 精确度: {precisions[3]:.4f}")
print(f"简短惩罚: {bp:.4f}")
print(f"BLEU 分数: {bleu_score:.4f}")
```

Listing 95: BLEU 分数从零实现

使用库实现:

```
from sacrebleu import BLEU

bleu = BLEU()

# 单个参考
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"
score = bleu.sentence_score(candidate, reference)
```

```
print(f"BLEU 分数: {score.score:.4f}")

# 多个参考
references = [
    ["the cat is on the mat"],
    ["there is a cat on the mat"]
]
score = bleu.sentence_score(candidate, references)
print(f"BLEU 分数（多参考）: {score.score:.4f}")
```

Listing 96: 使用 sacrebleu 库

计算案例:

案例 1:

- 参考: "the cat is on the mat"
- 候选: "the cat the cat on the mat"

逐步计算:

1-gram:

- 候选: the(2), cat(2), the(2), cat(2), on(1), the(2), mat(1)
- 参考: the(2), cat(1), is(1), on(1), the(2), mat(1)
- 截断计数: the(2), cat(1), on(1), mat(1) = 5
- 总数: 7
- $P_1 = 5/7 = 0.7143$

2-gram:

- 候选:(the cat)(2), (cat the)(1), (the cat)(2), (cat on)(1), (on the)(1), (the mat)(1)
- 参考: (the cat)(1), (cat is)(1), (is on)(1), (on the)(1), (the mat)(1)
- 截断计数: (the cat)(1), (cat on)(0), (on the)(1), (the mat)(1) = 3
- 总数: 6

- $P_2 = 3/6 = 0.5000$

简短惩罚：

- $c = 7, r = 6$
- $\text{BP} = e^{1-6/7} = e^{0.1429} = 1.1537$

BLEU 分数：

$$\text{BLEU} = \text{BP} \times \exp\left(\frac{1}{4}[\log P_1 + \log P_2 + \log P_3 + \log P_4]\right) \tag{306}$$

$$= 1.1537 \times \exp\left(\frac{1}{4}[\log(0.7143) + \log(0.5) + \log(0) + \log(0)]\right) \tag{307}$$

$$= 0 \quad （因为 P_3 = P_4 = 0） \tag{308}$$

代码验证：

```
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"1-gram 精确度: {precisions[0]:.4f}")  # 0.7143
print(f"2-gram 精确度: {precisions[1]:.4f}")  # 0.5000
print(f"简短惩罚: {bp:.4f}")  # 1.1537
print(f"BLEU 分数: {bleu_score:.4f}")  # 0.0000（因为3-gram和4-gram为0）
```

Listing 97: 案例 1 代码验证

案例 2：更好的匹配

参考：`"the cat sat on the mat"`
候选：`"the cat sat on the mat"`

这是完美匹配，BLEU 分数应该接近 1.0。

逐步计算：

1-gram：

- 候选：`the(2), cat(1), sat(1), on(1), the(2), mat(1)`
- 参考：`the(2), cat(1), sat(1), on(1), the(2), mat(1)`
- 截断计数：所有匹配 = 6

- 总数：6
- $P_1 = 6/6 = 1.0$

2-gram：

- 候选：(the cat)(1), (cat sat)(1), (sat on)(1), (on the)(1), (the mat)(1)
- 参考：(the cat)(1), (cat sat)(1), (sat on)(1), (on the)(1), (the mat)(1)
- 截断计数：所有匹配 = 5
- 总数：5
- $P_2 = 5/5 = 1.0$

类似地，$P_3 = 1.0$，$P_4 = 1.0$。

简短惩罚：

- $c = 6, r = 6$
- BP = 1.0（因为 $c = r$）

BLEU 分数：

$$\text{BLEU} = 1.0 \times \exp\left(\frac{1}{4}[\log(1.0) + \log(1.0) + \log(1.0) + \log(1.0)]\right) \tag{309}$$

$$= 1.0 \times \exp(0) \tag{310}$$

$$= 1.0 \tag{311}$$

代码验证：

```
reference = ["the cat sat on the mat"]
candidate = "the cat sat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"所有 n-gram 精确度: {precisions}")  # [1.0, 1.0, 1.0, 1.0]
print(f"简短惩罚: {bp:.4f}")  # 1.0000
print(f"BLEU 分数: {bleu_score:.4f}")  # 1.0000
```

Listing 98: 案例 2 代码验证

案例 3：部分匹配

参考：`"the cat is sitting on the mat"`
候选：`"a cat sits on mat"`

逐步计算：

1-gram：

- 候选：`a(1), cat(1), sits(1), on(1), mat(1)`
- 参考：`the(2), cat(1), is(1), sitting(1), on(1), the(2), mat(1)`
- 匹配：`cat(1), on(1), mat(1)` $= 3$
- 总数：5
- $P_1 = 3/5 = 0.6$

简短惩罚：

- $c = 5, r = 7$
- $\text{BP} = e^{1-7/5} = e^{-0.4} = 0.6703$

由于 2-gram、3-gram、4-gram 匹配较少，最终 BLEU 分数较低。

58.3 ROUGE 分数

概念解释：ROUGE（Recall-Oriented Understudy for Gisting Evaluation）主要用于文本摘要评估，关注召回率。

ROUGE-N：

$$\text{ROUGE-N} = \frac{\sum_{S \in \text{References}} \sum_{\text{n-gram} \in S} \text{Count}_{\text{match}}(\text{n-gram})}{\sum_{S \in \text{References}} \sum_{\text{n-gram} \in S} \text{Count}(\text{n-gram})} \tag{312}$$

ROUGE-L（最长公共子序列）：

$$\text{ROUGE-L} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{313}$$

其中：

$$R_{lcs} = \frac{LCS(X,Y)}{m} \quad \text{（召回率）} \tag{314}$$

$$P_{lcs} = \frac{LCS(X,Y)}{n} \quad \text{（精确率）} \tag{315}$$

$LCS(X,Y)$ 是最长公共子序列长度，m 是参考长度，n 是候选长度。

从零实现：

```
from collections import Counter

def lcs_length(x, y):
    """计算最长公共子序列长度"""
    m, n = len(x), len(y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i-1] == y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

def rouge_n(reference, candidate, n=1):
    """计算 ROUGE-N"""
    ref_ngrams = Counter(get_ngrams(reference.split(), n))
    cand_ngrams = Counter(get_ngrams(candidate.split(), n))

    matches = sum(min(ref_ngrams[ngram], cand_ngrams[ngram])
                  for ngram in ref_ngrams)
    total = sum(ref_ngrams.values())

    return matches / total if total > 0 else 0.0

def rouge_l(reference, candidate, beta=1.2):
    """计算 ROUGE-L"""
    ref_tokens = reference.split()
    cand_tokens = candidate.split()

    lcs_len = lcs_length(ref_tokens, cand_tokens)

    if len(ref_tokens) == 0 or len(cand_tokens) == 0:
        return 0.0
```

```
    recall = lcs_len / len(ref_tokens)
    precision = lcs_len / len(cand_tokens)

    if recall + precision == 0:
        return 0.0

    f_score = (1 + beta**2) * recall * precision / (recall + beta**2 *
   precision)
    return f_score, recall, precision

# 测试
reference = "the cat is on the mat"
candidate = "the cat sat on the mat"

rouge_1 = rouge_n(reference, candidate, n=1)
rouge_2 = rouge_n(reference, candidate, n=2)
rouge_l_score, recall, precision = rouge_l(reference, candidate)

print(f"ROUGE-1: {rouge_1:.4f}")
print(f"ROUGE-2: {rouge_2:.4f}")
print(f"ROUGE-L: {rouge_l_score:.4f} (Recall: {recall:.4f}, Precision: {
   precision:.4f})")
```

Listing 99: ROUGE 分数从零实现

计算案例:

案例 1:

- 参考: "the cat is on the mat"
- 候选: "the cat sat on the mat"

ROUGE-1 计算:

- 参考 1-gram: the(2), cat(1), is(1), on(1), mat(1), 共 6 个
- 候选 1-gram: the(2), cat(1), sat(1), on(1), mat(1), 共 6 个
- 匹配: the(2), cat(1), on(1), mat(1) = 5
- ROUGE-1 = $5/6 = 0.8333$

ROUGE-L 计算：

- 参考序列：[the, cat, is, on, the, mat]
- 候选序列：[the, cat, sat, on, the, mat]
- LCS：[the, cat, on, the, mat]，长度为 5
- Recall = 5/6 = 0.8333
- Precision = 5/6 = 0.8333
- ROUGE-L = $\frac{2\times0.8333\times0.8333}{0.8333+0.8333}$ = 0.8333

代码验证：

```
reference = "the cat is on the mat"
candidate = "the cat sat on the mat"

rouge_1 = rouge_n(reference, candidate, n=1)
rouge_2 = rouge_n(reference, candidate, n=2)
rouge_l_score, recall, precision = rouge_l(reference, candidate)

print(f"ROUGE-1: {rouge_1:.4f}")  # 0.8333
print(f"ROUGE-2: {rouge_2:.4f}")
print(f"ROUGE-L: {rouge_l_score:.4f}")  # 0.8333
print(f"Recall: {recall:.4f}, Precision: {precision:.4f}")
```

Listing 100: ROUGE 案例 1 代码验证

案例 2：

- 参考："the cat is on the mat"
- 候选："cat mat"

ROUGE-1 计算：

- 参考 1-gram：6 个
- 候选 1-gram：cat(1), mat(1)，共 2 个
- 匹配：cat(1), mat(1) = 2

- ROUGE-1 = $2/6 = 0.3333$

ROUGE-L 计算:

- LCS: `[cat, mat]`，长度为 2
- Recall = $2/6 = 0.3333$
- Precision = $2/2 = 1.0$
- ROUGE-L = $\frac{2\times0.3333\times1.0}{0.3333+1.0} = 0.5000$

58.4 METEOR 分数

概念解释: METEOR 考虑同义词匹配，比 BLEU 更灵活。

数学公式:

$$\text{METEOR} = (1 - \text{Penalty}) \times \text{F}_{\text{mean}} \tag{316}$$

其中:

$$\text{F}_{\text{mean}} = \frac{P \times R}{\alpha P + (1-\alpha)R} \tag{317}$$

$$\text{Penalty} = 0.5 \times \left(\frac{\text{chunks}}{\text{unigrams\_matched}}\right)^3 \tag{318}$$

58.5 BERTScore

概念解释: BERTScore 使用 BERT 嵌入计算语义相似度。

数学公式:

$$\text{Precision} = \frac{1}{|\hat{\mathbf{x}}|}\sum_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \max_{\mathbf{x}_j \in \mathbf{x}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \tag{319}$$

$$\text{Recall} = \frac{1}{|\mathbf{x}|}\sum_{\mathbf{x}_j \in \mathbf{x}} \max_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \tag{320}$$

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{321}$$

59 评测基准与数据集

59.1 GLUE 和 SuperGLUE

GLUE：General Language Understanding Evaluation，包含 9 个自然语言理解任务。

SuperGLUE：GLUE 的升级版，包含更具挑战性的任务。

59.2 MMLU

概念解释：Massive Multitask Language Understanding，包含 57 个任务，涵盖数学、物理、历史等多个领域。

59.3 HumanEval 和 MBPP

HumanEval：164 个 Python 编程问题，评估代码生成能力。

MBPP：974 个 Python 编程问题。

59.4 MT-Bench 和 AlpacaEval

MT-Bench：多轮对话评估基准。

AlpacaEval：指令遵循能力评估。

60 QA Pair（问答对）

60.1 LoRA 相关问答

Q1: 什么是 LoRA？它的核心思想是什么？

A: LoRA（Low-Rank Adaptation）是一种参数高效微调方法。核心思想是：对于预训练权重矩阵 $\mathbf{W}$，不直接更新它，而是学习一个低秩分解的增量 $\Delta\mathbf{W} = \mathbf{BA}$，其中 $\mathbf{A} \in \mathbb{R}^{r\times k}$，$\mathbf{B} \in \mathbb{R}^{d\times r}$，$r \ll \min(d, k)$。这样只需要训练 $r(d + k)$ 个参数，而不是 dk 个参数。

Q2: LoRA 和全参数微调的区别是什么？

A:

- **参数量**：LoRA 只更新 0.1-1% 的参数，全参数微调更新 100% 的参数

- **内存占用**：LoRA 内存占用大幅降低
- **训练速度**：LoRA 训练更快
- **效果**：LoRA 效果通常接近全参数微调（95-99%）
- **灵活性**：LoRA 可以保存多个适配器，快速切换任务

Q3: 如何选择 LoRA 的 rank?

A: rank 的选择需要权衡：

- **较小的 rank (4-8)**：参数量少，训练快，但可能表达能力不足
- **中等 rank (16-32)**：平衡性能和效率，适用于大多数任务
- **较大的 rank (64-128)**：表达能力更强，但参数量增加
- 建议从 $r = 8$ 开始，根据效果调整

60.2 评估指标相关问答

Q4: 如何计算 BLEU 分数？请给出详细步骤。

A: BLEU 分数计算步骤：

1. 计算各阶 n-gram（1-4）的精确度 P_n
2. 计算简短惩罚 BP
3. 计算几何平均：$\exp(\frac{1}{4}\sum_{n=1}^{4}\log P_n)$
4. 最终 BLEU = BP × 几何平均

Q5: 困惑度和 BLEU 的区别是什么?

A:

- **困惑度**：评估语言模型的预测不确定性，不需要参考文本，值越小越好
- **BLEU**：评估生成文本与参考文本的相似度，需要参考文本，值越大越好（0-1）
- **应用场景**：困惑度用于语言模型评估，BLEU 用于翻译和文本生成任务

61 综合练习

61.1 概念理解题

1. 解释 LoRA 的数学原理，说明为什么低秩分解能够有效。
2. 比较 QLoRA 和 LoRA 的区别，说明量化的作用。
3. 解释 Flash Attention 如何减少内存占用。
4. 说明 BLEU 分数的简短惩罚的作用。
5. 解释 ROUGE-L 和 ROUGE-N 的区别。

61.2 计算题

1. **困惑度计算**：
   - 给定序列长度为 100，平均对数概率为 -2.5，计算困惑度
   - 手算并编写代码验证
2. **BLEU 计算**：
   - 参考：`"the cat sat on the mat"`
   - 候选：`"a cat sat on mat"`
   - 计算 1-gram 到 4-gram 的精确度、简短惩罚和 BLEU 分数
   - 手算并编写代码验证
3. **ROUGE 计算**：
   - 参考：`"the cat is on the mat"`
   - 候选：`"cat mat"`
   - 计算 ROUGE-1、ROUGE-2 和 ROUGE-L
   - 手算并编写代码验证

61.3 代码实现题

1. 实现一个完整的 LoRA 训练脚本，包括数据加载、模型配置、训练循环和评估。
2. 实现 BLEU 分数计算函数，支持多个参考文本。

3. 实现 ROUGE 分数计算函数，包括 ROUGE-N 和 ROUGE-L。

4. 实现困惑度计算函数，支持批量计算。

61.4 案例分析题

1. 给定一个文本生成任务，设计完整的评估方案，包括选择合适的评估指标、实现评估代码、分析结果。

2. 分析 LoRA 在不同任务上的效果，比较不同 rank 设置的影响。

3. 设计一个模型部署方案，包括模型压缩、服务化部署和性能优化。

通过完成以上练习，读者可以深入理解大语言模型的先进技术和评估方法，掌握从模型微调到评估部署的完整流程。