東南大學
SOUTHEAST UNIVERSITY

模式识别实验报告

专业: 人工智能

学号: 58122204

年级: 大二

姓名: 谢兴

签名:

时间:

目录

1 实验一 KNN Classification

1.1 问题描述

1.2 概述

利用 KNN 算法，对 Iris 鸢尾花数据集中的测试集进行分类。

1.3 任务说明

1. 利用欧式距离作为 KNN 算法的度量函数，对测试集进行分类。实验报告中，要求在验证集上分析近邻数 k 对 KNN 算法分类精度的影响。

2. 利用马氏距离作为 KNN 算法的度量函数，对测试集进行分类。

3. 基于 MindSpore 平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore 的心得和建议。

4.（加分项）使用 MindSpore 平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法并与 MindSpore 平台上的官方实现算法进行对比，并进一步分析差异及其成因。

1.4 实现步骤与流程

1.4.1 实验思路

1. 导入必要的库，包括 numpy, pandas, matplotlib, plotly 和 seaborn；

2. 数据加载和基本信息显示；

   (a) 从 data/train.csv 文件中加载训练数据集

   (b) 显示数据集的前几行数据

(c) 显示数据集的描述性统计信息

(d) 显示数据集的基本信息，包括数据类型和缺失值情况

3. 数据可视化；

(a) 使用 seaborn 绘制数据集的特征两两关系图，并按标签着色

(b) 使用 plotly 绘制数据分布的饼图

(c) 分别绘制每个特征（萼片长度、萼片宽度、花瓣长度、花瓣宽度）的箱线图和直方图

4. 实现基于 Euiclidean 距离和基于 Mahalanobis 距离的 KNN 算法；

5. 数据处理和预测；

(a) 加载测试数据集并进行必要的类型转换和缺失值检查

(b) 使用预训练模型对测试数据进行预测，并将预测结果保存到 CSV 文件中

6. 最后对比欧式距离和马氏距离两种度量方式的分类效果和差异

(a) 比较多个预测结果文件 task1_test_prediction.csv 和 task2_test_prediction.csv 之间的差异，找出不同的行和列，并打印出不同值的位置

1.4.2 数学模型

KNN 算法的数学模型如下：

给定一个测试样本 x，KNN 算法通过计算 x 与训练集中所有样本之间的距离（常用欧氏距离），选择距离最近的 k 个样本，然后通过多数投票法决定 x 的类别。欧氏距离的计算公式为：

$$d(x_1, x_2) = \sqrt{\sum_{i=1}^{n}(x_{1i} - x_{2i})^2}$$

马氏距离的计算公式为：

$$d(x_1, x_2) = \sqrt{(x_1 - x_2)^T \Sigma^{-1} (x_1 - x_2)}$$

其中，Σ 为协方差矩阵。

1.4.3 关键难点

1. 如何高效地计算欧氏距离。

2. 如何在较大的数据集上进行快速的邻居搜索。

1.4.4 算法描述

基于 Euiclidean 距离和 Mahalanobis 距离的 KNN 算法的伪代码分别见算法 1 和算法 2。

Algorithm 1 K-Nearest Neighbors Based on Euclidean Distance

初始化 KNN 分类器，邻居数为 k
procedure 拟合 (X_train, y_train)
    存储训练数据和标签
end procedure
procedure 预测 (X)
    for 每个测试数据 x **do**
        计算 x 与所有训练样本之间的欧氏距离
        对距离进行排序，选择最近的 k 个邻居
        对这 k 个邻居的标签进行多数投票
        将多数投票结果赋予 x
    end for
    return 预测的标签
end procedure

Algorithm 2 K-Nearest Neighbors Based on Mahalanobis Distance

初始化 KNN 分类器，邻居数为 k，矩阵 A 的维度为 e，学习率为 η，最大迭代次数为 max_iter
procedure 拟合 (X_train, y_train)
    存储训练数据和标签
    初始化矩阵 A 为随机值
    for 迭代次数 $iteration = 1, 2, \ldots, max\_iter$ **do**
        初始化梯度矩阵 ∇A 为零
        for 每个训练样本 x_i **do**
            获取与 x_i 同类的样本索引 $same\_class\_indices$
            for 每个同类样本 x_j **do**
                if $i == j$ **then**
                    跳过
                end if
                计算 p_{ij} 值
                计算样本差异 $diff = x_i - x_j$
                更新梯度 $\nabla A += 2 \cdot p_{ij} \cdot (A \cdot diff) \cdot diff^T$
            end for
        end for
        按学习率更新矩阵 A: $A = A - \eta \cdot \nabla A / n$
    end for
end procedure
procedure 预测 (X)
    for 每个测试数据 x **do**
        计算 x 与所有训练样本之间的马氏距离
        对距离进行排序，选择最近的 k 个邻居
        对这 k 个邻居的标签进行多数投票
        将多数投票结果赋予 x
    end for
    return 预测的标签
end procedure

1.4.5 马氏距离梯度计算公式推导

假设我们有训练数据集 $\{(x_i, y_i)\}_{i=1}^n$，其中 $x_i \in \mathbb{R}^d$ 为样本特征，y_i 为样本类别。为了优化马氏距离下的 KNN 算法，我们需要学习一个矩阵 $A \in \mathbb{R}^{e \times d}$，使得同类样本之间的距离最小化。马氏距离的计算公式为：

$$d_M(x_i, x_j) = \sqrt{(x_i - x_j)^\top A^\top A (x_i - x_j)} \tag{1}$$

为了优化矩阵 A，我们使用如下的目标函数：

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j \in \mathcal{N}(i)} p_{ij} \cdot d_M^2(x_i, x_j) \tag{2}$$

其中，$\mathcal{N}(i)$ 表示与 x_i 同类的样本索引集合，p_{ij} 为权重，定义为：

$$p_{ij} = \frac{\exp(-d_M^2(x_i, x_j))}{\sum_{k \in \mathcal{N}(i)} \exp(-d_M^2(x_i, x_k))} \tag{3}$$

首先，我们对 $d_M^2(x_i, x_j)$ 进行展开：

$$d_M^2(x_i, x_j) = (x_i - x_j)^\top A^\top A (x_i - x_j) \tag{4}$$

为了计算梯度 $\nabla_A \mathcal{L}$，我们需要对 $\mathcal{L}$ 关于 A 求导：

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j \in \mathcal{N}(i)} p_{ij} (x_i - x_j)^\top A^\top A (x_i - x_j) \tag{5}$$

对 A 求导时，需要使用链式法则：

$$\frac{\partial \mathcal{L}}{\partial A} = \sum_{i=1}^{n} \sum_{j \in \mathcal{N}(i)} \left(\frac{\partial p_{ij}}{\partial A} (x_i - x_j)^\top A^\top A (x_i - x_j) + p_{ij} \frac{\partial ((x_i - x_j)^\top A^\top A (x_i - x_j))}{\partial A} \right) \tag{6}$$

首先计算 p_{ij} 对 A 的导数。由于 p_{ij} 包含在指数函数内，我们得到：

$$\frac{\partial p_{ij}}{\partial A} = p_{ij} \left(- \sum_{k \in \mathcal{N}(i)} p_{ik} \cdot 2(x_i - x_k)^\top A^\top \cdot (x_i - x_k) + 2(x_i - x_j)^\top A^\top \cdot (x_i - x_j) \right) \tag{7}$$

然后计算 $(x_i - x_j)^\top A^\top A (x_i - x_j)$ 对 A 的导数：

$$\frac{\partial ((x_i - x_j)^\top A^\top A (x_i - x_j))}{\partial A} = 2A(x_i - x_j)(x_i - x_j)^\top \tag{8}$$

将以上结果代入梯度公式中，我们得到：

$$\begin{aligned} \nabla_A \mathcal{L} = \sum_{i=1}^{n} \sum_{j \in \mathcal{N}(i)} \Bigg[p_{ij} \Bigg(& - \sum_{k \in \mathcal{N}(i)} p_{ik} \cdot 2(x_i - x_k)^\top A^\top \cdot (x_i - x_k) \\ & + 2(x_i - x_j)^\top A^\top \cdot (x_i - x_j) \Big) + 2 p_{ij} A (x_i - x_j)(x_i - x_j)^\top \Bigg] \end{aligned} \tag{9}$$

整理后得到最终的梯度公式：

$$\nabla_A \mathcal{L} = 2 \sum_{i=1}^{n} \sum_{j \in \mathcal{N}(i)} p_{ij} \left[A(x_i - x_j)(x_i - x_j)^\top - \sum_{k \in \mathcal{N}(i)} p_{ik} A (x_i - x_k)(x_i - x_k)^\top \right] \tag{10}$$

1.5 实验结果与分析

1.5.1 数据集的部分可视化分析

1. train.csv 文件中训练数据的 pairplot 图如图 1 所示。

2. 训练数据的分布情况如图 2 所示，可以看出这三类鸢尾花的数据分布比例是不完全一致的，但三类的数据量大致相同。

1.5.2 实验结果的分析

1. 对于基于欧氏距离的 KNN 算法，当 $k = 3$ 时，测试集的准确率为 93.33%；

2. 对于基于马氏距离的 KNN 算法，当 $k = 3$ 时，测试集的准确率为 93.33%；

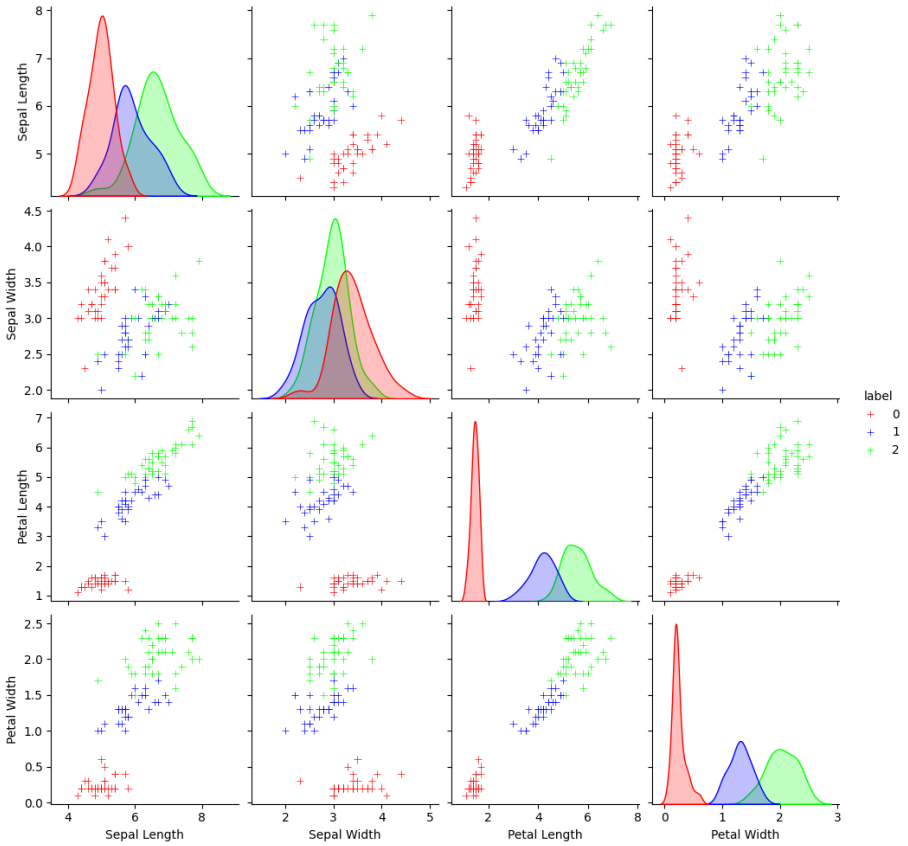


图 1: train.csv 训练数据的 pairplot 图

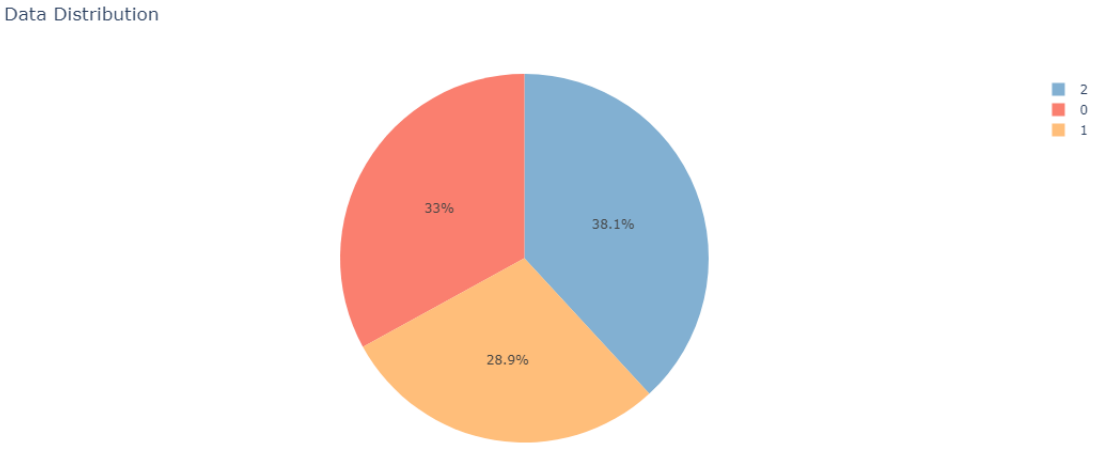


图 2: train.csv 训练数据分布比例

3. 将 k 从 1 到 50 遍历，分析出基于欧氏距离的 KNN 算法对 Iris 数据集进行分类的准确率随 k 的变化情况，如图 3 所示。

显然，基于欧式距离的最佳 k 值为 5 或 27 或 29，此时的准确率最高，为 100%，说明 $k = 5$，$k = 27$，$k = 29$ 时能完全正确的将 Iris 数据集中三类鸢尾花进行分类。

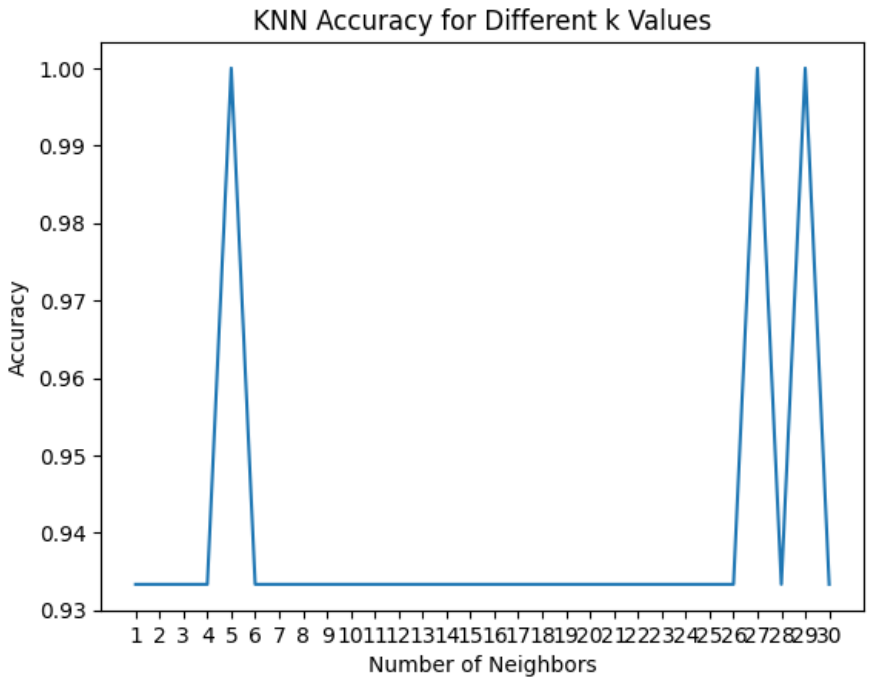


图 3: 分类准确率随 k 的变化情况

1.5.3 手动实现算法的评价

传统的 KNN 方法的不足之处主要包括：

1. 分类速度慢：最近邻分类器是基于实例学习的懒惰学习方法，因为它是根据所给训练样本构造的分类器，是将所有训练样本首先存储起来，当要进行分类时，就临时进行计算处理。需要计算待分样本与训练样本库中每一个样本的相似度，才能求得与其最近的 K 个样本。对于高维样本或样本集规模较大的情况，其时间和空间复杂度较高，时间代价为 $O(mn)$，其中 m 为向量空间模型空间特征维数，n 为训练样本集大小。

2. 样本库容量依赖性较强：对 KNN 算法在实际应用中的限制较大：有不少类别无法提供足够的训练样本，使得 KNN 算法所需要的相对均匀的特征空间条件无法得到满足，使得识别的误差较大。

3. 特征作用相同：与决策树归纳方法和神经网络方法相比，传统最近邻分类器认为每个属性的作用都是相同的（赋予相同权重）。样本的距离是根据样本的所有特征（属性）计算的。在这些特征中，有些特征与分类是强相关的，有些特征与分类是弱相关的，还有一些特征（可能是大部分）与分类不相关。这样，如果在计算相似度的时候，按所有特征作用相同来计算样本相似度就会误导分类过程。

4. K 值的确定：KNN 算法必须指定 K 值，K 值选择不当则分类精度不能保证。

KNN 的改进：对于 KNN 分类算法的改进方法主要可以分为加快分类速度、对训练样本库的维护、相似度的距离公式优化和 K 值确定四种类型。

1. 加快 KNN 算法的分类速度

就学习而言，懒惰学习方法比积极学习方法要快，就计算量而言，它要比积极学习方法慢许多，因为懒惰学习方法在进行分类时，需要进行大量的计算。针对这一问题，到目前为止绝大多数解决方法都是基于减少样本量和加快搜索 K 个最近邻速度两个方面考虑的：

(a) 浓缩训练样本

当训练样本集中样本数量较大时，为了减小计算开销，可以对训练样本集进行编辑处理，即从原始训练样本集中选择最优的参考子集进行 K 最近邻寻找，从而减少训练样本的存储量和提高计算效率。这类方法主要包括 Condensing 算法、WilSon 的 Editing 算法和 Devijver 的 MultiEdit 算法，Kuncheva 使用遗传算法在这方面也进行了一些研究。

(b) 加快 K 个最近邻的搜索速度

这类方法是通过快速搜索算法，在较短时间内找到待分类样本的 K 个最近邻。在具体进行搜索时，不要使用盲目的搜索方法，而是要采用一定的方法加快搜索速度或减小搜索范围，例如可以构造交叉索引表，利用匹配成功与否的历史来修改样本库的结构，使用样本和概念来构造层次或网络来组织训练样本。

2. 相似度的距离公式的优化

为了改变传统 KNN 算法中特征作用相同的缺陷，可在相似度的距离公式中给特征赋予不同权重，例如在欧氏距离公式中给不同特征赋予不同权重。特征的权重一般根据各个特征在分类中的作用设定，可根据特征在整个训练样本库中的所起的作用大小来确定权重，也可根据在训练样本的局部样本（靠近待测试样本的样本集合）中的分类作用确定权重。

3. 对训练样本库的维护

对训练样本库进行维护以满足 KNN 算法的需要，包括对训练样本库中的样本进行添加或删除。对样本库的维护并不是简单的增加删除样本，而是可采用适当的办法来保证空间的大小，如符合某种条件的样本可以加入数据库中，同时可以对数据库库中已有符合某种条件的样本进行删除。从而保证训练样本库中的样本提供 KNN 算法所需要的相对均匀的特征空间。

4. K 值选择 K 的选择原则一般为：

(a) K 的选择往往通过大量独立的测试数据、多个模型来验证最佳的选择；

(b) K 值一般事先确定，也可以使用动态的，例如采用固定的距离指标，只对小于该指标的样本进行分析。

1.6 MindSpore 学习使用心得体会

1.7 代码附录（数据加载可视化展示部分，具体见 knn.ipynb 文件）

1.7.1 knn.ipynb

```
# 实验一代码
```

1.7.2 knn_mindspore.ipynb

```
# 实验一代码
```

2 实验二 Naïve Bayes Classification

2.1 问题描述

2.1.1 概述

利用朴素贝叶斯算法，对 MNIST 数据集中的测试集进行分类。

2.1.2 任务说明

1. 在课程学习中同学们已经学习了贝叶斯分类理论并掌握了其基本原理，即利用贝叶斯公式

$$p(\omega_j|x) = \frac{p(x|\omega_j)p(\omega_j)}{p(x)}$$

对 $p(\omega_j|x)$ 作出预测。由于 $p(x)$ 为一固定值，所以一般不在计算过程中求得 $p(x)$ 的具体值。在实际运用中，为了方便计算，通常假设数据特征之间相互独立，即

$$p(x|\omega_j) = p(x_1|\omega_j) \cdot p(x_2|\omega_j) \cdots p(x_d|\omega_j), \quad x \in \mathbb{R}^d,$$

这便是著名的朴素贝叶斯算法。

2. MNIST 数据集本身以二进制形式保存，所以首先需要选择合适的编程语言编写读写二进制数据的程序完成对图片、标记信息的初步提取工作。读取了图片信息后，发现每个像素点的值在 [0,1] 区间内，这是图像压缩后的结果，所以可以先将像素值乘以 255 再取整，得到每一个点的灰度值。将图像二值化，得到可以用于分类的 28×28 个特征向量以及对应的标签数据，之后便可以交由贝叶斯分类器进行学习。

3. 基于 MindSpore 平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因，给出使用 MindSpore 的心得和建议。

4.（加分项）使用 MindSpore 平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法并与 MindSpore 平台上的官方实现算法进行对比，并进一步分析差异及其成因。

2.2 实现步骤与流程

2.2.1 实验思路

1. 读取数据集的图片和标签信息；

2. 对图片信息进行预处理，包括归一化、二值化和将图像展开成一维向量；

3. 实现朴素贝叶斯算法，包括拟合和预测两个步骤；

4. 使用预训练模型对测试数据进行预测，计算准确率；

5. 可视化部分模型对测试数据的预测结果。

2.2.2 数学模型

朴素贝叶斯学习步骤如下。先计算类先验概率分布：

$$P(Y=c_k)=\frac{1}{N}\sum_{i=1}^{N}I(\hat{y}_i=c_k),\quad k=1,2,\cdots,K \tag{11}$$

其中 c_k 表示第 k 个类别，y_i 表示第 i 个样本的类标记。类先验概率分布可以通过极大似然估计得到。

然后计算类条件概率分布：

$$P(X=x|Y=c_k)=P(X^{(1)}=x^{(1)},\cdots,X^{(n)}=x^{(n)}|Y=c_k),\quad k=1,2,\cdots,K \tag{12}$$

直接对 $P(X=x|Y=c_k)$ 进行估计不太可行，因为参数量太大。但是朴素贝叶斯的一

个最重要的假设就是条件独立性假设，即：

$$P(X=x|Y=c_k)=P(X^{(1)}=x^{(1)},\cdots,X^{(n)}=x^{(n)}|Y=c_k)=\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k) \tag{13}$$

有了条件独立性假设之后，便可基于极大似然估计计算类条件概率。

类先验概率分布和类条件概率分布都计算得到之后，基于贝叶斯公式即可以计算类后验概率：

$$P(Y=c_k|X=x)=\frac{P(X=x|Y=c_k)P(Y=c_k)}{\sum_k P(X=x|Y=c_k)P(Y=c_k)} \tag{14}$$

代入类条件计算公式，有：

$$P(Y=c_k|X=x)=\frac{\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)P(Y=c_k)}{\sum_k\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)P(Y=c_k)} \tag{15}$$

基于上述公式即可以学习一个朴素贝叶斯模型。给定新的数据样本时，计算其最大后验概率即可：

$$\hat{y}=\arg\max_{c_k}\frac{\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)P(Y=c_k)}{\sum_k\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)P(Y=c_k)} \tag{16}$$

其中，分母对于所有的 $\hat{y}$ 都是一样的，所以上述式可进一步简化为：

$$\hat{y}=\arg\max_{c_k}\prod_{j=1}^{n}P(X^{(j)}=x^{(j)}|Y=c_k)P(Y=c_k) \tag{17}$$

方程 17 即为朴素贝叶斯算法的预测公式。

2.2.3 关键难点

朴素贝叶斯算法的难点在于如何高效地计算类条件概率分布。由于朴素贝叶斯算法的条件独立性假设，可以将类条件概率分布分解为各个特征的条件概率分布的乘积。这样可以大大减少计算量。

2.2.4 算法描述

朴素贝叶斯算法实现的伪代码如算法 3 所示。

Algorithm 3 Optimized Multinomial Naive Bayes

Input: 平滑参数 α
Initialize:
    类别数 $n\_classes$
    特征数 $n\_features$
    类别计数 $class\_count$
    特征计数 $feature\_count$
    类别对数先验 $class\_log\_prior$
    特征对数概率 $feature\_log\_prob$
procedure 拟合 (X, y)
    获取唯一类别 $classes = \text{np.unique}(y)$
    初始化类别计数 $class\_count$ 和特征计数 $feature\_count$
    for 每个类别 $c \in classes$ **do**
        获取属于类别 c 的样本 X_c
        更新类别计数 $class\_count[c]$
        更新特征计数 $feature\_count[c, :]$
    end for
    计算类别对数先验 $class\_log\_prior$
    计算特征对数概率 $feature\_log\_prob$
end procedure
procedure 预测 (X)
    计算对数似然 $log\_likelihood = X \times feature\_log\_prob^T$
    计算对数后验概率 $log\_posterior = log\_likelihood + class\_log\_prior$
    返回类别 $classes[\text{np.argmax}(log\_posterior, axis = 1)]$
end procedure

2.3 实验结果与分析

2.3.1 手动实现朴素贝叶斯算法

利用课程所提供的 MNIST 数据集，我们手动实现了朴素贝叶斯算法，并在测试集上进行了分类。可以看出实验的准确率为 **70.26%**。

为便于观察手动实现朴素贝叶斯算法模型的性能，我们在部分验证集上进行了测试，测试结果如图 4 所示。可以看出在多数图片的预测中，手动实现的朴素贝叶斯算法还是能够将 MNIST 数据集中的数字进行正确分类的。

2.3.2 手动实现算法的评价

本实验要求手动实现朴素贝叶斯算法，这里我实现的是基于多项式朴素贝叶斯算法，并进行了 label smoothing。实验结果表明，手动实现的朴素贝叶斯算法在 MNIST 数据集上的分类准确率为 70.26%。这个准确率相对较低，可能是因为手动实现的朴素贝叶斯算法对数据集的特征提取和分类效果不够好。

未来可以采用半朴素贝叶斯算法，适当考虑一部分属性间的相互依赖信息，从而既不需要进行完全联合概率计算，又不至于彻底忽略了比较强的相互依赖关系。因为朴素贝叶斯党的假设是：假设属性之间相互独立，但实际上属性之间是有关联的，所以半朴素贝叶斯算法是对朴素贝叶斯算法的一种改进。

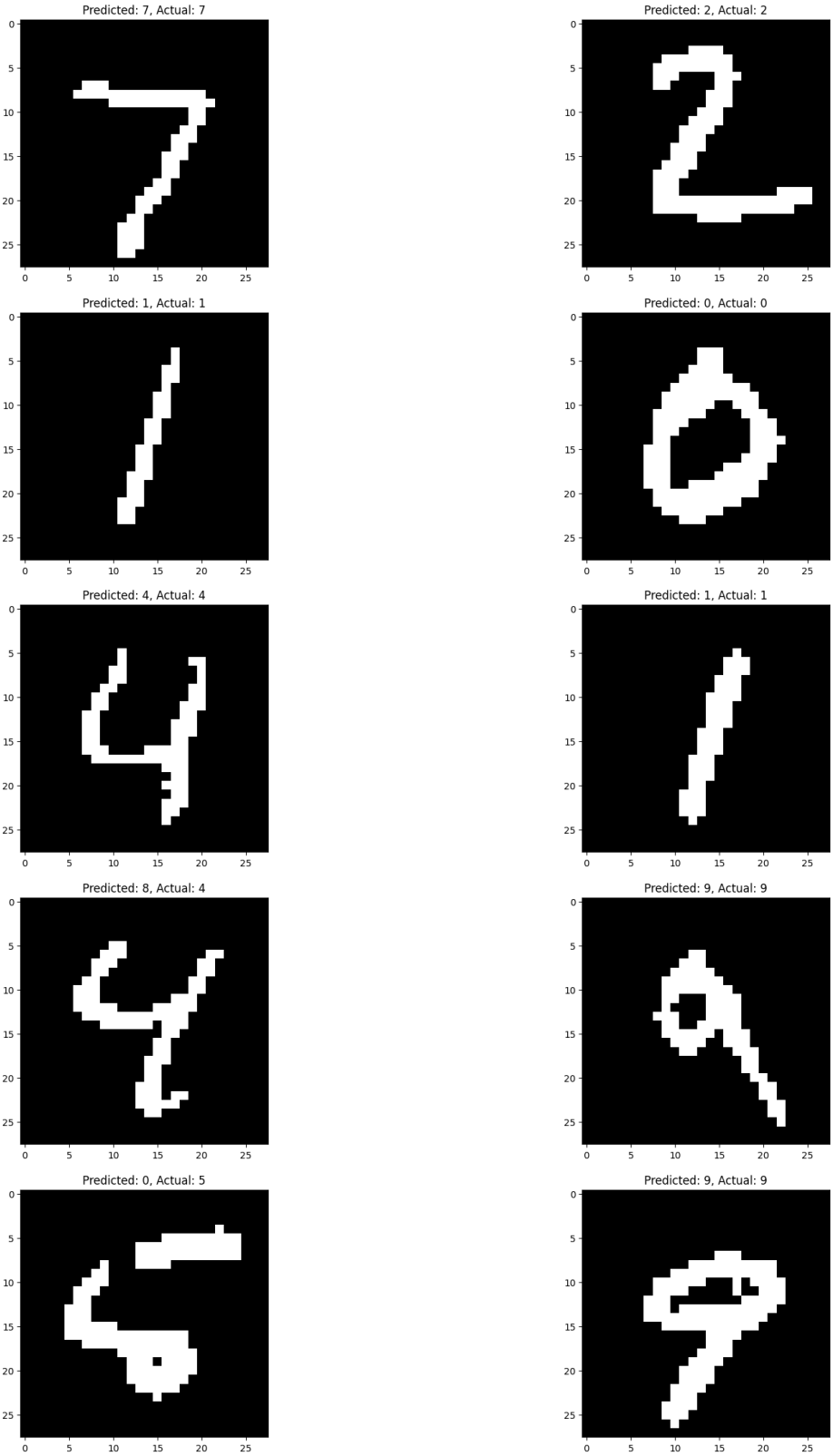
2.4 MindSpore 学习使用心得体会

2.5 代码附录

2.5.1 naive_bayes.ipynb

```
import numpy as np
import struct
import gzip
import matplotlib.pyplot as plt
def read_images(file_path):
```

图 4: 手动实现 Naïve Bayes 测试结果



```
    with gzip.open(file_path, 'rb') as f:
        # 读取文件头信息：魔数和图片数量
        magic, num_images = struct.unpack(">II", f.read(8))
        # 读取图片的行数和列数
        num_rows, num_cols = struct.unpack(">II", f.read(8))
        print(f"Magic number: {magic}, Number of images: {num_images}, Rows: {num_rows}, Columns: {num_cols}")
        # 读取图片数据
        images = np.frombuffer(f.read(), dtype=np.uint8).reshape(num_images, num_rows, num_cols)
        return images

def read_labels(file_path):
    with gzip.open(file_path, 'rb') as f:
        # 读取文件头信息：魔数和标签数量
        magic, num_labels = struct.unpack(">II", f.read(8))
        print(f"Magic number: {magic}, Number of labels: {num_labels}")
        # 读取标签数据
        labels = np.frombuffer(f.read(), dtype=np.uint8)
        return labels

# 设置数据集文件路径
train_images_path = 'data/train-images-idx3-ubyte.gz'
train_labels_path = 'data/train-labels-idx1-ubyte.gz'
test_images_path = 'data/t10k-images-idx3-ubyte.gz'
test_labels_path = 'data/t10k-labels-idx1-ubyte.gz'

# 读取数据集
X_train = read_images(train_images_path)
y_train = read_labels(train_labels_path)
X_test = read_images(test_images_path)
y_test = read_labels(test_labels_path)

print(f"Training data shape: {X_train.shape}")
print(f"Training labels shape: {y_train.shape}")
print(f"Test data shape: {X_test.shape}")
print(f"Test labels shape: {y_test.shape}")
# 将像素值归一化到[0, 1]范围
X_train = X_train / 255.0
X_test = X_test / 255.0

```

```
# 将图像二值化
X_train = (X_train > 0.5).astype(int)
X_test = (X_test > 0.5).astype(int)

# 将图像展开成一维向量
X_train = X_train.reshape(X_train.shape[0], -1)
X_test = X_test.reshape(X_test.shape[0], -1)
class OptimizedMultinomialNaiveBayes:
    def __init__(self, alpha=1.0):
        self.alpha = alpha  # 平滑参数
        self.classes = None
        self.class_count = None
        self.feature_count = None
        self.class_log_prior = None
        self.feature_log_prob = None

    def fit(self, X, y):
        self.classes = np.unique(y)
        n_classes = len(self.classes)
        n_features = X.shape[1]
        self.class_count = np.zeros(n_classes)
        self.feature_count = np.zeros((n_classes, n_features))

        for idx, c in enumerate(self.classes):
            X_c = X[y == c]
            self.class_count[idx] = X_c.shape[0]
            self.feature_count[idx, :] = np.sum(X_c, axis=0)

        self.class_log_prior = np.log(self.class_count / np.sum(self.
 class_count))
        self.feature_log_prob = np.log((self.feature_count + self.alpha) / (
 self.class_count[:, None] + self.alpha * n_features))

    def predict(self, X):
        log_likelihood = X @ self.feature_log_prob.T
        log_posterior = log_likelihood + self.class_log_prior
        return self.classes[np.argmax(log_posterior, axis=1)]
# 实例化并训练优化后的多项式朴素贝叶斯分类器
omnb = OptimizedMultinomialNaiveBayes(alpha=1.0)
omnb.fit(X_train, y_train)

```

```
# 在测试集上进行预测
predictions = omnb.predict(X_test)

# 计算准确率
accuracy = np.mean(predictions == y_test)
print(f'Accuracy: {accuracy}')
# 可视化前几个预测结果
for i in range(10):
    plt.imshow(X_test[i].reshape(28, 28), cmap='gray')
    plt.title(f'Predicted: {predictions[i]}, Actual: {y_test[i]}')
    plt.show()
```

3 实验三 Neural Network Image Classification

3.1 问题描述

3.2 概述

利用神经网络算法，对 CIFAR 数据集中的测试集进行分类。

3.3 任务说明

1. 基于神经网络模型及 BP 算法，根据训练集中的数据对你设计的神经网络模型进行训练，随后对给定的打乱的测试集中的数据进行分类。

2. 基于 MindSpore 平台提供的官方模型库，对相同的数据集进行训练，并与自己独立实现的算法对比结果（包括但不限于准确率、算法迭代收敛次数等指标），并分析结果中出现差异的可能原因。

3.（加分项）使用 MindSpore 平台提供的相似任务数据集（例如，其他的分类任务数据集）测试自己独立实现的算法并与 MindSpore 平台上的官方实现算法进行对比，并进一步分析差异及其成因。

3.4 实现步骤与流程

3.4.1 实验环境

实验环境见表 1。

3.4.2 实验思路

1. 数据加载与预处理:

   (a) 从 CIFAR-10 数据集中加载图像数据和标签。

表 1: Experiment Environment

Items	Version
CPU	Intel Core i5-1135G7
RAM	16 GB
Python	3.11.5
Operating system	Windows11

(b) 将图像数据从原始格式转换为适用于神经网络的格式（$32 \times 32 \times 3 \rightarrow 3072 \times 1$）。

(c) 将图像数据进行归一化处理，将像素值缩放到 [0, 1] 范围内。

2. 数据划分:

(a) 将加载的训练数据划分为训练集和验证集，以便在训练过程中进行模型评估。

3. 定义全连接神经网络结构:

(a) 输入层：包含 3072 个神经元，对应每张图像的 3072 个像素值。

(b) 隐藏层：包含 100 个神经元，使用 ReLU 激活函数。

(c) 输出层：包含 10 个神经元，对应 10 个类别，使用 Softmax 激活函数。

4. 定义前向传播与反向传播:

(a) 前向传播：计算输入数据通过网络后的输出。

(b) 反向传播：根据预测结果和实际标签计算损失，并更新网络权重。

5. 模型训练:

(a) 使用小批量梯度下降优化网络权重。

(b) 在每个 epoch 结束后，计算并记录训练损失、验证损失和验证准确率。

6. 模型评估:

(a) 在测试集上评估模型性能，计算并输出测试集准确率。

(b) 绘制训练过程中损失和准确率的变化曲线。

7. 混淆矩阵:

(a) 计算混淆矩阵，分析分类结果的具体表现。

(b) 绘制混淆矩阵，直观展示模型在各个类别上的分类效果。

8. 预测结果展示

(a) 展示模型在测试集部分图片上的分类结果。

本实验的评估指标和超参数如表 2 所示。

表 2: 实验评估指标和超参数

参数	值
learning rate	0.01
epochs	100
loss function	Cross Entropy
performance	accuracy
batch size	64
num hiddens	128

3.4.3 数学模型

BP 神经网络的数学模型如图 5 所示。

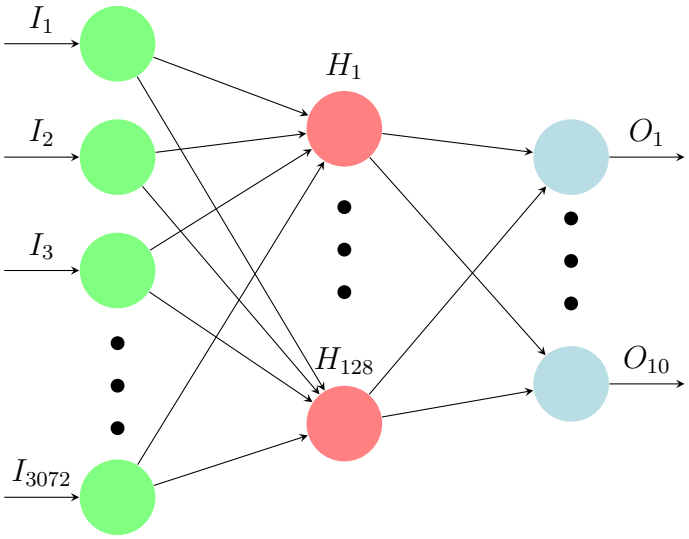


图 5: 全连接神经网络结构示意图

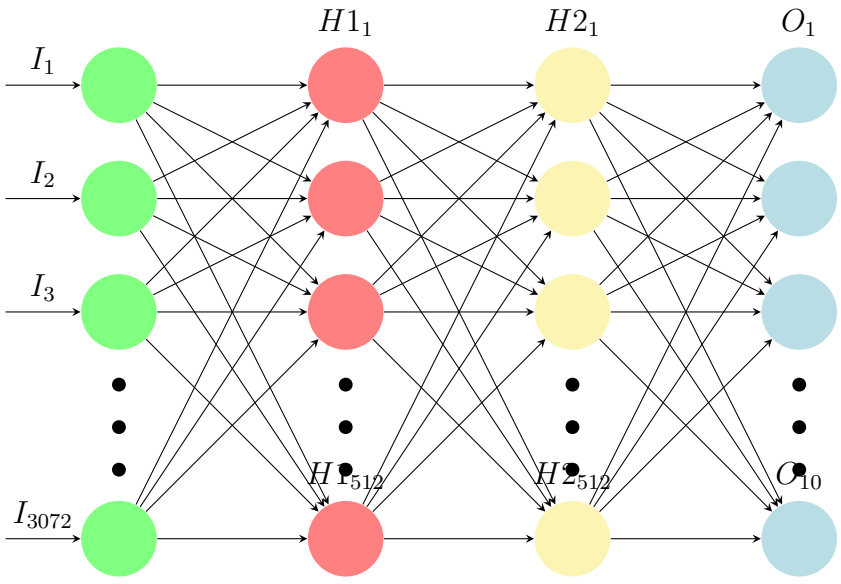


图 6: 基于 MindSpore 平台实现的多隐藏层神经网络结构示意图

3.4.4 关键难点

在训练过程中实时评估模型的性能，并根据评估结果调整模型的超参数，如学习率、批次大小、隐藏层神经元数量等。需要平衡模型的复杂度和泛化能力，避免过拟合或欠拟合。

3.4.5 算法描述

使用 BP 神经网络训练算法，如算法 4 所示。

3.5 实验结果与分析

3.5.1 实验结果展示

1. 准确率曲线本实验训练集和验证集上的准确率随 epoch 的变化曲线如图 7 所示。从图中可以看出，随着 epoch 的增加，验证集的准确率逐渐提高，最终收敛到 **50%**。
2. 损失曲线本实验验证集上的交叉熵损失随 epoch 的变化曲线如图 8 所示。从图中可以看出，随着 epoch 的增加，训练接和验证集的交叉熵损失逐渐降低，最终收敛到一个稳定值。其中训练集交叉熵损失稳定在 **1.0** 左右，验证集交叉熵损失稳定在 **1.4** 左右。
3. 混淆矩阵本实验的混淆矩阵如图 9 所示。从图中可以看出，模型在 CIFAR-10 数据集上的分类效果较好，大部分类别的分类准确率较高。

3.5.2 进一步探究

虽然上述基于全连接神经网络的分类器在 CIFAR-10 数据集上取得了一定的分类效果，但是其准确率仍然较低，仅为 50% 左右。结合机器学习课程所学知识，可以尝试以下方法进一步提高分类器的性能：

- 尝试使用更复杂的神经网络结构，如 CNN、RNN、Transformers 等
- 尝试使用更高级的优化算法，如 Adam、RMSprop 等

Algorithm 4 全连接神经网络训练算法

初始化网络的权重 **W1**, **W2** 和偏置 **b1**, **b2** 为随机值
procedure 训练 (X_train, Y_train, X_val, Y_val, batch_size, learning_rate, epochs)
    for 每个 epoch **do**
        for 每个 mini-batch **do**
            前向传播:
                设输入 **X**
                计算第一层的输出和激活值:
                $\mathbf{Z1} = \mathbf{XW1} + \mathbf{b1}$
                $\mathbf{A1} = \max(0, \mathbf{Z1})$ ▷ ReLU 激活函数
                计算第二层的输出和激活值:
                $\mathbf{Z2} = \mathbf{A1W2} + \mathbf{b2}$
                $\mathbf{A2} = \text{softmax}(\mathbf{Z2})$
            计算损失:
                使用交叉熵损失函数计算损失 L
            反向传播:
                计算输出层的误差 $\delta^{(2)}$
                计算隐藏层的误差 $\delta^{(1)}$
                计算梯度:
                $\nabla_{\mathbf{W2}} = \mathbf{A1}^T \delta^{(2)}$
                $\nabla_{\mathbf{b2}} = \sum \delta^{(2)}$
                $\nabla_{\mathbf{W1}} = \mathbf{X}^T \delta^{(1)}$
                $\nabla_{\mathbf{b1}} = \sum \delta^{(1)}$
            更新权重和偏置:
                使用学习率 η 更新 **W1**, **b1**, **W2**, **b2**
        end for
        计算训练集和验证集的损失和准确率
    end for
end procedure

- 尝试使用更复杂的数据增强技术，如旋转、平移、缩放等
- 尝试使用更复杂的模型评估指标，如 F1-score、ROC 曲线等
- 尝试使用更复杂的模型融合技术，如集成学习、模型融合等
- 尝试使用更复杂的超参数调优技术，如网格搜索、贝叶斯优化等
- 尝试使用更复杂的模型解释技术，如 LIME、SHAP 等

3.5.3 手动实现算法的评价

本实验实现了单隐藏层前馈神经网络对 CIFAR-10 数据集进行分类，取得了 50% 左右的准确率。与 mindspore 平台相比，手动实现的算法在准确率和收敛速度上均有所不足，可能的原因如下：

1. 模型复杂度：手动实现的算法只使用了单隐藏层的前馈神经网络，模型复杂度较低，难以捕捉数据集的复杂特征。
2. 优化算法：手动实现的算法使用了简单的小批量梯度下降优化算法，收敛速度较慢，难以达到较高的准确率。而 mindspore 中泽用的是 SGD、Adam 等高级的优化算法。
3. 超参数调优：手动实现的算法中的超参数（学习率、批次大小、隐藏层神经元数量等）未经过充分调优，可能导致模型性能不佳。

3.6 MindSpore 学习使用心得体会

3.7 代码附录

3.7.1 cifar-10-scratch.ipynb

```
import pickle
import numpy as np
import os
```

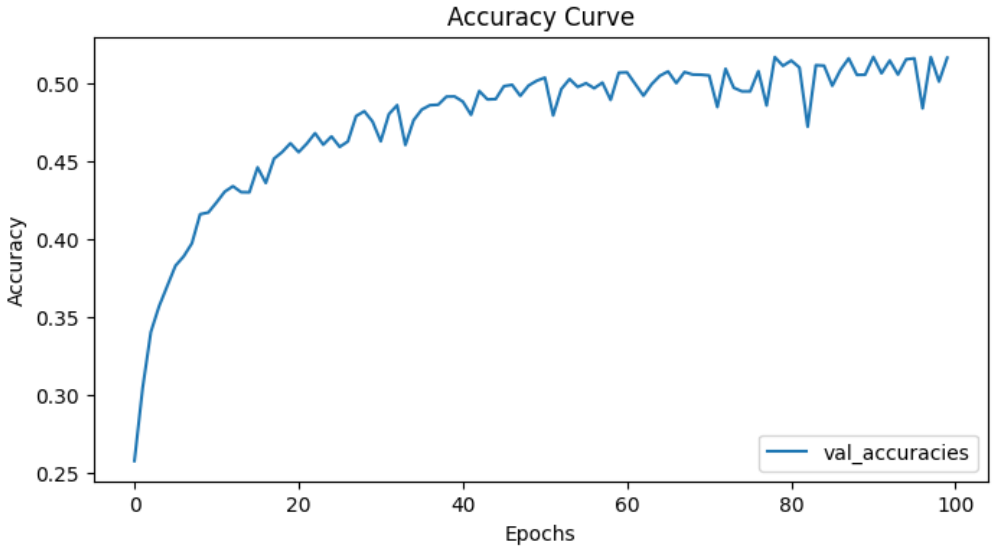


图 7: Accuracy 随 epoch 的变化曲线

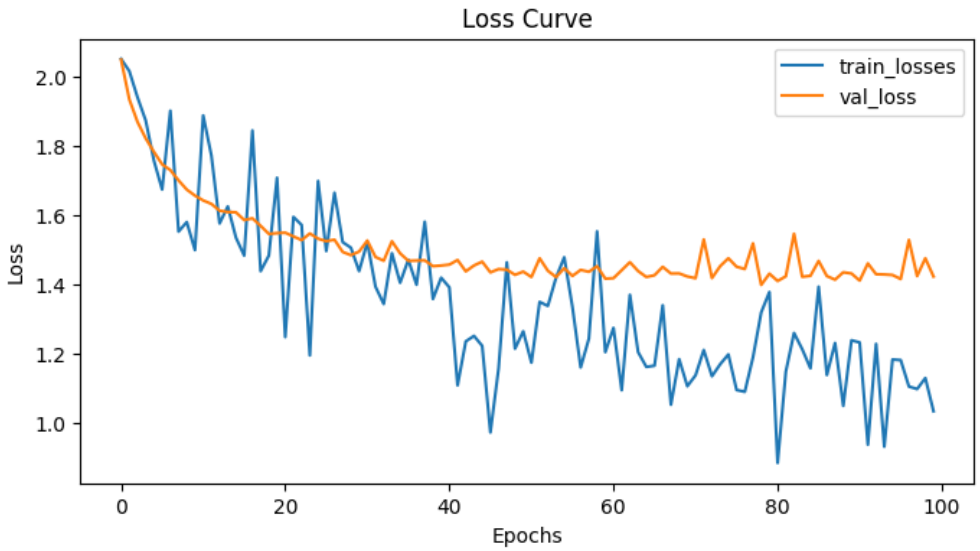


图 8: Loss 随 epoch 的变化曲线

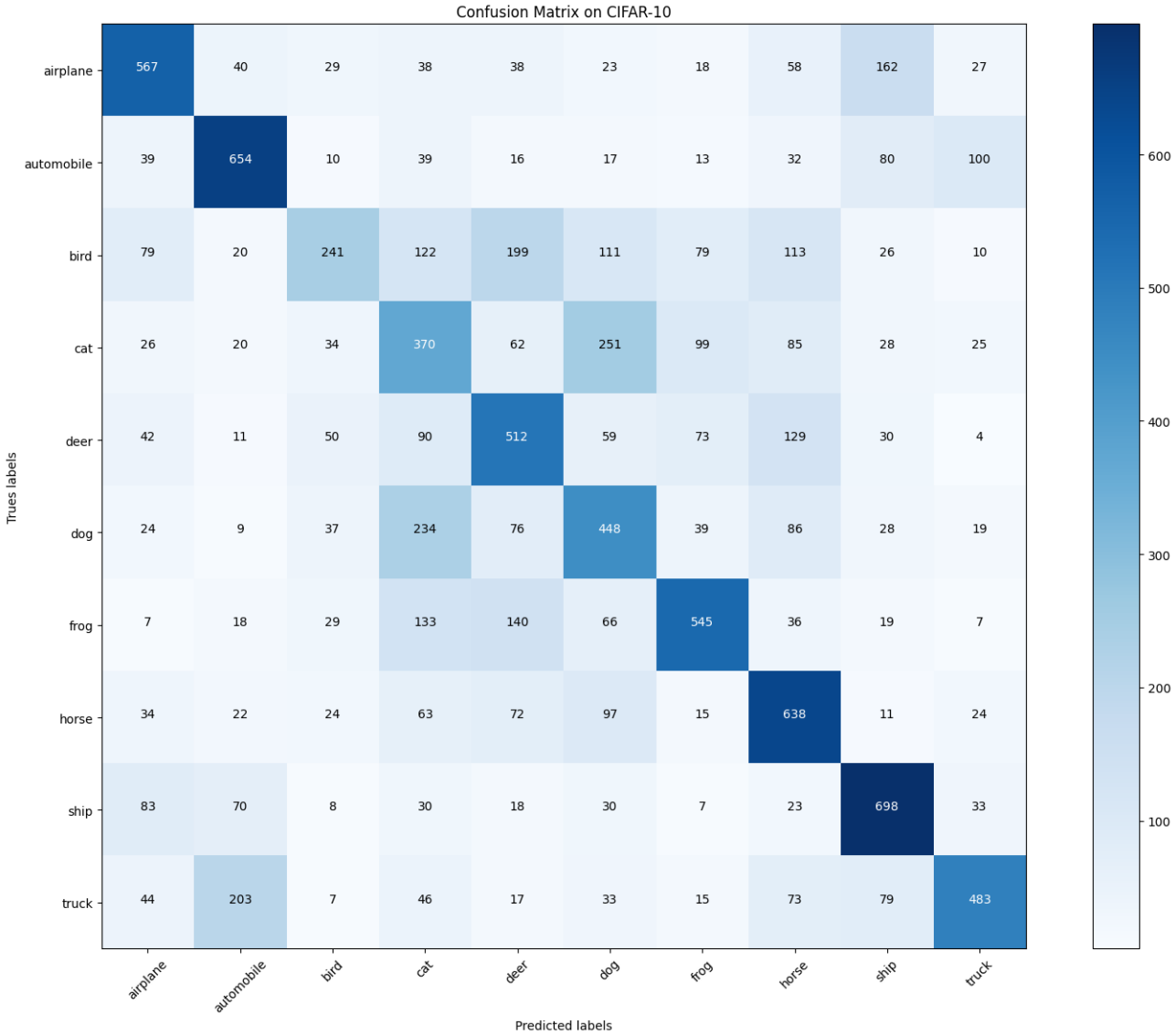


图 9: Confusion Matrix on CIFAR-10

```python
import matplotlib.pyplot as plt

# 加载 CIFAR-10 批次数据
def load_cifar10_batch(filename):
    with open(filename, 'rb') as f:
        datadict = pickle.load(f, encoding='bytes')
        X = datadict[b'data']
        Y = datadict[b'labels']
        X = X.reshape(10000, 3, 32, 32).astype("float")
        Y = np.array(Y)
        return X, Y

# 加载所有 CIFAR-10 数据
def load_cifar10(ROOT):
    xs = []
    ys = []
    for b in range(1, 6):
        f = os.path.join(ROOT, 'data_batch_%d' % (b,))
        X, Y = load_cifar10_batch(f)
        xs.append(X)
        ys.append(Y)
    Xtr = np.concatenate(xs)
    Ytr = np.concatenate(ys)
    del X, Y
    Xte, Yte = load_cifar10_batch(os.path.join(ROOT, 'test_batch'))
    return Xtr, Ytr, Xte, Yte

ROOT = './cifar-10-batches-py'
X_train, y_train, X_test, y_test = load_cifar10(ROOT)

# 全连接神经网络类
class FullyConnectedNN:
    def __init__(self, input_size, hidden_size, output_size):
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))

    def relu(self, Z):
        return np.maximum(0, Z)

```

```
    def softmax(self, Z):
        expZ = np.exp(Z - np.max(Z))
        return expZ / expZ.sum(axis=1, keepdims=True)

    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = self.relu(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = self.softmax(self.Z2)
        return self.A2

    def compute_loss(self, Y, Y_hat):
        m = Y.shape[0]
        log_likelihood = -np.log(Y_hat[range(m), Y])
        loss = np.sum(log_likelihood) / m
        return loss

    def backward(self, X, Y, Y_hat):
        m = X.shape[0]
        dZ2 = Y_hat
        dZ2[range(m), Y] -= 1
        dZ2 /= m

        dW2 = np.dot(self.A1.T, dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        dA1 = np.dot(dZ2, self.W2.T)
        dZ1 = dA1 * (self.Z1 > 0)

        dW1 = np.dot(X.T, dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2

    def compute_accuracy(self, X, Y):
        Y_hat = self.forward(X)
        predictions = np.argmax(Y_hat, axis=1)
        accuracy = np.mean(predictions == Y)
```

```
        return accuracy

    def train(self, X_train, Y_train, X_val, Y_val, epochs=300, learning_rate=0.01):
        self.learning_rate = learning_rate
        train_losses = []
        val_losses = []
        val_accuracies = []

        for epoch in range(epochs):
            # 打乱训练数据
            indices = np.arange(X_train.shape[0])
            np.random.shuffle(indices)
            X_train = X_train[indices]
            Y_train = Y_train[indices]

            # 小批量梯度下降
            for start_idx in range(0, X_train.shape[0], batch_size):
                end_idx = min(start_idx + batch_size, X_train.shape[0])
                X_batch = X_train[start_idx:end_idx]
                Y_batch = Y_train[start_idx:end_idx]

                # 前向传播
                Y_hat_train = self.forward(X_batch)
                train_loss = self.compute_loss(Y_batch, Y_hat_train)

                # 反向传播
                self.backward(X_batch, Y_batch, Y_hat_train)

            # 每个 epoch 结束后计算验证集上的损失和准确率
            Y_hat_val = self.forward(X_val)
            val_loss = self.compute_loss(Y_val, Y_hat_val)
            val_accuracy = self.compute_accuracy(X_val, Y_val)

            # 存储损失和准确率
            train_losses.append(train_loss)
            val_losses.append(val_loss)
            val_accuracies.append(val_accuracy)

            # 打印每个 epoch 的损失和准确率
```

```
            print(f'Epoch [{epoch + 1}], train_loss: {train_loss:.4f}, val_loss: {val_loss:.4f}, val_acc: {val_accuracy:.4f}')

        return train_losses, val_losses, val_accuracies

# 预处理数据
X_train = X_train.reshape(X_train.shape[0], -1) / 255.0
X_test = X_test.reshape(X_test.shape[0], -1) / 255.0

# 将训练数据分成训练集和验证集
split_index = int(0.8 * X_train.shape[0])
X_val, y_val = X_train[split_index:], y_train[split_index:]
X_train, y_train = X_train[:split_index], y_train[:split_index]

# 超参数
input_size = 3072  # 32*32*3
hidden_size = 100
output_size = 10
learning_rate = 0.01
epochs = 100
batch_size = 64

# 初始化并训练模型
model = FullyConnectedNN(input_size, hidden_size, output_size)
train_losses, val_losses, val_accuracies = model.train(X_train, y_train, X_val, y_val, epochs, learning_rate)

# 在测试集上进行预测
y_pred = np.argmax(model.forward(X_test), axis=1)
accuracy = np.mean(y_pred == y_test)
print(f'测试集准确率: {accuracy:.4f}')

# 绘制损失曲线
epochs_range = range(epochs)
plt.figure(figsize=(8, 4))
plt.plot(epochs_range, train_losses, label='训练损失')
plt.plot(epochs_range, val_losses, label='验证损失')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.title('损失曲线')
```

```
plt.show()

# 绘制准确率曲线
plt.figure(figsize=(8, 4))
plt.plot(epochs_range, val_accuracies, label='验证准确率')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.title('准确率曲线')
plt.show()


# 计算混淆矩阵
def compute_confusion_matrix(y_true, y_pred, num_classes):
    cm = np.zeros((num_classes, num_classes), dtype=int)
    for i in range(len(y_true)):
        cm[y_true[i], y_pred[i]] += 1
    return cm

# 绘制混淆矩阵
def plot_confusion_matrix(cm, classes, title='混淆矩阵', cmap=plt.cm.Blues):
    plt.figure(figsize=(16, 12))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = 'd'
    thresh = cm.max() / 2.
    for i, j in np.ndindex(cm.shape):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('真实标签')
    plt.xlabel('预测标签')
    plt.tight_layout()
    plt.show()

```

```
# CIFAR-10 标签
cifar10_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', '
 frog', 'horse', 'ship', 'truck']

# 计算并绘制混淆矩阵
cm = compute_confusion_matrix(y_test, y_pred, len(cifar10_labels))
plot_confusion_matrix(cm, classes=cifar10_labels)
```

4 心得体会

在使用华为 mindspore 平台时，我觉得数据集的加载相比于手动实现的方式更加方便，同时 mindspore 平台提供了丰富的 API 接口，可以方便地实现神经网络的训练和评估。在本次实验中，我实现了一个基于全连接神经网络的分类器，并在 CIFAR-10 数据集上进行了训绋和评估。通过本次实验，我对神经网络的训练过程有了更深入的理解，同时也学习了如何使用 mindspore 平台实现神经网络模型。希望在以后的实验中，我能够更加熟练地使用 mindspore 平台，实现更加复杂的神经网络模型。但是相比于 PyTorch, Tensorflow 等主流深度学习算法框架，mindspore 仍有部分数据集的加载仍然比较复杂，希望有关开发人员能够进一步优化 API 接口，提高 mindspore 的易用性。

4.1 关于上课的体会

4.2 关于实验的体会

4.3 总的体会