模式识别实验报告

专业：	人工智能
学号：	58122310
年级：	大二
姓名：	唐梓烨

签名：唐梓烨

时间：2024.6.7

实验一 数据降维与分类实验

1. 问题描述

(1) 目标：利用两种降维技术对葡萄酒数据进行降维，并对降维前后的数据进行分类。

(2) 数据集描述：所给数据集包含两个.csv 文件，分别为红葡萄酒数据（包括 1599 个样本）和白葡萄酒数据（包含 4898 个样本），每个样本含有 11 个特征（文件的前 11 列）：固定酸度、挥发酸度、柠檬酸、残糖、氯化物、游离二氧化硫、总二氧化硫、密度、pH 值、硫酸盐、酒精和专家对此葡萄酒的打分。

2. 实现步骤与流程

(1) 实验思路：

在 utils.py 中定义 PCA 和 LDA 降维函数，它们能将数据降维到指定维度。其中 PCA 是无监督降维，它根据数据的协方差矩阵的特征值和特征向量来选择主要成分。LDA 则是监督降维，通过最大化类间距离和最小化类内距离来确定最佳的投影方向。除了定义降维函数，还定义了 logistic 回归分类器，利用反向传播来训练用于分类的回归分类器。由于不能使用 pytorch 等高级机器学习包，我还定义了一些辅助函数，例如随机分割训练集和测试集的函数，模型训练过程中用到的交叉熵损失函数、softmax 函数、独热编码函数等。在 train.py 中使用在 utils.py 中定义的 PCA、LDA 函数对训练集进行降维，再将降维后的数据输入 logistic 回归分类器进行训练，利用测试集对模型进行评估。

(2) 实验流程：

i. 准备数据：从.csv 文件中加载红葡萄酒和白葡萄酒数据。每个数据集包括 1599 个红葡萄酒样本和 4898 个白葡萄酒样本，每个样本含有 11 个特征。

ii. 数据预处理：标准化数据集，确保 PCA 和 LDA 处理的有效性。

iii. 实现降维：使用定义好的 PCA 和 LDA 函数对数据集进行降维。选择将特征降维到 2 维，以便于后续的可视化和分析。

iv. 分类与评估：在降维后的数据上使用逻辑回归进行分类，并计算分类准确率。比较降维前后的准确率变化，通过可视化展示这些变化。

(3) 实验要点/难点：

i. 对于二分类问题的逻辑回归问题，使用的是线性回归的方法来计算输入特征的加权和，然后通过一个非线性函数，“sigmoid”函数将这个结果转换为概率值输出。在参数优化上可以使用交叉熵损失作为损失函数，使用反向传播算法来优化参数。

ii. 实验的数据集并没有标签信息，因此在读取时要在数据集中加入标签信息。由于白葡萄酒和红葡萄酒是分开在两个文件中的，我在红葡萄酒数据集中新增了‘label’列为 0，在白葡萄酒数据集中新增了‘label’列为 1。由于使用了交叉熵损失，标签数据还需要经过独热编码。

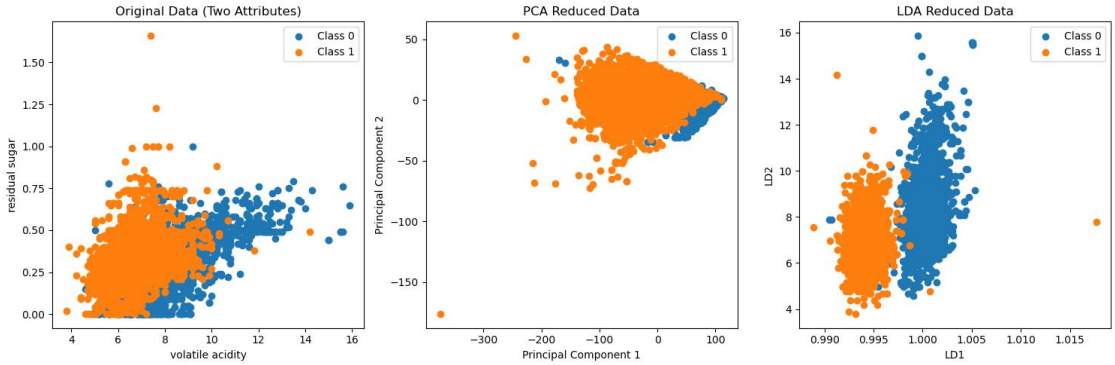
iii. 为了展示原始数据和降维后的数据分布，我将降维后的维度设置为了二维以便可视化。在实际 PCA 降维时，还可以采用主成分能保留原始数据 95%的

方差为标准。

iv. 由于 PCA 降维和 LDA 降维在实现形式上的多样性，比如是否对原始数据进行标准化、选择的投影方向可以反向等，实际结果和一些机器学习库自带的降维方法结果不同，这也是正常的情况。

3. 实验结果与分析

降维后的数据可视化：

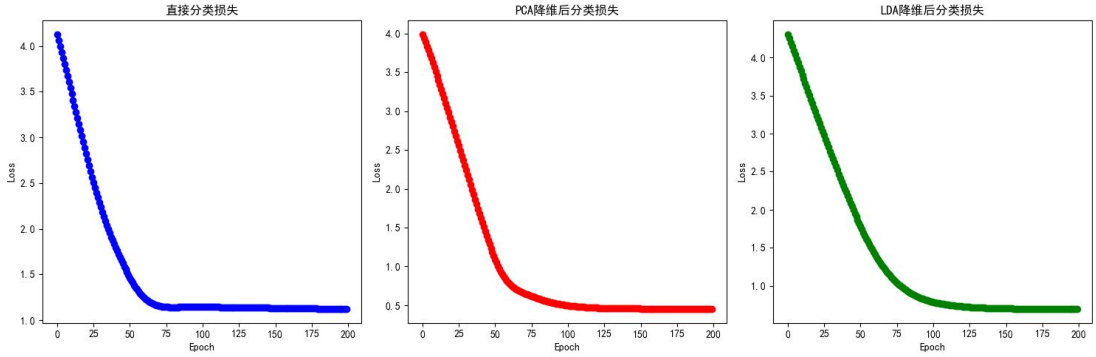


结果分析：

PCA 降维后数据在两个主成分上分布更加集中，这符合 PCA 的主成分的定义，这两个主成分可以很好的代表数据的特征。

LDA 降维后数据很明显的被分隔开了，这符合 LDA 的定义，它的目标函数就是使得类间尽可能的远同时类内尽可能的近。

降维后的数据对训练损失的影响：



结果分析：

在相同参数 lr=0.001 情况下，仅改变输入数据对训练时的分类损失的走势影响不大，训练损失在 0~75 轮次迅速减小，在 100 轮次后趋近收敛。

降维后的分类损失收敛时会明显比未降维的数据更低，这说明降低了数据的维度后，模型更容易进行拟合，更容易达到较低的分类损失

降维后的数据对分类准确率的影响：

```
直接分类准确率：0.939183987682833，直接分类训练时间：7.074042797088623
PCA降维后的分类准确率：0.9214780600461894，PCA降维后训练时间：6.7040791511535645
LDA降维后的分类准确率：0.7829099307159353，LDA降维后训练时间：6.310725927352905
```

由于在增加训练轮次时，模型准确率在不断提高，在提高到 10000 轮时还未

出现过拟合线性，因此我在此展示的是经过了 10000 轮次的训练后的分类准确率。并且在 10000 轮次的条件下，可以看到降维后的数据在训练时时间会比原始数据短一些，说明降维后的数据在计算代价上需求更小，但由于模型的计算大头在于梯度的计算，模型的参数量其实相同，所以数据维度对训练时间的影响其实微乎其微。

4. MindSpore 学习使用心得体会

在这里我会介绍我是如何使用 mindspore 库实现上面相同任务的，重点介绍一些不同点。 我的 mindspore 代码在 utils_ms.py, train_ms.py 文件中，文件中实现的函数与 utils.py, train.py 基本相同。

要用 mindspore 实现上面相同的任务，在这个实验中主要是利用 mindspore.numpy 来代替 numpy。我的大致思路就是将原先用到 numpy 的地方改为 mindspore.numpy，但由于 mindspore.numpy 提供的 api 相比 numpy 少了很多，使用起来并不方便。

不方便处举例：

mindspore.numpy 没有实现特征值计算和奇异值分解的 api，这在降维中是很重要的步骤。

mindspore.numpy 的基本数据类型为 mindspore.Tensor 而不是 array，这导致很多 ndarray 能进行的操作在使用 mindspore.numpy 后需要更改实现方法。

Mindspore 使用心得：

初学者要尝试使用 mindspore.numpy 可以在先写 numpy 代码的前提下，将代码中的 ndarray 数据类型通过 Tensor() 方法转换为 Tensor，在遇到 mindspore.numpy 没有提供的 api 时可以将原有代码中使用的 api 的操作转为手动实现，这增加了代码量和代码复杂度，但属于无奈之举。还可以多尝试使用 Tensor 数据类型的.asnumpy()方法，这个方法可以将很好的适配一些 ndarray 的操作。

5. 代码附录

Utils.py:

```
import numpy as np

class LogisticRegression:
    def __init__(self, lr=0.01, epochs=10):
        self.lr = lr
        self.epochs = epochs
        self.weight = None
        self.history = []

    def init_weights(self, num_features, num_classes):
        self.weight = np.random.randn(num_features, num_classes)
```

```
    def forward(self, X):
        return X @ self.weight

    def backward(self, X, y_true, y_preds):
        num_samples = X.shape[0]
        grad = X.T @ (y_preds - y_true) / num_samples
        self.weight -= self.lr * grad

    # 训练步骤代码，包含前向传播和反向传播
    def train(self, X, y):
        X = np.c_[X, np.ones((X.shape[0], 1))]
        num_samples, num_features = X.shape
        num_classes = len(np.unique(y))
        self.init_weights(num_features, num_classes) # 初始化权重

        y_one_hot = one_hot_encode(y, num_classes)

        for _ in range(self.epochs):
            model = self.forward(X)
            y_preds = softmax(model)
            loss = cross_entropy_loss(y_one_hot, y_preds)
            self.history.append(loss) # 保存训练损失
            self.backward(X, y_one_hot, y_preds)

    # 预测步骤代码，用于评估模型，计算准确度
    def predict(self, X):
        X = np.c_[X, np.ones((X.shape[0], 1))]
        model = self.forward(X)
        y_preds = softmax(model)
        return np.argmax(y_preds, axis=1)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))
    return exp_z / exp_z.sum(axis=1, keepdims=True)

def cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    loss = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
    return loss

def one_hot_encode(y, num_classes = 2):
    return np.eye(num_classes)[y]
```

```
    def PCA(X, n_components):
        # 数据标准化
        X_meaned = X - np.mean(X, axis=0)
        # 计算协方差矩阵
        cov_matrix = np.cov(X_meaned, rowvar=False)
        # 特征值分解
        eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)
        # 特征向量排序
        sorted_index = np.argsort(eigen_values)[::-1]
        sorted_eigenvectors = eigen_vectors[:, sorted_index]
        # 选择前 n 个特征向量（主成分）
        eigenvector_subset = sorted_eigenvectors[:, :n_components]
        # 投影数据
        X_pca = np.dot(X_meaned, eigenvector_subset)
        return X_pca

    def LDA(X, y, n_components):
        # 计算类别相同的葡萄酒的均值向量
        class_labels = np.unique(y)
        mean_vectors = []
        for label in class_labels:
            mean_vectors.append(np.mean(X[y == label], axis=0))
        # 计算类内散布矩阵 S_W
        S_W = np.zeros((X.shape[1], X.shape[1]))
        for label, mean_vec in zip(class_labels, mean_vectors):
            class_scatter = np.zeros((X.shape[1], X.shape[1]))
            for row in X[y == label]:
                row, mean_vec = row.reshape(X.shape[1], 1), mean_vec.reshape(X.shape[1],
1)
                class_scatter += (row - mean_vec).dot((row - mean_vec).T)
            S_W += class_scatter
        # 计算类间散布矩阵 S_B
        μ1 = mean_vectors[0].reshape(X.shape[1], 1)
        μ2 = mean_vectors[1].reshape(X.shape[1], 1)
        S_B = np.dot((μ1-μ2),(μ1-μ2).T)
        # 使用 SVD 计算特征值
        inv_S_W = np.linalg.pinv(S_W)
        mat = inv_S_W.dot(S_B)
        U, s, Vh = np.linalg.svd(mat)
        # 选择前 n 个特征向量
        W = U[:, :n_components]
        # 投影数据
        X_lda = np.dot(X, W)
        return X_lda
```

```
def train_test_split(X, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]
    indices = np.arange(n_samples)
    # 通过打乱索引来实现随机抽取
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_size = n_samples - test_size

    X_train = X[indices[0:train_size]]
    X_test = X[indices[train_size:]]
    y_train = y[indices[0:train_size]]
    y_test = y[indices[train_size:]]

    return X_train, X_test, y_train, y_test

def Accuracy(y_true, y_pred):
    correct_pred = sum(y_t == y_p for y_t, y_p in zip(y_true, y_pred))
    total_pred = len(y_true)
    accuracy = correct_pred / total_pred
    return accuracy
```

Train.py

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
from utils import *
import time

# 读取 CSV 文件
df_red = pd.read_csv(r'data/winequality-red.csv', sep=';')
df_white = pd.read_csv(r'data/winequality-white.csv', sep=';')

df_red['label'] = 0
df_white['label'] = 1

df = pd.concat([df_red, df_white], axis=0, ignore_index=True)
```

```
# 提取特征和标签
X = df.drop('label', axis=1).values
y = df['label'].values

X_pca = PCA(X,n_components=2)
X_lda = LDA(X,y,n_components=2)

# 可视化原始数据（指定两个属性）
feature1 = 0 # "volatile acidity"
feature2 = 2 # "residual sugar"
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
for label in np.unique(y):
    plt.scatter(X[y == label, feature1], X[y == label, feature2], label=f'Class {label}')
plt.title('Original Data (Two Attributes)')
plt.xlabel(f'volatile acidity')
plt.ylabel(f'residual sugar')
plt.legend()

# 可视化 PCA 降维到二维的数据
plt.subplot(1, 3, 2)
for label in np.unique(y):
    plt.scatter(X_pca[y == label, 0], X_pca[y == label, 1], label=f'Class {label}')
plt.title('PCA Reduced Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()

# 可视化 LDA 降维到二维的数据
plt.subplot(1, 3, 3)
for label in np.unique(y):
    plt.scatter(X_lda[y == label, 0], X_lda[y == label, 1], label=f'Class {label}')
plt.title('LDA Reduced Data')
plt.xlabel('LD1')
plt.ylabel('LD2')
plt.legend()

plt.tight_layout()
plt.show()

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y, test_size=0.2,
random_state=42)
```

```
X_train_lda, X_test_lda, y_train_lda, y_test_lda = train_test_split(X_lda, y, test_size=0.2,
random_state=42)

# 训练原始数据的分类器
start_time1 = time.time()
classifier = LogisticRegression(lr=0.001,epochs=200)
classifier.train(X_train, y_train)
end_time1 = time.time()
train_time1 = end_time1 - start_time1
y_pred = classifier.predict(X_test)
accuracy = Accuracy(y_test, y_pred)
loss1 = classifier.history

# 训练 PCA 降维数据的分类器
start_time2 = time.time()
classifier_pca = LogisticRegression(lr=0.001,epochs=200)
classifier_pca.train(X_train_pca, y_train_pca)
end_time2 = time.time()
train_time2 = end_time2 - start_time2
y_pred_pca = classifier_pca.predict(X_test_pca)
accuracy_pca = Accuracy(y_test_pca, y_pred_pca)
loss2 = classifier_pca.history

# 训练 LDA 降维数据的分类器
start_time3 = time.time()
classifier_lda = LogisticRegression(lr=0.001,epochs=200)
classifier_lda.train(X_train_lda, y_train_lda)
end_time3 = time.time()
train_time3 = end_time3 - start_time3
y_pred_lda = classifier_lda.predict(X_test_lda)
accuracy_lda = Accuracy(y_test_lda, y_pred_lda)
loss3 = classifier_lda.history

# 绘制损失变化图
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
rcParams['font.sans-serif'] = ['SimHei']

axs[0].plot(loss1, marker='o', linestyle='-', color='blue')
axs[0].set_title('直接分类损失')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Loss')

axs[1].plot(loss2, marker='o', linestyle='-', color='red')
axs[1].set_title('PCA 降维后分类损失')
```

```
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Loss')

axs[2].plot(loss3, marker='o', linestyle='-', color='green')
axs[2].set_title('LDA 降维后分类损失')
axs[2].set_xlabel('Epoch')
axs[2].set_ylabel('Loss')

plt.tight_layout()

plt.show()

print(f'直接分类准确率: {accuracy}，直接分类训练时间: {train_time1}')
print(f'PCA 降维后的分类准确率: {accuracy_pca}, PCA 降维后训练时间: {train_time2}')
print(f'LDA 降维后的分类准确率: {accuracy_lda}, LDA 降维后训练时间: {train_time3}')
```

Utlis_ms.py

```
import mindspore.numpy as mnp
import numpy as np
from mindspore import Tensor, context
from mindspore.common import dtype as mstype
from mindspore.ops import operations as P
from mindspore.ops import functional as F

context.set_context(mode=context.PYNATIVE_MODE)

class LogisticRegression:
    def __init__(self, lr=0.01, epochs=10):
        self.lr = lr
        self.epochs = epochs
        self.weight = None
        self.history = []

    def init_weights(self, num_features, num_classes):
        self.weight = np.random.randn(num_features, num_classes)
        self.weight = Tensor(self.weight,dtype=mstype.float32)

    def forward(self, X):
        return X @ self.weight

    def backward(self, X, y_true, y_preds):
        num_samples = X.shape[0]
```

```
            grad = X.T @ (y_preds - y_true) / num_samples
            self.weight -= self.lr * grad

    # 训练步骤代码，包含前向传播和反向传播
    def train(self, X, y):
        X = mnp.column_stack([X, mnp.ones((X.shape[0], 1))])
        num_samples, num_features = X.shape
        num_classes = len(mnp.unique(y))
        self.init_weights(num_features, num_classes) # 初始化权重

        y_one_hot = one_hot_encode(y, num_classes)

        for _ in range(self.epochs):
            model = self.forward(X)
            y_preds = softmax(model)
            loss = cross_entropy_loss(y_one_hot, y_preds)
            self.history.append(loss) # 保存训练损失
            self.backward(X, y_one_hot, y_preds)

    # 预测步骤代码，用于评估模型，计算准确度
    def predict(self, X):
        X = mnp.column_stack([X, mnp.ones((X.shape[0], 1))])
        model = self.forward(X)
        y_preds = softmax(model)
        return mnp.argmax(y_preds, axis=1)

def softmax(z):
    exp_z = mnp.exp(z - mnp.max(z, axis=1, keepdims=True))
    return exp_z / mnp.sum(exp_z, axis=1, keepdims=True)

def cross_entropy_loss(y_true, y_pred):
    epsilon = 1e-15
    y_pred = mnp.clip(y_pred, epsilon, 1 - epsilon)
    loss = -mnp.sum(y_true * mnp.log(y_pred)) / y_true.shape[0]
    return loss

def one_hot_encode(y, num_classes=2):
    return mnp.eye(num_classes)[Tensor(y, dtype=mnp.int32)]

def PCA(X, n_components):
    # 数据标准化
    X_meaned = X - np.mean(X, axis=0)
    # 计算协方差矩阵
    cov_matrix = np.cov(X_meaned, rowvar=False)
```

```
    # 特征值分解
    eigen_values, eigen_vectors = np.linalg.eigh(cov_matrix)
    # 特征向量排序
    sorted_index = np.argsort(eigen_values)[::-1]
    sorted_eigenvectors = eigen_vectors[:, sorted_index]
    # 选择前 n 个特征向量（主成分）
    eigenvector_subset = sorted_eigenvectors[:, :n_components]
    # 投影数据
    X_pca = np.dot(X_meaned, eigenvector_subset)
    return X_pca

def LDA(X, y, n_components):
    # 计算类别相同的葡萄酒的均值向量
    class_labels = np.unique(y)
    mean_vectors = []
    for label in class_labels:
        mean_vectors.append(np.mean(X[y == label], axis=0))
    # 计算类内散布矩阵 S_W
    S_W = np.zeros((X.shape[1], X.shape[1]))
    for label, mean_vec in zip(class_labels, mean_vectors):
        class_scatter = np.zeros((X.shape[1], X.shape[1]))
        for row in X[y == label]:
            row, mean_vec = row.reshape(X.shape[1], 1), mean_vec.reshape(X.shape[1], 1)
            class_scatter += (row - mean_vec).dot((row - mean_vec).T)
        S_W += class_scatter
    # 计算类间散布矩阵 S_B
    μ1 = mean_vectors[0].reshape(X.shape[1], 1)
    μ2 = mean_vectors[1].reshape(X.shape[1], 1)
    S_B = np.dot((μ1-μ2),(μ1-μ2).T)
    # 使用 SVD 计算特征值
    inv_S_W = np.linalg.pinv(S_W)
    mat = inv_S_W.dot(S_B)
    U, s, Vh = np.linalg.svd(mat)
    # 选择前 n 个特征向量
    W = U[:, :n_components]
    # 投影数据
    X_lda = np.dot(X, W)
    return X_lda

def train_test_split(X, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)

    n_samples = X.shape[0]
```

```
    # 通过打乱索引来实现随机抽取
    indices = np.arange(n_samples)
    np.random.shuffle(indices)

    test_size = int(n_samples * test_size)
    train_size = n_samples - test_size

    X_train = X[indices[0:train_size]]
    X_test = X[indices[train_size:]]
    y_train = y[indices[0:train_size]]
    y_test = y[indices[train_size:]]

    return X_train, X_test, y_train, y_test

def Accuracy(y_true, y_pred):
    # 将预测结果转化为同一形式
    if y_pred.shape != y_true.shape:
        y_pred = P.Argmax(axis=1)(y_pred)
    # 计算正确预测的数量
    correct_pred = F.reduce_sum(F.cast(y_true == y_pred, mnp.float32))
    # 总预测数量
    total_pred = y_true.shape[0]
    # 计算精确度
    accuracy = correct_pred / total_pred
    return accuracy
```

Train_ms.py

```
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib import rcParams
import mindspore.numpy as np
from mindspore import Tensor
from utils_ms import *

# 读取 CSV 文件
df_red = pd.read_csv(r'data/winequality-red.csv', sep=';')
df_white = pd.read_csv(r'data/winequality-white.csv', sep=';')

df_red['label'] = 0
df_white['label'] = 1

df = pd.concat([df_red, df_white], axis=0, ignore_index=True)
```

```
# 提取特征和标签
X = df.drop('label', axis=1).values
y = df['label'].values

X_pca = PCA(X,n_components=2)
X_lda = LDA(X,y,n_components=2)

# 可视化数据
feature1 = 0 # "volatile acidity"
feature2 = 2 # "residual sugar"
# 可视化原始数据（指定两个属性）
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
for label in np.unique(y):
    plt.scatter(X[y == label, feature1], X[y == label, feature2], label=f'Class {label}')
plt.title('Original Data (Two Attributes)')
plt.xlabel(f'volatile acidity')
plt.ylabel(f'residual sugar')
plt.legend()

# 可视化 PCA 降维到二维的数据
plt.subplot(1, 3, 2)
for label in np.unique(y):
    plt.scatter(X_pca[y == label, 0], X_pca[y == label, 1], label=f'Class {label}')
plt.title('PCA Reduced Data')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()

# 可视化 LDA 降维到二维的数据
plt.subplot(1, 3, 3)
for label in np.unique(y):
    plt.scatter(X_lda[y == label, 0], X_lda[y == label, 1], label=f'Class {label}')
plt.title('LDA Reduced Data')
plt.xlabel('LD1')
plt.ylabel('LD2')
plt.legend()

plt.tight_layout()
plt.show()

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca, y, test_size=0.2,
random_state=42)
X_train_lda, X_test_lda, y_train_lda, y_test_lda = train_test_split(X_lda, y, test_size=0.2,
random_state=42)
X_train = Tensor(X_train,dtype=mstype.float32)
X_test = Tensor(X_test,dtype=mstype.float32)
y_train = Tensor(y_train,dtype=mstype.float32)
X_train = Tensor(X_train,dtype=mstype.float32)
y_test = Tensor(y_test,dtype=mstype.float32)
X_train_pca = Tensor(X_train_pca,dtype=mstype.float32)
X_test_pca = Tensor(X_test_pca,dtype=mstype.float32)
y_train_pca = Tensor(y_train_pca,dtype=mstype.float32)
y_test_pca = Tensor(y_test_pca,dtype=mstype.float32)
X_train_lda = Tensor(X_train_lda,dtype=mstype.float32)
X_test_lda = Tensor(X_test_lda,dtype=mstype.float32)
y_train_lda = Tensor(y_train_lda,dtype=mstype.float32)
y_test_lda = Tensor(y_test_lda,dtype=mstype.float32)

classifier = LogisticRegression(lr=0.001,epochs=10000)
classifier.train(X_train, y_train)
y_pred = classifier.predict(X_test)
accuracy = Accuracy(y_test, y_pred)
loss1 = classifier.history

classifier_pca = LogisticRegression(lr=0.001,epochs=10000)
classifier_pca.train(X_train_pca, y_train_pca)
y_pred_pca = classifier_pca.predict(X_test_pca)
accuracy_pca = Accuracy(y_test_pca, y_pred_pca)
loss2 = classifier_pca.history

classifier_lda = LogisticRegression(lr=0.001,epochs=10000)
classifier_lda.train(X_train_lda, y_train_lda)
y_pred_lda = classifier_lda.predict(X_test_lda)
accuracy_lda = Accuracy(y_test_lda, y_pred_lda)
loss3 = classifier_lda.history

# 绘制损失变化图
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
rcParams['font.sans-serif'] = ['SimHei']

axs[0].plot(loss1, marker='o', linestyle='-', color='blue')
axs[0].set_title('直接分类损失')
axs[0].set_xlabel('Epoch')
axs[0].set_ylabel('Loss')
```

```
axs[1].plot(loss2, marker='o', linestyle='-', color='red')
axs[1].set_title('PCA 降维后分类损失')
axs[1].set_xlabel('Epoch')
axs[1].set_ylabel('Loss')

axs[2].plot(loss3, marker='o', linestyle='-', color='green')
axs[2].set_title('LDA 降维后分类损失')
axs[2].set_xlabel('Epoch')
axs[2].set_ylabel('Loss')

plt.tight_layout()

plt.show()

print(f'直接分类准确率: {accuracy}')
print(f'PCA 降维后的分类准确率: {accuracy_pca}')
print(f'LDA 降维后的分类准确率: {accuracy_lda}')
```

实验二 朴素贝叶斯分类实验

1. 问题描述

(1) 目标：利用朴素贝叶斯算法，对 MNIST 数据集中的测试集进行分类。

(2) 数据集描述：MNIST 数据集是一个广泛用于手写数字识别的数据集。它由美国高中生和美国人口调查局的员工手写的数字组成。MNIST 数据集包含了 70,000 张黑白图像，其中每张图像的大小为 28x28 像素。这些图像被分为两个部分：训练集包含 60,000 张图像，测试集包含 10,000 张图像。

每个图像都是一个手写的数字（从 0 到 9），并且每个图像都有一个与之对应的标签，指示图像中的数字。这些图像已经经过大小归一化并且位于中心位置，以便于进行机器学习模型的训练。

2. 实现步骤与流程

(1) 实验思路：

在 utils.py 中定义朴素贝叶斯分类器，包含其参数初始化，计算先验概率，计算类条件概率密度（pdf），利用计算的先验概率和类条件概率密度计算后验概率的函数。利用后验概率的值选择最大化后验概率的类即可作为预测值输出。除此之外我还在 utils.py 中封装了加载数据集的步骤。

在 train.py 中调用 utils.py 中的数据集加载函数，读取 MNIST 数据集，并且将图像数据从二进制格式转换为灰度值，每个像素点的值经过缩放和整数化处理后用于分类。然后调用朴素贝叶斯分类器类进行概率值的计算，将概率值计算好后存储在朴素贝叶斯分类器中，这样的朴素贝叶斯分类器就可以用来分类了。

(2) 实验步骤：

i. 加载 MNIST 数据集并进行数据预处理

ii. 基于贝叶斯定理，假设各特征之间相互独立，对特征和类进行概率建模

iii. 包括计算先验概率：对于每个类别C_k，计算其在训练集中出现的概率 $\mathrm{P}(C_k)$，即先验概率。计算条件概率：对于每个类别 C_k，计算给定类别下每个特征的条件概率 $\mathrm{P}(\mathrm{x_i}|C_k)$。由于特征是连续的，还需要假设其分布（正态分布）并估计相关参数（均值和方差）。应用朴素贝叶斯分类规则：对于一个新的样本，通过计算并比较每个类别的后验概率 $\mathrm{P}(C_k|x)$，选择具有最大后验概率的类别作为预测结果。后验概率可以通过贝叶斯定理计算：

$$\mathrm{P}(C_k|x) = \frac{\mathrm{P}(\mathrm{x}|C_k)\mathrm{P}(C_k)}{\mathrm{P}(\mathrm{x})}$$

iv. 使用建模好的后验概率值来进行预测。

(3) 实验要点/难点：

i. 由于图片有 28×28 个像素，也就是 784 个特征，其实在计算 $\mathrm{P}(\mathrm{x}|C_k)$时可以朴素的假设特征之间相互独立，得到，

$$P(x|C_k) = \prod_{i=1}^{n} P(x_i|C_k)$$

ii. 由于朴素贝叶斯分类器在是在学习中利用计算的概率值更新参数，所以它并不需要使用 BP 等优化算法来更新参数，在训练完训练集中全部数据后即可用来分类。

3. 实验结果与分析

MNIST 测试集分类准确度：

在使用 numpy 下的分类准确度和使用 mindspore.numpy 下的分类准确度相同，为 51.1%。

可视化部分分类结果：

可以看出朴素贝叶斯分类器在该数据集上表现中规中矩，在分类时会偏向错分为 9。这也是假设图片中的特征是相互独立的等朴素假设导致的模型过于简单，难以学习到图片特征中的深层规律，与朴素贝叶斯网络的特点相符。

4. MindSpore 学习使用心得体会

在这里我会介绍我是如何使用 mindspore 库实现上面相同任务的，重点介绍一些不同点。

我的 mindspore 代码在 utils_ms.py, train_ms.py 文件中，文件中实现的函数与 utils.py, train.py 基本相同。

在使用 mindspore 完成上述实验时，我同样感到有很多地方受限。

Mindspore 的基本数据类型是 Tensor，但这个数据类型在使用时对其内部的基本数据类型极其敏感，Tensor 内部的数值可以是 uint, int, float，只有内部数据类型相同时两个 Tensor 才能进行计算，并且它没有自动调整内部数据类型以适配要求的算法，需要手动调整，这导致使用起来比较繁杂。

由上面这个问题衍生出来的问题就是 Tensor 数据类型转换时的计算开销还很大，这导致代码中浪费了大量的时间在数据类型转换上，代码执行所需时间也明显增多。

在实验中还有一点就是 Tensor 中的数据要进行相等判断时常常出错，这可能也和其内部数据类型有关，这时候先将 Tensor 数据使用.asnumpy()方法进行转换再比较就能准确进行。

5. 代码附录

Utils.py

```
import numpy as np
import struct

class CustomGaussianNB:
    def __init__(self):
        # 初始化模型参数
        self.means = None
        self.variances = None
        self.priors = None

    def fit(self, X, y):
        n_samples, n_features = X.shape  # 获取样本数和特征数
        self.classes = np.unique(y)  # 获取所有类标签
        n_classes = len(self.classes)  # 类别数

        # 初始化均值、方差和先验概率矩阵
        self.means = np.zeros((n_classes, n_features), dtype=np.float64)
        self.variances = np.zeros((n_classes, n_features), dtype=np.float64)
        self.priors = np.zeros(n_classes, dtype=np.float64)

        for idx, c in enumerate(self.classes):
            X_c = X[y == c]  # 获取属于类 c 的所有样本
            self.means[idx, :] = X_c.mean(axis=0)  # 计算均值
            self.variances[idx, :] = X_c.var(axis=0) + 1e-9  # 计算方差并添加一个小数以避
免除以零
            self.priors[idx] = X_c.shape[0] / float(n_samples)  # 计算先验概率

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]  # 对每个样本进行预测
        return np.array(y_pred)

    def _predict(self, x):
        posteriors = []  # 后验概率列表

        for idx, c in enumerate(self.classes):
            prior = np.log(self.priors[idx])  # 计算先验概率的对数
            class_conditional = np.sum(np.log(self._pdf(idx, x)))  # 计算类条件概率的对数
            posterior = prior + class_conditional  # 计算后验概率
            posteriors.append(posterior)
```

```
        return self.classes[np.argmax(posteriors)]  # 返回后验概率最大的类标签

    def _pdf(self, class_idx, x):
        mean = self.means[class_idx]
        var = self.variances[class_idx]
        numerator = np.exp(- (x - mean) ** 2 / (2 * var))  # 计算分子
        denominator = np.sqrt(2 * np.pi * var)  # 计算分母
        return numerator / denominator  # 返回概率密度函数值

def load_mnist_images(filename):
    with open(filename, 'rb') as f:
        magic, num, rows, cols = struct.unpack(">IIII", f.read(16))
        images = np.fromfile(f, dtype=np.uint8).reshape(num, rows * cols)
    return images

def load_mnist_labels(filename):
    with open(filename, 'rb') as f:
        magic, num = struct.unpack(">II", f.read(8))
        labels = np.fromfile(f, dtype=np.uint8)
    return labels

def Accuracy(y_true, y_pred):
    correct_pred = sum(y_t == y_p for y_t, y_p in zip(y_true, y_pred))
    total_pred = len(y_true)
    accuracy = correct_pred / total_pred
    return accuracy
```

Train.py

```
from utils import *
import matplotlib.pyplot as plt

# 指定数据文件的路径
train_images_path = './data/train-images.idx3-ubyte'
train_labels_path = './data/train-labels.idx1-ubyte'
test_images_path = './data/t10k-images.idx3-ubyte'
test_labels_path = './data/t10k-labels.idx1-ubyte'

# 读取数据
train_images = load_mnist_images(train_images_path)
train_labels = load_mnist_labels(train_labels_path)
test_images = load_mnist_images(test_images_path)
test_labels = load_mnist_labels(test_labels_path)
```

```
# 初始化自定义朴素贝叶斯分类器
custom_gnb = CustomGaussianNB()

# 训练模型
custom_gnb.fit(train_images, train_labels)

# 预测测试集
y_pred = custom_gnb.predict(test_images)

# 计算准确率
accuracy = Accuracy(test_labels, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# 可视化部分测试样本及其预测结果
def plot_samples(X, y, y_pred, n_samples=10):
    plt.figure(figsize=(10, 10))
    for i in range(n_samples):
        plt.subplot(1, n_samples, i + 1)
        plt.imshow(X[i].reshape(28, 28), cmap='gray')
        plt.title(f"Label: {y[i]}\n Pred: {y_pred[i]}")
        plt.axis('off')
    plt.show()

# 展示前 10 个测试样本及其预测结果
plot_samples(test_images, test_labels, y_pred, n_samples=10)
```

Utils_ms.py

```
import mindspore.numpy as mnp
from mindspore import Tensor, context
from mindspore.common import dtype as mstype
from mindspore.ops import operations as P
from mindspore.ops import functional as F
import numpy as np
import struct

context.set_context(mode=context.PYNATIVE_MODE)

class CustomGaussianNB:
    def __init__(self):
        # 初始化模型参数
        self.means = None
```

```
        self.variances = None
        self.priors = None

    def fit(self, X, y):
        n_samples, n_features = X.shape # 获取样本数和特征数
        self.classes = mnp.unique(y) # 获取所有类标签
        n_classes = len(self.classes.asnumpy())  # 类别数

        # 初始化均值、方差和先验概率矩阵
        self.means = mnp.zeros((n_classes, n_features), dtype=mnp.float32)
        self.variances = mnp.zeros((n_classes, n_features), dtype=mnp.float32)
        self.priors = mnp.zeros(n_classes, dtype=mnp.float32)

           for idx, c in enumerate(self.classes.asnumpy()):  # Convert to numpy array for enumeration
            # 获取属于类 c 的所有样本
            X_c = []
            for i in range(len(y)):
                if(y[i].asnumpy()==c):
                    X_c.append(X[i])
            X_c = Tensor(X_c,dtype=mstype.float32)
            self.means[idx, :] = mnp.mean(X_c, axis=0) # 计算均值
            self.variances[idx, :] = mnp.var(X_c, axis=0) + 1e-9
            self.priors[idx] = X_c.shape[0] / float(n_samples) # 计算先验概率

    def predict(self, X):
        y_pred = [self._predict(x) for x in X] # 对每个样本进行预测
        return mnp.array(y_pred)

    def _predict(self, x):
        posteriors = [] # 后验概率列表

        for idx, c in enumerate(self.classes.asnumpy()):
            prior = mnp.log(self.priors[idx]) # 计算先验概率的对数
             class_conditional = mnp.sum(mnp.log(self._pdf(idx, x))) # 计算类条件概率的对数
            posterior = prior + class_conditional  # 计算后验概率
            posteriors.append(posterior)

        return self.classes[mnp.argmax(posteriors)] # 返回后验概率最大的类标签

    def _pdf(self, class_idx, x):
        mean = self.means[class_idx]
        var = self.variances[class_idx]
```

```
        numerator = mnp.exp(- (x - mean) ** 2 / (2 * var))
        denominator = mnp.sqrt(2 * mnp.pi * var)
        return numerator / denominator # 返回概率密度函数值

def load_mnist_images(filename):
    with open(filename, 'rb') as f:
        magic, num, rows, cols = struct.unpack(">IIII", f.read(16))
        images = np.fromfile(f, dtype=np.uint8).reshape(num, rows * cols)
    return images

def load_mnist_labels(filename):
    with open(filename, 'rb') as f:
        magic, num = struct.unpack(">II", f.read(8))
        labels = np.fromfile(f, dtype=np.uint8)
    return labels

def Accuracy(y_true, y_pred):
    # 将预测结果转化为同一形式（例如，通过取最大概率的索引）
    if y_pred.shape != y_true.shape:
        y_pred = P.Argmax(axis=1)(y_pred)
    # 计算正确预测的数量
    correct_pred = F.reduce_sum(F.cast(y_true == y_pred, mnp.float32))
    # 总预测数量
    total_pred = y_true.shape[0]
    # 计算精确度
    accuracy = correct_pred / total_pred
    return accuracy
```

Train_ms.py

```
from utils_ms import *
import matplotlib.pyplot as plt
from mindspore import Tensor

# 指定数据文件的路径
train_images_path = './data/train-images.idx3-ubyte'
train_labels_path = './data/train-labels.idx1-ubyte'
test_images_path = './data/t10k-images.idx3-ubyte'
test_labels_path = './data/t10k-labels.idx1-ubyte'

# 读取数据
train_images = load_mnist_images(train_images_path)
train_labels = load_mnist_labels(train_labels_path)
test_images = load_mnist_images(test_images_path)
```

```
test_labels = load_mnist_labels(test_labels_path)

train_images_Tensor = Tensor(train_images,dtype=mstype.float32)
train_labels_Tensor = Tensor(train_labels,dtype=mstype.float32)
test_images_Tensor = Tensor(test_images,dtype=mstype.float32)
test_labels_Tensor = Tensor(test_labels,dtype=mstype.float32)

# 初始化自定义朴素贝叶斯分类器
custom_gnb = CustomGaussianNB()

# 训练模型
custom_gnb.fit(train_images_Tensor, train_labels_Tensor)

# 预测测试集
y_pred = custom_gnb.predict(test_images_Tensor)

# 计算准确率
accuracy = Accuracy(test_labels_Tensor, y_pred)
accuracy_scalar = accuracy.asnumpy().item()
print(f"Accuracy: {accuracy_scalar * 100:.2f}%")

# 可视化部分测试样本及其预测结果
def plot_samples(X, y, y_pred, n_samples=10):
    plt.figure(figsize=(10, 10))
    for i in range(n_samples):
        plt.subplot(1, n_samples, i + 1)
        plt.imshow(X[i].reshape(28, 28), cmap='gray')
        plt.title(f"Label: {y[i]}\n Pred: {y_pred[i]}")
        plt.axis('off')
    plt.show()

# 展示前 10 个测试样本及其预测结果
plot_samples(test_images, test_labels, y_pred, n_samples=10)
```

实验三 神经网络

1. 问题描述

(1) 目标：利用神经网络算法，对 CIFAR-10 数据集中给定的测试集进行分类。

(2) 数据集描述：CIFAR-10 是一个描述自然图像的数据集，一共包含 10 个类别的彩色图片：飞机（ airplane ）、汽车（ automobile ）、鸟类（ bird ）、猫（ cat ）、鹿（ deer ）、狗（ dog ）、蛙类（ frog ）、马（ horse ）、船（ ship ）和卡车（ truck ）。每个图片的尺寸为 32 × 32，每个类别有 6000 个图像，数据集中一共有 50000 张训练图片和 10000 张测试图片。

2. 实现步骤与流程

(1) 实验思路:

在 utils.py 中定义一个结构确定的全连接神经网络类。由于本实验数据集和任务已经确定，因此我没有采用模块化神经网络层的设计，而是直接定义了一个单隐层的全连接神经网络，显示地更新其中的权值和偏置。其中包含参数初始化，前向传播，反向传播的函数实现。由于我固定了损失函数为交叉熵损失，在反向传播时梯度计算也是固定的，其中还有很多重复计算的值可以只计算一次，因此反向传播算法可以较为轻松的实现。除此之外我还仿照 pytorch 的 Dataset，Dataloader 结构写了一个用于本次实验的数据读取和加载的类，实现了批训练。

在 train.py 中则可以使用 utils.py 中定义的类来读取和加载 cifar10 数据集，初始化全连接神经网络后进行批训练循环，更新网络参数，最后用训练好的模型在测试集上进行评估。

(2) 实验流程：

i. 加载 cifar10 数据集，在本次实验中我使用的是官网下载的 pickle 格式的数据集。

ii. 给定输入层、隐藏层、输出层节点个数来初始化全连接神经网络，在此要注意神经网络参数的初始化方法，我使用的是随机高斯分布初始化权值，偏置初始化为全 0。

iii. 使用反向传播算法来计算梯度并进行参数更新，在此我同时进行了前向传播，记录训练过程中的交叉熵损失。

iv. 训练完成后使用测试集对模型进行评估，记录测试集上的损失和分类准确度。

(3) 实验要点/难点

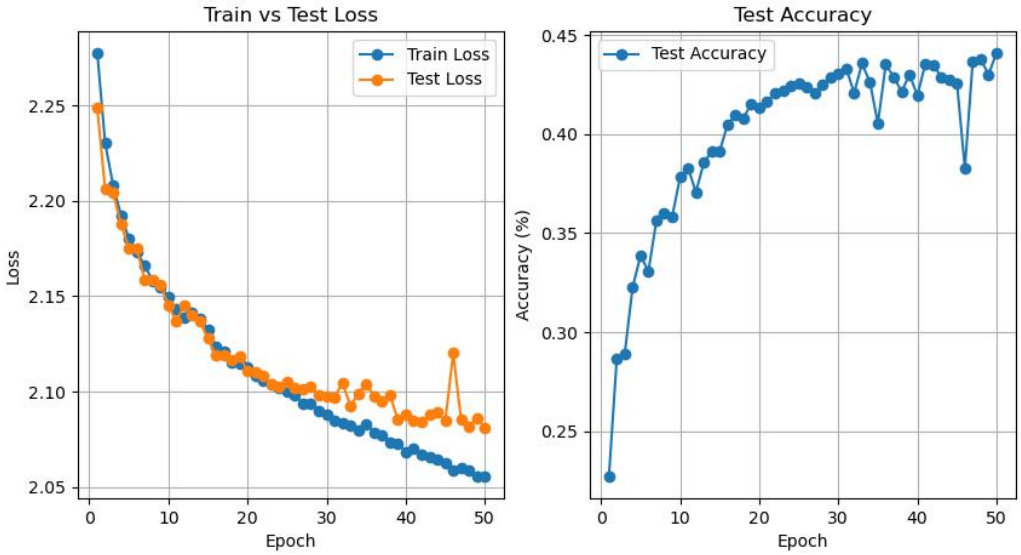
i. 由于实验要求不可使用 pytorch 等机器学习库，因此需要手动实现数据集的读取和加载，特别是在加载时的批处理。

ii. 实验中反向传播算法中的梯度计算要正确，在前期我使用前向传播中已经计算的参数来简化梯度计算，但由于前向传播时计算的参数没有及时更新导致梯度下降方向计算错误。在经过 debug 后我在反向传播算法实现时重新计算了要使用的参数，使损失正确减少。

iii. 采用批处理时要注意数据维度的转换，每次进入训练的数据会多一个批大小的维度，在数据展平等操作时要注意。而且由于批量学习，得到的损失、梯度也要注意除以批大小。

3. 实验结果与分析

实验损失和分类准确度可视化：

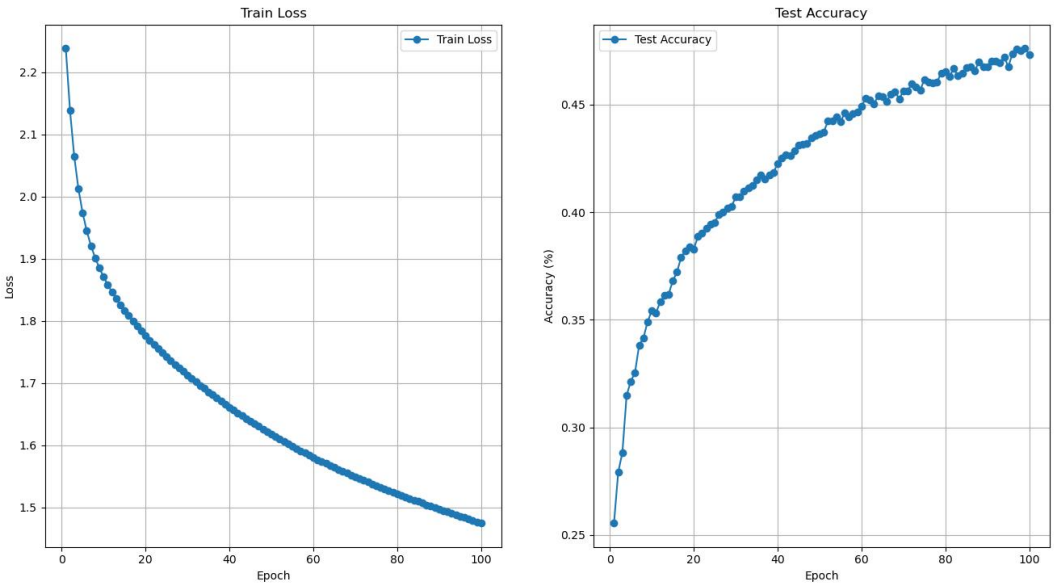


上图时学习率为 0.001 时的损失下降曲线和测试准确度曲线，可以看到模型在训练轮次达到 40 时，测试误差趋于收敛，最终分类精确度在 45%左右。

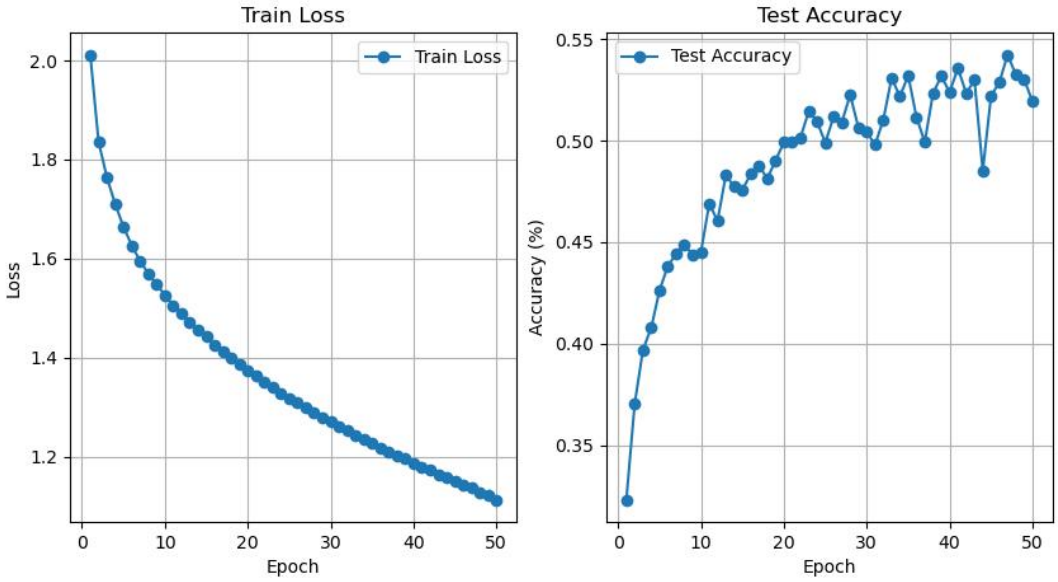
分类准确率仅有 45%的原因可能是单隐层全连接神经网络过于简单，对于该任务难以胜任。一是该图片数据为彩色数据，有三个通道，在 32×32 的像素下，图片特征数也达到了 3072，而输出为 10 个类，这导致隐藏层节点个数设置需要考量。对于 MNIST 数据集更好的方法是采用 CNN 来训练，它能将图片中的特征先进行提取，这样再输入全连接神经网络时输入节点数能显著降低，但由于我尝试进行纯手搓 CNN 的实现时出现了一些困难，最终选择了使用较为简单的全连接神经网络来实现。

在 mindspore 平台上的训练损失和测试准确度：

学习率为 0.001，训练轮次 100：



学习率为 0.01，训练轮次 50：



与手动实现的模型训练效果对比：

Mindspore 和手动实现的模型在学习率都为 0.001 时，mindspore 实现的模型损失和精度变化更加平滑，而且 mindspore 实现的模型在 100 轮次时还未达到收敛，其收敛速度更慢，能达到更高的期望精度。

Mindspore 实现模型间对比：

通过改变学习率可以很明显的看出学习率对于训练的影响。学习率越大，模型收敛越快，但在训练时精度的波动也更大。学习率为 0.01 时，在 40 轮次左右精确度达到最优在 54%左右。

4. MindSpore 学习使用心得体会

在这里我会介绍我是如何使用 mindspore 库实现上面相同任务的，重点介绍一些不同点。

我的 mindspore 实现代码在 train_ms.py 一个文件中。

由于 mindspore 是类似于 pytorch 的高级机器学习库，我可以使用其中高集成度的代码来实现神经网络。包括使用 mindspore.dataset 来读取数据，让模型继承 mindspore.nn.Cell 类来实现模块化搭建神经网络，使用 mindspore.ops 来进行梯度相关的计算。使用 mindspore 减轻了我手动搭建网络和一些辅助函数的工作量，也不需要手动进行复杂的梯度计算了，极大的方便了机器学习相关的代码搭建。

主要思路是使用 Minspore.dataset.Cifar10Dataset 类进行数据读取，这里由于官方提供的 api 只支持读取.bin 数据，所以 mindspore 读取的数据集的格式和 pandas 读取的数据集的格式有细微区别。然后对读取的数据进行预处理，包括归一化和 Tensor 内部数据类型变化，再使用 Mindspore.nn.Cell 进行神经网络的搭建。最后在进行反向传播时使用 Mindspore.ops 进行梯度计算和参数更新。

Mindspore 使用心得体会：

Mindspore 虽然和 pytorch 等库一样是为了便于相关人员进行机器学习开发，

但其实它们的框架还是有所不同的。

优点：在使用过程中我可以感受到 mindspore 还想在 pytorch 的基础上进一步提高集成性来简化代码。例如 mindspore 提供了一个 SoftmaxCrossEntropyWithLogits()函数，可以在未进行独热编码和归一化的情况下计算交叉熵损失。Mindspore 还提供了自动进行梯度下降功能，不需要显示实现。而且 mindspore 提供了 Model 类，在这个类中你可以输入你的网络类型和训练轮次和一些检查点信息，它能自动帮你训练模型，记录模型评估参数变化，定时保存模型，这些在几行代码内可以实现确实很方便。Minspore 的 dataset 类还提供了一些常见数据集的提取器，例如本次实验就可以使用 Minspore.dataset.Cifar10Dataset 直接读取数据。

缺点：虽然 mindspore 提供了很多便捷的、集成度高的代码，但这些代码在使用过程中其实是有些难用的。一是对其数据类型 Tensor 的内部数据类型限制很严重，常常出现将 Tensor 数据输入它提供的 api 中出现数据类型不匹配的情况，它也没有自动进行数据类型转换的功能，需要手动进行转换。二是这些高度集成的类的报错信息却过少，我在使用 Model 类时出现了输入参数不符合要求的报错，但这个报错居然没有任何提示信息，只有一个 No Description 的提示，不知道是不是我的 mindspore 版本或者代码编辑器的问题，我按照它官网提供的 api 要求传入数据出现报错还没提示信息，使得 debug 及其困难。

5. 代码附录

Utils.py

```
import numpy as np
import pickle
import math

def relu(x):
    return np.maximum(0, x)

# relu 函数的导数
def relu_derivative(x):
    return np.where(x > 0, 1, 0)

def one_hot_encode(labels, num_classes=10):
    # 创建一个形状为 (len(labels), num_classes) 的零矩阵
    one_hot_labels = np.zeros((len(labels), num_classes))
    # 设置对应的索引为 1
    one_hot_labels[np.arange(len(labels)), labels] = 1
    return one_hot_labels

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
```

```
    return np.exp(x) / np.sum(np.exp(x), axis=1, keepdims=True)

def cross_entropy_loss(y_pred, y_true):
    m = y_true.shape[0]
    p = softmax(y_pred)
    log_likelihood = -np.sum(y_true * np.log(p + 1e-12))
    loss = log_likelihood / m
    return loss

class FullyConnectedNN:
    def __init__(self, input_size, hidden_size, output_size, lr=0.001):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.lr = lr

        self.w1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros(hidden_size)
        self.w2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros(output_size)

    def __call__(self, inputs):
        return self.forward(inputs)

    # 前向传播
    def forward(self, x):
        # 输入层到隐藏层
        self.z1 = x @ self.w1 + self.b1
        self.a1 = relu(self.z1)
        # 隐藏层到输出层
        self.z2 = self.a1 @ self.w2 + self.b2
        self.a2 = softmax(self.z2)
        return self.a2

    # 反向传播算法
    def backward(self, x, y):
        # 在反向传播前进行了一次前向传播，利用其中的数据可以简化梯度计算
        z1 = x @ self.w1 + self.b1
        a1 = relu(z1)
        z2 = a1 @ self.w2 + self.b2
        a2 = softmax(z2)

        # 反向传播到输出层
        batch_size = x.shape[0]
```

```
        delta2 = a2 - y
        grad_w2 = a1.T @ delta2 / batch_size
        grad_b2 = np.sum(delta2, axis=0) / batch_size

        # 反向传播到隐藏层
        delta1 = (delta2 @ self.w2.T) * relu_derivative(z1)
        grad_w1 = x.T @ delta1 / batch_size
        grad_b1 = np.sum(delta1, axis=0) / batch_size

        # 更新参数
        self.w1 -= self.lr * grad_w1
        self.b1 -= self.lr * grad_b1
        self.w2 -= self.lr * grad_w2
        self.b2 -= self.lr * grad_b2


class CIFAR10Dataset:
    def __init__(self, file_paths):
        self.data, self.labels = self.load_data(file_paths)

    def load_data(self, file_paths):
        data = []
        labels = []
        for file_path in file_paths:
            with open(file_path, 'rb') as file:
                batch = pickle.load(file, encoding='bytes')
                labels.extend(batch[b'labels'])
                images = batch[b'data']
                images = images.reshape((-1, 3, 32, 32)).transpose(0, 2, 3, 1) # WHC2CWH
                data.extend(images)
        return np.array(data), np.array(labels)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, index):
        # 返回第 index 个图像及其标签
        return self.data[index], self.labels[index]

class DataLoader:
    def __init__(self, dataset, batch_size, shuffle=True):
        self.dataset = dataset
        self.batch_size = batch_size
        self.shuffle = shuffle
```

```
        self.indices = list(range(len(dataset)))
        if self.shuffle:
            np.random.shuffle(self.indices)

    # 创造迭代器
    def __iter__(self):
        self.current_idx = 0
        return self

    def __next__(self):
        if self.current_idx >= len(self.dataset):
            raise StopIteration

        indices = self.indices[self.current_idx:self.current_idx + self.batch_size]
        batch = [self.dataset[idx] for idx in indices]
        data_batch = np.array([item[0] for item in batch])
        labels_batch = np.array([item[1] for item in batch])
        labels_batch = one_hot_encode(labels_batch)
        self.current_idx += self.batch_size
        return data_batch, labels_batch

    def __len__(self):
        return math.ceil(len(self.dataset) / self.batch_size)
```

Train.py

```
from utils import *
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)

# 加载数据
train_files = [f'data/cifar-10-batches-py/data_batch_{i}' for i in range(1, 6)]
test_file = ['data/cifar-10-batches-py/test_batch']

train_dataset = CIFAR10Dataset(train_files)
test_dataset = CIFAR10Dataset(test_file)

train_dataloader = DataLoader(train_dataset, batch_size=512, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=512, shuffle=False)

# 实例化网络
network = FullyConnectedNN(3072, 512, 10)
```

```
epochs = 20

# 训练循环
history = [] # 用于记录评估参数
for epoch in range(epochs):
    total_loss = 0
    # 批处理，批大小为 512
    for data, label in train_dataloader:
        predictions = network.forward(data.reshape(data.shape[0], -1))
        loss = cross_entropy_loss(predictions, label)
        network.backward(data.reshape(data.shape[0], -1), label)
        total_loss += loss
    trian_loss = total_loss / len(train_dataloader)
    print(f'Epoch {epoch + 1}, Train Loss: {trian_loss}')

    # 评估模型
    total_loss = 0
    correct_predictions = 0
    total_samples = 0
    for data, labels in test_dataloader:
        predictions = network.forward(data.reshape(data.shape[0], -1))
        loss = cross_entropy_loss(predictions, labels)
        total_loss += loss
        predicted_labels = np.argmax(predictions, axis=1) # 选择最大概率值对应的标签作为预测
        labels = np.argmax(labels, axis=1)
        correct_predictions += np.sum(predicted_labels == labels)
        total_samples += len(labels)
    test_loss = total_loss / len(test_dataloader) # 测试损失
    test_accuracy = correct_predictions / total_samples # 测试精度
    print(f'Test Loss: {test_loss}, Test Accuracy: {test_accuracy}')

    history.append([epoch + 1,trian_loss,test_loss,test_accuracy])

epochs = [x[0] for x in history]
train_losses = [x[1] for x in history]
test_losses = [x[2] for x in history]
test_accuracies = [x[3] for x in history]

# 创建图表
plt.figure(figsize=(10, 5))

# 绘制训练损失和测试损失
```

```
plt.subplot(1, 2, 1)
plt.plot(epochs, train_losses, label='Train Loss', marker='o')
plt.plot(epochs, test_losses, label='Test Loss', marker='o')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Train vs Test Loss')
plt.grid(True)
plt.legend()

# 绘制测试精度
plt.subplot(1, 2, 2)
plt.plot(epochs, test_accuracies, label='Test Accuracy', marker='o')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Test Accuracy')
plt.grid(True)
plt.legend()

# 展示图表
plt.show()
```

Train_ms.py

```
from mindspore import nn, ops
import mindspore.dataset as ds
from mindspore.dataset import vision
from mindspore import dtype as mstype
import matplotlib.pyplot as plt

# 继承 nn.Cell，类似 pytorch.nn.model
class FullyConnectedNN(nn.Cell):
    def __init__(self, input_size, hidden_size, output_size, lr=1e-2):
        # 模块化搭建神经网络
        super(FullyConnectedNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Dense(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Dense(hidden_size, output_size)
        self.loss_fn = nn.SoftmaxCrossEntropyWithLogits(sparse=True, reduction='mean')
        self.optimizer = nn.SGD(self.trainable_params(), learning_rate=lr)

    # 完成 construct 函数，类似 forward 函数，用于前向传播
    def construct(self, x):
        x = self.flatten(x)
```

```
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x

    # ops.value_and_grad 中的优化函数，，mindspore 可以在此自动进行梯度下降来降低损失
    def optimization_function(self, data, label):
        output = self(data)
        loss = self.loss_fn(output, label)
        return loss, output

# 实例化网络
network = FullyConnectedNN(3072, 512, 10, lr=0.01)

# 读取数据
cifar10_dataset_dir = "./data/cifar10"

# 使用 Cifar10Dataset 读取数据并进行数据预处理
train_dataset =
ds.Cifar10Dataset(dataset_dir=cifar10_dataset_dir,usage='train',shuffle=True)
train_dataset = train_dataset.map(vision.Rescale(1.0 / 255.0, 0), 'image')
train_dataset = train_dataset.batch(batch_size=64, drop_remainder=True) # 批处理

test_dataset = ds.Cifar10Dataset(dataset_dir=cifar10_dataset_dir,usage='test')
test_dataset = test_dataset.map(vision.Rescale(1.0 / 255.0, 0), 'image')
test_dataset = test_dataset.batch(batch_size=64)

# 训练和评估网络
epochs=50
history = []
for epoch in range(epochs):
    network.set_train() # 设置为训练模式
    grad_fn = ops.value_and_grad(network.optimization_function, None,
network.optimizer.parameters, has_aux=True)
    total_loss = 0
    for images, labels in train_dataset.create_tuple_iterator(): # 产生迭代器，可自动进行批加载
        labels = labels.astype(mstype.int32)
        (loss, _), grads = grad_fn(images, labels)
        total_loss += ops.depend(loss, network.optimizer(grads))
    train_loss = total_loss / train_dataset.get_dataset_size()

    network.set_train(False) # 调整为评估模式
```

```
    total, correct = 0, 0
    for images, labels in test_dataset.create_tuple_iterator():
        labels = labels.astype(mstype.int32)
        output = network(images)
        predicted = output.argmax(axis=1)
        correct += (predicted == labels).asnumpy().sum()
        total += len(images)
    accuracy = correct / total
    print(f"Epoch {epoch + 1}, Loss: {train_loss}, Test Accuracy: {accuracy}")
    history.append([epoch + 1, train_loss ,accuracy])

epochs = [x[0] for x in history]
train_losses = [x[1] for x in history]
test_accuracies = [x[2] for x in history]

# 创建图表
plt.figure(figsize=(10, 5))

# 绘制训练损失
plt.subplot(1, 2, 1)
plt.plot(epochs, train_losses, label='Train Loss', marker='o')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Train Loss')
plt.grid(True)
plt.legend()

# 绘制测试精度
plt.subplot(1, 2, 2)
plt.plot(epochs, test_accuracies, label='Test Accuracy', marker='o')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.title('Test Accuracy')
plt.grid(True)
plt.legend()

# 展示图表
plt.show()
```

心得体会

本次实验的特色点在于禁止使用 pytorch 等机器学习库，这使得我在实现数据降维，构建逻辑回归分类器、构建朴素贝叶斯分类器、构建神经网络时需要手动写其中的反向传播算法，计算梯度并更新参数。缺少了自动追踪的梯度自然在代码搭建时提高了挑战性，但手动实现相关算法的同时也是对算法的一次加深理解，只有在掌握这些算法背后的数学原理时才能解决模型搭建和调试时出现的问题。

在本次实验中我还注重总结各个模型之间的共同点，提高代码的复用率。本次实验选题中有很多分类问题，因此例如交叉熵损失函数、独热编码函数都可以复用。而这些模型搭建时大多使用到了梯度下降法，因此基本都包含 forward，backward 函数，这些共性可以帮助我熟练进行相似模型的搭建，锻炼了我的相关代码能力。

在本次实验中还鼓励我在 mindspore 平台上进行实验，使用华为开发的 mindspore 机器学习库进行模型搭建。目前我的感受是使用起来还有一定的困难。一是其与 numpy、pytorch 库有一定的区别，在进行两种思路的转换中还需要进一步熟悉。二是 mindspore 的一些细节处用起来还没有 pytorch 那么智能，例如在数据类型的限制上不能自动转换以适应。三是 mindspore 的社区没有 pytorch 完善，代码报错的提示信息也不够清晰，这使得调试难度大大增加。Mindspore 也有其独特的见解，在一些方面的集成度上是有优势的，但整体来说要想更好的使用还需要我进一步的熟悉和 mindspore 的进一步优化更新。

本次实验是一次难得的开发体验，实验任务也从易到难，在整个过程中学习和调试是一次宝贵的经验。