人工智能导论课程项目报告

学　　院　信息科学与工程学院

年级专业　计算机科学与技术 23-5

组　　号　二组

姓名学号　李济岑 202211090252

康心柔 202311040121

那圣奇 202311040130

张连兴 202311050349

报告日期　2014 年 11 月 7 日

一. 摘要

本报告是针对《人工智能导论》课程中的三个三级项目的研究总结。项目旨在通过实践应用，加深我们对人工智能基本理论和方法的理解，并提升使用 Python 深度学习库 PyTorch 解决实际问题的能力。

第一个项目聚焦于 BP 神经网络的构建，利用波士顿房价数据集的前 300 条数据进行模型训练，并预测剩余数据的房价，最终统计预测误差。第二个项目则关注卷积神经网络模型的构建与应用。使用 CIFAR10 数据集进行模型训练，并评估模型在测试集上的分类准确率。第三个项目涉及长短期记忆网络（LSTM）模型的构建与评估。使用 THUCNews 中文新闻数据集进行模型训练，并在测试集上进行评估，以给出预测的准确率。

通过这这门课程的三个实践，提升了我们在人工智能领域的基本技能和综合素质的提升，同时也为未来的学习和研究奠定了坚实的基础。

关键词：人工智能、神经网络、模型训练与评估、基本技能与综合素质

二. 项目流程介绍

1. BP 神经网路房价预测

这段代码实现了一个基于 PyTorch 的 BP 神经网络模型，用于对波士顿房价数据集进行训练和预测。整个流程涵盖了数据预处理、模型定义、训练、评估及参数调优。以下是代码实现步骤和思路的详细解读：

1.1 主要第三方库

基于 pytorch 深度学习框架的的 BP 神经网络（多层前馈神经网络），构建了 3 层前馈神经网络并独立设定神经元个数，并通过 forward（前向传播函数）和 backward（反向传播函数）进行训练和反馈学习。

基于 numpy 和 pandas 的数据提取和处理，从 14 列数据中提取 13 列特征数据并通过基于 sklearn 的数据分割和数据加载器进行数据预处理分为 300 个训练集特征，训练集房价和 206 个测试集特征，测试集房价。

基于 matplotlib 的可视化及数据绘图处理，将最后求出模型 loss（损失率）和

MSE（预测房价均方误差）的波动曲线和散点图。

基于 gc 的内存垃圾回收机制实现模型多次运行求出最佳参数及平均 loss（损失率）和 MSE（预测房价均方误差）。

1.2 数据加载和预处理

数据加载：getData 方法读取 dat 文件，提取特征和标签，进行归一化处理，然后将数据分割为训练集和测试集。

数据预处理：读取目标路径 data_url 下的 boston.dat 文件并提取其中前 13 列特征数据并进行归一化处理和最后 1 列房价数据。

数据拆分：通过 train_test_split 函数进行分割为训练集数据和测试集数据。

数据转换和封装：数据被转换为 torch 的 tencor 张量，并封装到 TensorDataset 中，再通过 DataLoader 封装成 train_loader 训练数据集加载器，支持批量训练和打乱数据。

类属性赋值:将 x_test, y_test, train_data, train_loader 赋值给类实例的属性，以便后续使用。

1.3 定义神经网络模型

Boston 类继承自 torch.nn.Module，定义了神经网络的结构。

使用 torch.nn.Sequential 容器来定义了一个前馈神经网络。

通过 torch.nn.Sequential 容器定义并存储了一个 BP 神经网络，包含了 3 层由 torch.nn.Linear 定义的前馈神经网络并设定不同神经元，同时选择 torch.nn.ReLU 函数作为 BP 神经网络的激活函数。

在整个学习框架中将以 torch.nn.Sequential 定义的神经网络作为模型的 forward（前向传播函数）并选择 torch.nn.MSELoss 函数作为模型的 backward（反向传播函数）。

1.4 定义模型

trainData 方法定义了整个神经网络训练过程，包括优化器选择（Adam）、损失函数（torch.nn.MSELoss）、反向传播（backward）、步进更新参数（step）和梯度清零（zero_grad）。

该方法启用 train 训练模式，通过循环 epoch 对单个训练数据进行训练，并循环

train_loader 进行分块训练再 loss（损失值）求和。储存每次的 loss 值并保存训练最后 number_of_sig_losser 个数的 loss 值的平均值作为该模型最终 loss 值。

重要参数：

trainData 方法用于训练模型。

epochs 是训练的轮数，即整个训练数据集将被遍历多少次。

learn_rate 是学习率，控制参数更新的步长。

number_of_sig_losser 是用于计算最终平均损失值的最后几个 epoch 的数量。

方法内部逻辑：

1 . **优化器：**在测试中我们尝试了 SGD 优化器和 Adam 优化器进行训练，而 Adam 优化器的效果更好且更适合线性回归模型故选择 Adam 优化器。Adam 是一种基于一阶和二阶矩估计的自适应学习率方法。

```
optimizer = torch.optim.Adam(self.parameters(), lr=learn_rate)
```

2. **损失函数：**使用均方误差损失（MSE Loss）作为损失函数，衡量模型预测值与实际值之间的差异。

```
loss_fun = torch.nn.MSELoss()
```

3. **训练模式：**将模型设置为训练模式。这对于某些特定的层（如 Dropout 和 BatchNorm）是必要的，它们在训练和评估时的行为不同。

4. **训练循环：**遍历每个 epoch，对每个 batch 的数据进行前向传播、计算损失、反向传播和参数更新。

在每个 epoch 中，遍历训练数据加载器 self. train_loader，它应该是一个提供输入数据 x 和目标数据 y 的迭代器。

计算损失：loss = loss_fun(self(x)， y)。

反向传播损失：loss. backward()。

使用优化器更新参数：optimizer. step()。

清零梯度：optimizer. zero_grad()，为下一个 batch 的梯度计算做准备。

累加每个 batch 的损失，计算整个 epoch 的平均损失。

5. 打印和记录：每 100 个 epoch 打印一次当前 epoch 的平均损失，并记录每个 epoch 的平均损失值。

6. 最终平均损失：计算并打印最后 number_of_sig_losser 个 epoch 的平均损失值。

1.5 模型评估

方法一：calMSE

calMSE 方法为评估模式，通过已训练好的模型对测试集进行预测，并计算预测房价和真实房价的均方误差（MSE），并按照传入路径绘制并存储预测值与真实值的散点图和回归散点图。

重要参数：

scatter_diagram_path：保存预测值与真实值对比散点图的路径。

regression_graph_path：保存回归散点图的路径。

方法内部逻辑：

将模型设置为评估模式（self.eval），这通常会影响某些层（如 Dropout 层）的行为，使它们在评估时不会随机丢弃神经元。

使用 with torch.no_grad 上下文管理器来禁止梯度计算，这是因为在评估模式下我们不需要梯度，禁用它们可以减少内存消耗并提高计算速度。

通过模型传递测试集数据（self.x_test），得到预测值（y_pred）。

计算预测值与真实值（self.y_test）之间的均方误差（MSE），并打印出来。

方法二：drawCurve

drawCurve 方法按照传入路径绘制并存储训练过程中的整体的 loss 值损失变化曲线和局部的 loss 值损失变化曲线（个数为 number_of_sig_losses 个）。

重要参数：

save_model_path：保存模型状态的路径。

save_img_path：保存整个训练过程损失值变化图的路径。

local_loss_img_path：保存最近 number_of_sig_losser 个 epoch 损失值变化图的路径。

number_of_sig_losser：决定局部损失图显示的 epoch 数量。

方法内部逻辑：

使用 torch.save(self.state_dict(), save_model_path)将模型的当前状态（参数、架构等）保存到 save_model_path。

创建一个大小为 10x5 英寸的图像，绘制整个训练过程中损失值的变化曲线，并保存到 save_img_path。

创建另一个大小为 10x5 英寸的图像，绘制最近 number_of_sig_losser 个 epoch 的损失值变化曲线，并保存到 local_loss_img_path。这有助于观察模型在训练接近结束时损失值的变化趋势。

1.6 程序运行

parameter 字典定义了各种参数，如特征数量、数据路径、训练参数等，方便固定在一处对模型参数进行修改和调试。包含如下重要参数：

1. epochs：数据训练周期，表示每个 batch 学习的次数。

2. learn_rate：更新模型参数时步长的大小，即模型在每次反向传播中调整权重的幅度。

3. batch_size：决定了每次前向和反向传播时输入神经网络的数据样本数量。

4. train_model：表示该主程序运行模式，1 为 modelTest 单次模型训练及预测，2 为 theBestMSEandLoss 多次训练寻找最佳 loss 和 MSE，3 为 findBestParameter 多次遍历寻找最佳参数。

5. 根据 train_model 参数的值，决定是单次训练、多次训练评估还是参数调优。

1. modelTest 函数

功能：modelTest 函数是单次训练模型并评估预测，根据 parameter 字典中的参数执行。在该函数中获取到 loss 值和 MSE 值并绘制单次运行曲线。

2. theBestMSEandLoss 函数

功能：函数是通过循环遍历多次运行 modelTest 实现多次运行，记录每次的 loss 值和 MSE 值，计算二者的平均值和最佳值，并绘制归一化后的 MSE 和 loss 变化散点图。

3. findBestParameter 函数

功能：findBestParameter 函数是通过 3 重循环遍历 epoch、batch_size、learn_rate 三个参数值并调用运行 modelTest 函数，以获取三者在某个值时能获得最佳 loss 值和 MSE 值。

4. 主程序

功能：根据 parameter 字典中的设置，执行上述三个功能之一。
过程：检查 parameter["train_model"]的值，决定执行哪个功能。

1.7 测试数据结果及可视化

在参数选择上，我们重点在 epoch、batch_size、learn_rate 三个数据上进行值的调整以求得最优参数。为了自动化操作，我们使用 findBestParameter 函数遍历三种值的各类数据，并记录最优 loss 值和最优 MSE 值对应的参数。

运行结果如下：

```
lowest loss: (0.0054, 15.5052): [5000, 0.004, 8]
lowest MSE:  (0.0371, 10.0855): [5000, 0.005, 8]
```

最终我们得出最优 epoch=5000，batch_size=8、learn_rate=0.05，最终经测试 batch_size=8 时会出现过拟合现象且运行时间过高，如下所示：

```
batch_size:8    time:120s    loss:0.02/MSE:10    (overfitting)
batch_size:16   time:80s     loss:0.02/MSE:10
```

故最终参数的选择我们决定为 epoch=5000，batch_size=16、learn_rate=0.05。以此参数我们选择 train_model=1 进行单次模型运行并获得 epoch-loss 值收敛曲线图如图 1-1：

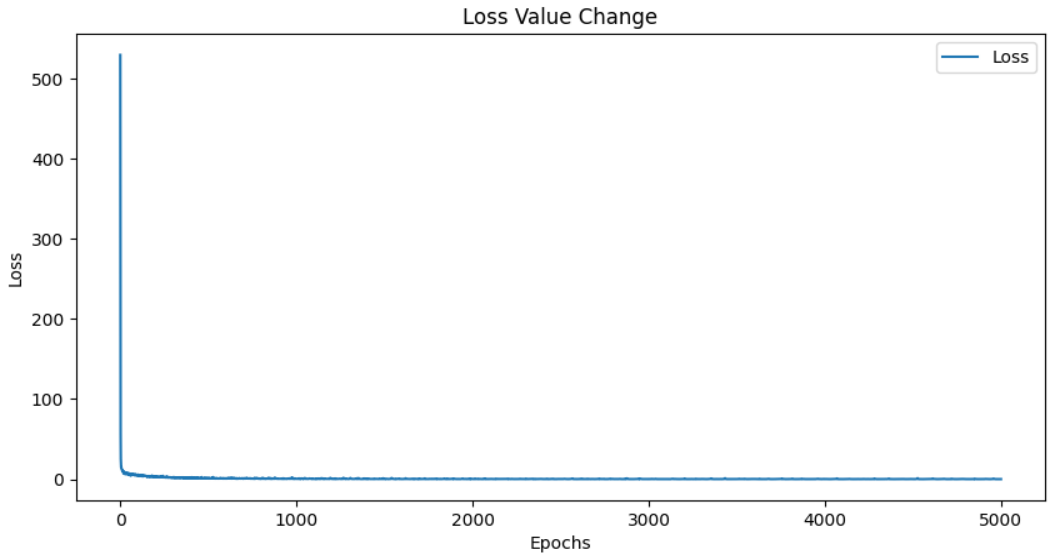


图 1-1

为使最终 loss 值在训练结束后趋于稳定，选定最后 30 组 epoch 的 loss 值取平均值作为最终训练的 loss 值，最后 30 组 epoch-loss 值收敛曲线如图 1-2：

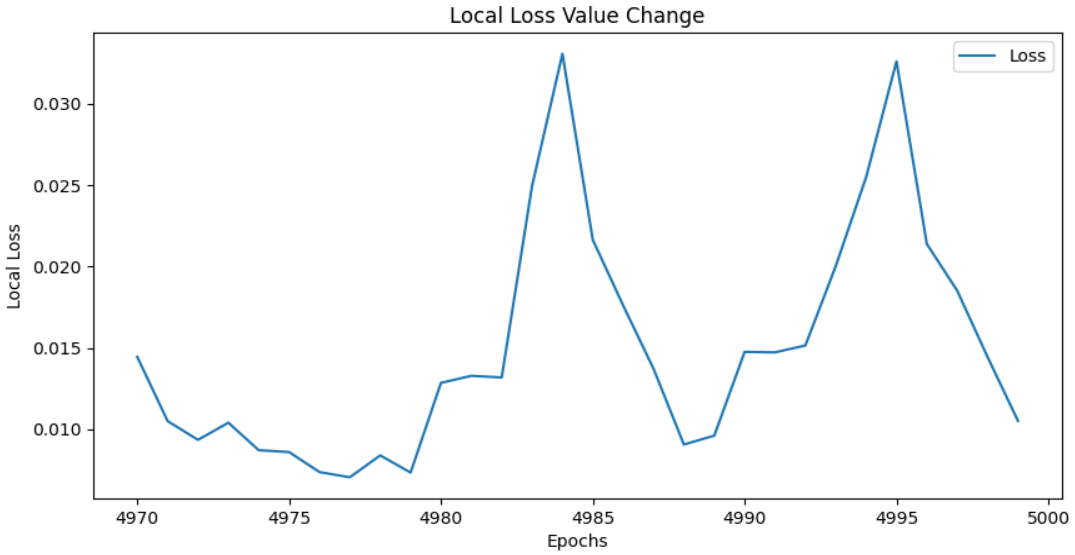


图 1-2

预测房价最终会回归成一条直线，房价回归曲线如图 1-3：

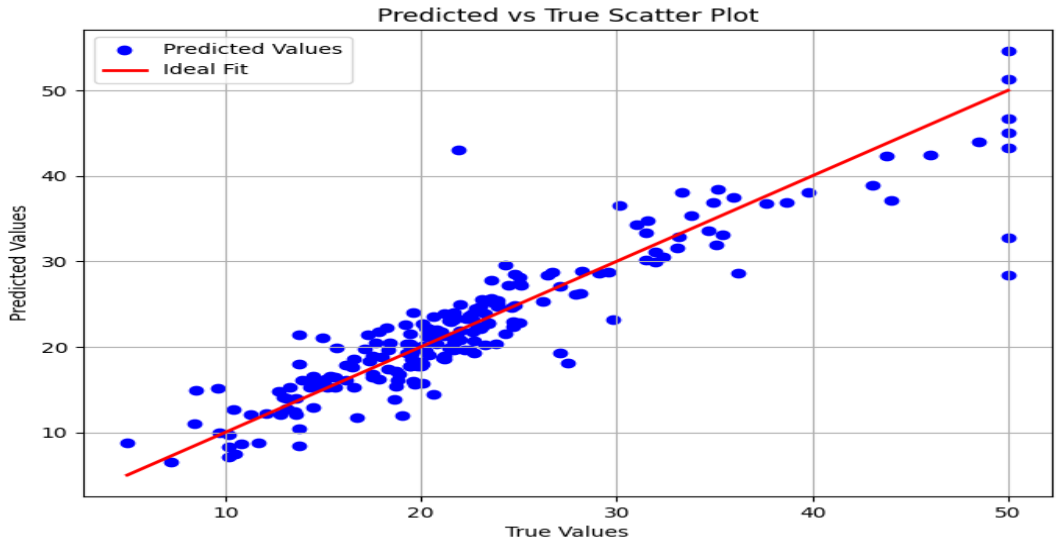


图 1-3

206 组测试集中预测房价和真实房价差值散点如图 1-4：

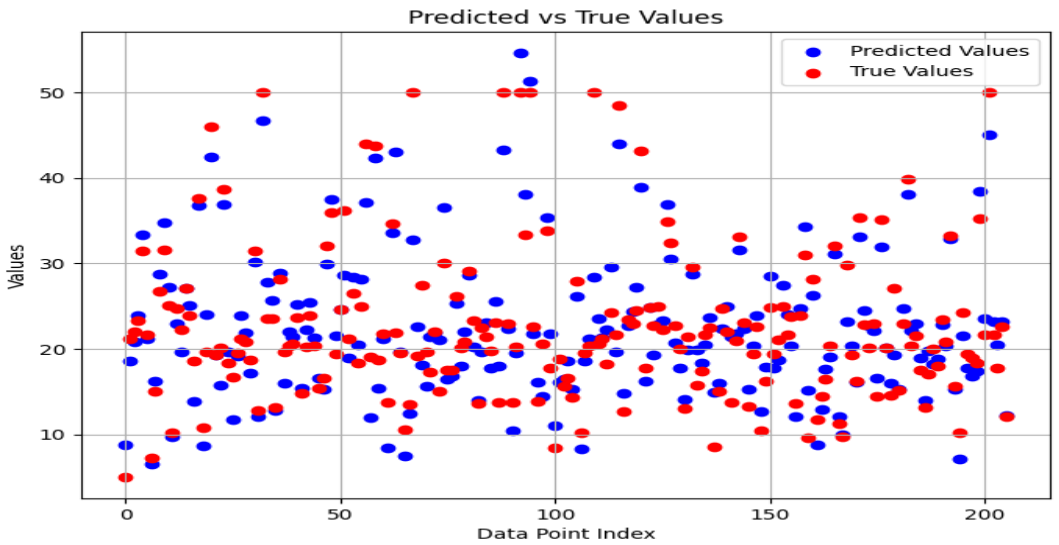


图 1-4

单次运行结果的最终平均 loss 值、MSE 值和单次运行时间如下：

```
Final average loss: 0.0058
The Test MSE: 13.2706
Take time:84.2377 s
```

在单次运行在中我们获得了 loss=0.0058，MSE=13.2706，但在实际效果中不同次运行会出现不同次效果，可能会导致误差，我们选择选择 train_model = 2 进行多次模型运行求得平均和最优 loss 值和 MSE 值，选择 train_times = 10 运行 10 次效果如下：

```
Run 10 times:
The min loss:0.0076
The average loss:0.0104
The min MSE:9.9589
The average MSE:10.8020
```

为求最佳和平均 loss 值和 MSE 值，选择 train_times = 200 运行 200 次效果，同时记录 200 次数据并将 loss 值和 MSE 值归一化进行直观获取最佳数据。

运行效果如下：

```
Run 200 times:
The min loss:0.0017
The average loss:0.0025
The min MSE:9.6185
The average MSE:10.6856
```

在运行 200 次后我们获得了 min_loss = 0.0017，average_loss = 0.0025，min_MSE = 9.6185，average_MSE = 10.6856。

最终为了使得 loss 值和 MSE 值在一定范围之内趋于稳定，将二者所得归一化后制成 loss 值和 MSE 值散点图如 1-5：

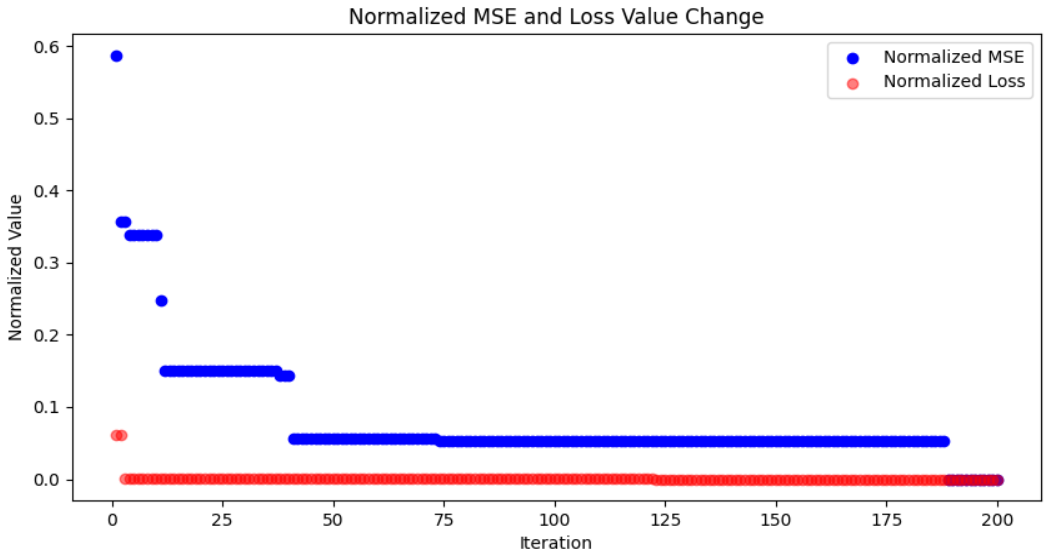


图 1-5

由模型训练测试得出当 epoch = 5000，batch_size = 16、learn_rate = 0.05 时，该预测模型在整体上表现出较高的准确性，预测值（蓝色点）与真实值（红色理想拟合线）之间呈现出较强的相关性。大部分数据点紧密围绕在理想拟合线周围，这证明了模型在预测真实值方面具备一定的可靠性。最终的 loss 值为 0.002 左右，MSE 值为 10 左右。

2. CNN 神经网络图像识别

在本项目中，我们定义了三种模型，Base_CNN,Onecyclelr_CNN,SE_CNN,并将 Base_CNN 作为 Baseline 进行模型对比优化。

2.1 Base_CNN

模型的总体设计思路

Base_CNN 主要用于处理 CIFAR-10 数据集的图像分类任务。它的结构设计基于层次化特征提取，逐层抽象出图像的低级到高级特征。模型包含四层卷积层和三层全连接层，通过逐步提取和处理图像的特征，最终映射到 10 个类别的分类任务上。

各层结构与功能说明

卷积层：每一层卷积层都包含一个卷积操作和一个激活函数（ReLU），以及批归一化层（Batch Normalization）。部分层通过通过堆叠 Conv-BN-ReLU-Pooling 的方式设置，卷积层的主要任务是通过卷积核提取图像的局部特征。

第 1 层卷积：输入图像为 RGB 彩色图（3 个通道），经过第 1 层卷积后，输出为 16 个特征图。

第 2 层卷积：通过更深的卷积操作，从 16 个通道变为 32 个通道，并添加池化层来降低图像分辨率，从而减少参数和计算量。

第 3 层卷积：继续增加通道数，将输入 32 通道特征图转变为 64 通道，进一步提取图像特征。

第 4 层卷积：最终将特征图转换为 128 个通道，再次通过池化层减少尺寸，以便进入全连接层进行分类。

池化层：池化操作的主要目的是对特征图进行降维，通过保留重要特征而丢弃细节，进一步提高模型的计算效率。在第 2 层和第 4 层卷积后添加了池化层。每次池化操作都将特征图的宽和高减半，同时保留重要的特征信息。

批归一化层：批归一化用于对特征进行标准化。归一化的作用是加速收敛，减少梯度消失或梯度爆炸问题的发生。每次卷积操作之后，批归一化对卷积结果进行归一化处理，从而提升训练效果和稳定性。

激活函数 ReLU：激活函数增加模型的非线性表达能力，使得神经网络能够学习更复杂的特征。ReLU（Rectified Linear Unit）是常用的激活函数，它的效果在于提升网络的训练速度并减少计算复杂度。

全连接层与分类

在经过多层卷积和池化后，输出的特征图展平为一维向量，然后通过三层全连接层将特征逐步映射到分类空间。

全连接层 1：首先将展平后的特征映射到一个高维空间（1024 维），以便模型能够有效地学习特征之间的复杂关系。

全连接层 2：将 1024 维特征进一步降维到 128 维，作为特征的更紧凑表示。

全连接层 3：最终将 128 维特征映射到 10 个输出类别，用于预测每个输入图像的分类结果。

损失函数与优化方法

损失函数：在训练过程中，模型使用交叉熵损失（Cross Entropy Loss）来衡量预测结果与真实标签的差距。交叉熵损失适合分类任务，通过最小化损失函数可以引导模型的权重更新，从而提升准确率。

优化算法：优化器采用随机梯度下降法（SGD）进行参数更新。学习率 learning_rate 设置为稳定的 0.01。

模型的训练与测试流程

在每个训练轮次（epoch）中，模型会：

前向传播：将训练数据传入模型，逐层进行卷积、池化、批归一化和全连接操作，得到输出预测。

计算损失：通过损失函数计算预测结果与实际标签的差距。

反向传播：根据损失对模型的参数进行梯度计算，优化器使用这些梯度来调整模型权重。

评估性能：在测试集上验证模型的表现，以监测是否存在过拟合问题。

训练和测试过程中的准确率和损失值在每个 epoch 被记录下来，用于观察模型的收敛情况，调整超参数以及模型结构。

模型效果可视化

在训练过程中，我们将训练和测试的损失与准确率可视化，以便直观地了解模型的学习进展，判断是否需要进一步优化结构或调整参数。

2.2 Onecyclelr_CNN

BaseCNN 和 Onecyclelr_CNN 主要区别在于数据增强和学习率调整策略。BaseCNN 是一个基础的卷积神经网络模型，使用固定学习率和基本数据增强，而 Onecyclelr_CNN 采用了更丰富的数据增强方式和动态学习率数据增强

BaseCNN

在 BaseCNN 中，数据处理主要包括：

```
transform = transforms.Compose([transforms.ToTensor()])
```

缺乏多样性：仅将数据转化为张量，可能导致模型对训练数据的特征过拟合。

Onecyclelr_CNN

在 Onecyclelr_CNN 中，数据增强通常包括：

```
# stats 变量存储了标准化时的均值和标准差
stats = ((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
train_data = torchvision.datasets.CIFAR10("./data", train=True,
transform=torchvision.transforms.Compose([
    torchvision.transforms.ColorJitter(0.5),  # 随机调整图像亮度、对比度、饱和度
    torchvision.transforms.RandomHorizontalFlip(),  # 随机水平翻转
    torchvision.transforms.ToTensor(),  # 将图像转化为张量
    torchvision.transforms.Normalize(*stats)  # 应用均值和标准差进行标准化
]))
```

随机水平翻转：增加了图像的随机水平翻转，以提高模型的泛化能力。

色彩抖动：随机改变图像的亮度、对比度、饱和度和色调，增加了训练数据的多样性，减少了过拟合的风险。

OneCycleLR 学习率策略

1.OneCycleLR 策略概述

OneCycleLR（单周期学习率）是一种动态的学习率调节策略。它先快速增大学习率，使模型能在最初阶段探索更大的参数空间，随后在训练的后期逐渐减小学习率，以便更稳定地收敛到全局或近似全局最优点。具体过程如下：

初期增大学习率：学习率从一个较小的初始值逐渐增大到一个指定的峰值（通常比传统的最大学习率大得多）。

后期逐渐减小学习率：在接下来的训练中，学习率从峰值逐步减小到一个非常小的值（接近 0），帮助模型更稳定地找到局部最优。

2.策略中的关键参数

optimizer：传递给 OneCycleLR 的优化器对象，用于在每一步更新时调整学习率。

max_lr：训练期间的最高学习率。一般来说，该值是通常学习率的 2 到 10 倍。调度器会逐渐将学习率提高到 max_lr，然后再逐步降低，直到训练结束。

epochs：总的训练周期数，即训练的轮次。指定该值可以帮助 OneCycleLR 在整个训练过程中适应学习率变化的周期性。

steps_per_epoch：每个周期的更新步数，即每轮训练中的批次数目。设置正确的步数让学习率能够平滑变化，不会出现突变。

代码实现思路

OneCycleLR 的实现需要在 CNN 模型、优化器和学习率调度器中进行配置。以下是具体的实现步骤：

定义模型和优化器：使用优化器 Adam，采用与 Base_CNN 相同的模型结构。

设置 OneCycleLR 调度器：在优化器之后，实例化 OneCycleLR 调度器，配置合适的 max_lr、steps_per_epoch 等参数。

训练循环中的学习率更新：在每个训练步中，按周期更新学习率，并监控其变化过程。通过 sched.step()逐步更新学习率

```
    # 定义 OneCycleLR 学习率调度器，用于动态调整学习率，使其在训练早期快速上升
sched = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,  # 要应用调度的优化器
    max_lr=0.01,  # 最大学习率，即训练过程中的峰值学习率
    epochs=epochs,  # 训练的总周期数
    steps_per_epoch=len(train_dataloader)  # 每个周期的更新步数
)
    # 调度器步进：更新学习率，遵循 OneCycleLR 策略
sched.step()
```

学习率曲线分析

在训练过程中，OneCycleLR 的学习率会遵循一个"单周期"曲线。将学习率的变化情况绘制出来，以更直观地看到以下几个阶段：

前期加速：学习率快速增大到最大值。

峰值探索：在较大学习率下探索参数空间。

后期收敛：学习率逐步衰减到一个很小的值，帮助模型稳定收敛

Onecyclelr_CNN 总结

相较于 Base_CNN,，Onecyclelr_CNN 通过更复杂的数据增强策略和动态学习率调整机制，显著提高了训练效率和模型性能。这种差异在复杂任务和大数据集上尤其明显，能够有效提升深度学习模型的实际应用效果。

2.3 SE_CNN 模型

SE_CNN 是一种在卷积神经网络中引入残差连接以及 Squeeze-and-Excitation 机制的模型，与 BaseCNN 和 Onecyclelr_CNN 相比，具有更强的特征重标定能力和更深层的网络结构。

残差结构的引进

在构建深度神经网络时，深度网络的训练往往会受到梯度消失和过拟合的影响。为了解决这一问题，残差连接（Residual Connection）被引入到 SE_CNN 中，使网络能够学习更深层次的特征。

残差结构的概述：

1.残差块: 通过引入短路连接，允许输入直接跳过某些层，从而有效缓解了深层网络中的梯度消失问题。

2.优势: 这种结构使得网络能够更容易地学习恒等映射，有助于提高训练效率和模型性能。

残差结构的代码分析：

```
self.res1 = nn.Sequential(conv_block(64, 64), conv_block(64, 64))   # 第一个残差块
#其余代码略
# 残差连接部分：output = self.res1(out) + out
out = self.res1(out) + out   # 第一个残差连接
```

在代码中，a.定义残差块：self.res1 = nn.Sequential(conv_block(64, 64), conv_block(64, 64))，即两个 conv_block 的序列，它们的输入和输出通道均为 64。

b.残差相加：out = self.res1(out)：将当前的 out 传递到 res1 中，即输入特征图经过两个卷积块的处理，输出为 res1_out。+ out：这个操作是残差连接，它将 res1_out 与原始输入 out 进行相加。这样，经过两个卷积层的输出（res1_out）与输入（out）相加，得到新的 out。这使得网络在学习时可以更容易地学习到恒等映射，从而减少梯度消失的问题。

Squeeze-and-Excitation（SE）模块的引入

SE 模块通过引入通道注意力机制，增强了卷积神经网络的能力，能够动态调整特征通道的重要性。

SE 模块的概述：

1.功能: SE 模块自适应地为每个通道分配权重，从而提升模型对关键特征的响应能力。

2.结构: 通过全局平均池化生成每个通道的描述符，再通过全连接层计算通道权重。

SE 模块的代码分析：

以下是 SE_CNN 中 SE 模块的实现示例

```
# 定义 Squeeze-and-Excitation (SE) 模块，用于提高模型关注度
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction_ratio=16):
        super(SEBlock, self).__init__()
        # 全局平均池化层，将空间信息压缩成单个值
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
        # 两层全连接网络，逐层减少和增加通道数。降维再升维，学习各通道的“重要性”权重
        self.fc = nn.Sequential(
            nn.Linear(in_channels, in_channels // reduction_ratio, bias=False),  # 降维：减少通道数
            nn.ReLU(inplace=True),  # 激活函数
            nn.Linear(in_channels // reduction_ratio, in_channels, bias=False),  # 升维：还原通道数
            nn.Sigmoid()  # 使用 Sigmoid 将权重值限制在 (0,1) 范围内
        )
```

关键代码分析：

Squeeze 操作：使用 AdaptiveAvgPool2d 对输入进行全局平均池化，生成通道描述符。

Excitation 操作：通过两个全连接层生成通道权重，使用 Sigmoid 函数将输出限制在 [0, 1] 之间。

重标定：最后将权重应用于原始特征图，实现特征的自适应重标定

SE 模块总结

通过引入残差结构和 SE 模块，SE_CNN 提高了模型的表达能力和训练效率。残差结构通过短路连接缓解了梯度消失问题，使得网络能够构建更深的结构，而 SE 模块通过动态调整特征通道的重要性进一步增强了特征提取能力。这些创新使得 SE_CNN 在图像分类等任务中展现出优越的性能。

2.4 模型测试结果可视化及对比

单模型效果图

Onecyclelr_CNN 效果如图 2-1 所示：

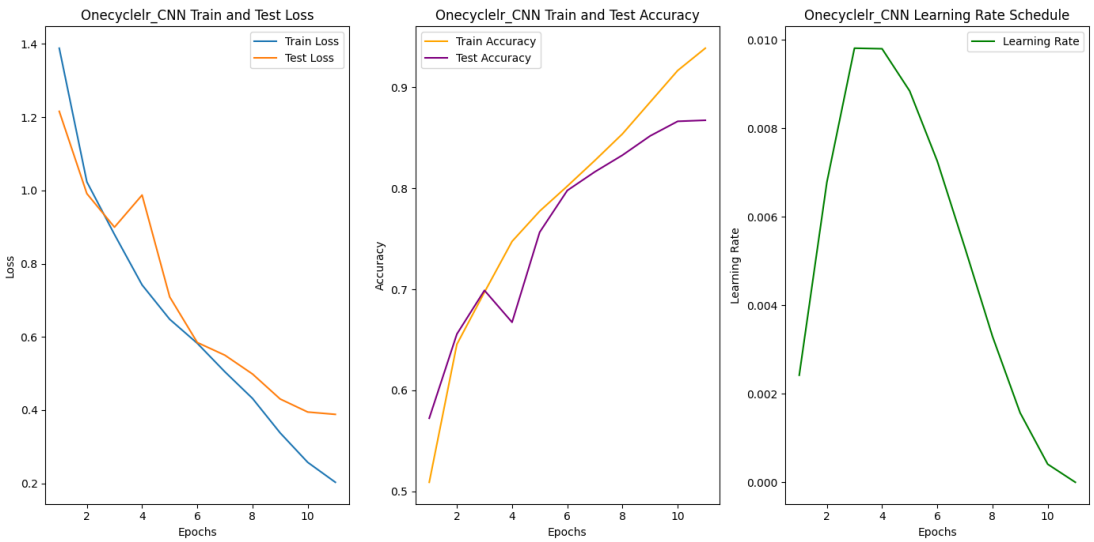


图 2-1

Base_CNN 效果如图 2-2 所示：

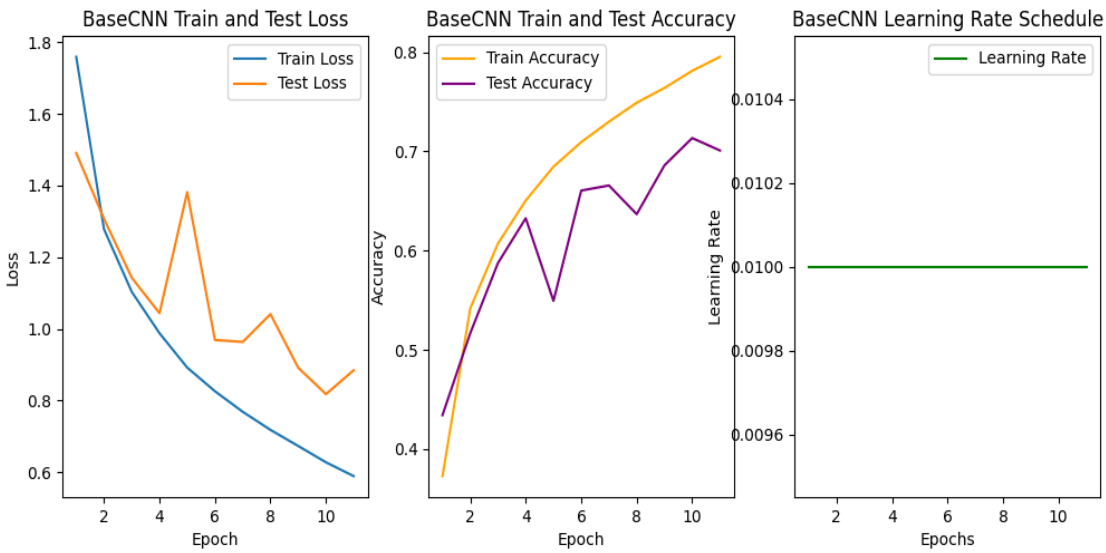


图 2-2

SE_CNN 效果如图 2-3 所示：

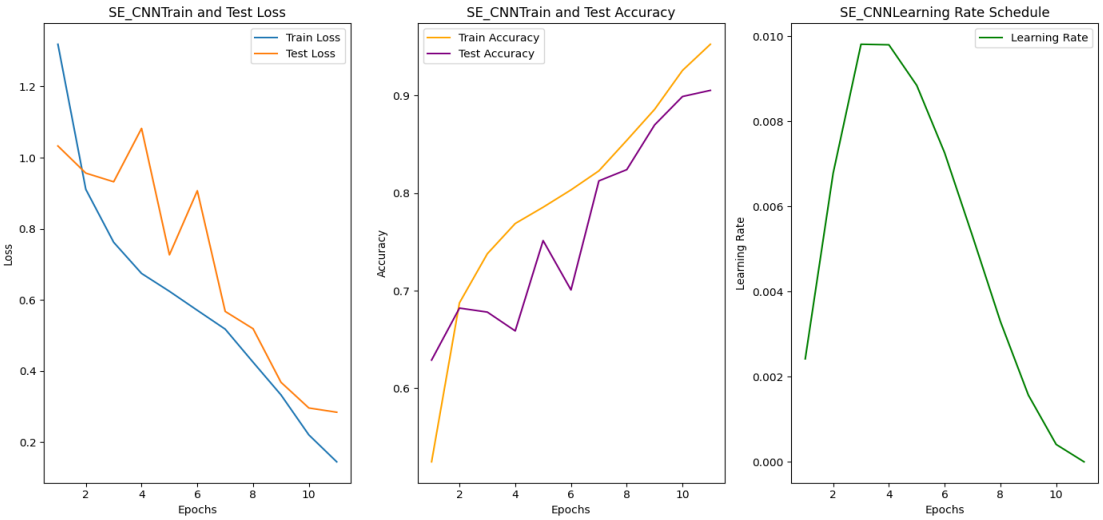


图 2-3

三模型效果对比图

精确度对比如图 2-4 所示:

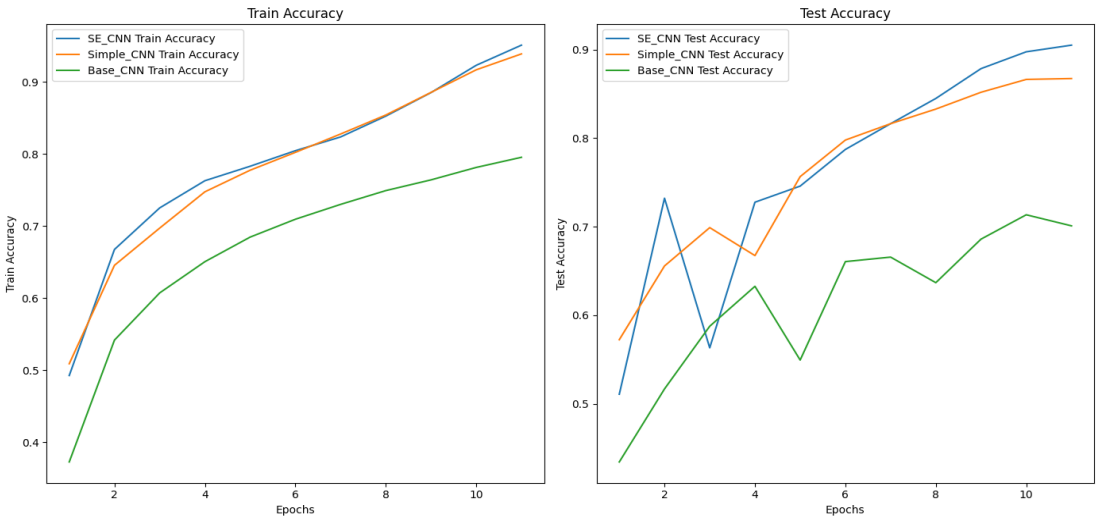


图 2-4

在训练准确率子图中，随着训练轮次的增加，三种模型的准确率均呈现上升趋势。其中，SE_CNN 表现尤为突出，其准确率曲线不仅稳步上升，而且在大部分轮次中均高于 Simple_CNN 和 Base_CNN。Simple_CNN 的训练准确率紧随其后，而 Base_CNN 虽然相对较低，但也显示出逐步提升的趋势。

测试准确率子图则揭示了模型在未见过的数据上的表现。与训练准确率类似，SE_CNN 在测试准确率方面也表现最佳，其曲线在大部分轮次中保持领先。Simple_CNN 的测试准确率次之，而 Base_CNN 的测试准确率相对较低。值得注意的是，测试准确率曲线在某些轮次中出现了波动，这可能反映了模型在这些阶段的不稳定性。

综上所述，SE_CNN 在训练和测试阶段均展现出最优的性能，Simple_CNN 次之，而 Base_CNN 虽然表现相对较弱，但也显示出一定的提升潜力。这些图表为模型的选择和进一步优化提供了直观的参考依据。

损失值对比如图 2-5 所示:

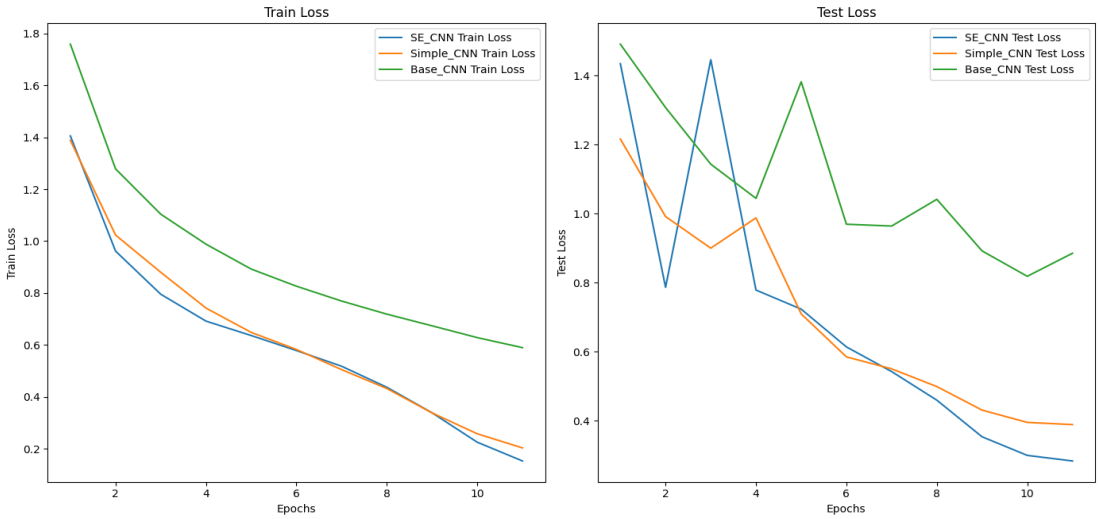


图 2-5

在训练损失方面，SE_CNN 和 Simple_CNN 的表现较为接近，且都优于 Base_CNN，说明前两者在训练集上的学习效果更好。而在测试损失方面，Simple_CNN 的表现最为稳定且最低，表明其泛化能力可能更强。SE_CNN 的测试损失虽然初期波动较大，但总体趋势也是下降的，表现次之。相比之下，Base_CNN 的测试损失在整个训练过程中都相对较高。

3.LSTM 神经网络文本处理

用于处理文本数据，训练一个 LSTM 模型来进行文本分类，并评估其性能。以下是逐步解释：

3.1 主要第三方库

NumPy：用于数据预处理和高效多维数组运算。

PyTorch：定义 LSTM 模型、创建数据加载器、训练及进行前向和反向传播的深度学习库。

Gensim：训练 Word2Vec 模型，将文本转换为向量表示的库。

Scikit-learn：提供标签编码、准确率计算和均方误差等机器学习工具和算法。

Matplotlib：用于绘制训练和测试过程中的图表和数据可视化的库。

TensorDataset 和 DataLoader：将数据包装成 tensor 对，并按批次加载数据以支持训练和评估。

Jieba：用于中文分词

3.2 数据加载和预处理

加载训练和测试数据，数据格式为标签和内容的分隔文本文件。

加载数据:使用 pandas 库的 read_csv 函数从指定路径加载训练和测试数据。train_data 和 test_data 分别存储训练和测试数据集，其中包含标签（label）和内容（content）。

获取内容和标签:从加载的数据集中提取出文本内容和对应的标签，分别存储在 train_texts、train_labels、test_texts 和 test_labels 中。

分词:定义了一个 tokenize 函数，使用 jieba 分词库对给定的文本进行分词，返回一个词列表。

```
def tokenize(text):
    return list(jieba.cut(text))
```

去除停用词:定义了一个 drop_stopword 函数，该函数首先尝试从指定路径加载停用词列表，然后对于给定的数据，去除每个文本中的停用词。停用词列表存储在. . /stopwords-master/cn_stopwords. txt 文件中，每个停用词占一行。

```
def drop_stopword(datas):
```

加载和保存数据:load_saved_data 函数用于从指定路径加载已预处理并保存的数据。数据按行读取，每行数据被分割成列表并存储在二维列表中。save_data 函数用于将给定的数据（二维列表）保存到指定路径。每行数据被转换为逗号分隔的字符串并写入文件。

数据预处理流程:代码首先检查已预处理的数据（分词并去除停用词后）是否已存在于指定路径（train_data_path 和 test_data_path）。如果已存在，则直接加载这些数据。如果不存在，则对原始文本数据进行分词和去除停用词的处理，然后将处理后的数据保存到指定路径。

3.3 构建 LSTM 模型

定义一个 LSTM 模型类，继承自 nn.Module。这个模型包含一个 LSTM 层和一个全连接层。

LSTM 层处理输入序列，并返回输出和隐藏状态。最后一个隐藏状态通过全连接层来产生最终输出。

```
def forward(self, x):
    x, (hn, cn) = self.lstm(x)
    x = hn[-1]  # 取最后一个时间步的输出
    x = self.fc(x)
    return x
```

self.lstm(x)执行 LSTM 层的前向传播，返回两个输出：x（所有时间步的输出）和(hn, cn)（隐藏状态和细胞状态）。由于我们只关心最后一个时间步的输出，所以忽略了 x。

hn 是一个元组，包含两个张量：最后一个时间步的隐藏状态和最后一个时间步的细胞状态。由于我们只关心隐藏状态，所以通过 hn[-1]取出最后一个隐藏状态（注意：这里的-1 是指取元组的最后一个元素，而不是时间步的最后一个元素；在 LSTM 中，hn 的形状通常是(num_layers, batch_size, hidden_dim)，其中 num_layers 是 LSTM 层的层数）。

self.fc(x)将最后一个隐藏状态通过全连接层映射到输出维度 output_dim，得到最终的预测结果。

3.4 文本向量化

要将文本数据转换为向量表示，并用于机器学习或深度学习模型中，Word2Vec 是一种非常有效的方法。Word2Vec 是一种语言建模和特征学习技术，在大量文本数据上训练后，能够将单词映射到高维空间中的向量，这些向量能够捕捉到单词间的语义关系。

需要训练一个 Word2Vec 模型。这通常涉及收集大量相关的文本数据，这些数据可以是文章、评论、社交媒体帖子等。使用 gensim 库中的 Word2Vec 类，我们可以很方便地训练模型。在训练过程中，模型会学习每个单词的向量表示，这些向量在后续的文本转换中将被使用。

定义一个函数 text_to_vector，这个函数接收一个文本列表、一个已经训练好的 Word2Vec 模型，以及一个指定的最大序列长度 max_length。函数的目的是将每个文本转换为向量表示，同时处理序列长度不一致的问题。

在函数内部，我们遍历文本列表中的每个文本。对于每个文本，我们检查其中的每个单词是否存在于 Word2Vec 模型的词汇表中。如果存在，我们就从模型中检索该单词的向量。这样，我们就得到了一个由单词向量组成的列表，这个列表的长度取决于文本中单词的数量。

然而，由于不同的文本可能包含不同数量的单词，我们需要对序列进行填充和截断，以确保它们具有相同的长度。这是通过将序列截断到 max_length 指定的长度，并在需要时用零向量填充来实现的。零向量是一个与 Word2Vec 向量具有相同维度，但所有元素都为零的向量。

最后，我们将处理后的向量列表转换为 NumPy 数组，并进一步转换为 PyTorch 张量。这个张量的形状是 (num_samples, max_length, embedding_dim)，其中 num_samples 是文本列表中的样本数，max_length 是我们指定的最大序列长度，embedding_dim 是 Word2Vec 向量的维度。

3.5 标签编码和数据准备

使用 LabelEncoder 将字符串标签编码为整数。

将编码后的标签转换为 PyTorch 张量，并确保它们是 long 类型，以便与损失函数兼容。

准备训练和测试数据的数据加载器（DataLoader），以便于批处理和随机排序。

3.6 训练模型

设置 LSTM 模型的超参数，包括嵌入维度（与 Word2Vec 向量大小匹配）、隐藏维度和输出维度（标签的唯一类数量）。

实例化 LSTM 模型，并将其移动到适当的设备上（CPU 或 GPU）。

定义损失函数（交叉熵损失）和优化器（Adam 优化器）。

训练模型：在指定的 epoch 数内迭代训练数据。对于每个批次，计算模型输出，计算损失，执行反向传播，并更新模型参数。

函数体

初始化性能指标列表：

这些列表用于存储每个 epoch 的训练和测试准确率、均方误差（MSE）和损失。

训练循环：

对于每个 epoch，将模型设置为训练模式（这会影响某些层，如 Dropout 和 BatchNorm 的行为）。

批次处理：

遍历训练数据加载器中的每个批次。每个批次包含一批输入数据和相应的标签。

前向传播、损失计算、反向传播和参数更新：

将输入和标签移动到指定的设备（如 GPU）。

计算模型输出。

使用预定义的损失函数计算损失。

清零梯度。

执行反向传播以计算梯度。

更新模型参数。

```
for inputs, labels in train_loader:
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    loss = criterion(outputs, labels)   # 计算损失
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

跟踪训练和预测：

累加损失以计算 epoch 的平均损失。

使用 torch. max 获取预测类别（输出中的最大值的索引）。

将预测和真实标签转换为 NumPy 数组并扩展到列表中。

计算训练集上的准确率和 MSE：

使用 accuracy_score（来自 sklearn. metrics）计算准确率。

注意：计算 MSE 通常不适用于分类任务，因为 MSE 是衡量连续值之间差异的标准，而分类标签是离散的。这里计算 MSE 可能是为了某种特定的目的或实验，但在常规分类评估中并不常见。

记录训练集的指标：

```
# 记录训练集的指标
train_accuracies.append(train_accuracy)
train_mses.append(train_mse)
train_losses.append(epoch_loss / len(train_loader))  # 计算平均损失
```

测试过程：

调用 evaluate_model 函数（未在代码中定义）来评估模型在测试集上的性能。该函数应返回测试准确率、测试 MSE、测试标签列表和测试预测列表。

打印进度信息：

打印当前 epoch 的训练损失、训练 MSE、训练准确率、测试准确率和测试 MSE。

返回性能指标：

```
return train_accuracies, train_mses, train_losses,
test_accuracies, test_mses, all_test_labels, all_test_preds
```

3.7 评估模型

在测试数据集上评估模型性能。将模型设置为评估模式，这将禁用 dropout 和批量规范化。

迭代测试数据加载器，计算预测，并收集真实标签和预测标签。

使用 accuracy_score 计算并打印模型的准确率。

函数定义：

model: 这是要评估的神经网络模型。

test_loader: 这是一个 DataLoader，它包含测试数据集。DataLoader 通常包括数据集的批次、打乱数据、并行加载等。

output_dim: 这是模型输出的维度，通常等于分类任务中的类别数。

评估模式：

将模型设置为评估模式。在评估模式下，像 dropout 和批量规范化这样的层会按照训练时的设定来运行（例如，dropout 在评估时不会丢弃任何单元）。

初始化列表：

初始化三个列表来存储所有的预测、真实标签和输出概率。

禁用梯度计算：

在这个上下文管理器内部，所有的计算都不会跟踪梯度，这有助于减少内存消

耗并加速计算，因为在评估阶段我们不需要梯度。

迭代测试数据:

迭代 test_loader 中的每一批数据。

将输入数据和标签移动到指定的设备（如 GPU）。

通过模型前向传播计算输出。

使用 torch.max 找到每个样本的预测类别（即概率最高的类别）。

将预测和真实标签转换为 NumPy 数组，并扩展到对应的列表中。

将输出概率也添加到 all_probs 列表中。

计算准确率:

使用 Scikit-learn 的 accuracy_score 函数计算准确率，即正确预测的比例。

计算均方误差 (MSE)

使用 np.vstack 将 all_probs 列表中的所有概率数组堆叠成一个二维数组。

使用 np.eye(output_dim)[all_labels]创建一个独热编码数组，其中 np.eye(output_dim)生成一个 output_dim x output_dim 的单位矩阵，然后通过索引 all_labels 选择对应的独热编码行。

使用 Scikit-learn 的 mean_squared_error 函数计算独热编码的真实标签和预测概率之间的均方误差。

返回结果:

函数返回准确率、均方误差、所有真实标签和所有预测标签。

3.8 测试数据结果及可视化

运用一个 Python 脚本，旨在自动化地将指定的 Python 代码文件（位于特定目录下）转换为图像文件，并运行这些代码文件，然后将它们的输出也转换为图像文件。

函数 save_output_as_image

将脚本的输出文本（包括用户输入提示和实际的输出内容）转换为图像文件。

在图像的第一行添加脚本路径，根据用户输入和提示性语句处理输出文本，最后添加 "Process finished with exit code 0"。

同样，计算图像尺寸并在黑色背景上绘制白色文字。

函数 get_input_prompt

使用正则表达式从脚本代码中提取所有 input() 函数的提示字符串。

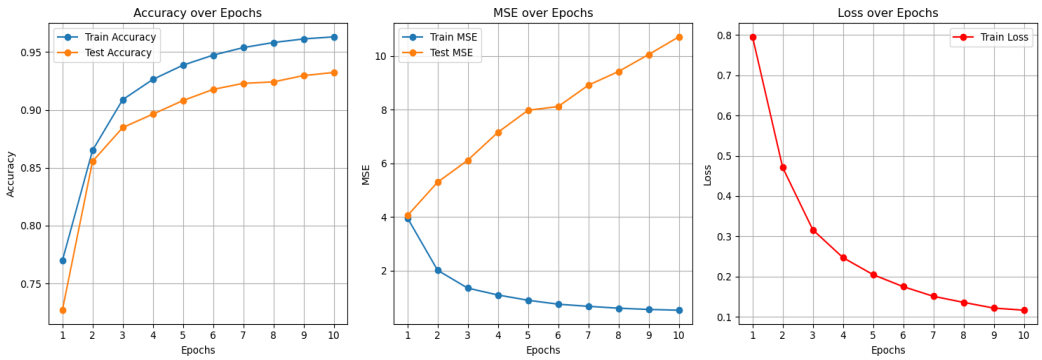
返回一个包含所有提示字符串的列表。

函数 run_script_with_input

使用 subprocess 模块运行指定的 Python 脚本，并通过 stdin 传递用户输入。

捕获脚本的标准输出和标准错误，并将它们组合成一个字符串返回。

运行结果如下：



左图描绘了准确率随训练轮次（epochs）的变化。随着训练的进行，无论是训练集还是测试集，准确率均呈现稳步上升的趋势。这表明模型在不断地学习并优化其预测能力。然而，训练集的准确率始终略高于测试集，这可能暗示了模型在训练集上存在一定的过拟合风险。

中图展示了均方误差（MSE）随训练轮次的变化。从图中可以看出，训练集和测试集的 MSE 均随着训练轮次的增加而逐渐降低。这反映了模型在训练过程中逐渐学习到了数据的内在特征，预测误差在逐渐减小。但同样地，训练集的 MSE 始终低于测试集，进一步印证了过拟合的可能性。

右图呈现了训练损失随训练轮次的变化。随着训练的进行，训练损失呈现出明显的下降趋势。这表明模型在不断地优化其参数，以减少预测误差，提高预测准确性。

三. 项目代码

1. BP 神经网络项目代码

1.1 结构图

```
-PBHB
  │  log.txt
  │  main.py
  │  model.py
  │
  ├─datas
  │      boston.dat
  │
  ├─imgs
  │      local_loss_img_path.png
  │      loss_curve.png
  │      Regression_graph.png
  │      scatter_diagram.png
  │      value_change.png
  │
  ├─model
  │      boston.pt
  │
  └─__pycache__
         findBestParameter.cpython-311.pyc
         model.cpython-311.pyc
```

main. py 调用 model. py 中的函数来定义和训练神经网络模型。

main. py 读取 datas/boston. dat 中的数据，用于训练和测试。

main. py 在训练过程中生成并保存图像文件到 imgs 文件夹。

model. py 可能调用 findBestParameter. py（通过其编译后的文件）来优化模型参数。

训练完成后，模型权重保存为 model/boston. pt，供后续使用。

1.2 完整代码

model.py

```
import torch
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
from torch.utils.data import DataLoader, TensorDataset
import matplotlib.pyplot as plt

class Boston(torch.nn.Module):
    def __init__(self, feature_col) -> None:
        super().__init__()
        # 构建神经网络
        self.fc = torch.nn.Sequential(
            torch.nn.Linear(in_features=feature_col, out_features=64),
            torch.nn.ReLU(),
            torch.nn.Linear(in_features=64, out_features=32),
            torch.nn.ReLU(),
            torch.nn.Linear(in_features=32, out_features=16),
            torch.nn.ReLU(),
            torch.nn.Linear(in_features=16, out_features=1),
            # torch.nn.Linear(in_features=feature_col, out_features=16),
            # torch.nn.ReLU(),
            # torch.nn.Linear(16, 1),
            # torch.nn.Sigmoid()
        )

    def getData(self, data_url, feature_col, test_size, batch_size):
        raw_df = pd.read_csv(data_url, sep="\s+", header=None)
        x_data = raw_df.iloc[:, :feature_col].values
        y_data = raw_df.iloc[:, -1].values
```

```
        # 归一化
        scalar = StandardScaler().fit(x_data)
        x_data = scalar.transform(x_data)

        # 拆分数据为训练集和测试集
        x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,
test_size=test_size, random_state=56)

        x_train = torch.from_numpy(x_train.astype(np.float32))
        y_train = torch.from_numpy(y_train.astype(np.float32)).view(-1, 1)
        self.x_test = torch.from_numpy(x_test.astype(np.float32))
        self.y_test = torch.from_numpy(y_test.astype(np.float32)).view(-1, 1)

        # 封装进数据集加载器
        self.train_data = TensorDataset(x_train, y_train)
        self.train_loader = DataLoader(dataset=self.train_data,
batch_size=batch_size, shuffle=True, drop_last=True)

# 前向传递函数
def forward(self, x):
        return self.fc(x)

def trainData(self, epochs, learn_rate, number_of_sig_losser):
        # 使用 Adam 优化器
        # optimizer = torch.optim.SGD(self.parameters(), lr=1e-3)
        optimizer = torch.optim.Adam(self.parameters(), lr=learn_rate)
        loss_fun = torch.nn.MSELoss()
        self.train()
        self.loss_values = []
        self.epochs = epochs
```

```
        for epoch in range(epochs):
            epoch_loss = 0
            for step, (x, y) in enumerate(self.train_loader):
                loss = loss_fun(self(x), y) #  计算损失
                loss.backward() #  反向传播
                optimizer.step() #  步进更新参数
                optimizer.zero_grad() #  梯度清零
                epoch_loss += loss.item()

            self.loss_values.append(epoch_loss / len(self.train_loader))

            if (epoch + 1) % 100 == 0:
                print(f"epoch:{epoch + 1}/{epochs}, loss:{epoch_loss /
len(self.train_loader)}")

        self.max_loss = max(self.loss_values)
        print(f"Final average loss:: {sum(self.loss_values[-
number_of_sig_losser:]) / number_of_sig_losser:.4f}")
        return sum(self.loss_values[-number_of_sig_losser:]) /
number_of_sig_losser

def drawCurve(self, save_model_path, save_img_path, local_loss_img_path,
number_of_sig_losser):
        torch.save(self.state_dict(), save_model_path)

        plt.figure(figsize=(10, 5))
        plt.plot(range(self.epochs), self.loss_values, label='Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title('Loss Value Change')
        plt.legend()
```

```
        plt.legend()
        plt.savefig(save_img_path)
        # plt.show()
        plt.close()

        plt.figure(figsize=(10, 5))
        plt.plot(range(self.epochs - number_of_sig_losser, self.epochs),
self.loss_values[-number_of_sig_losser:], label='Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Local Loss')
        plt.title('Local Loss Value Change')
        plt.legend()
        plt.savefig(local_loss_img_path)
        # plt.show()
        plt.close()

def calMSE(self, scatter_diagram_path, regression_graph_path):
        self.eval()
        with torch.no_grad(): # 评估模式禁止梯度计算
                y_pred = self(self.x_test)  # 获取预测结果
                mse = torch.nn.functional.mse_loss(y_pred, self.y_test)  # 计算
均方误差
                print(f'The Test MSE: {mse.item():.4f}')

                # 测试散点图
                indices = range(len(self.y_test))
                plt.figure(figsize=(8, 6))
plt.scatter(indices, y_pred.numpy(), color='blue', label='Predicted Values')
                plt.scatter(indices, self.y_test.numpy(), color='red', label='True
Values')

                plt.xlabel('Data Point Index')
```

```
            plt.xlabel('Data Point Index')
            plt.ylabel('Values')
            plt.title('Predicted vs True Values')
            plt.legend()
            plt.grid(True)
            plt.savefig(scatter_diagram_path)
            plt.close()

            # 回归散点图
            plt.figure(figsize=(8, 6))
            plt.scatter(self.y_test.numpy(), y_pred.numpy(), color='blue',
label='Predicted Values')

            # 绘制理想线
            plt.plot([self.y_test.min(), self.y_test.max()], [self.y_test.min(),
self.y_test.max()],
             color='red', linewidth=2, label='Ideal Fit')  # 理想线

# 设置图表的标签和标题
            plt.xlabel('True Values')
            plt.ylabel('Predicted Values')
            plt.title('Predicted vs True Scatter Plot')
            plt.legend()

            # 添加网格线
            plt.grid(True)

            # 保存图像
            plt.savefig(regression_graph_path)
            plt.close()

            return mse.item()
```

main.py

```
from model import Boston
import sys
import matplotlib.pyplot as plt
import time
from sklearn.preprocessing import MinMaxScaler
import gc

#  参数字典
# Best: epochs:5000, learn_rate:4/3 * 1e-3, batch_size:8
parameter = {
        "find_best_parameter" : [9, 5, 9],
        "feature_col" : 13,
        "data_file_path" : "./datas/boston.dat",
        "epochs" : 5000,
        "learn_rate" : 4e-3,
        "number_of_sig_losses" : 30,
        "test_size" : 0.407,
        "batch_size" : 16,
        "save_model_path" : "./model/boston.pt",
        "save_img_path" : "./imgs/loss_curve.png",
        "local_loss_img_path" : "./imgs/local_loss_img_path.png",
        "scatter_diagram_path" : "./imgs/scatter_diagram.png",
        "regression_graph_path" : "./imgs/Regression_graph.png",
        "value_change_img_path" : "./imgs/value_change.png",
        "train_times" : 10,    #  注意此处不能为1 或0 !!!
        "train_model" : 2    # 1:单次 2:多次 3:寻找最优参数
}

#  初始化-训练-评估
def modelTest(epochs=parameter["epochs"],
learn_rate=parameter["learn_rate"], batch_size=parameter["batch_size"]):
```

```
    model = Boston(parameter["feature_col"])

    model.getData(parameter["data_file_path"],
                  parameter["feature_col"],
                  parameter["test_size"],
                  batch_size)

    value_loss = model.trainData(epochs,
                                 learn_rate,
parameter["number_of_sig_losses"])

    value_MSE = model.calMSE(parameter["scatter_diagram_path"],
                             parameter["regression_graph_path"])

    model.drawCurve(parameter["save_model_path"],
                    parameter["save_img_path"],
                    parameter["local_loss_img_path"],
                    parameter["number_of_sig_losses"])

    del model
    gc.collect()  # 强制进行垃圾回收

    return value_MSE, value_loss

# 获取最优及平均MSE 和 loss
def theBestMSEandLoss():
    ls_MSE = []
    ls_loss = []
    min_MSE = sys.maxsize
    min_loss = sys.maxsize
    max_MSE = -sys.maxsize
```

```
    max_loss = -sys.maxsize
    times = parameter["train_times"]

    for i in range(times):
        print(f"--------------------{i + 1}----------------------")
        time_MSE, time_loss = modelTest()
        min_MSE = min(min_MSE, time_MSE)
        min_loss = min(min_loss, time_loss)
        max_MSE = max(max_MSE, time_MSE)
        max_loss = max(max_loss, time_loss)

        ls_MSE.append(min_MSE)
        ls_loss.append(min_loss)

    aver_MSE = sum(ls_MSE) / len(ls_MSE)
    aver_loss = sum(ls_loss) / len(ls_loss)

    # 归一化 MSE 和 Loss 值
    norm_MSE = [(mse - min_MSE) / (max_MSE - min_MSE) for mse in
ls_MSE]
    norm_loss = [(loss - min_loss) / (max_loss - min_loss) for loss in
ls_loss]

    # 绘制归一化后的 MSE 和 Loss 值的变化散点图
    plt.figure(figsize=(10, 5))
    plt.scatter(range(1, times + 1), norm_MSE, label='Normalized MSE',
color='blue')
    plt.scatter(range(1, times + 1), norm_loss, label='Normalized Loss',
color='red', alpha=0.5)
    plt.xlabel('Iteration')
    plt.ylabel('Normalized Value')
    plt.title('Normalized MSE and Loss Value Change')
```

```
        plt.legend()

        # 保存图片到指定路径
        plt.savefig(parameter["value_change_img_path"])
        # plt.show()
        plt.close()

        print()
        print(f"Run {times} times: ")
        print(f"The min loss:{min_loss:.4f}")
        print(f"The average loss:{aver_loss:.4f}")
        print(f"The min MSE:{min_MSE:.4f}")
        print(f"The average MSE:{aver_MSE:.4f}")

# 测试 epochs, learn_rate, batch_size
def findBestParameter(times):
        min_loss = sys.maxsize
        min_MSE = sys.maxsize
        min_loss_MSE = sys.maxsize
        min_MSE_loss = sys.maxsize
        min_loss_par = []
        min_MSE_par = []
        epoch_unit = 1000
        learn_rate_unit = 1e-3

        for i in range(1, times[0] + 1, 2): # epochs i * epoch_unit
                for j in range(3, times[1] + 1): # batch_size pow(2, j)
                        for k in range(1, times[2] + 1): # learn_rate k *
learn_rate_unit
                                print(f"----epoch:{i * epoch_unit}--batch_size:{pow(2,
j)}--lr:{k * learn_rate_unit}----")
```

```
                        value_MSE, value_loss = modelTest(i * epoch_unit,
k * learn_rate_unit, pow(2, j))

                    if min_loss > value_loss:
                        min_loss, min_loss_MSE = value_loss, value_MSE
                        min_loss_par = [i * epoch_unit, k * learn_rate_unit,
pow(2, j)]

                    if min_MSE > value_MSE:
                        min_MSE, min_MSE_loss = value_MSE,
value_loss
                        min_MSE_par = [i * epoch_unit, k *
learn_rate_unit, pow(2, j)]

    print(f"lowest loss: ({min_loss:.4f}, {min_loss_MSE:.4f}):
{min_loss_par}")
    print(f"lowest MSE:    ({min_MSE_loss:.4f}, {min_MSE:.4f}):
{min_MSE_par}")

if __name__ == "__main__":
    start_time = time.time()

    if parameter["train_model"] == 1:
        modelTest()
    elif parameter["train_model"] == 2:
        theBestMSEandLoss()
    else:
        findBestParameter(times=parameter["find_best_parameter"])

    end_time = time.time()
    print(f"Take time:{end_time - start_time:.4f} s")
```

2. CNN 神经网络项目代码

2.1 结构图

```
-CIFAR10（1）
  ├─data
  │  └─MNIST
  │      └─raw
  ├─Model
  │  │  BaseCNN.py
  │  │  Compare_SE_One.py
  │  │  Onecyclelr_CNN.py
  │  │  SE_CNN.py
  │  │
  │  └─data
  │      │  cifar-10-python.tar.gz
  │      │
  │      └─cifar-10-batches-py
  │              batches.meta
  │              data_batch_1
  │              data_batch_2
  │              data_batch_3
  │              data_batch_4
  │              data_batch_5
  │              readme.html
  │              test_batch
  │
  ├─单模型效果图
  │      BaseLine.png
  │      Onecyclelr_CNN.png
  │      SE_CNN.png
  │
  ├─模型对比图
  │      ACC.png
  │      lEARN.png
  │      LOSS.png
  │
  └─演示视频
          演示1 (online-video-cutter.com)(1).mp4
```

首先会解压数据集文件 cifar-10-python.tar.gz，并加载训练和测试数据。

接着，根据研究或实验需求，运行 BaseCNN.py、Onecyclelr_CNN.py、SE_CNN.py 来训练模型。

在训练过程中，使用 Compare_SE_One.py 生成模型对比图

2.2 完整代码

```
# CIFAR-10 数据集及其增强
# stats 变量存储了标准化时的均值和标准差
stats = ((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
train_data = torchvision.datasets.CIFAR10("./data", train=True,
transform=torchvision.transforms.Compose([
    torchvision.transforms.ColorJitter(0.5),  # 随机调整图像亮度、对比度、饱和度
    torchvision.transforms.RandomHorizontalFlip(),  # 随机水平翻转
    torchvision.transforms.ToTensor(),  # 将图像转化为张量
    torchvision.transforms.Normalize(*stats)  # 应用均值和标准差进行标准化
]))
# 定义测试集，仅进行标准化处理
test_data = torchvision.datasets.CIFAR10("./data", train=False,
transform=torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(*stats)
]))
train_dataloader = DataLoader(train_data, batch_size=128, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=128)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")#使用gpu 加速

# 定义 Squeeze-and-Excitation (SE) 模块，用于提高模型关注度
class SEBlock(nn.Module):
    def __init__(self, in_channels, reduction_ratio=16):
        super(SEBlock, self).__init__()
        # 全局平均池化层，将空间信息压缩成单个值
        self.global_avg_pool = nn.AdaptiveAvgPool2d(1)
        # 两层全连接网络，逐层减少和增加通道数。降维再升维，学习各通道的“重要性”权重
```

```
                self.fc = nn.Sequential(
                    nn.Linear(in_channels, in_channels //
reduction_ratio, bias=False),  # 降维：减少通道数
                    nn.ReLU(inplace=True),  # 激活函数
                    nn.Linear(in_channels // reduction_ratio,
in_channels, bias=False),  # 升维：还原通道数
                    nn.Sigmoid()  # 使用 Sigmoid 将权重值限制在
(0,1) 范围内
                )

# 定义改进后的 SE-CNN 模型
class SEModel(nn.Module):
    def __init__(self):
        super(SEModel, self).__init__()
        # 定义每层卷积和残差块
        self.conv1 = conv_block(3, 32)  # 输入层，通道数从 3 扩展到
32
        self.conv2 = conv_block(32, 64, pool=True)  # 第二层，通道数
扩展至 64，池化降低分辨率
        self.res1 = nn.Sequential(conv_block(64, 64), conv_block(64, 64))
# 第一个残差块
        self.conv3 = conv_block(64, 128)  # 第三层
        self.conv4 = conv_block(128, 256, pool=True)  # 第四层
        self.res2 = nn.Sequential(conv_block(256, 256), conv_block(256,
256))  # 第二个残差块
        self.conv5 = conv_block(256, 512)  # 第五层
        self.conv6 = conv_block(512, 1024, pool=True)  # 第六层
        self.res3 = nn.Sequential(conv_block(1024, 1024),
conv_block(1024, 1024))  # 第三个残差块
        # 最后线性层，用于分类输出
        self.linear1 = nn.Sequential(nn.MaxPool2d(4),  # 池化将分辨率
降到最低
```

```
                                nn.Flatten(),  # 展平张量用于全连接层
                                nn.Dropout(0.2),  # Dropout 防止过拟合
                                nn.Linear(1024, 10))  # 分类为10 类

def forward(self, x):
    # 执行前向传播过程
    out = self.conv1(x)
    out = self.conv2(out)
    out = self.res1(out) + out  # 第一个残差连接,将残差块的输出与输
入相加
    out = self.conv3(out)
    out = self.conv4(out)
    out = self.res2(out) + out  # 第二个残差连接
    out = self.conv5(out)
    out = self.conv6(out)
    out = self.res3(out) + out  # 第三个残差连接
    out = self.linear1(out)# 通过全连接层输出
    return out
# 定义 SimpleCNN 模型
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        self.layer1 = conv_block(3, 32,se_module=False)
        self.layer2 = conv_block(32, 64, pool=True,se_module=False)
        self.layer3 = conv_block(64, 128,se_module=False)
        self.layer4 = conv_block(128, 256, pool=True,se_module=False)
        self.fc = nn.Sequential(
            nn.MaxPool2d(8),
            nn.Flatten(),
            nn.Linear(256, 10)
        )
```

```
        def forward(self, x):
            x = self.layer1(x)
            x = self.layer2(x)
            x = self.layer3(x)
            x = self.layer4(x)
            x = self.fc(x)
            return x

epochs=11

# 训练过程
def train_and_evaluate(model, name):
    # 将模型加载到指定设备 (GPU 或 CPU) 上
    model = model.to(device)

    # 定义损失函数为交叉熵损失，用于多分类任务
    loss_fn = nn.CrossEntropyLoss().to(device)

    # 定义优化器为 Adam，使用学习率 0.01 和权重衰减 1e-4 防止过拟合
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-4)
    # 定义 OneCycleLR 学习率调度器，用于动态调整学习率，使其在训练早期快速上升
sched = torch.optim.lr_scheduler.OneCycleLR(
    optimizer,    # 要应用调度的优化器
    max_lr=0.01,    # 最大学习率，即训练过程中的峰值学习率
    epochs=epochs,    # 训练的总周期数
    steps_per_epoch=len(train_dataloader)    # 每个周期的更新步数
)

# 初始化用于存储每个 epoch 的训练/测试损失和准确率的列表
```

```
    # 初始化用于存储每个 epoch 的训练/测试损失和准确率的列表
train_losses, test_losses, train_accuracies, test_accuracies, learning_rates =
[], [], [], [], []

    # 开始循环训练每个 epoch
for epoch in range(epochs):
    model.train()  # 设置模型为训练模式，以启用 Dropout 等训练特
有的层
    total_train_loss = 0  # 用于累积训练损失
    total_accuracy = 0  # 用于累积训练准确率

    # 循环遍历每批训练数据
    for imgs, targets in train_dataloader:
        imgs, targets = imgs.to(device), targets.to(device)  # 将数据和标
签加载到设备上
        optimizer.zero_grad()  # 清除上一步的梯度，以免梯度累积

        # 前向传播：将输入图像传入模型获取预测
        output = model(imgs)
        # 计算损失：预测输出和真实标签之间的交叉熵损失
        loss = loss_fn(output, targets)
        # 反向传播：计算每个参数的梯度
        loss.backward()
        # 参数更新：根据梯度和学习率更新参数
        optimizer.step()
        # 调度器步进：更新学习率，遵循 OneCycleLR 策略
        sched.step()

        # 累积训练损失：将当前批次的损失值加入到总损失中
        total_train_loss += loss.item()
        # 计算批次正确预测的数量并累加
        total_accuracy += (output.argmax(1) == targets).sum().item()
```

```
# 计算当前 epoch 的平均训练损失
avg_train_loss = total_train_loss / len(train_dataloader)
train_losses.append(avg_train_loss)  # 记录当前 epoch 的平均训练损失

# 计算训练集准确率：正确预测数 / 总样本数
train_acc = total_accuracy / len(train_data)
train_accuracies.append(train_acc)  # 记录当前 epoch 的训练准确率

# 模型进入评估模式：禁用 Dropout 等训练特定层
model.eval()
total_test_loss = 0  # 用于累积测试损失
correct_predictions = 0  # 用于累积正确的测试预测数

# 禁用梯度计算，以节省内存和加快计算速度
with torch.no_grad():
    for imgs, targets in test_dataloader:
        imgs, targets = imgs.to(device), targets.to(device)
        output = model(imgs)  # 前向传播获取输出
        loss = loss_fn(output, targets)  # 计算测试损失
        total_test_loss += loss.item()  # 累积测试损失
        correct_predictions += (output.argmax(1) == targets).sum().item()
# 计算正确预测数
# 计算当前 epoch 的平均测试损失
avg_test_loss = total_test_loss / len(test_dataloader)
test_losses.append(avg_test_loss)  # 记录当前 epoch 的平均测试损失

# 计算测试集准确率
test_acc = correct_predictions / len(test_data)
test_accuracies.append(test_acc)  # 记录当前 epoch 的测试准确率

# 获取当前学习率并记录
learning_rates.append(sched.get_last_lr())
```

```
        # 获取当前学习率并记录
        learning_rates.append(sched.get_last_lr())

            print(
                f"Epoch [{epoch + 1}/{epochs}], Train Loss:
{avg_train_loss:.4f}, Test Loss: {avg_test_loss:.4f},
Train_Accuracy:{train_acc:.4f},Test_Accuracy: {test_acc:.4f}")

    plot_metrics(train_losses, test_losses, train_accuracies, test_accuracies,
learning_rates, name)
def plot_metrics( train_losses, test_losses, train_accuracies,test_accuracies,
learning_rates, model_name):
    # 使用 matplotlib 绘制损失和准确率图像
    epochs_range = range(1, epochs + 1)
    plt.figure(figsize=(14, 7))

    # 训练损失
    plt.subplot(1, 3, 1)
    plt.plot(epochs_range, train_losses, label="Train Loss")
    plt.plot(epochs_range, test_losses, label="Test Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.title(f"{model_name} Train and Test Loss")

    # 准确率
    plt.subplot(1, 3, 2)
    plt.plot(epochs_range, train_accuracies, label="Train Accuracy",
color='orange')
    plt.plot(epochs_range, test_accuracies, label="Test Accuracy",
color='purple')
    plt.xlabel("Epochs")
```

```
        plt.ylabel("Accuracy")
        plt.legend()
        plt.title(f"{model_name} Train and Test Accuracy")

        # 学习率
        plt.subplot(1, 3, 3)
        plt.plot(epochs_range, learning_rates, label="Learning Rate",
color='green')
        plt.xlabel("Epochs")
        plt.ylabel("Learning Rate")
        plt.legend()
        plt.title(f"{model_name} Learning Rate Schedule")

        plt.tight_layout()
        plt.savefig(f"{model_name}_running.png")
        print(f"{model_name}_running.png saved successfully.")
        plt.show()

# 训练两种模型
se_model = SEModel()
simple_cnn = SimpleCNN()
train_and_evaluate(se_model, "SE_CNN")

train_and_evaluate(simple_cnn, "Onecyclelr_CNN")
```

3. LSTM 神经网络项目代码

3.1 结构图

```
—THUCNews
    |  Code screenshot revise.py
    |  Figure_1.png
    |  test.txt
    |  train.txt
    |
    ├—cnews
    |      cnews.test.txt
    |      cnews.train.txt
    |      cnews.val.txt
    |      cnews.vocab.txt
    |
    ├—codes
    |   |  date_process.py
    |   |  main.py
    |   |
    |   └—__pycache__
    |          date_process.cpython-311.pyc
    |
    ├—imgs
    |   ├—code_imgs
    |   |      date_process_code.png
    |   |      main_code.png
    |   |
    |   └—effect_imgs
    |          date_process_output.png
    |          main_output.png
    |
    |
    ├—stopwords-master
    |      baidu_stopwords.txt
    |      cn_stopwords.txt
    |      hit_stopwords.txt
    |      README.md
    |      scu_stopwords.txt
    |
    └—__pycache__
```

cnews. train. txt、cnews. test. txt、cnews. val. txt：分别存储训练集、测试集和验证集的文本数据，这些数据将被用于训练和评估 LSTM 模型。

cnews. vocab. txt：包含词汇表，定义了项目中使用的所有单词或标记，是构建模型输入的重要参考。

Figure_1. png、date_process_code. png、main_code. png 等图像文件：可能是代码截图、数据处理流程示意图或模型架构图，用于文档记录或展示项目进展。

main. py 会首先调用 date_process. py 进行数据处理，然后加载 cnews. train. txt 等数据集，接着初始化并训练 LSTM 模型。

在数据处理过程中，main. py 或 date_process. py 会参考 baidu_stopwords. txt 等停用词表来去除文本中的无关词汇。

训练完成后，main. py 会调用测试集 cnews. test. txt 来评估模型性能，并生成如 main_output. png 的可视化结果。

revise. py 可能在项目迭代过程中被频繁调用，以修正和优化代码。

3. 2 完整代码

date_process.py

```
import os
import jieba
import pickle
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import numpy as np
import torch.nn as nn
```

```
# 加载数据
train_data = pd.read_csv('../cnews/cnews.train.txt', sep='\t', header=None,
names=["label", "content"])
test_data = pd.read_csv('../cnews/cnews.test.txt', sep='\t', header=None,
names=["label", "content"])
train_data_path, test_data_path = '../train.txt', '../test.txt'
# 获取内容和标签
train_texts = train_data['content'].values
train_labels = train_data['label'].values
test_texts = test_data['content'].values
test_labels = test_data['label'].values
# 分词
def tokenize(text):
    return list(jieba.cut(text))


def drop_stopword(datas):
    try:
        with open('../stopwords-master/cn_stopwords.txt', 'r',
encoding='UTF-8') as f:
            stop_words = [word.strip() for word in f.readlines()]
        print('Successfully loaded stopwords')
    except Exception as e:
        print(f"Error loading stopwords: {e}")
    return [[word for word in data if word not in stop_words] for data in
datas]


# 加载数据
def load_saved_data(path):
    data_list = []nhj
    with open(path, 'r', encoding='UTF-8') as lines:
```

```
                for line in lines:
                    # 移除行尾的换行符，并按逗号分割
                    line_data = line.strip().split(',')
                    # 将分割后的数据添加到二维列表中
                    data_list.append(line_data)
            return data_list

# 保存数据
def save_data(datax, path):
    with open(path, 'w', encoding='UTF-8') as f:
        for lines in datax:
            f.write(','.join(map(str, lines)) + '\n')
    print('Successfully saved data')


if os.path.exists(train_data_path) and os.path.exists(test_data_path):
    print('Loading preprocessed data...')
    train_texts = load_saved_data(train_data_path)
    test_texts = load_saved_data(test_data_path)
else:


    # 分词
    train_texts = [tokenize(text) for text in train_texts]
    test_texts = [tokenize(text) for text in test_texts]
    # 筛选
    train_texts = drop_stopword(train_texts)
    test_texts = drop_stopword(test_texts)
    save_data(train_texts, train_data_path)
    save_data(test_texts, test_data_path)
    print('Successfully saved data')
```

main.py

```
import torch
import numpy as np
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
from gensim.models import Word2Vec
from torch.nn.utils.rnn import pad_sequence
from sklearn.preprocessing import LabelEncoder
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import accuracy_score, mean_squared_error
from date_process import train_texts, test_texts, train_labels, test_labels

# NumPy：Python 中的数值运算库。
# PyTorch：一个深度学习框架，提供用于构建和训练神经网络的工具。
# Gensim：一个用于主题建模和文档相似性分析的库，此处用于 Word2Vec。
# Scikit-learn：一个机器学习库，提供用于预处理和评估模型的工具。
# DataLoader 和 TensorDataset：PyTorch 中用于处理批量数据的实用程序。
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# 构建 LSTM
# LSTMModel：nn.定义 LSTM 网络的模块。
# embedding_dim：输入嵌入的维数。
# hidden_dim：LSTM 处于隐藏状态的特征数。
# output_dim：输出类的数量。
class LSTMModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, output_dim):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(embedding_dim, hidden_dim,
batch_first=True)
```

```
    # forward
    # method：定义输入数据如何通过模型。
    # LSTM
    # 处理输入序列并返回输出和隐藏状态。
    # 最后一个隐藏状态通过一个全连接层来产生最终输出。
    def forward(self, x):
        x, (hn, cn) = self.lstm(x)
        x = hn[-1]  # 取最后一个时间步的输出
        x = self.fc(x)
        return x


# 文本转向量
# 使用经过训练的 Word2Vec 模型将文本列表转换为向量。
# 对于文本中的每个单词，如果模型中存在该向量，则它会检索该向量。
# 如果文本短于 max_length，则填充向量为零;如果更长，它会截断。
# 返回 shape (num_samples, max_length, embedding_dim) 为 的张量。
def text_to_vector(texts, model, max_length):
    vectors = []
    for text in texts:
        vecs = [model.wv[word] for word in text if word in model.wv]
        # 填充和截断
        vecs = vecs[:max_length] + [np.zeros(model.vector_size)] *
max(max_length - len(vecs), 0)
        vectors.append(vecs)
    vectors = np.array(vectors)
    return torch.tensor(vectors, dtype=torch.float32)


def train_model(model, train_loader, test_loader, epochs, output_dim):
    train_accuracies = []
```

```
# 训练模型
# 训练循环：运行指定数量的 epoch。
# 将模型设置为训练模式 （model.train（））。
# 对于每个批次：
#    重置渐变。
#    计算模型输出。
#    计算损失。
#    执行反向传播。
#    更新模型参数。
for epoch in range(epochs):
    model.train()
    all_preds = []
    all_labels = []
    epoch_loss = 0.0  # 初始化每个 epoch 的损失

    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)  # 计算损失
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()  # 累加损失
        _, predicted = torch.max(outputs, 1)
        all_preds.extend(predicted.cpu().numpy())
        all_labels.extend(labels.cpu().numpy())
    # 计算训练集上的准确率和 MSE
train_accuracy = accuracy_score(all_labels, all_preds)
train_mse = mean_squared_error(all_labels, all_preds)  # 使用预测和真实
标签计算 MSE
```

```
    # 记录训练集的指标
        train_accuracies.append(train_accuracy)
        train_mses.append(train_mse)
        train_losses.append(epoch_loss / len(train_loader))  # 计算平均损失

        # 测试过程
        test_accuracy, test_mse, all_test_labels, all_test_preds =
evaluate_model(model, test_loader, output_dim)
        test_accuracies.append(test_accuracy)
        test_mses.append(test_mse)

        print(
            f'Epoch {epoch + 1}/{epochs},Train Loss: {epoch_loss /
len(train_loader):.4f},Train MSE: {train_mse:.4f}, '
            f'Train Accuracy: {train_accuracy:.4f},     Test Accuracy:
{test_accuracy:.4f}, Test MSE: {test_mse:.4f}')

return train_accuracies, train_mses, train_losses, test_accuracies, test_mses,
all_test_labels, all_test_preds
# 评估模型
# 在测试数据集上评估模型。
# 将模型设置为评估模式 （model.eval（））， 这将禁用 dropout 和批
量规范化。
# 迭代测试 DataLoader， 计算预测， 并收集真实标签和预测标签。
# 打印每个样本的真实标签和预测标签。
# 使用 Scikit-learn 中的 accuracy_score 返回准确率分数。
def evaluate_model(model, test_loader, output_dim):
        model.eval()
        all_preds = []
        all_labels = []
        all_probs = []  # 确保初始化为列表
```

```
        with torch.no_grad():
        for inputs, labels in test_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                _, predicted = torch.max(outputs, 1)
                all_preds.extend(predicted.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
                all_probs.append(outputs.cpu().numpy())  # 收集输出概率

                # 打印每个样本的真实标签和预测标签
                # for true_label, pred_label in zip(labels.cpu().numpy(),
predicted.cpu().numpy()):
                #         print(f"True label: {true_label}, Predicted label:
{pred_label}")
        # 统计每个标签的真实数量和预测数量

# 计算准确率
accuracy = accuracy_score(all_labels, all_preds)

# 计算均方误差 (MSE)
all_probs = np.vstack(all_probs)  # 将所有概率输出堆叠成一个数组
one_hot_labels = np.eye(output_dim)[all_labels]  # 将真实标签转换为独
热编码
mse = mean_squared_error(one_hot_labels, all_probs)  # 计算 MSE

return accuracy, mse, all_labels, all_preds
def plot_metrics(train_accuracies, train_mses, train_losses, test_accuracies,
test_mses, all_test_labels,
                                all_test_preds):
        epochs = range(1, len(train_accuracies) + 1)

        plt.figure(figsize=(15, 5))
```

```
# 准确率图
plt.subplot(1, 3, 1)
plt.plot(epochs, train_accuracies, label='Train Accuracy', marker='o')
plt.plot(epochs, test_accuracies, label='Test Accuracy', marker='o')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.xticks(epochs)
plt.legend()
plt.grid()

# MSE 图
plt.subplot(1, 3, 2)
plt.plot(epochs, train_mses, label='Train MSE', marker='o')
plt.plot(epochs, test_mses, label='Test MSE', marker='o')
plt.title('MSE over Epochs')
plt.xlabel('Epochs')
plt.ylabel('MSE')
plt.xticks(epochs)
plt.legend()
plt.grid()

# 损失图
plt.subplot(1, 3, 3)
plt.plot(epochs, train_losses, label='Train Loss', marker='o', color='red')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.xticks(epochs)
plt.legend()
plt.grid()
plt.tight_layout()
plt.show()
```

```
true_counts = np.bincount(all_test_labels, minlength=output_dim)
pred_counts = np.bincount(all_test_preds, minlength=output_dim)

# 绘制真实标签与预测标签数量的比较图
labels = np.arange(output_dim)  # 标签的索引
x = np.arange(len(labels))  # x 轴位置

plt.figure(figsize=(10, 5))
width = 0.35  # 条形宽度

# 绘制条形图
plt.bar(x - width / 2, true_counts, width, label='True Counts', alpha=0.7,
color='blue')
plt.bar(x + width / 2, pred_counts, width, label='Predicted Counts',
alpha=0.7, color='orange')

plt.title('True vs Predicted Counts per Label')
plt.xlabel('Labels')
plt.ylabel('Counts')
plt.xticks(x, labels)
plt.legend(loc='best')
plt.grid(axis='y')
plt.tight_layout()
plt.show()
# 标签编码
# 使用 LabelEncoder 将字符串标签编码为整数。
le = LabelEncoder()
train_labels_encoded = le.fit_transform(train_labels)
test_labels_encoded = le.transform(test_labels)
# 将编码后的标签转换为 PyTorch 张量，确保它们是 long 类型，以
便与损失函数兼容。
Y_train_tensor = torch.tensor(train_labels_encoded, dtype=torch.long)  #确
保标签是 long 类型
```

```
Y_test_tensor = torch.tensor(test_labels_encoded, dtype=torch.long)  # 确保标签是 long 类型

# 训练 Word2Vec 模型
# 在训练文本上训练 Word2Vec 模型。
# vector_size：单词向量的维度。
# window：句子中当前单词和预测单词之间的最大距离。
# min_count：忽略总频率低于此频率的所有单词。
# workers：用于训练模型的工作线程数。
word2vec_model = Word2Vec(sentences=train_texts, vector_size=100, window=5, min_count=1, workers=8)

# 将文本转换为 Word2Vec 向量
# 使用先前训练的 Word2Vec 模型将训练和测试文本转换为矢量表示。
# 确保所有序列的长度相同 （max_length）
max_length = 100  # 最大序列长度
X_train_vec = text_to_vector(train_texts, word2vec_model, max_length)
X_test_vec = text_to_vector(test_texts, word2vec_model, max_length)

# 超参数
# 设置 LSTM 模型的超参数：
# embedding_dim：应与 Word2Vec 向量大小匹配。
# hidden_dim：处于 LSTM 隐藏状态的特征数。
# output_dim：训练标签中唯一类的数量。
embedding_dim = 100  # 与 Word2Vec 的 vector_size 一致
hidden_dim = 64
output_dim = len(np.unique(train_labels_encoded))

# 实例化模型
model = LSTMModel(embedding_dim, hidden_dim, output_dim).to(device)
```

```
# 创建数据加载器
# 将训练数据包装到 TensorDataset 中，以便于批处理。
# 为训练数据集创建 DataLoader，允许进行批处理和随机排序。
train_dataset = TensorDataset(X_train_vec.to(device),
Y_train_tensor.to(device))
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
# 定义损失函数和优化器
# Loss Function：使用交叉熵损失，适用于多类分类。
# Optimizer：使用 Adam 优化器，它对训练深度学习模型非常有效。
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
# 评估模型
test_dataset = TensorDataset(X_test_vec, Y_test_tensor)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=True)
# 使用示例
train_accuracies, train_mses, train_losses, test_accuracies, test_mses,
all_test_labels, all_test_preds = train_model(
    model, train_loader, test_loader,
    num_epochs, output_dim)

plot_metrics(train_accuracies, train_mses, train_losses, test_accuracies,
test_mses, all_test_labels, all_test_preds)
```

四. 小组分工及结语

1. 分工

组员	主要工作	工作量占比
李济岑	BP 神经网络项目的制作及测试调优	25%
康心柔	CNN 神经网络项目制作及测试调优	25%
那圣奇	LSTM 神经网络项目制作及测试调优	25%
张连兴	项目整理及 word 报告制作	25%

2. 结语

在本次人工智能导论课程的小组合作项目中，我们成功地设计并实施了一个涵盖 BP（反向传播）神经网络、CNN（卷积神经网络）以及 LSTM（长短时记忆网络）的综合性三级项目。这一过程不仅加深了我们对神经网络基本原理的理解，还让我们在实践中探索了不同网络架构在解决特定问题上的优势与局限。在整个项目实施过程中，我们不仅学会了如何选择合适的模型、调整超参数以优化性能，还锻炼了团队协作、问题解决以及项目管理的能力。面对模型调优中的种种挑战，我们不断尝试、迭代，这种“试错”的学习过程让我们更加珍惜每一次小小的进步。

本次项目不仅是一次技术上的探索之旅，更是一次对自我能力的全面提升。它不仅增强了我们对人工智能领域的热情，更为我们未来的学术研究和职业发展奠定了坚实的基础。展望未来，我们将带着这份宝贵的经验和知识，继续在人工智能的广阔天地中探索前行，期待为解决现实世界中的复杂问题贡献自己的力量。