# 目录

第 1 章 新冠病例预测项目概述 ..... 1
1.1 新冠病例预测实践项目简介 ..... 1
1.2 新冠病例预测实践项目研究现状和意义 ..... 1
1.3 新冠预测实践项目开发环境以及技术 ..... 2
第 2 章 数据探索与数据预处理 ..... 3
2.1 新冠预测实践项目数据探索 ..... 3
2.2 数据预处理 ..... 5
第 3 章 相关性分析与特征值选取 ..... 10
3.1 相关系数矩阵热力图 ..... 10
3.2 特征值相关性得分分析 ..... 14
3.3 特征值选取 ..... 16
第 4 章 基于线性回归的新冠病例预测分析 ..... 17
4.1 线性回归模型的原理和公式 ..... 17
4.2 定义线性回归模型的网络结构 ..... 18
4.3 描述线性回归模型的训练 ..... 18
4.4 模型优化与调整参数设置 ..... 23
4.5 生成预测结果并导出文件 ..... 34
第 5 章 基于多层感知机的新冠病例预测分析 ..... 36
5.1 多层感知机模型的原理和公式 ..... 36
5.2 定义多层感知机模型的网络结构 ..... 37
5.3 描述多层感知机模型的训练 ..... 38
5.4 模型优化与调整参数设置 ..... 43
5.5 生成预测结果并导出文件 ..... 52
第 6 章 基于循环神经网络的新冠病例预测分析 ..... 54
6.1 循环神经模型的原理和公式 ..... 54
6.2 循环神经网络模型的网络结构 ..... 55
6.3 描述循环神经网络模型的训练 ..... 56
6.4 模型优化与调整参数设置 ..... 60

6.5 生成预测结果并导出文件 ..... 63
第 7 章 基于卷积神经网络的新冠病例预测分析 ..... 64
7.1 卷积神经网络模型的原理和公式 ..... 64
7.2 卷积神经网络模型的网络结构 ..... 65
7.3 描述卷积神经网络模型的训练 ..... 67
7.4 模型优化与调整参数设置 ..... 72
7.5 生成预测结果并导出文件 ..... 76
第八章 实验结果分析 ..... 77
8.1 分析相同模型不同参数的效果 ..... 77
8.2 对比不同模型的实验结果 ..... 85
第九章 结论 ..... 87
参考文献 ..... 89

# 第 1 章 新冠病例预测项目概述

## 1.1 新冠病例预测实践项目简介

本次新冠病例预测实践项目目的是在通过使用深度学习模型，针对 40 个州的特征数据，预测新冠病例在第三天的确诊情况。实践项目将利用前三天的特征数据以及相应的标签（新冠确诊）进行分析和预测。

新冠病例的预测对于疫情控制和公共卫生管理具有重要意义。通过准确预测新冠病例的数量，政府和卫生部门可以更好地制定防控措施，并优化资源的分配。同时，医疗机构和卫生部门可以根据预测结果合理规划医疗资源，以应对未来病例的增长和变化。此外，新冠病例预测还可以用于评估疫情对经济和社会的影响，为决策者提供重要的数据支持。

在本项目中，我们将进行数据预处理、特征分析和选择，并构建深度神经网络模型，包括线性回归、多层感知机、循环神经网络和卷积神经网络，来实现新冠病例的预测任务。通过对模型进行训练和预测，我们将评估不同模型的性能和准确性，并进行可视化分析，以选择最佳的模型和参数设置。

## 1.2 新冠病例预测实践项目研究现状和意义

### 1.2.1 研究现状

在后疫情时代，新冠病例预测成为一项重要的研究任务，并在疫情控制、医疗资源规划和经济社会影响评估等方面具有广泛的应用。许多研究人员通过应用机器学习和深度学习技术，对新冠病例进行预测和分析，以提供决策支持和指导。近年来，深度学习模型在新冠病例预测中展现出了巨大的潜力[1]。

深度学习模型以其强大的学习能力和表征能力在新冠病例预测中取得了显著的成果。通过利用大规模的数据集和复杂的神经网络结构，这些模型能够从海量的特征数据中提取有效的特征，并捕捉到数据中的非线性关系和时序模式。例如，循环神经网络（RNN）能够处理时间序列数据，捕捉病例数量随时间的变化趋势。卷积神经网络（CNN）则适用于处理空间相关的特征，如地理位置和人口密度等。此外，多层感知机（MLP）和深度残差网络（ResNet）等模型也被广泛应用于新冠病例预测任务中[2]。

### 1.2.2 研究意义

在后疫情时代，新冠病例预测对于疫情控制和公共卫生管理至关重要。准确预测未来的病例数量可以帮助政府和卫生部门制定更精确和有针对性的防控措施。通过预测病例数量的增长趋势和传播速度，可以优化资源分配，确保医疗设施、防护用品和检测能力等在需要的地区得到充分的供给。同时，新冠病例预测还可以为疫情预警系统提供数据支持，帮助及早发现和控制潜在的疫情爆发风险[3]。

新冠病例预测在医疗资源规划方面具有重要作用。通过准确预测未来病例数量，医疗机构和卫生部门可以合理规划医疗资源，包括床位、医护人员、药品和医疗设备等。这有助于应对未来病例的增长和变化，避免资源短缺和医疗系统过载的问题。通过合理规划资源分配，可以提高病患的治疗效果和生存率，为患者提供更好的医疗服务[4]。

新冠病例预测对于评估疫情对经济和社会的影响至关重要。在后疫情时代，疫情对经济和社会的影响仍然存在。准确预测新冠病例的数量和传播趋势，可以为政府和决策者提供重要的数据支持。根据预测结果，他们可以制定相应的经济政策、社会措施和恢复计划，以减轻疫情对经济和社会的冲击。预测结果还可以用于评估疫情对不同行业、就业和消费等方面的影响，从而更好地应对疫情带来的挑战和变化[5]。

## 1.3 新冠预测实践项目开发环境以及技术

本次新冠预测实践项目使用了 Python3 开发，采用 Jupyter Notebook 这一交互式的开发环境。分析过程中使用了以下扩展包。

Numpy 扩展库，NumPy（Numerical Python）是 Python 的一种开源的数值计算扩展库，其中提供了许多向量和矩阵操作，不仅方便易用而且效率更高[6]。

Pandas 扩展库：Pandas 是一个强大的分析结构化数据的工具集；它的使用基础是 Numpy（提供高性能的矩阵运算）；用于数据挖掘和数据分析，同时也提供数据清洗功能[7]。

Matplotlib 扩展库，Python 优秀的数据可视化第三方库。Matplotlib 库由各种可视化类构成，内部结构复杂，受 Matlab 启发[8]。

Scikit-learn 扩展库：Scikit-learn 是 Python 中常用的机器学习库，提供了丰富的机器学习算法和工具，用于模型训练、评估和预测等任务。

PyTorch 扩展库，PyTorch 是一个用于构建深度学习模型的开源机器学习库，提供了灵活的张量计算和自动求导功能[9]。

# 第 2 章 数据探索与数据预处理

## 2.1 新冠预测实践项目数据探索

本次新冠预测实践项目针对提供的数据集进行了数据探索性分析，经过分析数据集包含：

1、训练数据表（covid.train.csv）

2、测试数据表（covid.test.csv）

3、样本提交表（sampleSubmission.csv）

使用 pandas 库将数据表导入到 jupyter notebook，并分别命名为 covid_train、covid_test、sampleSubmission，导入数据表代码如图 2-1 所示。

```
#导入数据表
#导入 covid.test 表
covid_test = pd.read_csv('data/COVID_19/covid.test.csv', sep = ',')
#导入 covid.train 表
covid_train = pd.read_csv('data/COVID_19/covid.train.csv', sep = ',')
#导入 sampleSubmission 表
sampleSubmission = pd.read_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\sampleSubmission.csv', sep = ',')
```

图 2-1 导入数据表

利用 covid_train.info()、covid_test.info()可以查看每个特征值的数据类型，数据行数以及是否存在空值，经过数据表的探索性分析的出结果：

1、训练数据表（covid.train.csv）的作用是为存储了给 40 个州的前三天的特征，共 2700 例数据，其内容如表 2-1 所示。

表 2-1 训练数据表数据构成

| 列号 | 说明 | 数据类型 | 是否项目需要 |
|---|---|---|---|
| 0 | Id | 整型 | 不需要 |
| 1-40 | 40 个州独热编码 | 整型 | 需要 |
| 41-59 | 第一天特征数据 | 整型 | 需要 |
| 60-78 | 第二天特征数据 | 整型 | 需要 |
| 79-94 | 第三天数据特征 | 整型 | 需要 |

下面针对分析后项目需要的数据进行分析

（1）40 个州独热编码：

该数据项是将 40 个州进行独热编码后的数据，通过数据探索发现该数据无空值，无异常值，符合项目需求，不需要进行数据过滤。

（2）第一天特征数据：

该数据项是由 COVID-like illness（类似新冠疾病）、Behavior Indicators（行为

指标）、Mental Health Indicators（心理健康指标）和 tested_positive（结果数据）组成。通过数据探索发现该数据项无空值，无异常值，符合项目需求，不需要进行数据过滤。

（3）第二天特征数据、第三天数据特征同第一天数据特征。

2、测试数据表（covid.test.csv）的作用是进行新冠预测，得到最终结果，给 40 个州的前三天的特征（缺少了第三天的结果，需要预测），共 893 例数据，其内容如表 2-2 所示。

**表 2-2 测试数据表数据构成**

| 列号 | 说明 | 数据类型 | 是否项目需要 |
|---|---|---|---|
| 0 | Id | 整型 | 不需要 |
| 1-40 | 40 个州独热编码 | 整型 | 需要 |
| 41-59 | 第一天特征数据 | 整型 | 需要 |
| 60-78 | 第二天特征数据 | 整型 | 需要 |
| 79-93 | 第三天数据特征 | 整型 | 需要 |

下面针对分析后项目需要的数据进行分析

（1）40 个州独热编码：

该数据项是将 40 个州进行独热编码后的数据，通过数据探索发现该数据无空值，无异常值，符合项目需求，不需要进行数据过滤。

（2）第一天特征数据：

该数据项是由 COVID-like illness（类似新冠疾病）、Behavior Indicators（行为指标）、Mental Health Indicators（心理健康指标）和 tested_positive（结果数据）组成。通过数据探索发现该数据项无空值，无异常值，符合项目需求，不需要进行数据过滤。

（3）第二天特征数据同第一天数据特征。

（4）第三天特征数据同第一天数据特征，但缺少第三天的结果数据，需要进行预测。

3、样本提交表（sampleSubmission.csv）的作用是存储测试训练集中通过预测得到的结果数据，以便用来提交结果数据，其内容如表 2-3 所示。

**表 2-3 样本提交表（sampleSubmission.csv）数据构成**

| 说明 | 数据类型 | 是否项目需要 |
|---|---|---|
| Id | 整型 | 需要 |
| tested_positive.2 | 空值 | 需要补充 |

下面针对分析后项目需要的数据进行分析

（1）id 列：该数据项是同测试数据集中的 id 一致共 893 例。

（2）tested_positive.2：该项数据为空值，需要预测完成后将预测结果导入。

## 2.2 数据预处理

根据数据探索分析结果针对训练数据集和测试数据集做数据预处理工作，工作包含删除无用列、特征值切分、数据集划分、标准化处理、转换数据类型等，方便数据分析。

### 2.2.1 删除无用列

根据数据探索分析结果针对训练数据集和测试数据集中的 id 列为无用列，需要将其删除，删除操作如图 2-2 所示。

```
: # 使用drop() 函数进行删除，默认为删除行，axis=1为删除列
  # 删除covid_train的id列
  covid_train1 = covid_train.drop('id',axis=1)
  # 删除covid_test的id列
  covid_test1 = covid_test.drop(['id'], axis=1)
```

图 2-2 删除 id 列操作

### 2.2.2 特征值切分

因为本次实验的特征值较多，去除 id 列后还存在 93 个特征值，若同时绘制所有特征值的相关系数矩阵热力图，会导致该图像可视性较差，达不到应有的效果，为了之后的相关系数矩阵热力图更加美观，便于查看各特征值之间的相关性，因此需要按照数据探索分析将训练数据表（covid_train）中 40 个州的独热编码、第一天特征数据、第二天特征数据和第三天特征数据进行切分，切分操作如图 2-3 所示。

```
]: # 特征值切分
   # 提取州数据
   covid_train_state = covid_train1.iloc[:,0:40]
   # 提取第一天数据
   covid_train_day1 = covid_train1.iloc[:,40:58]
   # 提取第二天数据
   covid_train_day2 = covid_train1.iloc[:,58:76]
   # 提取第三天数据
   covid_train_day3 = covid_train1.iloc[:,76:94]
```

按照列的索引对特征值进行切分

图 2-3 切分特征值操作

### 2.2.3 分离特征值与目标变量

为了便于之后的相关性分析以及划分训练集、测试集，因此需要将训练数据表（covid_train）中的特征值与目标变量（target）进行分离，分离操作如图 2-4 所示。

```
# 分离特征值与目标变量
# 获得特征值
train = covid_train1.drop(['tested_positive.2'], axis=1)
# 获得目标变量
target = covid_train['tested_positive.2']
```

图 2-4 分离特征值与目标变量操作

### 2.2.4 标准化处理

标准化处理数据的作用是将不同特征之间的数值范围统一，以便更好地进行比较和分析。它可以帮助提升模型的收敛速度（加快梯度下降的求解速度）、提升模型的精度（消除量级和量纲的影响）、同时还能简化计算。

标准化处理的数学原理是将数据按列进行零均值和单位方差的缩放。这可以通过以下两个步骤实现。

零均值化（Zero Mean）、单位方差化（Unit Variance），通过这两个步骤，可以使得数据的每个特征都具有相似的尺度，消除特征之间的量纲差异，使得模型更容易学习特征之间的关系。

本次实验从 sklearn.preprocessing 库引入了 StandardScaler 类，用于对数据进行标准化处理。根据上述数据预处理步骤，已将需要进行标准化的数据准备好，分别为训练数据表（covid_train1）中的特征值 train，以及测试数据表（covid_test1）中的特征值。然后创建一个 StandardScaler 对象，并使用 fit_transform 方法对训数据进行标准化处理，将特征数据按列进行零均值和单位方差的缩放。然后，我们使用同一个标准化对象的 transform 方法对测试数据进行相同的标准化处理，标准化处理操作如图 2-5 所示。

```
from sklearn.preprocessing import StandardScaler
# 创建StandardScaler对象
scaler = StandardScaler()
# 对训练数据表（covid_train1）中的特征值train进行标准化处理
train1 = scaler.fit_transform(train1)
# 对测试数据表（covid_test1）中的特征值进行标准化处理
test3 = scaler.fit_transform(test2)
```

图 2-5 数据标准化操作

## 2.2.5 划分训练集与测试集

本次实验使用 strain_test_split 函数划分训练集与测试集，train_test_split 函数是来自 sklearn.model_selection 模块的用于划分数据集的方法。它的作用是将原始数据集按照指定的比例或数量随机分割成训练集和测试集。本次实验将标准化之后的训练数据表（covid_train1）中的特征值和目标变量（target）进行划分，将 20%的样本作为测试集，80%的样本作为训练集。同时为了确保结果的可重复性，设置了 random_state=42，这个参数可以控制随机划分过程的种子值，保证每次运行时划分结果相同，然后通过调用.shape 方法，您检查切割后训练集和测试集中样本的形态，即它们的维度信息，划分数据集操作如图 2-6 所示。

```
# 切割数据样本 20%验证集；80%训练集
X_train, X_test, y_train, y_test=train_test_split(train1, target, test_size=0.2, random_state=42)
#检查训练集和验证集中样本的形态
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

设置比例为0.2　设置随机

```
((2160, 29), (540, 29), (2160,), (540,))
```

输出样本形态，便于确认数据集划分是否正确

图 2-6 划分数据集操作

## 2.2.6 转换数据类型

为了后续进行预测分析，需要将原来的数据类型转换为张量的形式，在循环神经网络和卷积神经网络中，还需要将张量的维度转换为该模型所需的维度。

1、线性回归和多层感知机的数据类型转换

线性回归模型和多层感知机模型对张量维度无特殊要求，因此只需将原始数据类型转换为张量即可，线性回归和多层感知机数据类型转换操作如图 2-7 所示。

```
# 将样本数据转换为 PyTorch 的张量类型
# 训练集数据
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
# 测试集数据
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
# 将需要预测数据转换为 PyTorch 的张量类型
test4 = torch.tensor(test3, dtype=torch.float32)
# 查看维度信息
X_train.shape, X_test.shape, y_train.shape, y_test.shape, test4.shape
```

转换为张量格式
并设置数据类型为float

查看维度信息

```
(torch.Size([2160, 29]),
 torch.Size([540, 29]),
 torch.Size([2160]),
 torch.Size([540]),
 torch.Size([893, 29]))
```

图 2-7 线性回归与多层感知机的数据类型转换操作

2、循环神经网络的数据类型转换

循环神经网络模型需要输入具有三个维度的数据，即(batch_size,

sequence_length, input_size)。这是因为 RNN 模型是基于时间序列数据的，它需要考虑输入数据的时序关系。

在本次新冠病例预测实践中，需要根据过去三天的特征来预测第三天的新冠确诊数量。因此，要将每个样本的特征序列视为一个时间序列，其中时间步的数量为 3。通过将特征序列调整为具有三个维度的形状，我们可以将其作为输入传递给 RNN 模型。

具体而言，我们将训练数据的形状转换为(样本数量,1,特征数量)，其中 1 表示时间序列的长度为 3。这样，RNN 模型可以按顺序处理每个时间步的输入，并在最后一个时间步输出预测结果。

同样，将目标值进行了相应的形状调整，以匹配 RNN 模型输出的形状，循环神经网络数据类型转换操作如图 2-8、图 2-9 所示。

```
# 将样本数据转换为 PyTorch 的张量类型
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
# 将测试数据转换为 PyTorch 的张量类型
test4 = torch.tensor(test4, dtype=torch.float32)
```

转换为张量数据类型设置为float

图 2-8 循环神经网络的数据类型转换操作

```
# 转换样本数据形状为RNN所需的维度
X_train = X_train.reshape(X_train.shape[0], 1, X_train.shape[1])
y_train = y_train.reshape(y_train.shape[0], 1, 1)
X_test = X_test.reshape(X_test.shape[0], 1, X_test.shape[1])
y_test = y_test.reshape(y_test.shape[0], 1, 1)
# 转换测试数据形状为RNN所需的维度
test4 = test4.reshape(test4.shape[0], 1, test4.shape[1])
# 查看维度信息
X_train.shape, X_test.shape, y_train.shape, y_test.shape, test4.shape
```

改变数据的维度

```
(torch.Size([2160, 1, 29]),
 torch.Size([540, 1, 29]),
 torch.Size([2160, 1, 1]),
 torch.Size([540, 1, 1]),
 torch.Size([893, 1, 29]))
```

查看数据的维度

图 2-9 循环神经网络的调整维度操作

3、卷积神经网络的数据类型转换

CNN 模型通常用于处理图像或其他类型的多通道数据，为了适应卷积神经网络（Convolutional Neural Network，CNN）模型的输入要求，需要将数据转换为特定的四维形状，通过以下操作来转换训练集、测试集样本维度。

CNN 模型期望输入数据具有四个维度，其中第一个维度表示样本数量，第二和第三个维度表示图像的高度和宽度（在这里我们使用 1 来表示），最后一个维度表示特征数量。

总结而言，通过将样本数据的形状转换为 CNN 所需的四维形状，我们能够满

足卷积操作和多通道处理的要求，并与模型的输入输出维度保持一致。这样可以确保数据能够正确传递给 CNN 模型，从而进行有效的特征提取和预测，卷积神经网络数据类型转换操作与调整维度操作如图 2-10、图 2-11 所示。

```
# 将样本数据转换为 PyTorch 的张量类型
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.float32)
# 将测试数据转换为 PyTorch 的张量类型
test4 = torch.tensor(test4, dtype=torch.float32)
```

转换为张量数据类型设置为float

图 2-10 循环神经网络的数据类型转换操作

```
# 转换样本数据形状为CNN所需的维度
# 第一个维度表示样本数量，
# 第二和第三个维度表示图像的高度和宽度（在这里我们使用1来表示）
# 最后一个维度表示特征数量。
X_train = X_train.reshape(X_train.shape[0], 1, 1, X_train.shape[1])
y_train = y_train.reshape(y_train.shape[0], 1)
X_test = X_test.reshape(X_test.shape[0], 1, 1, X_test.shape[1])
y_test = y_test.reshape(y_test.shape[0], 1)
# 转换测试集数据形状为CNN所需的维度
test4 = test4.reshape(test4.shape[0], 1, 1, test4.shape[1])
# 查看维度信息
X_train.shape, X_test.shape, y_train.shape, y_test.shape, test4.shape
```

转换数据维度

```
(torch.Size([2160, 1, 1, 29]),
 torch.Size([540, 1, 1, 29]),
 torch.Size([2160, 1]),
 torch.Size([540, 1]),
 torch.Size([893, 1, 1, 29]))
```

查看数据维度

图 2-11 卷积神经网络的调整维度操作

# 第 3 章 相关性分析与特征值选取

本次新冠预测实践项目使用 Pandas 库（pandas）中 corr()函数计算了 40 个州、第一天特征数据、第二天特征数据、第三天特征数据和目标变量之间的相关系数矩阵。该方法显示了每对变量之间的线性相关程度，可以理解变量之间的关系，并使用 Seaborn 库的 heatmap 函数创建热力图，分别绘制了相关系数矩阵热力图。

通过 Scikit-learn 库中的 f_regression 函数计算特征与目标变量之间的相关性并返回了特征值与分数，然后通过 Scikit-learn 库中的 SelectKBest 类，根据指定的评分函数选择最重要的 k 个特征。本次实验使用 SelectKBest 选取了 30 个特征值。

最终通过综合分析相关系数矩阵热力图与 f_regression 函数得到的相关性分数，选取了 29 个特征值进行模型训练。

## 3.1 相关系数矩阵热力图

### 3.1.1 各州与目标变量的相关系数矩阵热力图

通过上述数据预处理中的特征值切分步骤，将 40 个周独热编码数据进行切分，并命名为 covid_train_state，然后与目标变量（tested_positive.2）进行合并，再利用的 corr()函数计算相关系数矩阵，heatmap 函数创建热力图，实现可视化，实现代码如图 3-1 所示。

```
# 40个州与第三天结果的相关系数矩阵热力图
# 数据准备，将目标结果添加到40个州的独热编码之后
covid_train_state['tested_positive.2'] = covid_train['tested_positive.2']
# 计算所有变量的相关系数，并将结果存储在 d 中
d = covid_train_state.corr()
# 创建图形，并设置其大小为 84x84 英寸
plt.subplots(figsize = (84, 84))
# 用Seaborn库的heatmap函数创建热力图
sns.heatmap(d, annot = True, vmax = 1, square = True, cmap = "Reds")
# 设置标题
plt.title('40个州与第三天结果的相关系数矩阵热力图', fontproperties='STKAITI', fontsize=25, pad=10)
# 设置x轴刻度线
plt.tick_params(axis='x', which='major', direction='inout', color='r', pad=10, size=8, width=3, labelsize=15, labelcolor='k')
# 设置y轴刻度线
plt.tick_params(axis='y', which='major', direction='inout', size=8, width=3, labelsize=15, labelcolor='k')
plt.show()
```

将目标变量插入到40各州的独热编码之后

使用corr函数，获得相关系数矩阵

使用heatmap函数创建热力图

图 3-1 各州与目标变量的相关系数矩阵热力图代码

得到的相关系数矩阵热力图如图 3-2 所示。

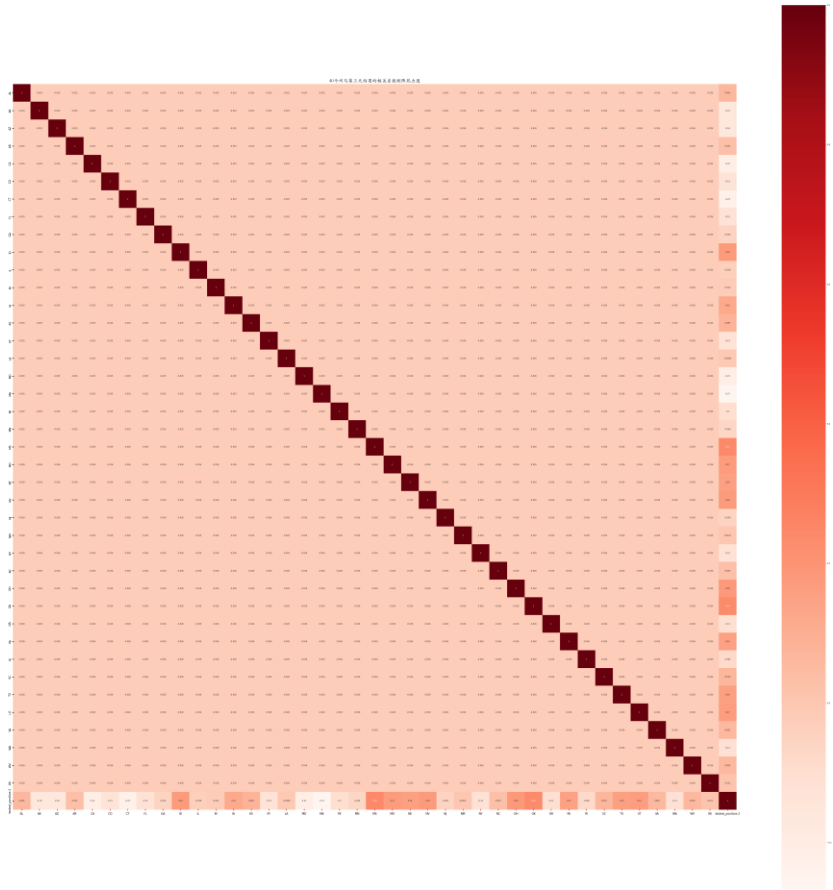

图 3-2 各州与目标变量的相关系数矩阵热力图

通过图像可以看出，各个州之间的相关性是一致的，同时各个州与目标变量之间的相关性也较低。

### 3.1.2 第一天特征值与目标变量的相关系数矩阵热力图

通过上述数据预处理中的特征值切分步骤，将第一天的数据进行切分，并命名为 covid_train_day1，然后与目标变量（tested_positive.2）进行合并，再利用的 corr()函数计算相关系数矩阵，heatmap 函数创建热力图，实现可视化，实现代码如图 3-3 所示。

```
# 第一天数据与第三天结果的相关系数矩阵热力图
#数据准备
covid_train_day1['tested_positive.2'] = covid_train['tested_positive.2']
d = covid_train_day1.corr()
plt.subplots(figsize = (40,40))
sns.heatmap(d,annot = True,vmax = 1,square = True,cmap = "Reds")
plt.show()
```

将目标变量插入第一天数据中

得到相关系数矩阵

绘制热力图

图 3-3 第一天特征值与目标变量的相关系数矩阵热力图代码

得到的相关系数矩阵热力图如图 3-4 所示。

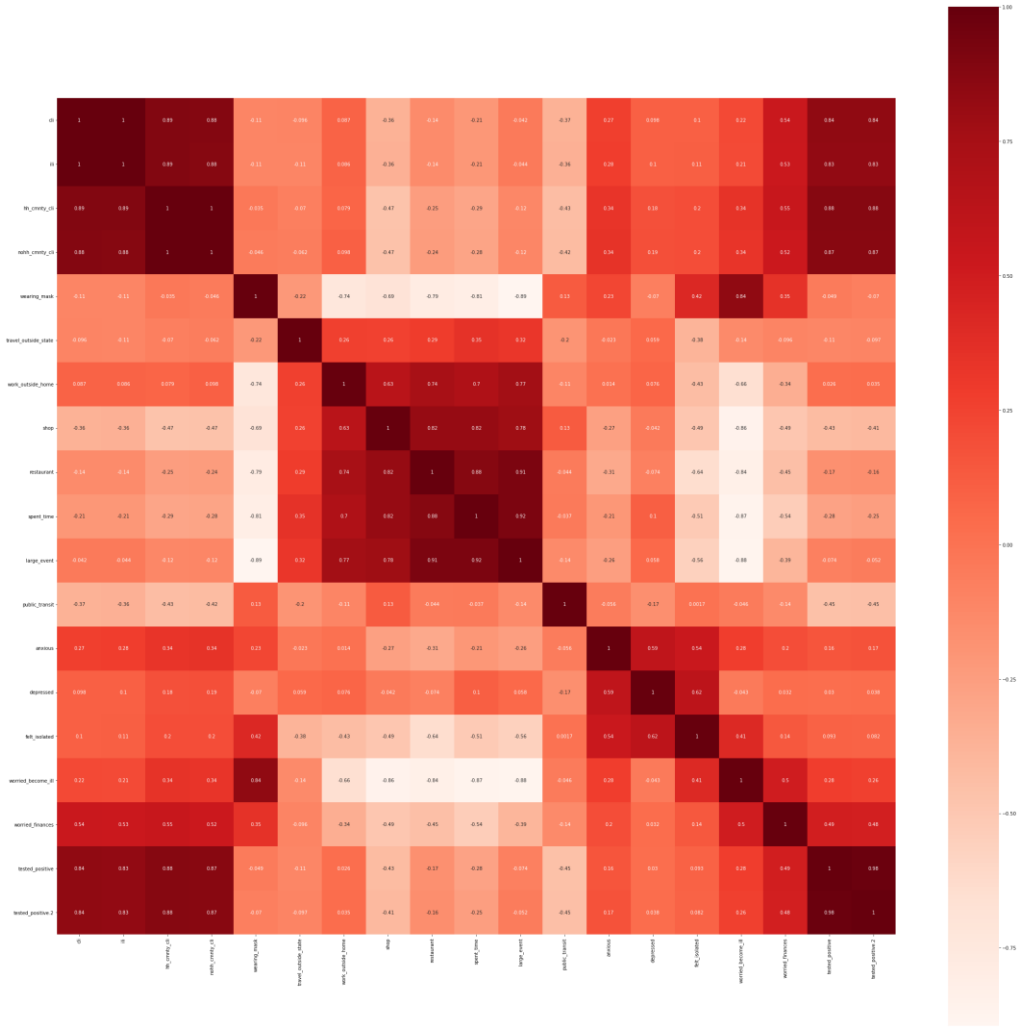

图 3-4 第一天特征值与目标变量的相关系数矩阵热力图

这里我们放大图像，来查看最后一行第一天特征值与目标变量的相关系数，如图 3-5 所示.

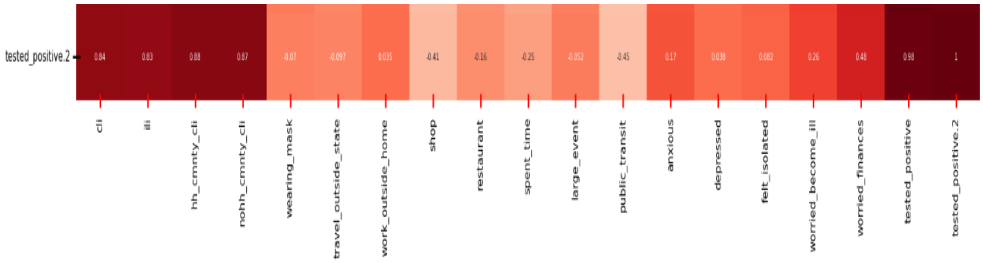


图 3-5 第一天特征值与目标变量的相关系数图

根据图像我们可以看出，类新冠特征 cli、ili、hh_cmnty_cli、nohh_cmnty_cli 与 tested_positive.2 的相关性较高相关性在均 0.80 以上，anxious、worried_become_ill、worried_finances、tested_positive 与 tested_positive.2 的相关性也很高，特别是 tested_positive，相关系数为 0.98，因此在选取特征值时，可以选择这几个特征值进行模型训练。

### 3.1.3 第二天特征值与目标变量的相关系数矩阵热力图

通过上述数据预处理中的特征值切分步骤，将第二天的数据进行切分，并命名为 covid_train_day2，然后与目标变量（tested_positive.2）进行合并，再利用的 corr()函数计算相关系数矩阵，heatmap 函数创建热力图，实现可视化，实现代码如图 3-6 所示。

```
#第二天数据与第三天结果的相关系数矩阵热力图
#数据准备
covid_train_day2['tested_positive.2'] = covid_train['tested_positive.2']
d = covid_train_day2.corr()
plt.subplots(figsize = (40,40))
sns.heatmap(d,annot = True,vmax = 1,square = True,cmap = "Reds")
# 设置标题
plt.title('第二天数据与第三天结果的相关系数矩阵热力图',fontproperties='STKAITI',fontsize=25,pad=10)
# 设置x轴刻度线
plt.tick_params(axis='x',which='major',direction='inout',color='r',pad=10,size=15,width=3,labelsize=15,labelcolor='k')
# 设置y轴刻度线
plt.tick_params(axis='y',which='major',direction='inout',size=15,width=3,labelsize=15,labelcolor='k')
plt.show()
```

将目标变量插入第二天数据中

利用corr函数获得相关系数矩阵

绘制热力图

图 3-6 第二天特征值与目标变量的相关系数矩阵热力图代码

得到的相关系数矩阵热力图同第一天效果一致，这里直接查看最后一行第二天特征值与目标变量的相关系数，如图 3-7 所示。

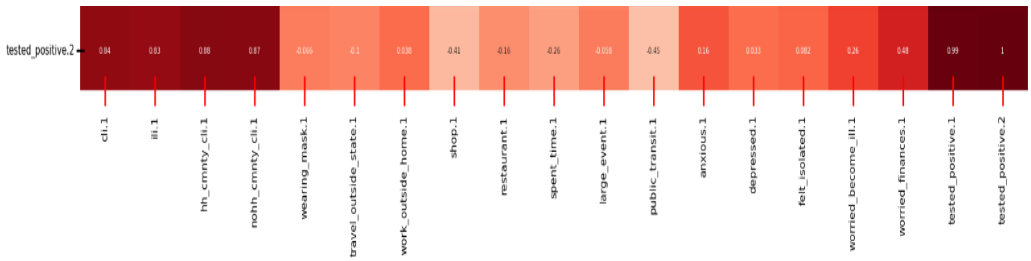


图 3-7 第二天特征值与目标变量的相关系数图

根据图像我们可以看出，类新冠特征 cli.1、ili.1、hh_cmnty_cli.1、nohh_cmnty_cli.1 与 tested_positive.2 的相关性较高，相关性均在 0.80 以上，anxious.1 、worried_become_ill.1 、worried_finances.1 、tested_positive.1 与 tested_positive.2 的相关性也很高，特别是 tested_positive.1，相关系数为 0.99，因此在选取特征值时，可以选择这几个特征值进行模型训练。

### 3.1.4 第三天特征值与目标变量的相关系数矩阵热力图

通过上述数据预处理中的特征值切分步骤，将第三天的数据进行切分，并命名为 covid_train_day3，再利用的 corr()函数计算相关系数矩阵，heatmap 函数创建热力图，实现可视化，实现代码如图 3-8 所示。

```
#第三天数据与第三天结果的相关系数矩阵热力图
# 获得相关系数矩阵
d = covid_train_day3.corr()
plt.subplots(figsize = (40,40))
sns.heatmap(d,annot = True,vmax = 1,square = True,cmap = "Reds")
# 设置标题
plt.title('第二天数据与第三天结果的相关系数矩阵热力图',fontproperties='STKAITI',fontsize=25,pad=10)
# 设置x轴刻度线
plt.tick_params(axis='x',which='major',direction='inout',color='r',pad=10,size=15,width=3,labelsize=15,labelcolor='k')
# 设置y轴刻度线
plt.tick_params(axis='y',which='major',direction='inout',size=15,width=3,labelsize=15,labelcolor='k')
plt.show()
```

获得相关系数矩阵

绘制热力图

图 3-8 第三天特征值与目标变量的相关系数矩阵热力图代码

得到的相关系数矩阵热力图同第一天、第二天效果一致，这里直接查看最后一行第三天特征值与目标变量的相关系数，如图 3-9 所示。

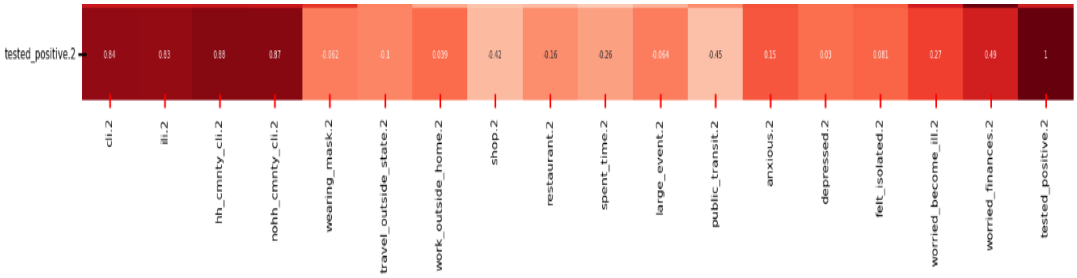


图 3-9 第三天特征值与目标变量的相关系数图

根据图像我们可以看出，类新冠特征 cli.2、ili.2、hh_cmnty_cli.2、nohh_cmnty_cli.2 与 tested_positive.2 的相关性较高，均在 0.80 以上，anxious.2 、worried_become_ill.2、worried_finances.2 与 tested_positive.2 的相关性也很高，因此在选取特征值时，可以选择这几个特征值进行模型训练。

## 3.2 特征值相关性得分分析

本次实验的相关性分数分析操作使用了 sklearn.feature_selection 中的 SelectKBest：特征选择库、f_regression：评分函数，创建了一个 SelectKBest 对象，并使用 f_regression 作为评分函数，选择最重要的前 30 个特征。使用训练数据集（train）和目标变量（target）来拟合 SelectKBest 模型，得到特征与目标变量之间的相关分数。然后将得分转换为 DataFrame 格式，并保存特征名称和对应的分数。将特征名称和分数合并成一个表格（featureScores）。然后对 featureScores 根据分数进行降序排序，以便查看分数最高的特征在前面。再输出对特征得分的统计摘要信息，包括计数、均值、方差、最小值、25% 分位数、50% 分位数（中位数）、75% 分位数和最大值。

得到特征得分的摘要信息之后，选取得分在 70-20000 之间的数据绘制特征得分条形图来观察不同特征值的分数差异，选取该得分区间的原因为得分前几名的分数较高，不设置得分上限绘制条形图可能导致得分低的特征值在条形图上无法

体现，导致可视化效果较差。经过调整参数，并最终根据输出的摘要信息，选取了分数大于前 50%与小于 20000 之间的得分绘制条形图，实现该操作的代码如图 3-10 所示。

```
]: # 特征选择库
from sklearn.feature_selection import SelectKBest
# 评分函数
from sklearn.feature_selection import f_regression
# 创建SelectKBest对象，使用f_regression作为评分函数，选择前30个最重要的特征
bestfeatures = SelectKBest(score_func=f_regression, k=30)
# 使用训练数据train和目标变量target拟合SelectKBest模型，得到特征与目标变量相关的分数
fit = bestfeatures.fit(train,target)
# 将得分转换为DataFrame格式，并保存特征名称和对应的分数
train_scores = pd.DataFrame(fit.scores_)  # 分数
train_columns = pd.DataFrame(train.columns)  # 对应的特征值
featureScores = pd.concat([train_columns,train_scores],axis=1)  # 合并表格
featureScores.columns = ['Specs','Score']  # 列命名
# 按照分数降序排序
featureScores = featureScores.sort_values(by='Score', ascending=False)
# 输出特征得分的统计摘要信息
featureScores.describe()
```

获得特征值相关性得分

```
]: # 绘制特征值分数条形图
# 选择分数大于150且小于20000的特征
filtered_features = featureScores[(featureScores['Score'] > 70) & (featureScores['Score'] < 20000)]
# 设置画布大小
plt.figure(figsize=(10, 6))
# 绘制条形图
plt.bar(filtered_features['Specs'], filtered_features['Score'])
plt.xlabel('特征值') # x轴标签
plt.ylabel('分数')  # y轴标签
plt.title('特征值分数 (150-20000)') # 设置标题
plt.xticks(rotation=90)
plt.show()
```

绘制条形图

图 3-10 特征值相关性得分代码图

得到特征值分数条形图如图 3-11 所示。

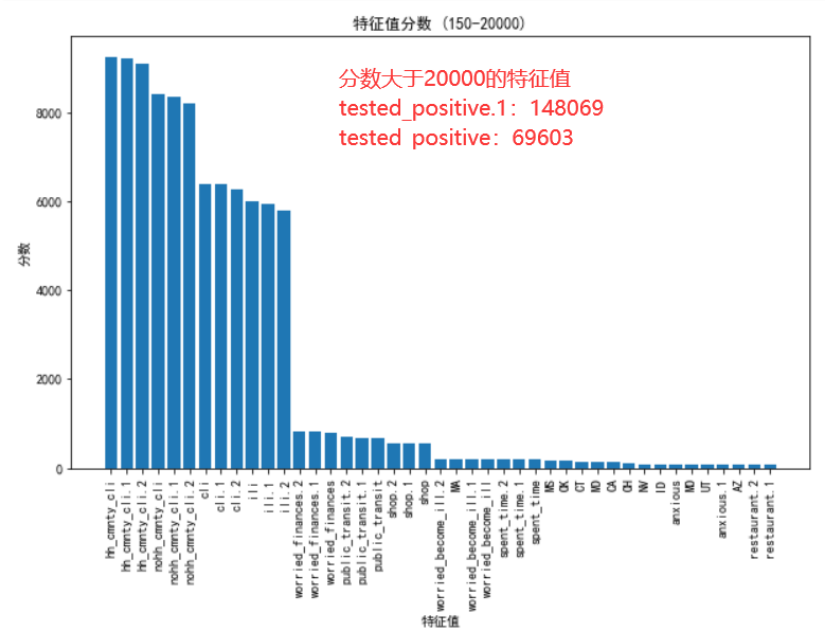


图 3-11 特征值分数条形图

## 3.3 特征值选取

1、根据相关系数矩阵热力图分析得出，各个州之间的相关性是一致的，同时各个州与目标变量之间的相关性也较低。因此在特征值选取时可以排除各个州的特征值；第一天特征值中的类新冠特征 cli、ili、hh_cmnty_cli、nohh_cmnty_cli 与 tested_positive.2 的相关性较高相关性在均 0.80 以上，anxious、worried_become_ill、worried_finances、tested_positive 与 tested_positive.2 的相关性也很高，特别是 tested_positive，相关系数为 0.98；第二天特征值中类新冠特征 cli.1、ili.1、hh_cmnty_cli.1、nohh_cmnty_cli.1 与 tested_positive.2 的相关性较高，相关性均在 0.80 以上，anxious.1 、worried_become_ill.1、worried_finances.1、tested_positive.1 与 tested_positive.2 的相关性也很高，特别是 tested_positive.1，相关系数为 0.99；第三天特征值中 cli.2、ili.2、hh_cmnty_cli.2、nohh_cmnty_cli.2 与 tested_positive.2 的相关性较高，均在 0.80 以上，anxious.2 、worried_become_ill.2、worried_finances.2 与 tested_positive.2 的相关性也很高

2、根据特征值相关性得分分析以及可视化条形图得出，得分前 30 的特征值分别为 tested_positive.1 、 tested_positive 、 hh_cmnty_cli 、 hh_cmnty_cli.1 、hh_cmnty_cli.2、nohh_cmnty_cli、nohh_cmnty_cli.1、nohh_cmnty_cli.2、cli、cli.1、cli.2、ili、ili.1、ili.2、worried_finances.2、worried_finances.1、worried_finances、public_transit.2 、 public_transit.1 、 public_transit 、 shop.2 、 shop.1 、 shop 、worried_become_ill.2、MA、worried_become_ill.1、worried_become_ill、spent_time.2、spent_time.1、spent_time。

综上所述，可以发现通过相关系数矩阵热力图与特征值相关性得分分析得到的特征值基本一致，因此，本次新冠预测实践将使用 SelectKBest 选取得分在前 30 的特征值，但由于这 30 个特征值中存在 MA 州因此需要将该州删除。实现特征值选取的代码如图 3-12 所示。

```
# 创建SelectKBest对象，使用f_regression作为评分函数，选择前30个最重要的特征
bestfeatures = SelectKBest(score_func=f_regression, k=30)
# 使用训练数据train和目标变量target拟合SelectKBest模型，得到特征与目标变量相关的分数
fit = bestfeatures.fit(train,target)
# 得到前30个特征值的索引
cols = bestfeatures.get_support(indices=True)
```

选取得分前30的特征值

```
# 为训练数据集选取特征值
train = train.iloc[:,cols]
# 为测试数据集选取特征值
test2 = covid_test1.iloc[:,cols]
# 去除MA州这一列
train1 = train.drop('MA',axis=1)
# 去除MA州这一列
test2 = test2.drop('MA',axis=1)
```

根据索引获得训练集、测试集数据

删除排名在前30中的MA州这一特征值

图 3-12 特征值选取代码图

# 第 4 章 基于线性回归的新冠病例预测分析

本次基于线性回归模型的新冠病例预测，简述了线性回归模型的原理以及公式，说明了线性回归模型是如何定义的，并详细描述了模型的训练过程，包括参数设置、模型初始化、优化器选择等，通过优化器选择和添加 L1 正则化优化了模型，并添加了 RMSE 指标，通过可视化分析进行了参数调整，得到了拟合效果较好的线性回归模型。

## 4.1 线性回归模型的原理和公式

线性回归模型是一种用于建立特征与目标变量之间线性关系的预测模型。其原理基于线性方程的表示和最小二乘法的优化。

线性回归模型的基本原理如下。

假设存在一个特征向量 $X$ 和一个目标变量 $y$ 之间的线性关系：$y=\beta_0+\beta_1x_1+\beta_2x_2+...+\beta_nx_n+\varepsilon$，其中 $\beta_0$ 是截距，$\beta_1$、$\beta_2$、...、$\beta_n$ 是特征的系数，$\varepsilon$ 是误差项或噪声。目标是通过找到最优的系数 $\beta_0$、$\beta_1$、$\beta_2$、...、$\beta_n$ 来拟合线性方程，使得预测值 $y$ 的平方误差最小化。最小二乘法是一种常用的方法，通过最小化观测值与预测值之间的残差平方和来选择最佳系数值，使得拟合线性方程最优。

线性回归模型的公式表示如下：

其中 $Y=\beta_0+\beta_1X_1+\beta_2X_2+...+\beta nXn+\varepsilon$，$Y$ 表示目标变量（要预测的值）；$X_1$、$X_2$、...、$X_n$ 表示特征变量；$\beta_0$、$\beta_1$，$\beta_2$、...、$\beta_n$ 表示特征的系数或权重，代表了每个特征对目标变量的影响程度；$\varepsilon$ 表示误差项或噪声，表示模型无法完全解释的观测值的变异。线性回归模型的目标是通过拟合数据集中的观测值来估计最优的系数值，从而使得预测值与实际观测值之间的差异最小化。这样可以建立一个线性方程模型，用于预测新的特征值对应的目标变量值。

在实际应用中，可以使用不同的方法和算法来拟合线性回归模型，如普通最小二乘法、梯度下降法等。这些方法都旨在找到最佳的系数值，以便使得线性方程模型与观测数据最好地拟合[10]。

## 4.2 定义线性回归模型的网络结构

本次新冠预测实践的线性回归模型包含一个线性层(nn.Linear)，并且在前向传播过程中直接返回线性层的输出。具体来说，使用了__init__方法nn.Linear(input_size, 1)定义了一个输入特征维度为 input_size，输出特征维度为 1 的线性层。在前向传播过程，将输入 x 通过线性层进行线性变换在 forward 方法中，通过 self.fc(x)调用线性层将输入 x 进行线性变换，并将结果作为模型的输出返回。

该线性回归模型可以接受 input_size 维度的输入，并生成一维的输出。在模型训练过程中，并使用损失函数（均方误差）和优化算法（如随机梯度下降 SGD 或 Adam 优化器）对模型进行训练，以使输出值与真实值之间的误差最小化，实现定义线性回归模型的代码如图 4-1 所示。

```
# 定义线性回归模型类
class LinearRegression(nn.Module):
    def __init__(self, input_size):
        # 调用父类nn.Module的构造方法，以确保正确初始化线性回归模型的属性和状态。
        super(LinearRegression, self).__init__()
        # 定义线性层，输入特征维度为input_size，输出特征维度为1
        self.fc = nn.Linear(input_size, 1)

    def forward(self, x):
        # 前向传播过程，将输入x通过线性层进行线性变换
        out = self.fc(x)
        return out
```

图 4-1 定义线性回归模型代码图

## 4.3 描述线性回归模型的训练

### 4.3.1 定义超参数

1、设置随机种子

通过设置随机种子，可以使得随机过程在每次运行时产生相同的随机数序列，从而使得结果可重现。在本次线性回归模型中，使用了 torch.manual_seed(42)将随机种子设置为 42。这意味着在后续的代码中，使用 PyTorch 库的随机函数时，将使用相同的随机数生成器种子，从而产生相同的随机数序列。设置随机种子对于调试和复现实验结果是非常有用的。通过固定随机种子，可以确保在多次运行相同代码时获得相同的随机初始化、随机采样或随机数生成结果，从而使得实验结果更加可靠和一致，设置随机种子代码如图 4-2 所示。

```
# 设置随机种子
torch.manual_seed(42)
```

```
<torch._C.Generator at 0x21340c7fd70>
```

图 4-2 设置随机种子代码图

2、设置超参数

X_train.shape[1]表示训练集的特征数据 X_train 的第二个维度，即特征的数量。将其赋值给 input_dim 变量，表示输入特征的维度。

lr = 0.03，将学习率设置为 0.03。学习率是控制模型在每次参数更新时的步长大小。较大的学习率可以加快模型的收敛速度，但可能会导致不稳定的训练过程。较小的学习率可以提高训练的稳定性，但可能需要更多的训练迭代次数才能达到较好的结果。学习率的选择需要根据具体问题和实验结果进行调整。

num_epochs=10001，设置了迭代次数为 10001，表示模型将进行 10001 轮的训练。迭代次数的选择需要根据具体数据集和模型的复杂程度进行调整，以获得最佳的模型性能。

通过设置输入特征的维度、学习率和迭代次数，我们可以对模型的输入、训练速度和训练轮数进行相应的配置，以满足实际需求和获得最佳的模型效果，设置超参数代码如图 4-3 所示。

```
# 设置输入特征的维度
input_dim = X_train.shape[1]
# 设置学习率
lr = 0.03
# 设置迭代次数
num_epochs = 10001
```

图 4-3 设置超参数代码图

### 4.3.2 模型初始化与优化器选择

模型初始化，通过使用之前定义的 LinearRegression 类创建了一个模型对象 model。这样做是为了构建一个线性回归模型，其中 input_dim 是输入特征的维度。

接下来，我们定义了损失函数和优化器。criterion 使用的是均方误差损失函数 MSE（Mean Squared Error），该损失函数用于衡量模型预测值和真实值之间的差异。optimizer 使用的是随机梯度下降（Stochastic Gradient Descent，SGD）优化器，用于优化模型的参数，其中 lr 是学习率。

通过定义损失函数和优化器，为本次新冠预测实践的线性回归模型训练提供

了计算损失和更新参数的工具。损失函数用于计算模型的预测值与真实值之间的误差，而优化器则使用该误差来更新模型的参数，以最小化损失函数。

这样就完成了线性回归模型的初始化和定义损失函数与优化器的过程，为接下来的模型训练做好了准备，模型初始化与优化器选择的代码如图 4-4 所示。

```
# 模型初始化
model = LinearRegression(input_dim)
# 定义损失函数和优化器
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr)
```

图 4-4 模型初始化与优化器选择代码图

### 4.3.3 训练模型与结果评估

在这段代码中，首先定义了存储损失的列表 train_losses、train_epochs，用来存储迭代次数和每次迭代的损失值，使用了一个 for 循环来进行模型的训练。每个训练周期（epoch）中，首先进行前向传播，计算模型的输出和损失。然后进行反向传播，通过调用 backward() 方法计算梯度并更新模型的参数。最后，通过调用 step() 方法，根据优化器的设定来更新模型的参数。

然后使用 if 条件语句设置每 100 次迭代为一个周期，在每个训练周期结束后，打印出当前周期的损失值。同时，将损失值和训练周期数存储在列表 train_losses 和 train_epochs 中，以便后续的可视化和分析。

通过这段代码，完成模型的训练过程，并在训练过程中实时监控损失值的变化。训练周期数和损失值的输出可以根据具体情况进行调整，训练模型的代码如图 4-5 所示。

```
# 定义存储损失的列表
train_losses = []
train_epochs = []
# 训练模型
for epoch in range(num_epochs):
    # 前向传播
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    # 反向传播和优化器更新
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# 打印损失
    if (epoch+1) % 100 == 0:
        train_losses.append(loss.item())
        train_epochs.append(epoch+1)

        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

图 4-5 训练模型代码图

得到的打印结果如图 4-6 所示。

```
Epoch [8900/10001], Loss: 57.8335
Epoch [9000/10001], Loss: 57.8335
Epoch [9100/10001], Loss: 57.8335
Epoch [9200/10001], Loss: 57.8335
Epoch [9300/10001], Loss: 57.8335
Epoch [9400/10001], Loss: 57.8335
Epoch [9500/10001], Loss: 57.8335
Epoch [9600/10001], Loss: 57.8335
Epoch [9700/10001], Loss: 57.8335
Epoch [9800/10001], Loss: 57.8335
Epoch [9900/10001], Loss: 57.8335
Epoch [10000/10001], Loss: 57.8335
```

图 4-6 训练模型结果图

通过打印结果发现，迭代次数 100 到 10000 次，函数的损失值均在 57.8 上下浮动，损失值较高，且不会随迭代次数明显下降，可能有以下几个原因：

模型过于简单：线性回归模型是一个比较简单的模型，对于复杂的数据集和非线性关系可能无法很好地拟合。如果数据集具有复杂的特征和模式，线性回归模型可能无法提供足够的灵活性来捕捉这些特征，导致损失值较高。

学习率过高或过低：学习率是优化算法中一个重要的超参数，控制每次参数更新的步长。如果学习率过高，参数更新可能会跳过最优点，导致损失值无法收敛；如果学习率过低，参数更新可能会非常缓慢，需要更多的迭代次数才能达到较好的结果。建议尝试调整学习率的大小，找到一个合适的值。

模型欠拟合：如果模型的表达能力不足，无法很好地拟合训练数据，也会导致损失值较高。可以考虑使用更复杂的模型，如多层感知机、循环神经网络或卷积神经网络等，以提高模型的表达能力。

因此需要进一步调整参数或修改模型来进行优化。

### 4.3.4 可视化分析

1、损失值函数图

绘制损失值函数图使用 Matplotlib 库绘制了损失函数随迭代次数的变化曲线。train_epochs 是存储训练轮数的列表，train_losses 是存储训练过程中损失函数值的列表。通过调用 plt.plot()函数，将训练轮数和对应的损失函数值传递给该函数，可以绘制出损失函数曲线。plt.xlabel()和 plt.ylabel()设置 x 轴和 y 轴的标签，plt.title()设置图表的标题，plt.legend()添加图例。使用 plt.yticks() 函数可以设置 y 轴刻度的范围和间隔。使用 np.arange(57.83, 57.84, 0.005) 生成了一个从 57.83 到 57.84 的刻度值列表，间隔为 0.005。根据实际情况，您可以根据需求调整刻度值的范围和间隔。最后，使用 plt.show() 显示绘制的图表，绘制损失值函数图代码如图 4-7

所示。

```
# 绘制损失函数曲线图
plt.plot(train_epochs, train_losses, label='训练损失')
plt.xlabel('迭代次数')
plt.ylabel('损失值')
plt.title('损失函数曲线图')
plt.yticks(np.arange(57.83, 57.84, 0.005))
plt.legend()
plt.show()
```

设置y轴区间

图 4-7 损失值函数代码图

得到的图像如图 4-8 所示。

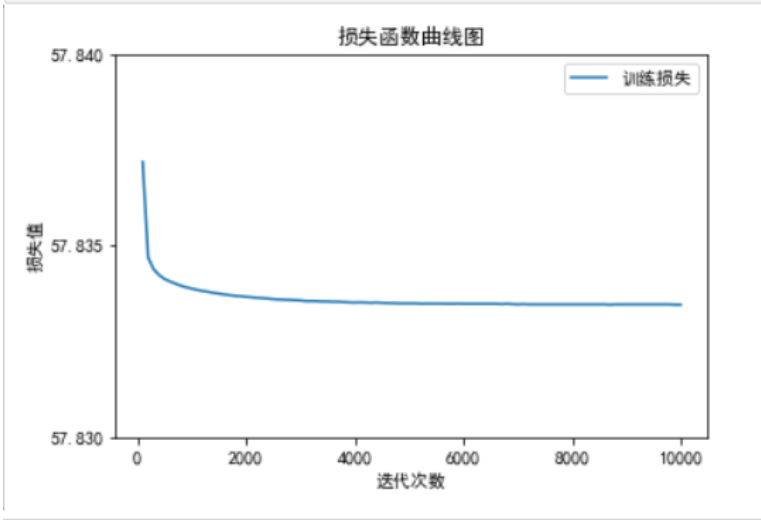


图 4-8 损失值函数图

通过图像可以看到，该线性回归模型的损失率只在 57.835 附近，损失值很高，且不会随着迭代次数而迅速下降，因此需要进一步优化该线性回归模型。

2、真实值与预测值对比图

通过数据预处理，将训练数据集（covid_train）进行了划分，将 20%的样本作为测试集，80%的样本作为训练集。使用训练集用来拟合模型，测试集来查看模型拟合效果。该步骤先将测试集数据 X_test 转换为 PyTorch 张量类型 torch.Tensor。然后，通过将测试集数据输入到模型中进行预测，使用 model(test_inputs) 得到预测结果。接着，通过 squeeze()方法将结果的维度压缩，使其成为一维数组。最后，使用 detach().numpy()将结果从 PyTorch 张量类型转换为 NumPy 数组，以便进行后续的处理和分析。可视化部分通过 plt.plot()函数分别传入真实值 y_test.squeeze()和预测值 predictions，绘制了两条曲线。并通过调用 plt.legend()添加图例，将真实值和预测值对应的图例显示在图表上方。最后，使用 plt.show()显示绘制的图表，以进行可视化分析。测试集模型预测和可视化的代码如图 4-9 所示。

```
# 模型预测
test_inputs = torch.Tensor(X_test)
predictions = model(test_inputs).squeeze().detach().numpy()
```

```
# 可视化分析
plt.plot(y_test.squeeze(), label='真实值')
plt.plot(predictions, label='预测值')
plt.title('真实值与预测值对比图')
plt.legend()
plt.show()
```

图 4-9 测试集模型预测和可视化的代码图

得到的结果如图 4-10 所示。

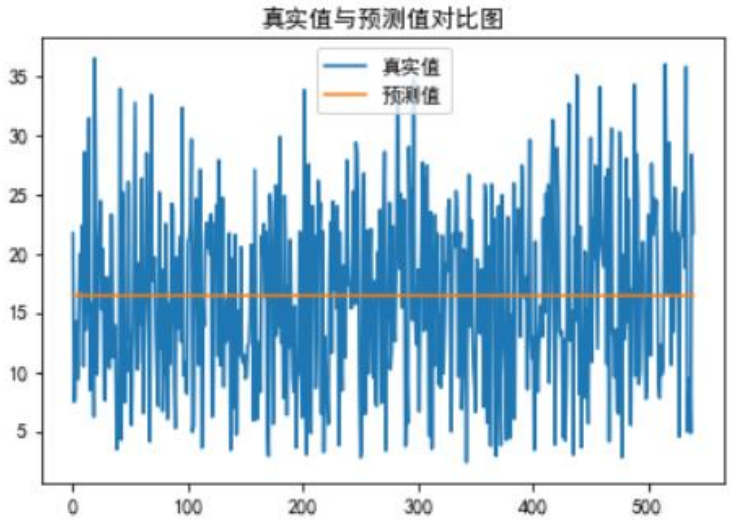


图 4-10 真实值与预测值对比图

通过图像可以看出，因为模型拟合效果较差，导致预测值基本在一条线上，与其对应的真实值存在很大的差别，因此需要调整参数和进行模型优化。

## 4.4 模型优化与调整参数设置

### 4.4.1 模型优化

1、优化器选择

上述步骤的优化器 optimizer 使用的是随机梯度下降（Stochastic Gradient Descent，SGD）优化器，优化后将选择 Adam 为优化器。SGD（随机梯度下降）和 Adam 是两种常用的优化器算法，用于训练神经网络模型时更新模型参数。它们在更新参数的方式和效果上存在一些区别。

SGD（随机梯度下降）：

（1）SGD 是一种基本的优化算法，在每个训练样本上计算梯度并更新参数。

每次迭代时，SGD 随机选择一个样本计算梯度，并根据梯度更新参数。

（2）SGD 的更新步长（学习率）是固定的，可以手动设置或随着训练过程进行调整。

（3）SGD 可能会受到局部极小值的影响，因为它只使用一个样本来更新参数，可能会导致参数更新的方向不够准确[11]。

Adam：

（1）Adam 是一种自适应学习率的优化算法，结合了动量方法和自适应学习率方法。

（2）Adam 在每个训练样本上计算梯度，并使用动量方法平均梯度和梯度平方的指数移动平均。

（3）Adam 根据梯度的一阶矩估计（平均梯度）和二阶矩估计（梯度平方的指数移动平均）自适应地调整学习率。

（4）Adam 可以更快地收敛，尤其是在具有大量参数和复杂结构的模型中。

但在某些情况下，Adam 可能会导致过度拟合，因为它会对梯度进行自适应调整。

总体而言，SGD 是一种简单的优化算法，适用于较小的数据集和简单的模型。Adam 是一种更复杂的优化算法，适用于大型数据集和复杂的深度学习模型[12]，修改优化器代码如图 4-11 所示。

```
# 使用Adam优化器
optimizer = torch.optim.Adam(model.parameters(), lr)
```

图 4-11 使用 Adam 优化器代码图

2、添加 L1 正则化

（1）L1 正则化的优点

L1 正则化是一种用于机器学习和统计建模的正则化方法，也被称为 Lasso 正则化。它通过在损失函数中添加 L1 范数惩罚来限制模型参数的大小。在 L1 正则化中，损失函数由两部分组成：第一部分是原始的目标函数，用于衡量模型在训练数据上的拟合程度；第二部分是 L1 范数惩罚项，用于惩罚模型参数的绝对值之和。L1 正则化鼓励模型参数稀疏化，即使得一些参数变为零，从而实现特征选择的效果。这是因为 L1 范数惩罚项具有一种稀疏化效果，能够将对模型贡献较小的参数压缩为零，只保留对模型贡献较大的参数。相比于 L2 正则化（Ridge 正则化），L1 正则化更适用于特征选择问题，因为它能够自动找到对预测结果具有重要影响的特征。另外，L1 正则化也可以用于解决过拟合问题，通过约束模型参数的大小，减少模型复杂性，提高泛化性能。

总结来说，L1 正则化是一种适用于特征选择和过拟合问题的正则化方法，通过添加 L1 范数惩罚项来限制模型参数的大小，并实现对模型参数的稀疏化[13]。

（2）L1 正则化权重的自动选取

该步骤使用了 scikit-learn 库的 LassoCV 实现实现 L1 正则化权重的自动选取，可以使用交叉验证来估计模型在不同正则化权重下的性能，并选择表现最佳的权重。在这个步骤中，我们使用了 LassoCV 类，它是基于交叉验证的 Lasso 回归模型。通过指定不同正则化路径（alphas）和交叉验证的折数（cv），LassoCV 会在训练数据上进行交叉验证，找到表现最佳的正则化权重。最后，通过访问 LassoCV 对象的 alpha_属性，就可以获取最佳正则化权重的值。这个值就是在交叉验证过程中得到的最优权重。

（3）计算 L1 正则化项，并将其添加到总损失中

首先，为了防止原位操作，需要创建一个名为 l1_regularization 的初始值为 0 的 Tensor 变量，用于累加每个模型参数的 L1 范数。

其次，在循环中，对模型的每个参数进行迭代，并使用 torch.norm 函数计算每个参数的 L1 范数，并将其加到 l1_regularization 上。

最后，通过将 best_l1_lambda（L1 正则化的权重）乘以 l1_regularization 并加到总损失 loss 上，计算带有 L1 正则化的总损失 l1_loss。

最后，将 l1_loss 作为总损失传递给优化器进行梯度下降，以更新模型的参数。

L1 正则化权重的自动选取代码如图 4-12 所示，计算带有 L1 正则化损失的代码如图 4-13 所示。

```
# L1正则化权重的自动选取
from sklearn.linear_model import LassoCV
# 创建一个LassoCV对象，指定正则化路径和交叉验证的折数
lasso_cv = LassoCV(alphas=[0.0001,0.001, 0.01, 0.1, 1.0], cv=5,max_iter=100000)
# 在训练数据上拟合模型
lasso_cv.fit(X_train, y_train)
# 选择最佳正则化权重
best_l1_lambda = lasso_cv.alpha_
# 打印输出最佳的L1正则化权重
print("best_l1_lambda:", best_l1_lambda)
```

```
best_l1_lambda: 0.001
```

图 4-12 L1 正则化权重的自动选取代码图

```
# 计算L1正则化项
# 创建新的变量以避免原位操作
l1_regularization = torch.tensor(0., dtype=torch.float32)
for param in model.parameters():
    l1_regularization += torch.norm(param, p=1)

# 计算带有L1正则化的总损失
l1_loss = loss + best_l1_lambda * l1_regularization

# 反向传播和优化器更新
optimizer.zero_grad()
l1_loss.backward()
optimizer.step()
```

l1_loss作为总损失传递给优化器进行梯度下降

图 4-13 计算带有 L1 正则化损失代码图

3、添加 RMSE 指标

（1）RMSE 的介绍

RMSE（Root Mean Squared Error）是一种常用的评估回归模型性能的指标，它衡量了预测值与实际观测值之间的误差。

RMSE 是通过以下步骤计算得出的：对于每个数据样本，计算预测值与实际观测值之间的差值（残差）。对这些差值进行平方。对所有平方差值求和。取平均值。对平均值求平方根。数学公式表示为：$RMSE=\sqrt{\frac{1}{N}\sum_{i=1}^{n}(Y_i-f(x_i))^2}$。其中，$N$ 是数据样本的数量，$Y_i$ 是预测值，$f(x_i)$ 是实际观测值。

RMSE 具有以下特点：RMSE 值始终为非负数。当预测值与实际观测值完全匹配时，RMSE 为 0，表示模型对数据的拟合非常好。RMSE 受异常值的影响较大，因为它使用了平方差值。RMSE 的单位与原始数据的单位相同，因此可以直观地理解预测误差的大小。RMSE 通常用于评估回归模型的性能，尤其在对误差敏感的问题中。它提供了一个统一的度量方式，可以比较不同模型的拟合能力，并帮助选择最佳模型或调整模型参数。此外，RMSE 还可以用作模型优化的损失函数，以指导模型在训练过程中的更新[14]。

（2）添加 RMSE 指标

首先，通过使用 torch.no_grad()上下文管理器来关闭梯度计算，在模型上进行推断得到训练集的预测值 train_predictions。

然后，使用定义好的损失函数 criterion 计算预测值与训练集标签 y_train 之间的均方误差（MSE）。

接下来，将训练集标签 y_train 转换为张量，并调整其形状为(-1, 1)，以确保与预测值相匹配。

最后，使用 torch.sqrt()函数计算均方误差的平方根，即均方根误差（RMSE），并将其赋值给变量 rmse。可以将 RMSE 添加到一个列表中以进行记录或进一步分析，添加 RMSE 指标的代码如图 4-14 所示。

```
# 在训练集上计算MSE
with torch.no_grad():
    train_predictions = model(X_train)
    mse = criterion(train_predictions, y_train)
# 将训练集标签转换为张量
y_train = y_train.view(-1, 1)
# 计RMSE并将其添加到列表中
rmse = torch.sqrt(mse)
```

图 4-14 添加 RMSE 指标代码图

4、优化后的模型训练与结果评估（学习率：0.03，迭代次数：10001）

该步骤是训练优化后的模型并记录损失的代码。以下是对每个步骤的解释：

（1）定义存储损失的列表。

train_losses：用于存储训练过程中的损失值。

train_epochs：用于存储每个 epoch 的数量。

rmse_list：用于存储每个 epoch 的 RMSE 值。

（2）开始训练循环（共进行 num_epochs 次迭代）。

进行前向传播以获取模型的输出。

计算损失函数，这里使用了交叉熵损失函数（criterion）。

（3）计算 L1 正则化项。

创建一个初始值为 0 的张量 l1_regularization。

对模型的每个参数进行遍历，并计算该参数的 L1 范数，将其加到 l1_regularization 中。

（4）计算带有 L1 正则化的总损失。

将 L1 正则化项乘以 l1_lambda，得到 L1 正则化惩罚项。

将 L1 正则化惩罚项与原始损失相加，得到带有 L1 正则化的总损失 l1_loss。

（5）反向传播和优化器更新。

清除优化器的梯度。

对 l1_loss 进行反向传播。

使用优化器进行参数更新。

（6）在训练集上计算 MSE 和 RMSE：

使用训练好的模型对训练集进行前向传播，得到预测值 train_predictions。

计算预测值与真实标签 y_train 之间的均方误差（MSE）。

（7）将训练集标签转换为张量：

将训练集标签 y_train 进行形状变换，将其从一维向量转换为二维张量。

（8）计算 RMSE 并将其添加到列表中：

通过对 MSE 进行平方根操作来计算 RMSE。

（9）打印损失：

每经过 100 个 epoch，将当前 epoch 的损失、RMSE 等信息打印出来，并将 loss.item()、rmse.item()等值添加到相应的列表中，以便于进行可视化操作，训练优化后的模型代码如图 4-15 所示。

```python
# 定义存储损失的列表
train_losses = []
train_epochs = []
rmse_list = []
for epoch in range(num_epochs):
    # 前向传播
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    # 计算L1正则化项
    # 创建新的变量以避免原位操作
    l1_regularization = torch.tensor(0., dtype=torch.float32)
    for param in model.parameters():
        l1_regularization += torch.norm(param, p=1)
    # 计算带有L1正则化的总损失
    l1_loss = loss + best_l1_lambda * l1_regularization
    # 反向传播和优化器更新
    optimizer.zero_grad()
    l1_loss.backward()
    optimizer.step()
    # 在训练集上计算MSE
    with torch.no_grad():
        train_predictions = model(X_train)
        mse = criterion(train_predictions, y_train)
    # 将训练集标签转换为张量
    y_train = y_train.view(-1, 1)
    # 计RMSE并将其添加到列表中
    rmse = torch.sqrt(mse)
    # 每迭代100次打印损失
    if (epoch+1) % 100 == 0:
        # 将数据存储到列表中，便于可视化分析
        train_losses.append(loss.item())
        train_epochs.append(epoch+1)
        rmse_list.append(rmse.item())
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, RMSE: {rmse:.4f}')
```

定义存储损失值的列表

开始循环

计算L1正则化项

计算带有L1正则化的损

对带有L1正则化的损失进行反向传播

对MSE进行开平方得到RMSE

图 4-15 训练优化后的模型代码图

得到优化后的模型打印结果如图 4-16 所示。

```
Epoch [100/10001], Loss: 192.4205, RMSE: 13.8426
Epoch [200/10001], Loss: 124.5816, RMSE: 11.1364
Epoch [300/10001], Loss: 77.8833, RMSE: 8.8036
Epoch [400/10001], Loss: 46.7216, RMSE: 6.8171
Epoch [500/10001], Loss: 26.7983, RMSE: 5.1618
Epoch [600/10001], Loss: 14.7387, RMSE: 3.8273
Epoch [700/10001], Loss: 7.8812, RMSE: 2.7985
Epoch [800/10001], Loss: 4.2360, RMSE: 2.0520
Epoch [900/10001], Loss: 2.4302, RMSE: 1.5550
Epoch [1000/10001], Loss: 1.5967, RMSE: 1.2615
Epoch [1100/10001], Loss: 1.2365, RMSE: 1.1110
Epoch [1200/10001], Loss: 1.0881, RMSE: 1.0427
Epoch [1300/10001], Loss: 1.0270, RMSE: 1.0132
Epoch [1400/10001], Loss: 0.9991, RMSE: 0.9995
Epoch [1500/10001], Loss: 0.9833, RMSE: 0.9915
```

```
Epoch [8500/10001], Loss: 0.9000, RMSE: 0.9487
Epoch [8600/10001], Loss: 0.9001, RMSE: 0.9488
Epoch [8700/10001], Loss: 0.9000, RMSE: 0.9487
Epoch [8800/10001], Loss: 0.9004, RMSE: 0.9489
Epoch [8900/10001], Loss: 0.9005, RMSE: 0.9488
Epoch [9000/10001], Loss: 0.9005, RMSE: 0.9489
Epoch [9100/10001], Loss: 0.9001, RMSE: 0.9487
Epoch [9200/10001], Loss: 0.9000, RMSE: 0.9487
Epoch [9300/10001], Loss: 0.9006, RMSE: 0.9490
Epoch [9400/10001], Loss: 0.9000, RMSE: 0.9487
Epoch [9500/10001], Loss: 0.9000, RMSE: 0.9487
Epoch [9600/10001], Loss: 0.9002, RMSE: 0.9488
Epoch [9700/10001], Loss: 0.9003, RMSE: 0.9488
Epoch [9800/10001], Loss: 0.9004, RMSE: 0.9488
Epoch [9900/10001], Loss: 0.9007, RMSE: 0.9489
Epoch [10000/10001], Loss: 0.9001, RMSE: 0.9487
```

图 4-16 优化后模型训练结果图（前后各 15 行）

通过打印的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型收敛后的损失值从 57.83 附近下降到了 0.90 附近，这说明通过添加 L1 正则化以及更换优化器使该线性回归模型的拟合效果有了很大的提升。但损失值在迭代 2000 次左右就已经收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型。

5、可视化分析

在 4.3.4 节可视化分析中，已经说明了该线性回归模型所用的可视化代码，这里不再进行说明。

（1）优化模型后的训练损失与均方根误差曲线图

通过可视化代码，得到优化模型后的训练损失与均方根误差曲线图，如图 4-17 所示。

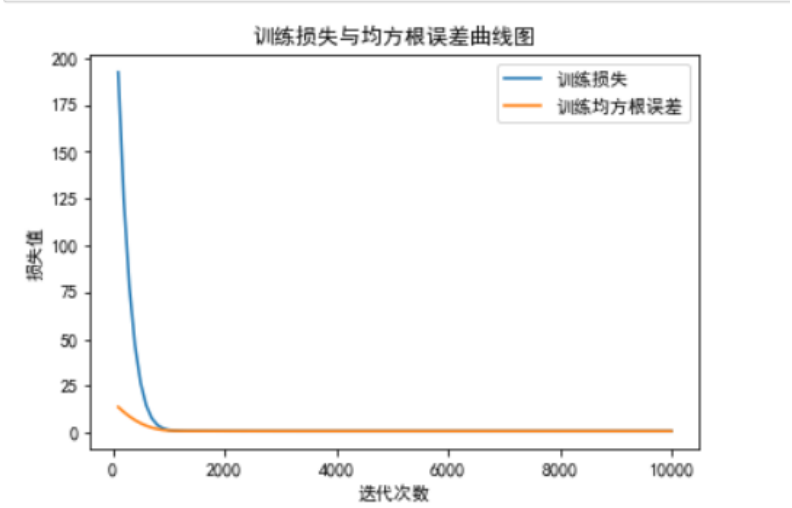


图 4-17 优化模型后的损失值与均方根误差图

通过可视化的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型收敛后的损失值从 57.83 附近下降到了 0 附近，这说明通过添加 L1 正则化以及更换优化器使该线性回归模型的拟合效果有了很大的提升。但损失值在迭代 2000 次左右就已经收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型。

（2）优化模型后的真实值与预测值对比图

通过可视化代码，得到优化模型后的真实值与预测值对比图，如图 4-18 所示。

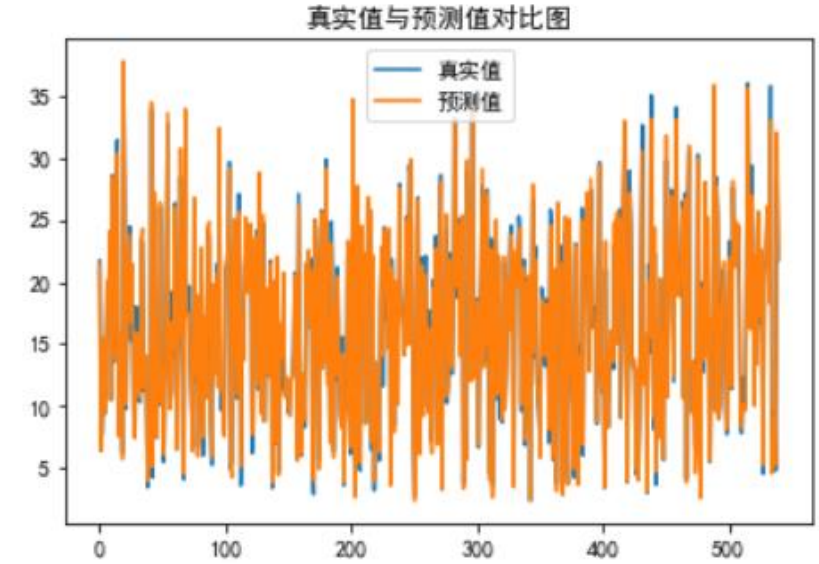


图 4-18 优化模型后的真实值与预测值对比图

通过图像可以得出，优化后的模型得到的预测值与真实值基本一致，说明该

线性回归模型拟合程度较高。

### 4.4.2 调整参数设置

通过模型优化可以确定，该优化模型拟合效果较好，但通过可视化的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，损失值在迭代 2000 次左右就已经收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型，获得效果最好的参数设置。

1、迭代次数：1001，学习率：0.05

（1）训练并打印结果

根据优化模型的结果，迭代次数在 10001 次、学习率为 0.03 时，模型损失值随迭代次数下降速度较快，损失值在迭代 2000 次左右就已经收敛，为了提高模型拟合效率，需要减少迭代次数，同时为了防止结果未收敛还需要提高学习率。因此选择迭代次数为 1001、学习率为 0.05，但由于优化模型时，迭代次数为 10001，所以选择每 100 次为周期打印结果，需要改变打印周期，以 10 次为一个周期打印结果。这里选择前 20 次和后 20 次的迭代结果，如图 4-19 所示。

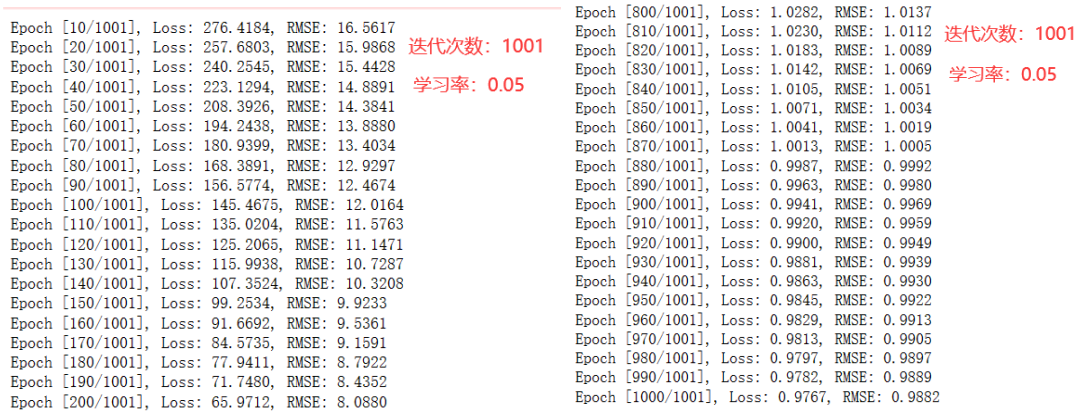


图 4-19 调整参数后的训练结果图（前、后各 20 行数据）

通过打印结果可以发现，在迭代次数为 1001 次、学习率为 0.05 时，损失值为 0.9767，该模型还未收敛，因此还需要提高学习率来加速收敛。

（2）可视化分析

由于真实值与预测值对比图只是查看模型拟合效果，其可视化分析对迭代次数、学习率的参数调整作用不大，因此，这里只分析损失值与均方误差的图像，修改参数后（迭代次数：1001，学习率：0.05）的损失值与均方误差图像如图 4-20 所示。

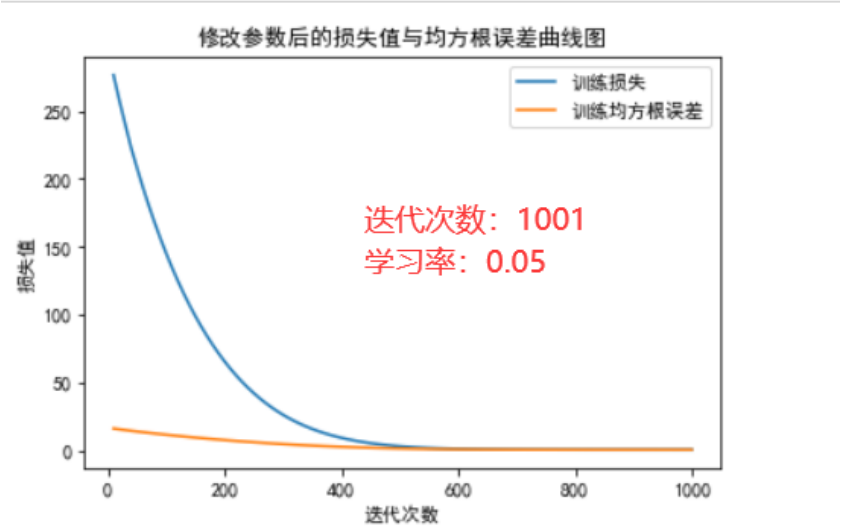


图 4-20 调整参数后的损失值与均方误差图

通过图像可以看出，迭代次数在 1001，学习率为 0.05 时，损失值随迭代次数的增大，其下降速度较为平缓，而均方误差随迭代次数的增大，其变化波动较小。

2、迭代次数：1001，学习率：0.3

（1）训练并打印结果

根据第一次调整参数的结果，迭代次数在 1001 次、学习率为 0.05 时，模型损失值随迭代次数下降速度较为平缓，但在迭代结束之后，损失值还未收敛，因此需要提高学习率，因此选择迭代次数为 1001、学习率为 0.3，这里选择前 20 次和后 20 次的迭代结果，如图 4-21 所示。

```
Epoch [10/1001], Loss: 217.6547, RMSE: 14.1216
Epoch [20/1001], Loss: 128.6456, RMSE: 11.2076
Epoch [30/1001], Loss: 77.2530, RMSE: 8.5777
Epoch [40/1001], Loss: 40.1514, RMSE: 6.0666
Epoch [50/1001], Loss: 18.7274, RMSE: 4.1817
Epoch [60/1001], Loss: 8.4123, RMSE: 2.7655
Epoch [70/1001], Loss: 3.5444, RMSE: 1.8176
Epoch [80/1001], Loss: 1.7925, RMSE: 1.3005
Epoch [90/1001], Loss: 1.2587, RMSE: 1.1124
Epoch [100/1001], Loss: 1.1369, RMSE: 1.0654
Epoch [110/1001], Loss: 1.1196, RMSE: 1.0576
Epoch [120/1001], Loss: 1.1114, RMSE: 1.0536
Epoch [130/1001], Loss: 1.1003, RMSE: 1.0484
Epoch [140/1001], Loss: 1.0894, RMSE: 1.0433
Epoch [150/1001], Loss: 1.0804, RMSE: 1.0390
Epoch [160/1001], Loss: 1.0727, RMSE: 1.0354
Epoch [170/1001], Loss: 1.0659, RMSE: 1.0321
Epoch [180/1001], Loss: 1.0594, RMSE: 1.0290
Epoch [190/1001], Loss: 1.0533, RMSE: 1.0260
Epoch [200/1001], Loss: 1.0474, RMSE: 1.0231
```

迭代次数：
学习率：0.3
前20行数据

```
Epoch [800/1001], Loss: 0.9073, RMSE: 0.9525
Epoch [810/1001], Loss: 0.9070, RMSE: 0.9523
Epoch [820/1001], Loss: 0.9067, RMSE: 0.9522
Epoch [830/1001], Loss: 0.9064, RMSE: 0.9520
Epoch [840/1001], Loss: 0.9061, RMSE: 0.9519
Epoch [850/1001], Loss: 0.9059, RMSE: 0.9518
Epoch [860/1001], Loss: 0.9056, RMSE: 0.9516
Epoch [870/1001], Loss: 0.9054, RMSE: 0.9515
Epoch [880/1001], Loss: 0.9052, RMSE: 0.9514
Epoch [890/1001], Loss: 0.9050, RMSE: 0.9513
Epoch [900/1001], Loss: 0.9048, RMSE: 0.9512
Epoch [910/1001], Loss: 0.9046, RMSE: 0.9511
Epoch [920/1001], Loss: 0.9044, RMSE: 0.9510
Epoch [930/1001], Loss: 0.9043, RMSE: 0.9509
Epoch [940/1001], Loss: 0.9041, RMSE: 0.9509
Epoch [950/1001], Loss: 0.9040, RMSE: 0.9508
Epoch [960/1001], Loss: 0.9039, RMSE: 0.9507
Epoch [970/1001], Loss: 0.9038, RMSE: 0.9507
Epoch [980/1001], Loss: 0.9036, RMSE: 0.9506
Epoch [990/1001], Loss: 0.9035, RMSE: 0.9505
Epoch [1000/1001], Loss: 0.9034, RMSE: 0.9505
```

迭代次数：1001
学习率：0.3
后20行数据

图 4-21 调整参数后的训练结果图（前、后各 20 行数据）

通过打印的结果可以得出，迭代次数在 1001、学习率为 0.3 时，该模型损失值随迭代次数下降速度较快，损失值已收敛。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：1001，学习率：0.3）的损失值与均方误差图像如图 4-22 所示。

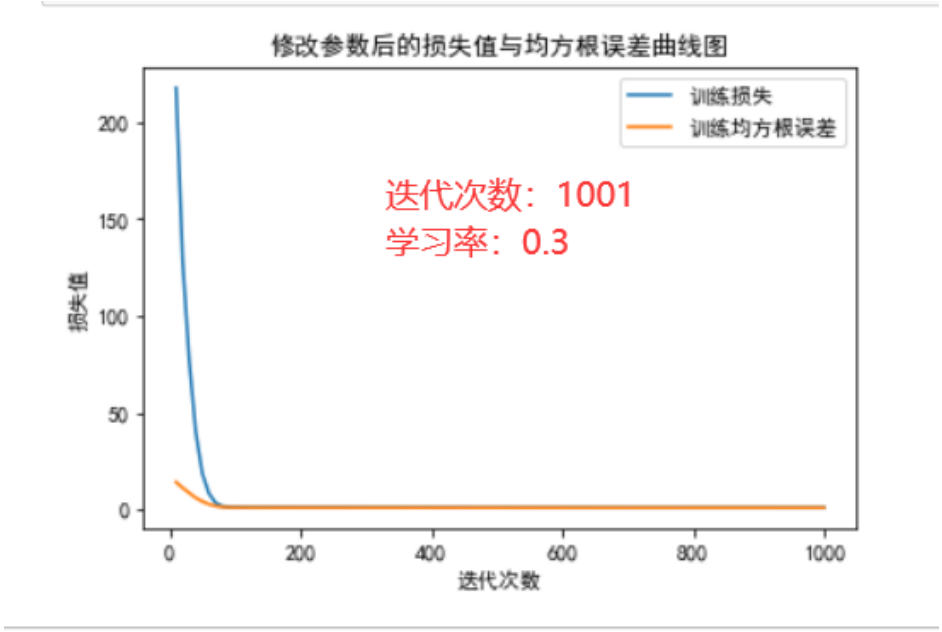


图 4-22 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数为 1001、学习率为 0.3 时的模型，在迭代开始时损失值随着迭代次数的增加迅速下降，在迭代次数为 200 次时就趋近于收敛，而均方误差在开始时波动较大，之后趋于平稳。

3、迭代次数：200，学习率：3

（1）训练并打印结果

根据第二次调整参数的结果，迭代次数在 1001 次、学习率为 0.3 时，模型损失值随迭代次数下降速度较快，在迭代次数 200 次时，已经趋近收敛。为了提高模型拟合的效率，需要再次减少迭代次数，同时为了防止结果未收敛还需要提高学习率，因此选择迭代次数为 200、学习率为 3。但由于调整参数时，迭代次数每 10 次为周期打印结果，而迭代次数为 200 时，需要每 2 次为周期打印结果。这里选择前 20 次和后 20 次的迭代结果，如图 4-23 所示。

```
Epoch [2/200], Loss: 5082.1968, RMSE: 25.2024
Epoch [4/200], Loss: 699.5397, RMSE: 50.0367
Epoch [6/200], Loss: 2352.8020, RMSE: 31.9057
Epoch [8/200], Loss: 86.6655, RMSE: 15.5461
Epoch [10/200], Loss: 978.8514, RMSE: 36.7117
Epoch [12/200], Loss: 1032.8461, RMSE: 20.6202
Epoch [14/200], Loss: 59.6871, RMSE: 12.5516
Epoch [16/200], Loss: 500.2067, RMSE: 26.3313
Epoch [18/200], Loss: 559.6713, RMSE: 15.6876
Epoch [20/200], Loss: 25.7314, RMSE: 6.8716
Epoch [22/200], Loss: 227.9897, RMSE: 18.9835
Epoch [24/200], Loss: 318.9878, RMSE: 12.5727
Epoch [26/200], Loss: 29.5147, RMSE: 5.6264
Epoch [28/200], Loss: 128.7184, RMSE: 14.2458
Epoch [30/200], Loss: 177.6501, RMSE: 9.1106
Epoch [32/200], Loss: 10.2051, RMSE: 4.0103
Epoch [34/200], Loss: 73.9701, RMSE: 10.5755
Epoch [36/200], Loss: 90.1342, RMSE: 5.9749
Epoch [38/200], Loss: 4.2309, RMSE: 4.4369
Epoch [40/200], Loss: 54.5346, RMSE: 8.1128
```

迭代次数：200
学习率：3
前20行数据

```
Epoch [160/200], Loss: 1.0306, RMSE: 1.0149
Epoch [162/200], Loss: 1.0293, RMSE: 1.0141
Epoch [164/200], Loss: 1.0276, RMSE: 1.0134
Epoch [166/200], Loss: 1.0263, RMSE: 1.0126
Epoch [168/200], Loss: 1.0247, RMSE: 1.0120
Epoch [170/200], Loss: 1.0233, RMSE: 1.0112
Epoch [172/200], Loss: 1.0218, RMSE: 1.0105
Epoch [174/200], Loss: 1.0204, RMSE: 1.0098
Epoch [176/200], Loss: 1.0190, RMSE: 1.0091
Epoch [178/200], Loss: 1.0176, RMSE: 1.0084
Epoch [180/200], Loss: 1.0162, RMSE: 1.0077
Epoch [182/200], Loss: 1.0148, RMSE: 1.0070
Epoch [184/200], Loss: 1.0135, RMSE: 1.0064
Epoch [186/200], Loss: 1.0121, RMSE: 1.0057
Epoch [188/200], Loss: 1.0107, RMSE: 1.0050
Epoch [190/200], Loss: 1.0094, RMSE: 1.0044
Epoch [192/200], Loss: 1.0081, RMSE: 1.0037
Epoch [194/200], Loss: 1.0068, RMSE: 1.0031
Epoch [196/200], Loss: 1.0055, RMSE: 1.0024
Epoch [198/200], Loss: 1.0042, RMSE: 1.0018
Epoch [200/200], Loss: 1.0029, RMSE: 1.0011
```

迭代次数：200
学习率：3
后20行数据

图 4-23 调整参数后的训练结果图（前、后各 20 行数据）

通过打印的结果可以得出，迭代次数在 200、学习率为 3 时，该模型损失值随迭代次数变化波动较大，且损失值未收敛。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：200，学习率：3）的损失值与均方误差图像如图 4-2 所示。

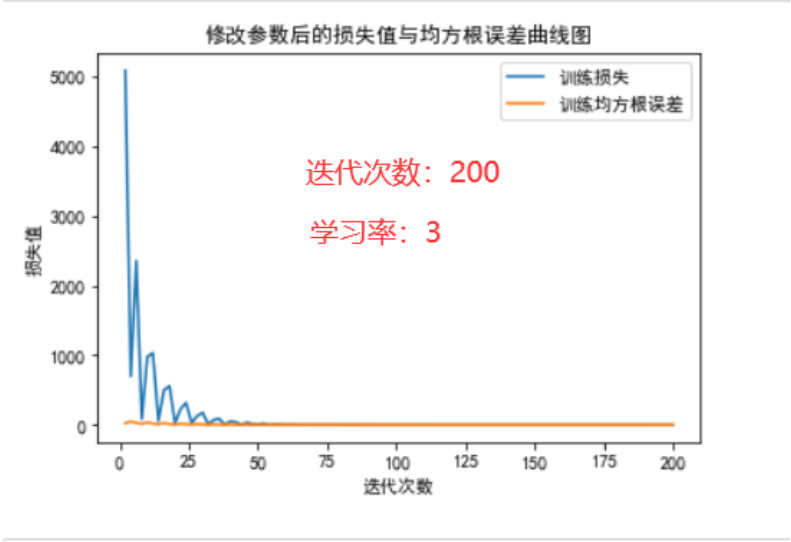


图 4-24 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数为 200、学习率为 3 时的模型，损失值随着迭代次数的变化波动很大，且损失值未收敛，因此该学习率较大，需要降低学习率，并增加迭代次数。

4、迭代次数：500，学习率：0.8

（1）训练并打印结果

根据第三次调整参数的结果，迭代次数在 201 次、学习率为 3 时，模型损失值随迭代次数变化波动很大，损失函数波动较大的原因可能是由于学习率过大导致的，因此选择迭代次数为 500、学习率为 0.8。但由于调整参数时，迭代次数每 2 次为周期打印结果，而迭代次数为 500 时，需要每 5 次为周期打印结果。这里选择前 20 次和后 20 次的迭代结果，如图 4-25 所示。

```
Epoch [5/501], Loss: 241.4730, RMSE: 14.7153
Epoch [10/501], Loss: 123.9267, RMSE: 9.7103
Epoch [15/501], Loss: 52.6473, RMSE: 5.7869
Epoch [20/501], Loss: 18.3382, RMSE: 3.0301
Epoch [25/501], Loss: 4.6796, RMSE: 1.5534
Epoch [30/501], Loss: 3.6083, RMSE: 1.9240
Epoch [35/501], Loss: 5.4594, RMSE: 2.4803
Epoch [40/501], Loss: 5.7744, RMSE: 2.4918
Epoch [45/501], Loss: 4.3155, RMSE: 2.0285
Epoch [50/501], Loss: 2.4858, RMSE: 1.4096
Epoch [55/501], Loss: 1.3362, RMSE: 1.0920
Epoch [60/501], Loss: 1.1812, RMSE: 1.1393
Epoch [65/501], Loss: 1.3896, RMSE: 1.1709
Epoch [70/501], Loss: 1.3385, RMSE: 1.1322
Epoch [75/501], Loss: 1.2068, RMSE: 1.0993
Epoch [80/501], Loss: 1.1344, RMSE: 1.0535
Epoch [85/501], Loss: 1.0775, RMSE: 1.0387
Epoch [90/501], Loss: 1.0814, RMSE: 1.0405
Epoch [95/501], Loss: 1.0752, RMSE: 1.0357
Epoch [100/501], Loss: 1.0640, RMSE: 1.0308
```

迭代次数：501

学习率：0.8

前20行数据

```
Epoch [400/501], Loss: 0.9082, RMSE: 0.9530
Epoch [405/501], Loss: 0.9079, RMSE: 0.9528
Epoch [410/501], Loss: 0.9075, RMSE: 0.9526
Epoch [415/501], Loss: 0.9072, RMSE: 0.9525
Epoch [420/501], Loss: 0.9070, RMSE: 0.9523
Epoch [425/501], Loss: 0.9067, RMSE: 0.9522
Epoch [430/501], Loss: 0.9065, RMSE: 0.9521
Epoch [435/501], Loss: 0.9062, RMSE: 0.9519
Epoch [440/501], Loss: 0.9060, RMSE: 0.9518
Epoch [445/501], Loss: 0.9058, RMSE: 0.9517
Epoch [450/501], Loss: 0.9056, RMSE: 0.9516
Epoch [455/501], Loss: 0.9054, RMSE: 0.9515
Epoch [460/501], Loss: 0.9052, RMSE: 0.9514
Epoch [465/501], Loss: 0.9051, RMSE: 0.9513
Epoch [470/501], Loss: 0.9049, RMSE: 0.9513
Epoch [475/501], Loss: 0.9048, RMSE: 0.9512
Epoch [480/501], Loss: 0.9047, RMSE: 0.9511
Epoch [485/501], Loss: 0.9045, RMSE: 0.9511
Epoch [490/501], Loss: 0.9044, RMSE: 0.9510
Epoch [495/501], Loss: 0.9043, RMSE: 0.9509
Epoch [500/501], Loss: 0.9042, RMSE: 0.9509
```

迭代次数：501

学习率：0.8

后20行数据

图 4-25 调整参数后的训练结果图（前、后各 20 行数据）

通过打印的结果可以得出，迭代次数在 501、学习率为 0.8 时，该模型损失值随迭代次数下降速度较快，损失值已收敛。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：501，学习率：0.8）的损失值与均方误差图像如图 4-26 所示。

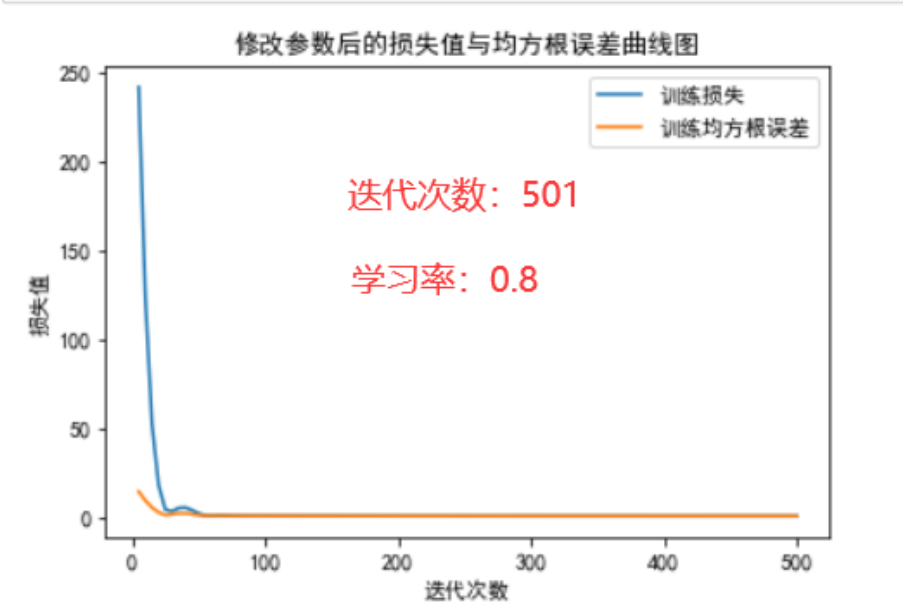


图 4-26 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数为 501、学习率为 0.8 时的模型，在迭代开始时损失值随着迭代次数的增加迅速下降，在迭代次数为 100 次时就趋近于收敛，而均方误差在开始时稍有波动，之后趋于平稳。

### 4.4.3 关于线性回归模型的结论

通过上诉模型的优化以及参数的调整，确定本次新冠预测实践的线性回归模型使用 Adam 优化器，并计算带有 L1 正则化的损失函数，确定了 L1 正则化权重为 0.001，通过参数调整以及可视化分析，确定了该模型在迭代次数为 500 次，学习率为 0.8 时的效果较好，同时效率较高。

## 4.5 生成预测结果并导出文件

1、通过优化后的线性回归模型得到预测结果

（1）通过调用 model.eval()方法，可以将模型设置为推断模式。在该模式下，其参数不会改变。这样可以确保模型在推断时表现一致而可预测。

（2）test_outputs=model(test4)用于对输入数据 test4 进行模型推断。test4 是需要预测的输入数据，通过将其传递给模型，模型会计算并返回相应的输出结果。

（3）predicted_labels=test_outputs.detach().numpy()用于从模型的输出结果中提

取预测标签。detach()方法用于将输出结果与计算图的连接断开，以便将其作为普通的 NumPy 数组使用。然后，通过 numpy()方法将结果转换为 NumPy 数组，以便下面的文件导出，得到预测结果的代码如图 4-27 所示。

```
# 设置推断模式
model.eval()
# 将处理好的测试集数据输入模型中
test_outputs = model(test4)
# 将得到的预测值转为Numpy类型
predicted_labels = test_outputs.detach().numpy()
```

图 4-27 得到预测结果代码图

2、导出文件

（1）创建一个名为 LR_sampleSubmission 的空 DataFrame，并将其保存为 CSV 文件，并保存到桌面的特定文件夹中。

（2）使用 sampleSubmission 文件中的“id”列来填充 LR_sampleSubmission 的“id”列。

（3）将模型预测的结果来填充到 LR_sampleSubmission 的“tested_positive”。

（4）将更新后的 LR_sampleSubmission 再次保存为 CSV 文件，覆盖之前创建的同名文件，导出预测结果的代码如图 4-28 所示。

```
# 创建新表
LR_sampleSubmission = pd.DataFrame()
LR_sampleSubmission .to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\LR_sampleSubmission.csv', index=False)
# 写入id列
LR_sampleSubmission["id"] =sampleSubmission["id"]
# 将数据写入特定一列
LR_sampleSubmission['tested_positive'] = predicted_labels[:len(sampleSubmission)]  # 只写入与数据帧行数相匹配的部分数组数据
# 写回 CSV 文件
LR_sampleSubmission.to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\LR_sampleSubmission.csv', index=False)
```

图 4-28 导出预测结果代码图

得到的 LR_sampleSubmission 文件如图 4-29 所示（只展示部分）。

| | A | B |
|---|---|---|
| 1 | id | tested_positive |
| 2 | 0 | 20.741913 |
| 3 | 1 | 2.8375185 |
| 4 | 2 | 2.7476969 |
| 5 | 3 | 11.67091 |
| 6 | 4 | 3.0571206 |
| 7 | 5 | 28.501812 |
| 8 | 6 | 26.166607 |
| 9 | 7 | 7.2413526 |
| 10 | 8 | 11.5804 |

图 4-29 导出的文件展示图

# 第 5 章 基于多层感知机的新冠病例预测分析

本次基于多层感知机模型的新冠病例预测，简述了多层感知机模型的原理以及公式，定义了多层感知机的模型，包括三个全连接层，并详细描述了模型的训练过程，包括参数设置、模型初始化、优化器选择等，通过优化器选择和添加 L1 正则化优化了模型，并添加了 RMSE 指标，通过可视化分析进行了参数调整，得到了拟合效果较好的多层感知机模型。

## 5.1 多层感知机模型的原理和公式

多层感知机（Multilayer Perceptron, MLP）是一种前馈神经网络模型，由多个全连接的隐藏层和一个输出层组成。它被广泛应用于各种任务，包括分类、回归和其他机器学习问题。

多层感知机模型通过层与层之间的权重参数进行信息传递和转换。每一层都由多个神经元组成，每个神经元接收上一层的输入，并通过激活函数对输入进行非线性变换后输出。这样通过多层的组合和嵌套，模型可以学习到更复杂的特征表示和决策边界。

假设我们有一个具有 L 个隐藏层的多层感知机，输入向量为$x$，第 l 层的权重矩阵为$W(l)$，偏置向量为$b(l)$，第 l 层的激活函数为 $f(l)$，输出为 $y$。

输入层到第一隐藏层的计算：

$$z(1) = W(1) * x + b(1),\ a(1) = f(1)(z(1))$$

第 l-1 隐藏层到第 l 隐藏层（2 <= l <= L）的计算：

$$z(l) = W(l) * a(l-1) + b(l),\ a(l) = f(l)(z(l))$$

最后一个隐藏层到输出层的计算：

$$z(L) = W(L) * a(L-1) + b(L),\ y = f(L)(z(L))$$

其中，*表示矩阵乘法，+表示矢量相加，$f(l)$表示第 l 层的激活函数。

在训练过程中，我们使用反向传播算法来更新权重和偏置参数，以最小化损失函数。常用的优化算法包括梯度下降法、随机梯度下降法和动量法等。

总的来说，多层感知机通过堆叠隐藏层和非线性激活函数来构建一个深层的神经网络模型，并通过反向传播算法来进行训练和优化。它可以学习到更复杂的

特征表示和决策边界，适用于各种机器学习任务[15]。

## 5.2 定义多层感知机模型的网络结构

本次新冠预测实践的多层感知机（MLP）模型包含输入层、隐藏层和输出层。

（1）__init__ 方法

使用了__init__ 方法用于初始化模型的参数。其中，input_size 是一个参数，用于指定输入特征的大小。在这个方法中，通过调用父类的 super().__init__() 来进行初始化，然后定义了三个全连接层。

（2）三个全连接层

输入层，将输入的特征大小为 input_size 的数据映射到一个包含 64 个神经元的隐藏层。

ReLU 激活函数实例化，并赋值给 self.relu，用于在前向传播过程中使用。

隐藏层，将输入大小为 64 的数据映射到一个包含 32 个神经元的隐藏层。

输出层，将输入大小为 32 的数据映射到一个包含 1 个神经元的输出层。

（3）forward 方法

forward 方法定义了模型的前向传播过程，即输入数据如何通过网络层进行计算得到输出。在这个方法中，首先对输入数据 x 应用 ReLU 激活函数，并经过第一个全连接层 self.fc1 得到隐藏层的输出。然后，将隐藏层的输出通过第二个全连接层 self.fc2 得到最终的中间结果，再经过第三个全连接层 self.fc3 得到模型的输出。

通过以上方法定义了一个简单的多层感知机模型，其中包含一个输入层、一个隐藏层和一个输出层，使用 ReLU 作为激活函数。在前向传播过程中，输入数据会经过各个全连接层并进行相应的计算，最终得到模型的输出结果。

该多层感知机模型可以接受 input_size 维度的输入，并生成一维的输出。在模型训练过程中，并使用损失函数（均方误差）和优化算法（如随机梯度下降 SGD 或 Adam 优化器）对模型进行训练，以使输出值与真实值之间的误差最小化，实现定义多层感知机模型的代码如图 5-1 所示。

```
]: # 定义多层感知机模型
class MLP(nn.Module):
    def __init__(self, input_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, 64)  # 输入层
        self.relu = nn.ReLU(),#激活函数ReLU
        self.fc2 = nn.Linear(64, 32)  # 隐藏层
        self.fc3 = nn.Linear(32, 1)  # 输出层

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

图 5-1 定义多层感知机模型代码图

## 5.3 描述多层感知机模型的训练

### 5.3.1 定义超参数

1、设置随机种子

通过设置随机种子，可以使得随机过程在每次运行时产生相同的随机数序列，从而使得结果可重现。在本次多层感知机模型中，使用了 torch.manual_seed(42)将随机种子设置为 42。这意味着在后续的代码中，使用 PyTorch 库的随机函数时，将使用相同的随机数生成器种子，从而产生相同的随机数序列。设置随机种子对于调试和复现实验结果是非常有用的。通过固定随机种子，可以确保在多次运行相同代码时获得相同的随机初始化、随机采样或随机数生成结果，从而使得实验结果更加可靠和一致，设置随机种子代码如图 5-2 所示。

```
# 设置随机种子
torch.manual_seed(42)
```

```
<torch._C.Generator at 0x256b4fbbd70>
```

图 5-2 设置随机种子代码图

2、设置超参数

X_train.shape[1] 表示训练集的特征数据 X_train 的第二个维度，即特征的数量。将其赋值给 input_dim 变量，表示输入特征的维度。

lr = 0.03，将学习率设置为 0.03。学习率是控制模型在每次参数更新时的步长大小。较大的学习率可以加快模型的收敛速度，但可能会导致不稳定的训练过程。较小的学习率可以提高训练的稳定性，但可能需要更多的训练迭代次数才能达到

较好的结果。学习率的选择需要根据具体问题和实验结果进行调整。

num_epochs = 10001，设置了迭代次数为 10001，表示模型将进行 10001 轮的训练。迭代次数的选择需要根据具体数据集和模型的复杂程度进行调整，以获得最佳的模型性能。

通过设置输入特征的维度、学习率和迭代次数，我们可以对模型的输入、训练速度和训练轮数进行相应的配置，以满足实际需求和获得最佳的模型效果，设置超参数代码如图 5-3 所示。

```
# 设置输入特征的维度
input_dim = X_train.shape[1]
# 设置学习率
lr = 0.03
# 设置迭代次数
num_epochs = 10001
```

图 5-3 设置超参数代码图

### 5.3.2 模型初始化与优化器选择

模型初始化，通过使用之前定义的 MLP 类创建了一个模型对象 model。这样做是为了构建一个多层感知机模型，其中 input_dim 是输入特征的维度。

接下来，我们定义了损失函数和优化器。criterion 使用的是均方误差损失函数（Mean Squared Error，MSE），该损失函数用于衡量模型预测值和真实值之间的差异。optimizer 使用的是随机梯度下降（Stochastic Gradient Descent，SGD）优化器，用于优化模型的参数，其中 lr 是学习率。

通过定义损失函数和优化器，为本次新冠预测实践的多层感知机模型训练提供了计算损失和更新参数的工具。损失函数用于计算模型的预测值与真实值之间的误差，而优化器则使用该误差来更新模型的参数，以最小化损失函数。

这样就完成了多层感知机模型的初始化和定义损失函数与优化器的过程，为接下来的模型训练做好了准备，模型初始化与优化器选择的代码如图 5-4 所示。

```
# 模型初始化
model = MLP(input_dim)
# 定义损失函数
criterion = nn.MSELoss()
# 使用SGD优化器
optimizer = torch.optim.SGD(model.parameters(), lr)
```

图 5-4 模型初始化与优化器选择代码图

### 5.3.3 训练模型与结果评估

在这段代码中，首先定义了存储损失的列表 train_losses、train_epochs，用来存储迭代次数和每次迭代的损失值，使用了一个 for 循环来进行模型的训练。每个训练周期（epoch）中，首先进行前向传播，计算模型的输出和损失。然后进行反向传播，通过调用 backward() 方法计算梯度并更新模型的参数。最后，通过调用 step() 方法，根据优化器的设定来更新模型的参数。

然后使用 if 条件语句设置每 100 次迭代为一个周期，在每个训练周期结束后，打印出当前周期的损失值。同时，将损失值和训练周期数存储在列表 train_losses 和 train_epochs 中，以便后续的可视化和分析。

通过这段代码，完成模型的训练过程，并在训练过程中实时监控损失值的变化。训练周期数和损失值的输出可以根据具体情况进行调整，训练模型的代码如图 5-5 所示。

```
[ ]: # 定义存储损失的列表
train_losses = []
train_epochs = []
# 进行训练
for epoch in range(num_epochs):
    # 前向传播
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    # 反向传播和优化
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    # 每迭代100次打印一次损失值
    if (epoch+1) % 100 == 0:
        # 将数据存储到列表中，便于可视化分析
        train_losses.append(loss.item())
        train_epochs.append(epoch+1)
        print(f'Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}')
```

图 5-5 训练模型代码图

得到的打印结果如图 5-6 所示。（这里选择前、后各 20 行数据）

```
Epoch 100/10001, Loss: 59.51937484741211
Epoch 200/10001, Loss: 58.22129440307617
Epoch 300/10001, Loss: 58.38158416748047
Epoch 400/10001, Loss: 58.07102966308594
Epoch 500/10001, Loss: 57.95632553100586
Epoch 600/10001, Loss: 58.00375747680664
Epoch 700/10001, Loss: 57.921630859375
Epoch 800/10001, Loss: 57.92576217651367
Epoch 900/10001, Loss: 57.88932800292969
Epoch 1000/10001, Loss: 57.894683837890625
Epoch 1100/10001, Loss: 57.88041305541992
Epoch 1200/10001, Loss: 57.873966217041016
Epoch 1300/10001, Loss: 57.86960220336914
Epoch 1400/10001, Loss: 57.863643646240234
Epoch 1500/10001, Loss: 57.86039733886719
Epoch 1600/10001, Loss: 57.86060333251953
Epoch 1700/10001, Loss: 57.857757568359375
Epoch 1800/10001, Loss: 57.85397720336914
Epoch 1900/10001, Loss: 57.851219177246094
Epoch 2000/10001, Loss: 57.848663330078125
```

迭代次数：10001

学习率：0.003

```
Epoch 8000/10001, Loss: 57.83528137207031
Epoch 8100/10001, Loss: 57.83525466918945
Epoch 8200/10001, Loss: 57.83523178100586
Epoch 8300/10001, Loss: 57.835205078125
Epoch 8400/10001, Loss: 57.83517837524414
Epoch 8500/10001, Loss: 57.83515930175781
Epoch 8600/10001, Loss: 57.83512878417969
Epoch 8700/10001, Loss: 57.83510971069336
Epoch 8800/10001, Loss: 57.83509063720703
Epoch 8900/10001, Loss: 57.83506774902344
Epoch 9000/10001, Loss: 57.83504867553711
Epoch 9100/10001, Loss: 57.83502197265625
Epoch 9200/10001, Loss: 57.83500671386719
Epoch 9300/10001, Loss: 57.83498001098633
Epoch 9400/10001, Loss: 57.834957122802734
Epoch 9500/10001, Loss: 57.83494567871094
Epoch 9600/10001, Loss: 57.834922790527344
Epoch 9700/10001, Loss: 57.83491134643555
Epoch 9800/10001, Loss: 57.83488464355469
Epoch 9900/10001, Loss: 57.834869384765625
Epoch 10000/10001, Loss: 57.83485412597656
```

迭代次数：10001

学习率：0.003

图 5-6 训练模型结果图

通过打印结果发现，函数的损失值均在 57.8 上下浮动，损失值较高，且不会随迭代次数明显下降，可能有以下几个原因。

模型过于简单，本次实现的是一个较为简单的多层感知机模型，对于复杂的数据集和非线性关系可能无法很好地拟合。如果数据集具有复杂的特征和模式，线性回归模型可能无法提供足够的灵活性来捕捉这些特征，导致损失值较高。

学习率过高或过低，学习率是优化算法中一个重要的超参数，控制每次参数更新的步长。如果学习率过高，参数更新可能会跳过最优点，导致损失值无法收敛；如果学习率过低，参数更新可能会非常缓慢，需要更多的迭代次数才能达到较好的结果。建议尝试调整学习率的大小，找到一个合适的值。

模型欠拟合，如果模型的表达能力不足，无法很好地拟合训练数据，也会导致损失值较高。可以考虑使用更复杂的模型，如多层感知机、循环神经网络或卷积神经网络等，以提高模型的表达能力。

因此需要进一步调整参数或修改模型来进行优化。

### 5.3.4 可视化分析

1、损失值函数图

绘制损失值函数图使用 Matplotlib 库绘制了损失函数随迭代次数的变化曲线。train_epochs 是存储训练轮数的列表，train_losses 是存储训练过程中损失函数值的列表。通过调用 plt.plot()函数，将训练轮数和对应的损失函数值传递给该函数，可以绘制出损失函数曲线。plt.xlabel()和 plt.ylabel()设置 x 轴和 y 轴的标签，plt.title()设置图表的标题，plt.legend()添加图例，最后，使用 plt.show()显示绘制的图表，绘制损失值函数图代码如图 5-7 所示。

```
# 绘制训练损失曲线图
plt.plot(train_epochs, train_losses, label='训练损失')
plt.xlabel('迭代次数')
plt.ylabel('损失值')
plt.title('损失值曲线图')
plt.legend()
plt.show()
```

图 5-7 损失值函数代码图

得到的图像如图 5-8 所示。

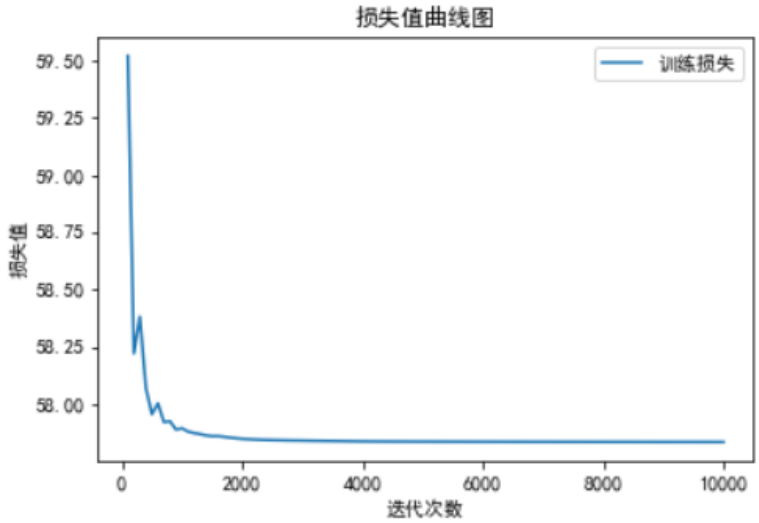


图 5-8 损失值函数图

通过图像可以看到，该线性回归模型的损失率只在 57.835 附近，损失值很高，且不会随着迭代次数而迅速下降，因此需要进一步优化该线性回归模型。

2、真实值与预测值对比图

通过数据预处理，将训练数据集（covid_train）进行了划分，将 20%的样本作为测试集，80%的样本作为训练集。使用训练集用来拟合模型，测试集来查看模型拟合效果。该步骤先将测试集数据 X_test 转换为 PyTorch 张量类型 torch.Tensor。然后，通过将测试集数据输入到模型中进行预测，使用 model(test_inputs) 得到预测结果。接着，通过 squeeze() 方法将结果的维度压缩，使其成为一维数组。最后，使用 detach().numpy() 将结果从 PyTorch 张量类型转换为 NumPy 数组，以便进行后续的处理和分析。可视化部分通过 plt.plot() 函数分别传入真实值 y_test.squeeze() 和预测值 predictions，绘制了两条曲线。并通过调用 plt.legend() 添加图例，将真实值和预测值对应的图例显示在图表上方。最后，使用 plt.show() 显示绘制的图表，以进行可视化分析，测试集模型预测和可视化的代码如图 5-9 所示。

```
[ ]: # 模型预测
test_inputs = torch.Tensor(X_test)
predictions = model(test_inputs).squeeze().detach().numpy()
```

```
[ ]: # 可视化分析
plt.plot(y_test.squeeze(), label='真实值')
plt.plot(predictions, label='预测值')
plt.title('真实值与预测值对比图')
plt.legend()
plt.show()
```

图 5-9 测试集模型预测和可视化的代码图

得到的结果如图 5-10 所示。

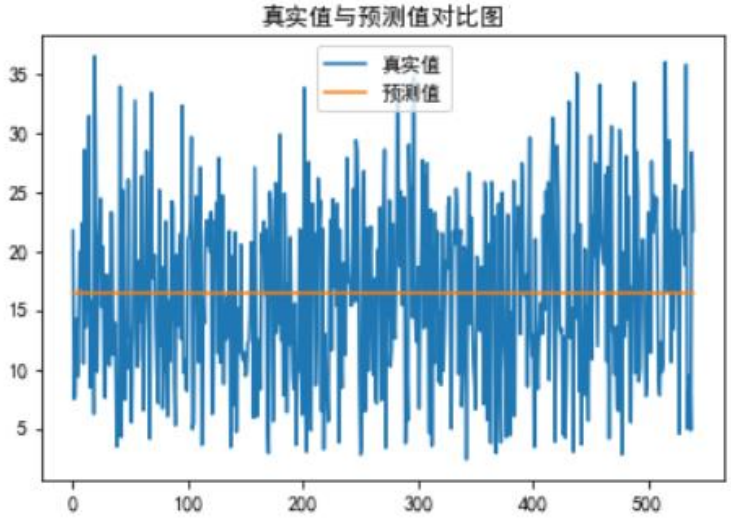


图 5-10 真实值与预测值对比图

通过图像可以看出，因为模型拟合效果较差，导致预测值基本在一条线上，与其对应的真实值存在很大的差别，因此需要调整参数和进行模型优化。

## 5.4 模型优化与调整参数设置

### 5.4.1 模型优化

1、优化器选择

同线性回归模型一致，改为使用 Adam 优化器，修改优化器代码如图 5-11 所示。

```
# 使用Adam优化器
optimizer = torch.optim.Adam(model.parameters(), lr)
```

图 5-11 使用 Adam 优化器代码图

2、添加 L1 正则化

（1）L1 正则化权重的自动选取

该步骤使用了 scikit-learn 库的 LassoCV 实现实现 L1 正则化权重的自动选取，可以使用交叉验证来估计模型在不同正则化权重下的性能，并选择表现最佳的权重。在这个步骤中，我们使用了 LassoCV 类，它是基于交叉验证的 Lasso 回归模型。通过指定不同正则化路径（alphas）和交叉验证的折数（cv），LassoCV 会在训练数据上进行交叉验证，找到表现最佳的正则化权重。最后，通过访问 LassoCV 对象的 alpha_属性，就可以获取最佳正则化权重的值。这个值就是在交叉验证过程中得到的最优权重。

（2）计算 L1 正则化项，并将其添加到总损失中

首先，为了防止原位操作，需要创建一个名为 l1_regularization 的初始值为 0 的 Tensor 变量，用于累加每个模型参数的 L1 范数。

其次，在循环中，对模型的每个参数进行迭代，并使用 torch.norm 函数计算每个参数的 L1 范数，并将其加到 l1_regularization 上。

然后，通过将 best_l1_lambda（L1 正则化的权重）乘以 l1_regularization 并加到总损失 loss 上，计算带有 L1 正则化的总损失 l1_loss。

最后，将 l1_loss 作为总损失传递给优化器进行梯度下降，以更新模型的参数。

L1 正则化权重的自动选取代码如图 5-12 所示，计算带有 L1 正则化损失的代码如图 5-13 所示。

```
# L1正则化权重的自动选取
from sklearn.linear_model import LassoCV
# 创建一个LassoCV对象，指定正则化路径和交叉验证的折数
lasso_cv = LassoCV(alphas=[0.0001,0.001, 0.01, 0.1, 1.0], cv=5,max_iter=100000)
# 在训练数据上拟合模型
lasso_cv.fit(X_train, y_train)
# 选择最佳正则化权重
best_l1_lambda = lasso_cv.alpha_
# 打印输出最佳的L1正则化权重
print("best_l1_lambda:", best_l1_lambda)
```

```
best_l1_lambda: 0.001
```

图 5-12 L1 正则化权重的自动选取代码图

```
# 计算L1正则化项
# 创建新的变量以避免原位操作
l1_regularization = torch.tensor(0., dtype=torch.float32)
for param in model.parameters():
    l1_regularization += torch.norm(param, p=1)

# 计算带有L1正则化的总损失
l1_loss = loss + best_l1_lambda * l1_regularization

# 反向传播和优化器更新
optimizer.zero_grad()
l1_loss.backward()
optimizer.step()
```

l1_loss作为总损失传递给优化器进行梯度下降

图 5-13 计算带有 L1 正则化损失代码图

3、添加 RMSE 指标

（1）通过使用 torch.no_grad()上下文管理器来关闭梯度计算，在模型上进行推断得到训练集的预测值 train_predictions。

（2）使用定义好的损失函数 criterion 计算预测值与训练集标签 y_train 之间的均方误差（MSE）。

（3）将训练集标签 y_train 转换为张量，并调整其形状为(-1, 1)，以确保与预

测值相匹配。

（4）使用 torch.sqrt()函数计算均方误差的平方根，即均方根误差（RMSE），并将其赋值给变量 rmse。可以将 RMSE 添加到一个列表中以进行记录或进一步分析，添加 RMSE 指标的代码如图 5-14 所示。

```
# 在训练集上计算MSE
with torch.no_grad():
    train_predictions = model(X_train)
    mse = criterion(train_predictions, y_train)
# 将训练集标签转换为张量
y_train = y_train.view(-1, 1)
# 计RMSE并将其添加到列表中
rmse = torch.sqrt(mse)
```

图 5-14 添加 RMSE 指标代码图

4、优化后的模型训练与结果评估（学习率：0.003，迭代次数：10001）

该步骤是训练优化后的模型并记录损失的代码。以下是对每个步骤的解释：

（1）定义存储损失的列表：

train_losses：用于存储训练过程中的损失值。

train_epochs：用于存储每个 epoch 的数量。

rmse_list：用于存储每个 epoch 的 RMSE 值。

（2）开始训练循环（共进行 num_epochs 次迭代）：

进行前向传播以获取模型的输出。

计算损失函数，这里使用了交叉熵损失函数（criterion）。

（3）计算 L1 正则化项：

创建一个初始值为 0 的张量 l1_regularization。

对模型的每个参数进行遍历，并计算该参数的 L1 范数，将其加到 l1_regularization 中。

（4）计算带有 L1 正则化的总损失：

将 L1 正则化项乘以 l1_lambda，得到 L1 正则化惩罚项。

将 L1 正则化惩罚项与原始损失相加，得到带有 L1 正则化的总损失 l1_loss。

（5）反向传播和优化器更新：

清除优化器的梯度。

对 l1_loss 进行反向传播。

使用优化器进行参数更新。

（6）在训练集上计算 MSE 和 RMSE：

使用训练好的模型对训练集进行前向传播，得到预测值 train_predictions。

计算预测值与真实标签 y_train 之间的均方误差（MSE）。

（7）将训练集标签转换为张量：

将训练集标签 y_train 进行形状变换，将其从一维向量转换为二维张量。

（8）计算 RMSE 并将其添加到列表中：

通过对 MSE 进行平方根操作来计算 RMSE。

（9）打印损失：

每经过 100 个 epoch，将当前 epoch 的损失、RMSE 等信息打印出来，并将 loss.item()、rmse.item()等值添加到相应的列表中，以便于进行可视化操作，训练优化后的模型代码如图 5-1 所示。

```
# 定义存储损失的列表
train_losses = []
train_epochs = []
rmse_list = []
for epoch in range(num_epochs):
    # 前向传播
    outputs = model(X_train)
    loss = criterion(outputs, y_train)
    # 计算L1正则化项
    # 创建新的变量以避免原位操作
    l1_regularization = torch.tensor(0., dtype=torch.float32)
    for param in model.parameters():
        l1_regularization += torch.norm(param, p=1)
    # 计算带有L1正则化的总损失
    l1_loss = loss + best_l1_lambda * l1_regularization
    # 反向传播和优化器更新
    optimizer.zero_grad()
    l1_loss.backward()
    optimizer.step()
    # 在训练集上计算MSE
    with torch.no_grad():
        train_predictions = model(X_train)
        mse = criterion(train_predictions, y_train)
    # 将训练集标签转换为张量
    y_train = y_train.view(-1, 1)
    # 计RMSE并将其添加到列表中
    rmse = torch.sqrt(mse)
    # 每迭代100次打印损失
    if (epoch+1) % 100 == 0:
        # 将数据存储到列表中，便于可视化分析
        train_losses.append(loss.item())
        train_epochs.append(epoch+1)
        rmse_list.append(rmse.item())
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}, RMSE: {rmse:.4f}')
```

定义存储损失值的列表

开始循环

计算L1正则化项

计算带有L1正则化的损

对带有L1正则化的损失进行反向传播

对MSE进行开平方得到RMSE

图 5-15 训练优化后的模型代码图

得到优化后的模型打印结果如图 5-16 所示。

```
Epoch [100/10001], Loss: 11.4940, RMSE: 3.3632
Epoch [200/10001], Loss: 3.2338, RMSE: 1.7875
Epoch [300/10001], Loss: 1.3479, RMSE: 1.1582
Epoch [400/10001], Loss: 1.0101, RMSE: 1.0042
Epoch [500/10001], Loss: 0.9178, RMSE: 0.9577
Epoch [600/10001], Loss: 0.8763, RMSE: 0.9359
Epoch [700/10001], Loss: 0.8550, RMSE: 0.9246
Epoch [800/10001], Loss: 0.8413, RMSE: 0.9172
Epoch [900/10001], Loss: 0.8316, RMSE: 0.9119
Epoch [1000/10001], Loss: 0.8237, RMSE: 0.9075
Epoch [1100/10001], Loss: 0.8171, RMSE: 0.9039
Epoch [1200/10001], Loss: 0.8112, RMSE: 0.9006
Epoch [1300/10001], Loss: 0.8059, RMSE: 0.8976
Epoch [1400/10001], Loss: 0.7991, RMSE: 0.8939
Epoch [1500/10001], Loss: 0.7930, RMSE: 0.8905
Epoch [1600/10001], Loss: 0.7876, RMSE: 0.8875
Epoch [1700/10001], Loss: 0.7825, RMSE: 0.8845
Epoch [1800/10001], Loss: 0.7784, RMSE: 0.8821
Epoch [1900/10001], Loss: 0.7734, RMSE: 0.8795
Epoch [2000/10001], Loss: 0.7698, RMSE: 0.8774
```

迭代次数：10001

学习率：0.003

前20行数据

```
Epoch [8000/10001], Loss: 0.6235, RMSE: 0.7889
Epoch [8100/10001], Loss: 0.6201, RMSE: 0.7881
Epoch [8200/10001], Loss: 0.6184, RMSE: 0.7864
Epoch [8300/10001], Loss: 0.6178, RMSE: 0.7857
Epoch [8400/10001], Loss: 0.6266, RMSE: 0.7912
Epoch [8500/10001], Loss: 0.6235, RMSE: 0.7912
Epoch [8600/10001], Loss: 0.6145, RMSE: 0.7840
Epoch [8700/10001], Loss: 0.6188, RMSE: 0.7852
Epoch [8800/10001], Loss: 0.6244, RMSE: 0.7889
Epoch [8900/10001], Loss: 0.6195, RMSE: 0.7876
Epoch [9000/10001], Loss: 0.6138, RMSE: 0.7828
Epoch [9100/10001], Loss: 0.6204, RMSE: 0.7861
Epoch [9200/10001], Loss: 0.6221, RMSE: 0.7885
Epoch [9300/10001], Loss: 0.6160, RMSE: 0.7832
Epoch [9400/10001], Loss: 0.6096, RMSE: 0.7813
Epoch [9500/10001], Loss: 0.6182, RMSE: 0.7850
Epoch [9600/10001], Loss: 0.6117, RMSE: 0.7833
Epoch [9700/10001], Loss: 0.6067, RMSE: 0.7788
Epoch [9800/10001], Loss: 0.6065, RMSE: 0.7804
Epoch [9900/10001], Loss: 0.6117, RMSE: 0.7833
Epoch [10000/10001], Loss: 0.6210, RMSE: 0.7831
```

迭代次数：10001

学习率：0.003

后20行数据

图 5-16 优化后模型训练结果图（前、后 20 行）

通过打印的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型收敛后的损失值从 57.83 附近下降到了 0.61 附近，这说明通过添加L1 正则化以及更换优化器使该线性回归模型的拟合效果有了很大的提升。但损失值在迭代 2000 次左右就已经趋于收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型。

5、可视化分析

（1）优化模型后的训练损失与均方根误差曲线图

通过可视化代码，得到优化模型后的训练损失与均方根误差曲线图，如图 5-17 所示。

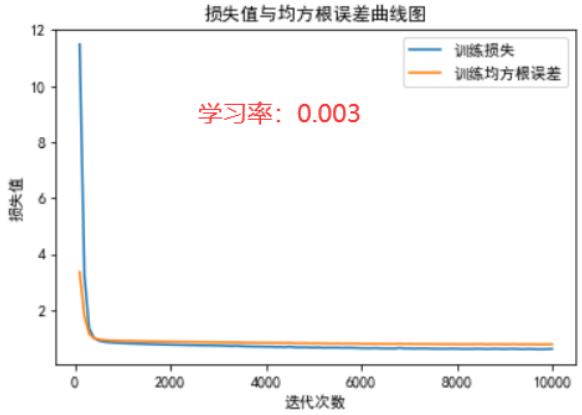


图 5-17 优化模型后的损失值与均方根误差图

通过可视化的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型收敛后的损失值从 57.83 附近下降到了 0.5 附近，这说明通过添加L1 正则化以及更换优化器使该线性回归模型的拟合效果有了很大的提升。但损失值在迭代 2000 次左右就已经趋于收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型。

（2）优化模型后的真实值与预测值对比图

通过可视化代码，得到优化模型后的真实值与预测值对比图，如图 5-18 所示。

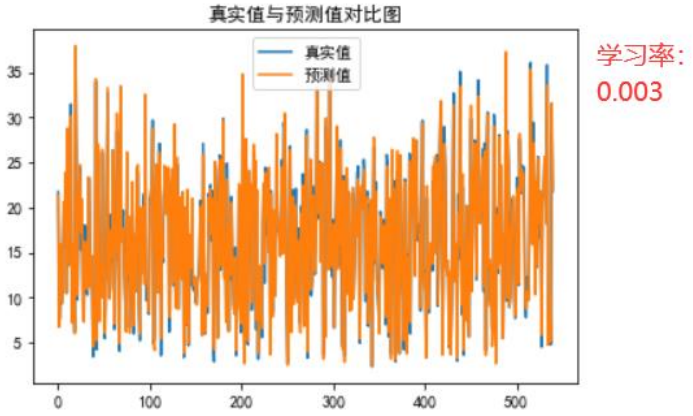


图 5-18 优化模型后的真实值与预测值对比图

通过图像可以得出，优化后的模型得到的预测值与真实值基本一致，说明该线性回归模型拟合程度较高。

### 5.4.2 调整参数设置

通过模型优化可以确定，该优化模型拟合效果较好，但通过可视化的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，损失值在迭代 2000 次左右就已经收敛，因此还需要继续调整学习率、迭代次数来优化线性回归模型，获得效果最好的参数设置。

1、迭代次数：1001，学习率：0.03

（1）训练并打印结果

根据优化模型的结果，迭代次数在 10001 次、学习率为 0.003 时，模型损失值随迭代次数下降速度较快，损失值在迭代 2000 次左右就已经趋于收敛，为了提高模型拟合效率，需要减少迭代次数，同时为了防止结果未收敛还需要提高学习率。因此选择迭代次数为 1001、学习率为 0.03，但由于优化模型时，迭代次数为 10001，所以选择每 100 次为周期打印结果，需要改变打印周期，以 10 次为一个周期打印结果。这里选择前 20 次和后 20 次的迭代结果，如图 5-19 所示。

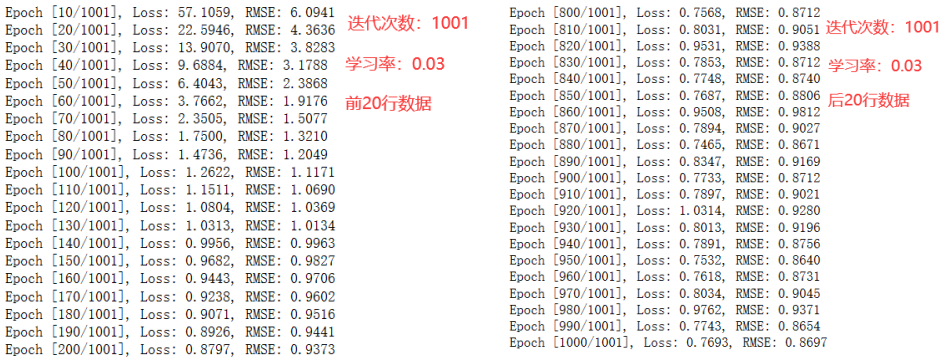


图 5-19 调整参数后的训练结果图（前、后各 20 行数据）

通过打印结果可以发现，在迭代次数为 1001 次、学习率为 0.03 时，损失值为 0.8797，该模型还未收敛，因此还需要提高学习率来加速收敛。

（2）可视化分析

这里只分析损失值与均方误差的图像，修改参数后（迭代次数：1001，学习率：0.03）的损失值与均方误差图像如图 5-20 所示。

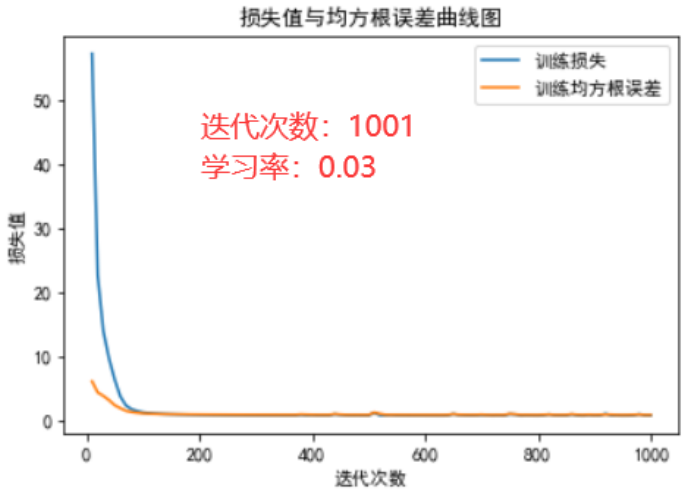


图 5-20 调整参数后的损失值与均方误差图

通过图像可以看出，迭代次数在 1001，学习率为 0.03 时，损失值随迭代次数的增大，其下降速度较为平缓，而均方误差随迭代次数的增大，其变化波动较小。

2、迭代次数：1001，学习率：0.3

（1）训练并打印结果

根据第一次调整参数的结果，迭代次数在 1001 次、学习率为 0.03 时，模型损失值随迭代次数下降速度较为平缓，但在迭代结束之后，损失值还未收敛，因此需要提高学习率，因此选择迭代次数为 1001、学习率为 0.3，这里选择前 20 次和后 20 次的迭代结果，如图 5-21 所示。

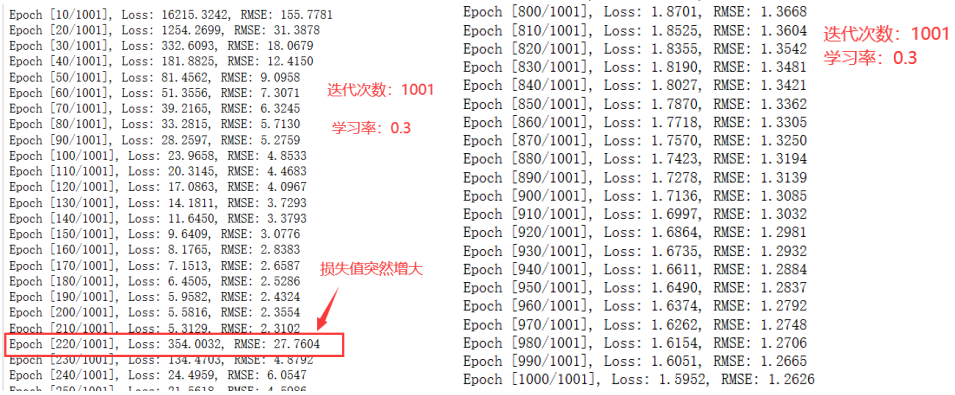


图 5-21 调整参数后的训练结果图（前、后各 20 行数据）

通过打印的结果可以得出，迭代次数在 220 次时，该模型损失值突然增大，且迭代次数为 1001 次时，损失值在 1.60 左右，相比迭代次数为 1001，学习率为 0.03 时，损失值不降反升。损失值突然上升，以及不降反升的原因可能为学习率过大，较大的学习率可能导致参数更新的步长过大，从而使模型在参数空间中跳过了最优解。当参数更新的幅度过大时，模型可能无法稳定地朝着最优解的方向前进，从而导致损失值增大。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：1001，学习率：0.3）的损失值与均方误差图像如图 5-22 所示。

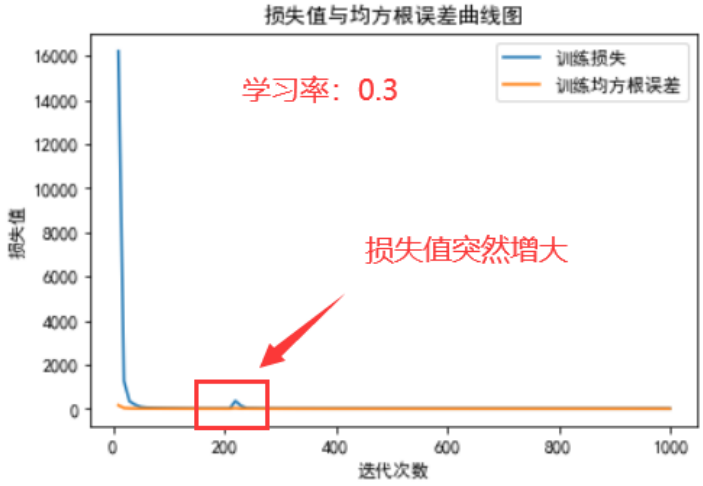


图 5-22 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数在 200 之后，损失值有突然增大的现象。因此需要降低学习率。

3、迭代次数：1001，学习率：0.1

（1）训练并打印结果

根据第二次调整参数的结果，迭代次数在 1001 次、学习率为 0.3 时，损失值不降反升。在迭代次数 200 次时，损失值突然上升。因此选择迭代次数为 1001、学习率为 0.1。这里选择部分迭代结果，如图 5-23 所示。

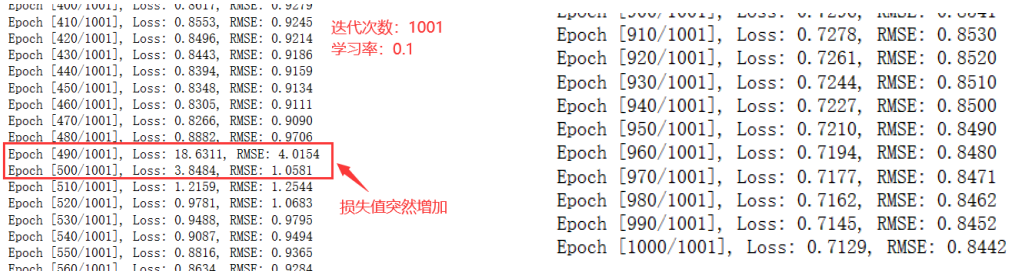


图 5-23 调整参数后的训练结果图（部分截图）

通过打印的结果可以得出，迭代次数在 1001、学习率为 0.1 时，在迭代次数在 500 次左右时，损失值仍然有突然增大的现象，但增大幅度有所减少，且最终结果并未收敛。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：1001，学习率：0.1）的损失值与均方误差图像如图 5-24 所示。

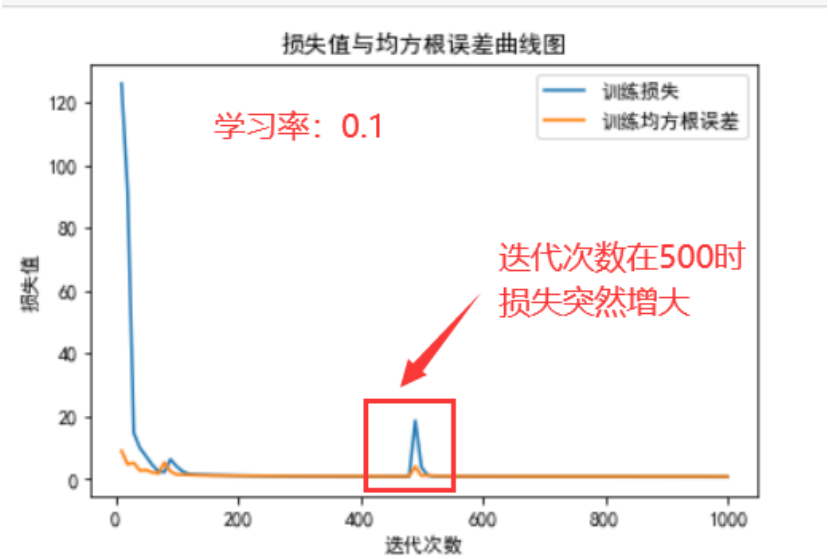


图 5-24 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数为 1001、学习率为 0.1 时的模型，在迭代次数在 500 次左右时，损失值仍然有突然增大的现象，但增大幅度相比之前较小，且结果为收敛，因此需要降低学习率，增加迭代次数。

4、迭代次数：10001，学习率：0.005

（1）训练并打印结果

经过多次训练，发现在学习率大于 0.01 时，损失值均会有一定的波动，因此最终选择迭代次数为 10001，学习率为 0.005。打印结果如图 5-25 所示。

```
Epoch [100/10001], Loss: 7.2545, RMSE: 2.6701
Epoch [200/10001], Loss: 1.5396, RMSE: 1.2349
Epoch [300/10001], Loss: 0.9884, RMSE: 0.9933
Epoch [400/10001], Loss: 0.9001, RMSE: 0.9485
Epoch [500/10001], Loss: 0.8697, RMSE: 0.9325
Epoch [600/10001], Loss: 0.8535, RMSE: 0.9238
Epoch [700/10001], Loss: 0.8412, RMSE: 0.9171
Epoch [800/10001], Loss: 0.8310, RMSE: 0.9116
Epoch [900/10001], Loss: 0.8220, RMSE: 0.9066
Epoch [1000/10001], Loss: 0.8138, RMSE: 0.9021
Epoch [1100/10001], Loss: 0.8060, RMSE: 0.8977
Epoch [1200/10001], Loss: 0.7987, RMSE: 0.8937
Epoch [1300/10001], Loss: 0.7944, RMSE: 0.8912
Epoch [1400/10001], Loss: 0.7865, RMSE: 0.8867
Epoch [1500/10001], Loss: 0.7801, RMSE: 0.8834
Epoch [1600/10001], Loss: 0.7766, RMSE: 0.8815
Epoch [1700/10001], Loss: 0.7690, RMSE: 0.8770
Epoch [1800/10001], Loss: 0.7686, RMSE: 0.8768
Epoch [1900/10001], Loss: 0.7595, RMSE: 0.8715
Epoch [2000/10001], Loss: 0.7545, RMSE: 0.8687
```

迭代次数：10001

学习率：0.005

前20行数据

```
Epoch [8000/10001], Loss: 0.6025, RMSE: 0.7735
Epoch [8100/10001], Loss: 0.6037, RMSE: 0.7792
Epoch [8200/10001], Loss: 0.6123, RMSE: 0.7809
Epoch [8300/10001], Loss: 0.5933, RMSE: 0.7718
Epoch [8400/10001], Loss: 0.5998, RMSE: 0.7735
Epoch [8500/10001], Loss: 0.6200, RMSE: 0.7890
Epoch [8600/10001], Loss: 0.5966, RMSE: 0.7709
Epoch [8700/10001], Loss: 0.6379, RMSE: 0.8027
Epoch [8800/10001], Loss: 0.5999, RMSE: 0.7724
Epoch [8900/10001], Loss: 0.6214, RMSE: 0.7836
Epoch [9000/10001], Loss: 0.6003, RMSE: 0.7770
Epoch [9100/10001], Loss: 0.5876, RMSE: 0.7665
Epoch [9200/10001], Loss: 0.5873, RMSE: 0.7664
Epoch [9300/10001], Loss: 0.5884, RMSE: 0.7668
Epoch [9400/10001], Loss: 0.6235, RMSE: 0.7914
Epoch [9500/10001], Loss: 0.6107, RMSE: 0.7752
Epoch [9600/10001], Loss: 0.5919, RMSE: 0.7680
Epoch [9700/10001], Loss: 0.6103, RMSE: 0.7808
Epoch [9800/10001], Loss: 0.6030, RMSE: 0.7729
Epoch [9900/10001], Loss: 0.5914, RMSE: 0.7710
Epoch [10000/10001], Loss: 0.6314, RMSE: 0.7959
```

图 5-25 调整参数后的训练结果图（前、后各 20 行数据）

通过打印的结果可以得出，迭代次数在 10001、学习率为 0.005 时，该模型损失值随迭代次数下降波动性较小，且损失值已趋于收敛。

（2）可视化分析

通过可视化得到损失值与均方误差的图像，修改参数后（迭代次数：10001，学习率：0.005）的损失值与均方误差图像如图 5-26 所示。

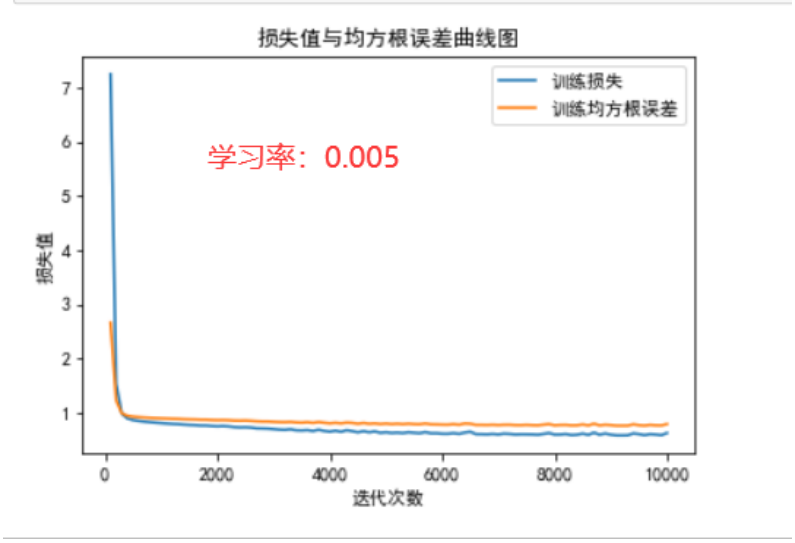


图 5-26 调整参数后的损失值与均方误差图

通过可视化的结果可以看到，迭代次数为 10001、学习率为 0.005 时的模型，在迭代开始时损失值随着迭代次数的增加迅速下降，在迭代次数为 2000 次时就趋近于收敛，而均方误差在开始时稍有波动，之后趋于平稳。

### 5.4.3 关于多层感知机模型的结论

通过上述模型的优化以及参数的调整，确定本次新冠预测实践的多层感知机模型使用 Adam 优化器，并计算带有 L1 正则化的损失函数，确定了 L1 正则化权重为 0.001，通过参数调整以及可视化分析，确定了该模型在迭代次数为 1001 次，学习率为 0.03 时的效果较好。

## 5.5 生成预测结果并导出文件

1、通过优化后的线性回归模型得到预测结果

（1）通过调用 model.eval()方法，可以将模型设置为推断模式。在该模式下，其参数不会改变。这样可以确保模型在推断时表现一致而可预测。

（2）test_outputs=model(test4)用于对输入数据 test4 进行模型推断。test4 是需要预测的输入数据，通过将其传递给模型，模型会计算并返回相应的输出结果。

（3）predicted_labels=test_outputs.detach().numpy()用于从模型的输出结果中提取预测标签。detach()方法用于将输出结果与计算图的连接断开，以便将其作为普通的 NumPy 数组使用。然后，通过 numpy()方法将结果转换为 NumPy 数组，以便下面的文件导出。得到预测结果的代码如图 5-27 所示。

```
# 设置推断模式
model.eval()
# 将处理好的测试集数据输入模型中
test_outputs = model(test4)
# 将得到的预测值转为Numpy类型
predicted_labels = test_outputs.detach().numpy()
```

图 5-27 得到预测结果代码图

2、导出文件

（1）创建一个名为 MLP_sampleSubmissionn 的空 DataFrame，并将其保存为 CSV 文件，并保存到桌面的特定文件夹中。

（2）使用 sampleSubmission 文件中的“id”列来填充 MLP_sampleSubmission 的"id"列。

（3）将模型预测的结果来填充到 MLP_sampleSubmission 的“tested_positive”列。

（4）将更新后的 MLP_sampleSubmission 再次保存为 CSV 文件，覆盖之前创建的同名文件，导出预测结果的代码如图 5-28 所示。

```
#导入 sampleSubmission 表
sampleSubmission = pd.read_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\sampleSubmission.csv', sep = ',')
# 创建新表
MLP_sampleSubmission = pd.DataFrame()
MLP_sampleSubmission .to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\MLP_sampleSubmission.csv', index=False)
# 写入id列
MLP_sampleSubmission["id"] =sampleSubmission["id"]
```

图 5-28 导出预测结果代码图

得到的 MLP_sampleSubmission 文件如图 5-29 所示（只展示部分）。

| | A | B | C |
|---|---|---|---|
| 1 | id | tested_positive | |
| 2 | 0 | 21.50941 | |
| 3 | 1 | 2.549297 | |
| 4 | 2 | 3.129495 | |
| 5 | 3 | 11.42882 | |
| 6 | 4 | 0.518095 | |
| 7 | 5 | 26.90776 | |
| 8 | 6 | 25.11659 | |
| 9 | 7 | 7.952799 | |
| 10 | 8 | 10.95943 | |
| 11 | 9 | 11.53081 | |
| 12 | 10 | 18.2854 | |
| 13 | 11 | 21.19814 | |

图 5-29 导出的文件展示图

# 第 6 章 基于循环神经网络的新冠病例预测分析

本次基于循环神经网络模型的新冠病例预测，简述了循环神经网络模型的原理以及公式，定义了一个简单的循环神经网络模型包括一个 RNN 层、一个全连接层，并详细描述了模型的训练过程，包括参数设置、模型初始化、优化器选择等，通过优化器选择和特征值选取优化了模型，得到了拟合效果较好的循环神经网络模型。

## 6.1 循环神经模型的原理和公式

循环神经网络（Recurrent Neural Network，RNN）是一种具有记忆性的神经网络，适用于处理序列数据，例如时间序列或自然语言文本。RNN 的原理是通过将网络的隐藏状态在时间上进行传递和更新，以捕捉输入序列中的时序信息。它使用相同的权重参数对每个时间步的输入进行处理，并将前一个时间步的隐藏状态作为当前时间步的输入。

RNN 的公式可以分为两个主要部分：前向传播和反向传播。

1、前向传播

在每个时间步，RNN 接收输入向量 $x(t)$ 和前一个时间步的隐藏状态 $h(t-1)$，计算当前时间步的隐藏状态 $h(t)$ 和输出 $y(t)$。具体公式如下：

$$h(t) = activation(W\_hh * h(t-1) + W\_xh * x(t) + b\_h)$$

$$y(t) = W\_hy * h(t) + b\_y$$

其中，$h(t)$ 表示当前时间步的隐藏状态，$activation$ 为激活函数，$W\_hh$ 是隐藏状态的权重参数，$W\_xh$ 是输入向量的权重参数，$b\_h$ 是偏置项参数，$W\_hy$ 是隐藏状态到输出的权重参数，$b\_y$ 是输出的偏置项参数。

2、反向传播

在反向传播过程中，根据损失函数计算梯度并更新权重参数。通过时间展开（Backpropagation Through Time, BPTT）算法，梯度可以从当前时间步一直传播到第一个时间步。

RNN 的主要特点是具有记忆性，能够捕捉输入序列中的长期依赖关系。然而，标准的 RNN 存在梯度消失或梯度爆炸的问题，导致难以处理长序列。为了克服这个问题，出现了一些改进的 RNN 变体，如长短期记忆网络（Long Short-Term Memory, LSTM）和门控循环单元（Gated Recurrent Unit, GRU）。这些改进的模型通过引入门控机制和记忆单元，有效地解决了梯度消失和梯度爆炸的问题，使得 RNN 在处理长序列时更加稳定和有效[16]。

## 6.2 循环神经网络模型的网络结构

本次新冠预测实践的循环神经网络（RNN）模型，该模型包含一个 RNN 层和一个全连接层。

1、__init__方法

__init__方法用于初始化模型的参数。其中，input_size 是输入特征的大小，hidden_size 是隐藏状态的大小。在这个方法中，通过调用父类的 super().__init__() 进行初始化，并定义了两个层：

（1）RNN 层

将输入的特征大小为 input_size 的数据按照时间步进行处理，每个时间步输出大小为 64 的隐藏状态，并且 batch_first=True 表示输入数据的维度顺序为 (batch_size, sequence_length, input_size)。

（2）全连接层，

将 RNN 层最后一个时间步的隐藏状态映射到一个标量输出。

2、forward 方法

定义了模型的前向传播过程，即输入数据如何通过网络层进行计算得到输出。在这个方法中，输入数据 x 经过 RNN 层得到所有时间步的输出结果 out，然后取最后一个时间步的输出 out[:,-1,:]，并经过全连接层 self.fc 得到模型的输出。

本次实践项目定义了一个简单的循环神经网络模型，输入数据经过 RNN 层进行时间步处理，然后通过全连接层得到输出结果。这个模型适用于序列数据的建模任务，如时间序列预测、自然语言处理等。该循环神经网络模型可以接受 input_size 维度的输入，并生成一维的输出。在模型训练过程中，并使用损失函数（均方误差）和优化算法（如随机梯度下降 SGD 或 Adam 优化器）对模型进行训练，以使输出值与真实值之间的误差最小化。

实现定义循环神经网络模型的代码如图 6-1 所示。

```python
# 构建循环神经网络
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNN, self).__init__()
        # 定义RNN层，输入维度为input_size，隐藏状态维度为64
        self.rnn = nn.RNN(input_size, 64, batch_first=True)
        # 定义RNN层，输入维度为input_size，隐藏状态维度为64
        self.fc = nn.Linear(64, 1)

    def forward(self, x):
        # 将输入数据x经过RNN层得到所有时间步的输出结果out
        out, _ = self.rnn(x)
        # 取最后一个时间步的输出，并通过全连接层得到模型的输出
        out = self.fc(out[:, -1, :])
        return out
```

图 6-1 定义循环神经网络模型代码图

## 6.3 描述循环神经网络模型的训练

### 6.3.1 定义超参数

1、设置超参数

X_train.shape[2] 表示训练集的特征数据 X_train 的第三个维度，即特征的数量。将其赋值给 input_dim 变量，表示输入特征的维度。

lr=0.001，将学习率设置为 0.001。学习率是控制模型在每次参数更新时的步长大小。较大的学习率可以加快模型的收敛速度，但可能会导致不稳定的训练过程。较小的学习率可以提高训练的稳定性，但可能需要更多的训练迭代次数才能达到较好的结果。学习率的选择需要根据具体问题和实验结果进行调整。

num_epochs=10001，设置了迭代次数为 10001，表示模型将进行 10001 轮的训练。迭代次数的选择需要根据具体数据集和模型的复杂程度进行调整，以获得最佳的模型性能。

通过设置输入特征的维度、学习率和迭代次数，我们可以对模型的输入、训练速度和训练轮数进行相应的配置，以满足实际需求和获得最佳的模型效果。

设置超参数代码如图 6-2 所示。

```python
# 设置输入特征的维度
input_size = X_train.shape[2]
# 设置学习率
lr = 0.001
# 设置迭代次数
num_epochs = 10001
```

图 6-2 设置超参数代码图

### 6.3.2 模型初始化与优化器选择

模型初始化，通过使用之前定义的 RNN 创建了一个模型对象 model。这样做是为了构建一个多层感知机模型，其中 input_size 是输入特征的维度。

接下来，我们定义了损失函数和优化器。criterion 使用的是均方误差损失函数，该损失函数用于衡量模型预测值和真实值之间的差异。optimizer 使用的是随机梯度下降优化器，用于优化模型的参数，其中 lr 是学习率。

通过定义损失函数和优化器，为本次新冠预测实践的循环神经网络模型训练提供了计算损失和更新参数的工具。损失函数用于计算模型的预测值与真实值之间的误差，而优化器则使用该误差来更新模型的参数，以最小化损失函数。

这样就完成了循环神经网络模型的初始化和定义损失函数与优化器的过程，为接下来的模型训练做好了准备。模型初始化与优化器选择的代码如图 6-3 所示。

```
# 初始化模型、损失函数和优化器
model = RNN(input_size)
# 定义损失函数
criterion = nn.MSELoss()
# 设置优化器
optimizer = torch.optim.Adam(model.parameters(), lr)
```

图 6-3 模型初始化与优化器选择代码图

### 6.3.3 训练模型与结果评估

1、实现模型训练代码

该步骤实现了循环神经网络模型训练的循环，并定义了存储损失的列表 train_losses 和训练轮数的列表 train_epochs。

在每个训练轮次中：

（1）将输入数据 X_train 转换为 torch.Tensor 类型，并将标签数据 y_train 也转换为 torch.Tensor 类型。

（2）清空优化器的梯度信息，以准备进行反向传播和参数更新。

（3）将输入数据 inputs 输入到模型中，得到模型的输出结果 outputs。

（4）计算模型的损失值，使用 criterion 定义的损失函数，并将输出和标签都压缩为一维张量进行比较。

（5）进行反向传播，计算参数的梯度。

（6）使用优化器 optimizer 更新模型的参数。

（7）每训练 100 个轮次，将当前的损失值 loss.item() 添加到 train_losses 列表中，并将当前的训练轮次 epoch+1 添加到 train_epochs 列表中。同时打印当前轮次的损失值。

通过循环迭代训练轮次，将训练过程中的损失值和轮次记录下来，可以用于后续的可视化和分析。

训练模型的代码如图 6-4 所示。

```
# 定义存储损失的列表
train_losses = []
train_epochs = []

# 模型训练
for epoch in range(num_epochs):
    inputs = torch.Tensor(X_train)  # 将输入数据转换为torch.Tensor类型
    labels = torch.Tensor(y_train)  # 将标签数据转换为torch.Tensor类型

    optimizer.zero_grad()  # 清空优化器的梯度信息，准备进行反向传播和参数更新

    outputs = model(inputs)  # 输入数据经过模型得到输出结果
    loss = criterion(outputs.squeeze(), labels.squeeze())  # 计算损失值

    loss.backward()  # 反向传播，计算参数的梯度
    optimizer.step()  # 使用优化器更新模型的参数

    if (epoch+1) % 100 == 0:
        train_losses.append(loss.item())  # 将当前的损失值添加到train_losses列表中
        train_epochs.append(epoch+1)  # 将当前的训练轮次添加到train_epochs列表中
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')  # 打印当前轮次的损失值
```

图 6-4 训练模型代码图

2、结果分析

得到的打印结果如图 6-5 所示。（这里选择前、后各 20 行数据）

```
Epoch [100/10001], Loss: 8.1838
Epoch [200/10001], Loss: 4.0856
Epoch [300/10001], Loss: 2.6727
Epoch [400/10001], Loss: 2.1687
Epoch [500/10001], Loss: 1.9785
Epoch [600/10001], Loss: 1.8803
Epoch [700/10001], Loss: 1.8112
Epoch [800/10001], Loss: 1.7544
Epoch [900/10001], Loss: 1.7049
Epoch [1000/10001], Loss: 1.6607
Epoch [1100/10001], Loss: 1.6207
Epoch [1200/10001], Loss: 1.5842
Epoch [1300/10001], Loss: 1.5506
Epoch [1400/10001], Loss: 1.5196
Epoch [1500/10001], Loss: 1.4907
Epoch [1600/10001], Loss: 1.4639
Epoch [1700/10001], Loss: 1.4388
Epoch [1800/10001], Loss: 1.4153
Epoch [1900/10001], Loss: 1.3932
Epoch [2000/10001], Loss: 1.3724
```

迭代次数：10001
学习率：0.001
优化器：SGD

```
Epoch [8000/10001], Loss: 0.9667
Epoch [8100/10001], Loss: 0.9649
Epoch [8200/10001], Loss: 0.9632
Epoch [8300/10001], Loss: 0.9615
Epoch [8400/10001], Loss: 0.9598
Epoch [8500/10001], Loss: 0.9582
Epoch [8600/10001], Loss: 0.9566
Epoch [8700/10001], Loss: 0.9551
Epoch [8800/10001], Loss: 0.9536
Epoch [8900/10001], Loss: 0.9521
Epoch [9000/10001], Loss: 0.9507
Epoch [9100/10001], Loss: 0.9493
Epoch [9200/10001], Loss: 0.9479
Epoch [9300/10001], Loss: 0.9465
Epoch [9400/10001], Loss: 0.9452
Epoch [9500/10001], Loss: 0.9440
Epoch [9600/10001], Loss: 0.9427
Epoch [9700/10001], Loss: 0.9415
Epoch [9800/10001], Loss: 0.9403
Epoch [9900/10001], Loss: 0.9391
Epoch [10000/10001], Loss: 0.9379
```

迭代次数：10001
学习率：0.001
优化器：SGD

图 6-5 训练模型结果图（前后各 20 行）

通过打印结果发现，损失值随着迭代次数平缓下降，且损失值也较小，甚至可以与优化后的线性回归模型、多层感知机模型相比。

## 6.3.4 可视化分析

1、损失值函数图

绘制损失值函数图使用 Matplotlib 库绘制了损失函数随迭代次数的变化曲线。train_epochs 是存储训练轮数的列表，train_losses 是存储训练过程中损失函数值的列表。通过调用 plt.plot() 函数，将训练轮数和对应的损失函数值传递给该函数，可以绘制出损失函数曲线。plt.xlabel() 和 plt.ylabel() 设置 x 轴和 y 轴的标

签，plt.title() 设置图表的标题，plt.legend() 添加图例，最后，使用 plt.show() 显示绘制的图表，绘制损失值函数图代码如图 6-6 所示。

```
# 绘制损失函数曲线图
plt.plot(train_epochs, train_losses, label='训练损失')
plt.xlabel('迭代次数')
plt.ylabel('损失值')
plt.title('训练损失曲线')
plt.legend()
plt.show()
```

图 6-6 损失值函数代码图

得到的图像如图 6-7 所示。

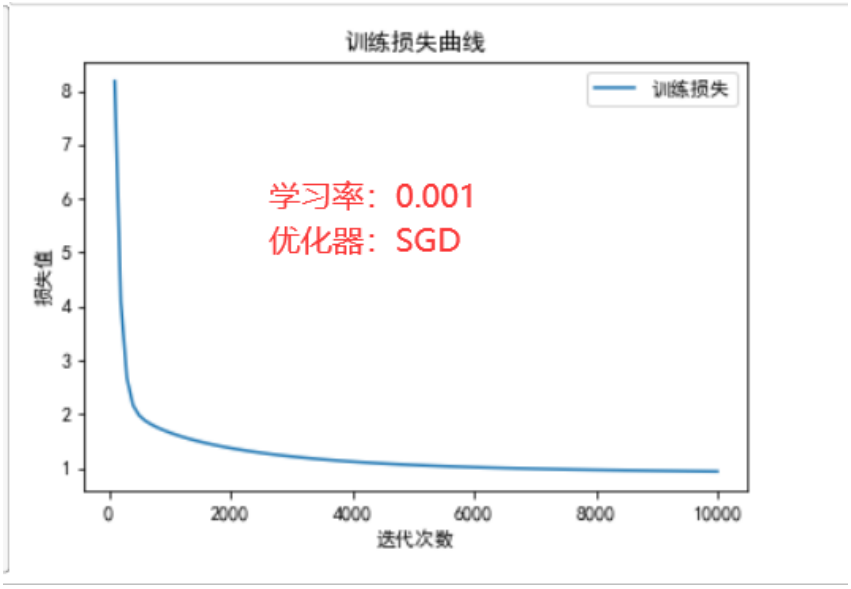


图 6-7 损失值函数图

通过图像可以看到，该线性回归模型的损失率随迭代次数平稳下降，损失值在 0.9 附近，较小。该模型效果较好。

2、真实值与预测值对比图

通过数据预处理，将训练数据集（covid_train）进行了划分，将 20%的样本作为测试集，80%的样本作为训练集。使用训练集用来拟合模型，测试集来查看模型拟合效果。该步骤先将测试集数据 X_test 转换为 PyTorch 张量类型 torch.Tensor。然后，通过将测试集数据输入到模型中进行预测，使用 model(test_inputs) 得到预测结果。接着，通过 squeeze() 方法将结果的维度压缩，使其成为一维数组。最后，使用 detach().numpy() 将结果从 PyTorch 张量类型转换为 NumPy 数组，以便进行后续的处理和分析。可视化部分通过 plt.plot() 函数分别传入真实值 y_test.squeeze() 和预测值 predictions，绘制了两条曲线。并通过调用 plt.legend() 添加图例，将真实值和预测值对应的图例显示在图表上方。最后，使用 plt.show() 显示绘制的图表，以进行可视化分析，测试集模型预测和可视化的代码如图 6-8

所示。

```
[ ]: # 模型预测
test_inputs = torch.Tensor(X_test)
predictions = model(test_inputs).squeeze().detach().numpy()
```

```
[ ]: # 可视化分析
plt.plot(y_test.squeeze(), label='真实值')
plt.plot(predictions, label='预测值')
plt.title('真实值与预测值对比图')
plt.legend()
plt.show()
```

图 6-8 测试集模型预测和可视化的代码图

得到的结果如图 6-9 所示。

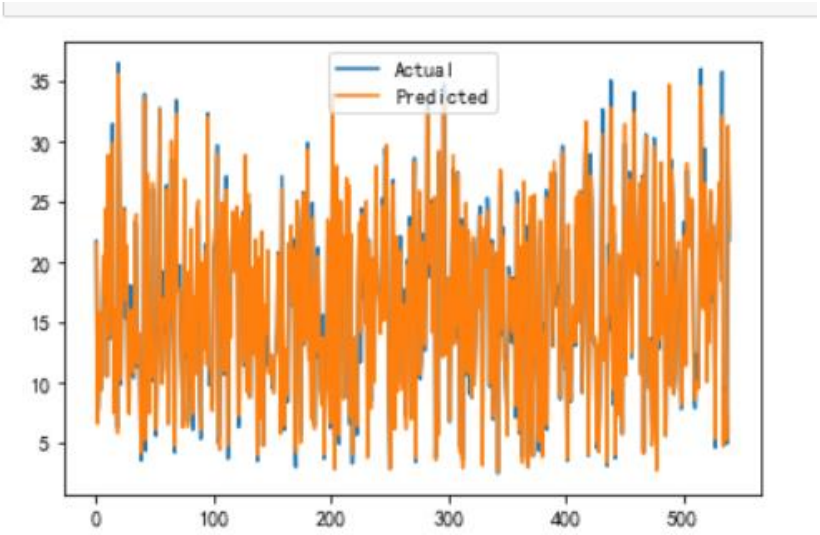


图 6-9 真实值与预测值对比图

通过图像可以看出，因为模型拟合效果较好，预测值基本与真实值一致。

## 6.4 模型优化与调整参数设置

### 6.4.1 优化器的选择

因上述模型拟合效果较好，因此只需更换优化器为 Adam 即可。

1、优化器选择

同线性回归模型一致，改为使用 Adam 优化器。修改优化器代码如图 6-10 所示。

```
# 使用Adam优化器
optimizer = torch.optim.Adam(model.parameters(), lr)
```

图 6-10 使用 Adam 优化器代码图

2、优化后的模型训练与结果评估（学习率：0.001，迭代次数：10001）

该步骤是训练优化后的模型并记录损失的代码。因为只修改了优化器，对训练过程并未修改，因此可以直接进行模型训练，得到优化后的模型打印结果如图 6-11 所示。


图 6-11 优化后模型训练结果图（前、后 20 行）

通过打印的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型迭代完成后的损失值从 0.9 附近下降到了 0.50 附近，这说明通过修改优化器使该循环神经网络模型的拟合效果有了很大的提升。

3、可视化分析

优化模型后的训练损失曲线图，通过可视化代码，得到优化模型后的训练损失曲线图，如图 6-12 所示。

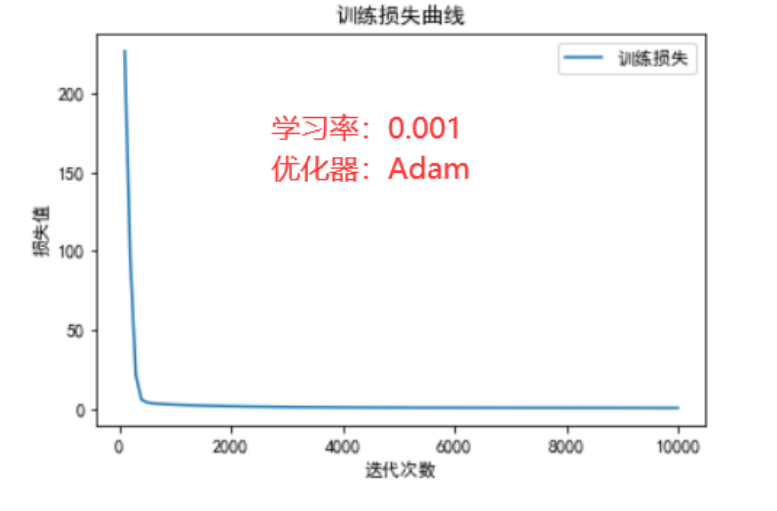


图 6-12 优化模型后的损失值曲线图

通过可视化的结果可以看到，优化后的模型损失值随迭代次数下降速度相比于未优化的模型更快一些。

## 6.4.2 调整输入的特征值数量

在本次新冠预测实践开始时，为了排除无关的特征值对模型预测的影响，使线性回归模型与多层感知机模型的拟合效果更好，我们通过使用 Sklearn 库中的 SelectKBest：特征选择库、f_regression：评分函数，以及综合考虑相关系数矩阵热力图，我们从 93 个特征值中选取了 29 个特征值进行模型拟合，而循环神经网络在自然语言处理、语音识别、时间序列预测、图像描述生成、推荐系统等领域都有广泛应用。而本次新冠预测实践模型对于循环神经网络来说较为简单，因此可以直接使用这 93 个特征值来拟合模型。查看特征值数的结果如图 6-13 所示。

```
(torch.Size([2160, 93]),
 torch.Size([540, 93]),
 torch.Size([2160]),
 torch.Size([540]))
```

图 6-13 查看特征值数量图

输入 93 个特征值后的训练结果如图 6-14 所示。

```
Epoch [100/10001], Loss: 199.5257
Epoch [200/10001], Loss: 84.4722
Epoch [300/10001], Loss: 36.1727
Epoch [400/10001], Loss: 19.3918
Epoch [500/10001], Loss: 12.4362
Epoch [600/10001], Loss: 9.1403
Epoch [700/10001], Loss: 6.6309
Epoch [800/10001], Loss: 4.7892
Epoch [900/10001], Loss: 3.7103
Epoch [1000/10001], Loss: 2.9437
Epoch [1100/10001], Loss: 2.3775
Epoch [1200/10001], Loss: 1.9472
Epoch [1300/10001], Loss: 1.6411
Epoch [1400/10001], Loss: 1.4217
Epoch [1500/10001], Loss: 1.2501
Epoch [1600/10001], Loss: 1.0862
Epoch [1700/10001], Loss: 0.9686
Epoch [1800/10001], Loss: 0.8685
Epoch [1900/10001], Loss: 0.7814
Epoch [2000/10001], Loss: 0.7212
```

学习率：0.001
优化器：Adam
输入的特征值数量：93

```
Epoch [8000/10001], Loss: 0.0244
Epoch [8100/10001], Loss: 0.0232
Epoch [8200/10001], Loss: 0.0220
Epoch [8300/10001], Loss: 0.0209
Epoch [8400/10001], Loss: 0.0199
Epoch [8500/10001], Loss: 0.0188
Epoch [8600/10001], Loss: 0.0179
Epoch [8700/10001], Loss: 0.0170
Epoch [8800/10001], Loss: 0.0163
Epoch [8900/10001], Loss: 0.0155
Epoch [9000/10001], Loss: 0.0148
Epoch [9100/10001], Loss: 0.0142
Epoch [9200/10001], Loss: 0.0136
Epoch [9300/10001], Loss: 0.0130
Epoch [9400/10001], Loss: 0.0125
Epoch [9500/10001], Loss: 0.0143
Epoch [9600/10001], Loss: 0.0115
Epoch [9700/10001], Loss: 0.0111
Epoch [9800/10001], Loss: 0.0107
Epoch [9900/10001], Loss: 0.0102
Epoch [10000/10001], Loss: 0.0099
```

学习率：0.001
优化器：Adam
输入的特征值数量：93

图 6-14 输入 93 个特征值的训练结果图

通过打印的训练结果可以看到，在迭代次数为 10000 时，损失值已经达到 0.0099，说明模型已经出现过拟合现象。

## 6.4.3 关于循环神经网络模型的结论

通过上诉模型的优化以及输入特征值数量的调整，确定本次新冠预测实践的循环神经网络模型使用 Adam 优化器，因为该模型拟合效果很好，因此这里再做参数调整并无实际意义，所以确定了该模型在迭代次数为 10001 次，学习率为 0.001，输入的特征值数量为 93 个时的效果较好。

## 6.5 生成预测结果并导出文件

1、通过优化后的线性回归模型得到预测结果

使用训练好的模型对测试数据进行预测，并将预测结果转换为 NumPy 数组。

具体来说，代码中的各个步骤如下：

（1）model(test4)：将测试数据 test4 输入到模型中，获取模型的预测输出。

（2）.squeeze()：将预测输出的维度为 1 的维度进行压缩，使得输出结果变为一维数组。

（3）.detach()：从计算图中分离出预测结果，以便后续的操作不会影响梯度计算。

（4）.numpy()：将预测结果转换为 NumPy 数组。

最终，变量 predictions 存储了模型对训练集数据的预测结果，以 NumPy 数组的形式保存。以便下面的文件导出。得到预测结果的代码如图 6-15 所示。

```
# 预测结果
predictions = model(test4).squeeze().detach().numpy()
```

图 6-15 得到预测结果代码图

2、导出文件

（1）创建一个名为 RNN_sampleSubmissionn 的空 DataFrame，并将其保存为 CSV 文件，并保存到桌面的特定文件夹中。

（2）使用 sampleSubmission 文件中的“id”列来填充 RNN_sampleSubmission 的“id”列。

（3）将模型的预测结果数组 predictions 的前 len(sampleSubmission)个值写入到新的样本提交表 RNN_sampleSubmission 的“tested_positive”列中。这里使用了切片操作确保只写入与原始样本提交表相匹配的部分数据。

（4）将更新后的 RNN_sampleSubmission 再次保存为 CSV 文件，覆盖之前创建的同名文件，导出预测结果的代码如图 6-16 所示。

```
#导入 sampleSubmission 表
sampleSubmission = pd.read_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\sampleSubmission.csv', sep = ',')
# 创建新表
RNN_sampleSubmission = pd.DataFrame()
RNN_sampleSubmission .to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\RNN_sampleSubmission.csv', index=False)
# 写入id列
RNN_sampleSubmission["id"] =sampleSubmission["id"]
# 将数据写入特定一列
RNN_sampleSubmission['tested_positive'] = predictions[:len(sampleSubmission)]  # 只写入与数据帧行数相匹配的部分数组数据
# 写回 CSV 文件
RNN_sampleSubmission.to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\RNN_sampleSubmission.csv', index=False)
```

图 6-16 导出预测结果代码图

# 第 7 章 基于卷积神经网络的新冠病例预测分析

本次基于卷积神经网络模型的新冠病例预测，简述了卷积神经网络模型的原理以及公式，定义了卷积神经网络模型，包括 2 个卷积层，每个卷积层还包含一个最大化池，以及一个展平层，一个全连接层，并详细描述了模型的训练过程，包括参数设置、模型初始化、优化器选择等，通过优化器选择、特征值选取和添加全连接层优化了模型，得到了拟合效果较好的卷积神经网络模型。

## 7.1 卷积神经网络模型的原理和公式

卷积神经网络（Convolutional Neural Network，CNN）是一种广泛用于处理图像和其他二维数据的深度学习模型。它通过使用卷积层、池化层和全连接层等组件来提取和学习图像中的特征。

下面是卷积神经网络的原理和公式：

1、卷积操作

卷积操作是卷积神经网络的核心操作之一，用于从输入数据中提取局部特征。卷积操作可以表示为以下公式：

$$C[i,j]=\sum_{m}\sum_{n}I[i+m,j+n]\cdot K[m,n]$$

其中，$C$ 是卷积结果，$I$ 是输入数据，$K$ 是卷积核（也称为滤波器）。卷积核与输入数据进行逐元素相乘后再求和，$C$ 得到卷积结果 的一个元素。卷积操作的本质是滑动窗口在输入数据上的移动，并在每个位置上与卷积核进行逐元素相乘求和。

2、激活函数

激活函数用于引入非线性变换，增加模型的表达能力。常用的激活函数有 ReLU、Sigmoid、Tanh 等。以 ReLU 函数为例，其公式如下：

$$f(x)=max(0,x)$$

激活函数将输入值 $x$ 转换为输出值 $f(x)$，当输入值小于零时，输出值为零，大于等于零时，输出值等于输入值。

3、池化操作

池化操作用于减小特征图的尺寸并保留主要特征，常用的池化操作有最大池化和平均池化，最大池化操作的公式：

$$M[i,j]=\max_{m,n}(I[i\cdot s+m,j\cdot s+n])$$

其中，$M$ 是池化结果，$I$ 是输入特征图，$s$ 是池化窗口的大小，$max$ 表示取窗口内的最大值。

4、全连接层

全连接层是卷积神经网络中的一种常见层，用于将之前的卷积和池化层提取的特征映射转换为最终的分类或回归结果。全连接层的公式如下：

$$O=WX+b$$

其中，$O$ 是输出结果，$W$ 是权重矩阵，$X$ 是输入向量，$b$ 是偏置向量。全连接层通过将输入向量 $X$ 与权重矩阵 $W$ 相乘，并加上偏置向量 $b$，得到输出结果 $O$。

综上所述，卷积神经网络通过多次堆叠卷积层、激活函数、池化层和全连接层等组件，实现了对图像特征的提取和高级表达能力的学习。这些操作和公式构成了卷积神经网络的基本原理[17]。

## 7.2 卷积神经网络模型的网络结构

本次新冠预测实践的卷积神经网络模型 CNNModel，包括两个卷积层、激活函数、最大池化层和一个全连接层。

具体来说，代码中的各个部分如下：

1、class CNNModel(nn.Module):

定义了一个继承自 nn.Module 的类 CNNModel，表示我们要构建一个卷积神经网络模型。

2、def __init__(self, input_channels, output_size):

定义了模型的初始化方法，在创建模型对象时会被调用。input_channels 表示输入数据的通道数，output_size 表示输出的维度或类别数量。

3、在初始化方法中定义了模型的各个组件：

（1）self.conv1 = nn.Conv2d(input_channels, 16, kernel_size=(1, 3))

第一个卷积层，输入通道数为 input_channels，输出通道数为 16，卷积核大小为 (1, 3)。

（2）self.relu1 = nn.ReLU()

激活函数，使用 ReLU 函数对卷积结果进行非线性变换。

（3）self.maxpool1 = nn.MaxPool2d(kernel_size=(1, 2))

最大池化层，使用大小为 (1, 2) 的池化窗口对特征图进行下采样。

（4）self.flatten = nn.Flatten()：

展平层，用于将输入的多维特征图转换为一维特征向量。在卷积神经网络中，卷积层输出的特征图通常是多维的，而全连接层需要接收一维向量作为输入，因

此需要使用展平层进行转换。

（5）self.fc1 = nn.Linear(40 * 4, 1)：

全连接层，用于将展平后的特征向量映射到最终的输出。nn.Linear 是 PyTorch 提供的线性变换操作，它接收一个输入维度和一个输出维度参数，表示输入特征的维度和输出结果的维度。在这里，输入维度为 40 * 4，输出维度为 1。

（6）self.relu3 = nn.ReLU()

这是全连接层后面的激活函数，对全连接层的输出进行非线性变换。nn.ReLU() 表示使用 ReLU 函数作为激活函数。激活函数的作用是引入非线性，增加模型的表达能力。

4、def forward(self, x):

定义了前向传播方法，该方法描述了模型从输入到输出的计算过程。

在前向传播方法中定义了模型的计算流程：

x = self.conv1(x)：第一个卷积层，将输入特征 x 通过卷积操作得到特征图。

x = self.relu1(x)：激活函数，对卷积结果进行非线性变换。

x = self.maxpool1(x)：最大池化层，对特征图进行下采样。

x = self.conv2(x)：第二个卷积层，将上一层的特征图作为输入进行卷积操作。

x = self.relu2(x)：激活函数，对卷积结果进行非线性变换。

x = self.maxpool2(x)：最大池化层，对特征图进行下采样。

x = self.flatten(x)：展平层，将多维的特征图转换为一维的特征向量。

x = self.fc1(x)：第一个全连接层，输入特征向量经过矩阵乘法和偏置相加得到输出结果。

x = self.relu3(x)：激活函数，对全连接层的输出进行非线性变换。

最后，返回输出结果 x。

通过以上代码，我们定义了一个简单的卷积神经网络模型，本次实践项目定义了一个简单的循环神经网络模型，并生成一维的输出。在模型训练过程中，使用损失函数（均方误差）和优化算法（如随机梯度下降 SGD 或 Adam 优化器）对模型进行训练，以使输出值与真实值之间的误差最小化，实现定义卷积神经网络模型的代码如图 7-1 所示。

```python
# 构建卷积神经网络
class CNNModel(nn.Module):
    def __init__(self, input_channels, output_size):
        super(CNNModel, self).__init__()
        # 第一个卷积层：输入通道数为input_channels，输出通道数为16，卷积核大小为(1, 3)
        self.conv1 = nn.Conv2d(input_channels, 16, kernel_size=(1, 3))
        self.relu1 = nn.ReLU()  # 激活函数
        self.maxpool1 = nn.MaxPool2d(kernel_size=(1, 2))  # 最大池化层

        # 第二个卷积层：输入通道数为16，输出通道数为32，卷积核大小为(1, 3)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=(1, 3))
        self.relu2 = nn.ReLU()  # 激活函数
        self.maxpool2 = nn.MaxPool2d(kernel_size=(1, 2))  # 最大池化层

        self.flatten = nn.Flatten()  # 展平层
        # 一个全连接层：输入维度为40 * 4，输出维度为
        self.fc1 = nn.Linear(40 * 4, 1)
        self.relu3 = nn.ReLU()  # 激活函数

    def forward(self, x):
        x = self.conv1(x)  # 第一个卷积层
        x = self.relu1(x)  # 激活函数
        x = self.maxpool1(x)  # 最大池化层

        x = self.conv2(x)  # 第二个卷积层
        x = self.relu2(x)  # 激活函数
        x = self.maxpool2(x)  # 最大池化层

        x = self.flatten(x)  # 展平
        x = self.fc1(x)  # 第一个全连接层
        x = self.relu3(x)  # 激活函数
        return x
```

图 7-1 定义循环神经网络模型代码图

## 7.3 描述卷积神经网络模型的训练

### 7.3.1 定义超参数

这部分定义了模型的超参数，包括输入通道数、输出大小、学习率和训练的总轮数。

input_channels = 1：输入通道数表示输入数据的通道维度。在卷积神经网络中，输入数据通常是多通道的（比如彩色图像有 RGB 三个通道），但这里要处理的数据为新冠预测中的数据集，并非是彩色图像，因此设定为 1，意味着输入数据是单通道的（只是简单是数据集）。

output_size = 1：输出大小表示模型最终输出的维度或类别数量。在这里，输出大小为 1，意味着模型是用于回归任务，输出一个实数值。

lr = 0.001：学习率表示模型在每次参数更新时所使用的学习步长。学习率决定了每次参数更新的幅度，较大的学习率可能导致参数更新过快而无法收敛，而较小的学习率可能导致收敛速度过慢。在这里，学习率设定为 0.001。

num_epochs = 10001：训练的总轮数，也称为训练的迭代次数。在这里，模型将会进行 10001 轮的训练，每一轮都会使用一个 mini-batch 的样本进行参数更新。

设置超参数代码如图 7-2 所示。

```
# 定义超参数
# 输入数据的通道维度
input_channels = 1
# 最终输出的维度
output_size = 1
# 学习率
lr = 0.001
# 迭代次数
num_epochs = 10001
```

图 7-2 设置超参数代码图

## 7.3.2 模型初始化与优化器选择

模型初始化，通过使用之前定义的 CNNModel 创建了一个模型对象 model。这样做是为了构建一个卷积神经网络模型，其中 input_channels 是输入数据的通道维度，output_size 是最终输出的维度

接下来，我们定义了损失函数和优化器。criterion 使用的是均方误差损失函数（Mean Squared Error，MSE），该损失函数用于衡量模型预测值和真实值之间的差异。optimizer 使用的是随机梯度下降（Stochastic Gradient Descent，SGD）优化器，用于优化模型的参数，其中 lr 是学习率。

通过定义损失函数和优化器，为本次新冠预测实践的卷积神经网络模型训练提供了计算损失和更新参数的工具。损失函数用于计算模型的预测值与真实值之间的误差，而优化器则使用该误差来更新模型的参数，以最小化损失函数。

这样就完成了卷积神经网络模型的初始化和定义损失函数与优化器的过程，为接下来的模型训练做好了准备，型初始化与优化器选择的代码如图 7-3 所示。

```
# 初始化模型
model = CNNModel(input_channels, output_size)
# 定义损失函数
criterion = nn.MSELoss()
# 设置优化器
optimizer = torch.optim.SGD(model.parameters(), lr)
```

图 7-3 模型初始化与优化器选择代码图

## 7.3.3 训练模型与结果评估

1、实现模型训练代码

该步骤实现了循环神经网络模型训练的循环，并定义了存储损失的列表

train_losses 和训练轮数的列表 train_epochs。

在每个训练轮次中：

（1)将输入数据 X_train 转换为 torch.Tensor 类型，并将标签数据 y_train 也转换为 torch.Tensor 类型。

（2）清空优化器的梯度信息，以准备进行反向传播和参数更新。

（3）将输入数据 inputs 输入到模型中，得到模型的输出结果 outputs。

（4）计算模型的损失值，使用 criterion 定义的损失函数，并将输出和标签都压缩为一维张量进行比较。

（5）进行反向传播，计算参数的梯度。

（6）使用优化器 optimizer 更新模型的参数。

（7）每训练 100 个轮次，将当前的损失值 loss.item() 添加到 train_losses 列表中，并将当前的训练轮次 epoch+1 添加到 train_epochs 列表中。同时打印当前轮次的损失值。

通过循环迭代训练轮次，将训练过程中的损失值和轮次记录下来，可以用于后续的可视化和分析，练模型的代码如图 7-4 所示。

```
# 定义存储损失的列表
train_losses = []
train_epochs = []

# 模型训练
for epoch in range(num_epochs):
    inputs = torch.Tensor(X_train)  # 将输入数据转换为torch.Tensor类型
    labels = torch.Tensor(y_train)  # 将标签数据转换为torch.Tensor类型

    optimizer.zero_grad()  # 清空优化器的梯度信息，准备进行反向传播和参数更新

    outputs = model(inputs)  # 输入数据经过模型得到输出结果
    loss = criterion(outputs.squeeze(), labels.squeeze())  # 计算损失值

    loss.backward()  # 反向传播，计算参数的梯度
    optimizer.step()  # 使用优化器更新模型的参数

    if (epoch+1) % 100 == 0:
        train_losses.append(loss.item())  # 将当前的损失值添加到train_losses列表中
        train_epochs.append(epoch+1)  # 将当前的训练轮次添加到train_epochs列表中
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')  # 打印当前轮次的损失值
```

图 7-4 训练模型代码图

2、结果分析

得到的打印结果如图 7-5 示。（这里选择前、后各 20 行数据）。

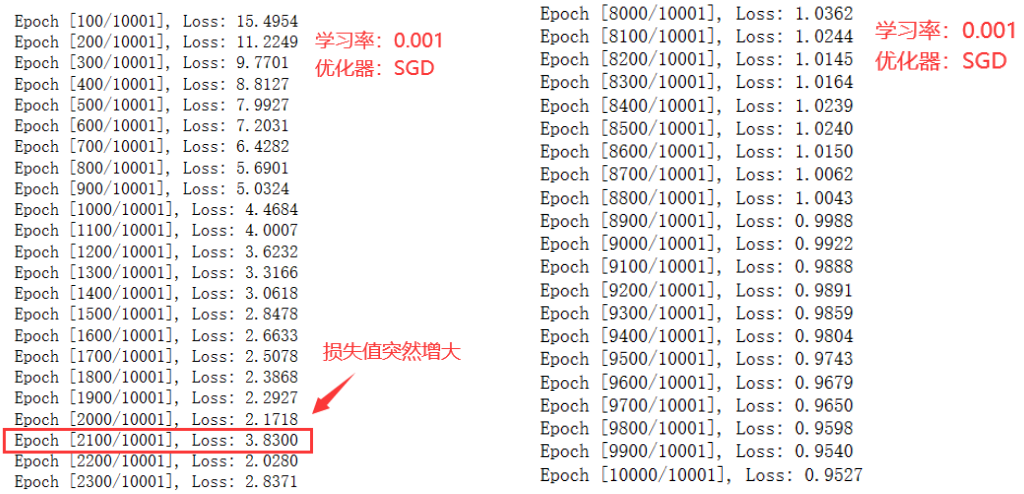


图 7-5 训练模型结果图（前后各 20 行）

通过打印结果发现，损失值随着迭代次数平缓下降，但在迭代次数为 2100 左右时，损失值存在波动，除此之外，无其它问题，损失值在 0.95 附近，甚至可以与优化后的线性回归模型、多层感知机模型相比。

## 7.3.4 可视化分析

1、损失值函数图

绘制损失值函数图使用 Matplotlib 库绘制了损失函数随迭代次数的变化曲线。train_epochs 是存储训练轮数的列表，train_losses 是存储训练过程中损失函数值的列表。通过调用 plt.plot() 函数，将训练轮数和对应的损失函数值传递给该函数，可以绘制出损失函数曲线。plt.xlabel() 和 plt.ylabel() 设置 x 轴和 y 轴的标签，plt.title() 设置图表的标题，plt.legend() 添加图例。最后，使用 plt.show() 显示绘制的图表。

绘制损失值函数图代码如图 7-6 所示。

```
# 绘制损失函数曲线图
plt.plot(train_epochs, train_losses, label='训练损失')
plt.xlabel('迭代次数')
plt.ylabel('损失值')
plt.title('训练损失曲线')
plt.legend()
plt.show()
```

图 7-6 损失值函数代码图

得到的图像如图 7-7 所示。

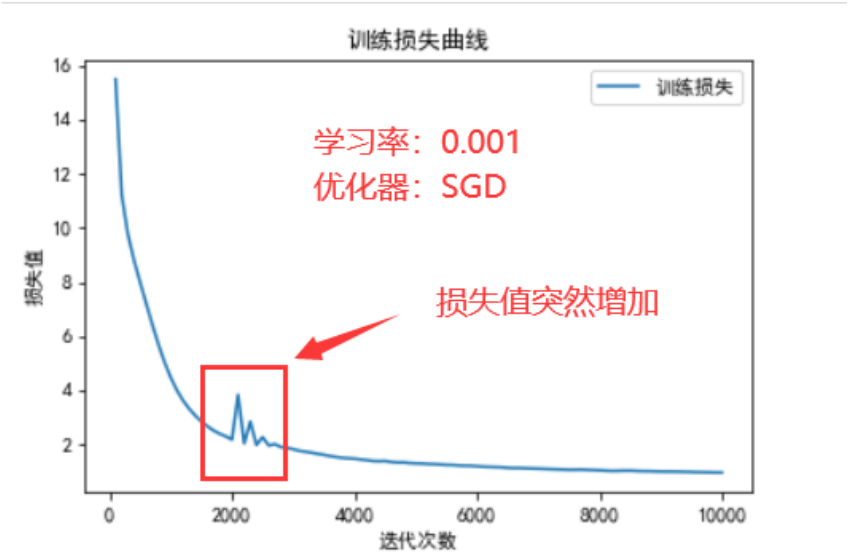


图 7-7 损失值函数图

通过图像可以看到，该卷积神经网络模型的损失率随迭代次数平稳下降，损失值在 0.95 附近，较小。该模型效果较好。

2、真实值与预测值对比图

通过数据预处理，将训练数据集（covid_train）进行了划分，将 20%的样本作为测试集，80%的样本作为训练集。使用训练集用来拟合模型，测试集来查看模型拟合效果。该步骤先将测试集数据 X_test 转换为 PyTorch 张量类型 torch.Tensor。然后，通过将测试集数据输入到模型中进行预测，使用 model(test_inputs) 得到预测结果。接着，通过 squeeze() 方法将结果的维度压缩，使其成为一维数组。最后，使用 detach().numpy() 将结果从 PyTorch 张量类型转换为 NumPy 数组，以便进行后续的处理和分析。可视化部分通过 plt.plot() 函数分别传入真实值 y_test.squeeze() 和预测值 predictions，绘制了两条曲线。使用 '真实值' 和 '预测值' 分别标记了两条曲线的图例。并通过调用 plt.legend() 添加图例，将真实值和预测值对应的图例显示在图表上方。最后，使用 plt.show() 显示绘制的图表，以进行可视化分析。测试集模型预测和可视化的代码如图 7-8 所示。

```
: # 模型预测
test_inputs = torch.Tensor(X_test)
predictions = model(test_inputs).squeeze().detach().numpy()
```

```
: # 可视化分析
plt.plot(y_test.squeeze(), label='真实值')
plt.plot(predictions, label='预测值')
plt.title('真实值与预测值对比图')
plt.legend()
plt.show()
```

图 7-8 测试集模型预测和可视化的代码图

得到的结果如图 7-9 所示。

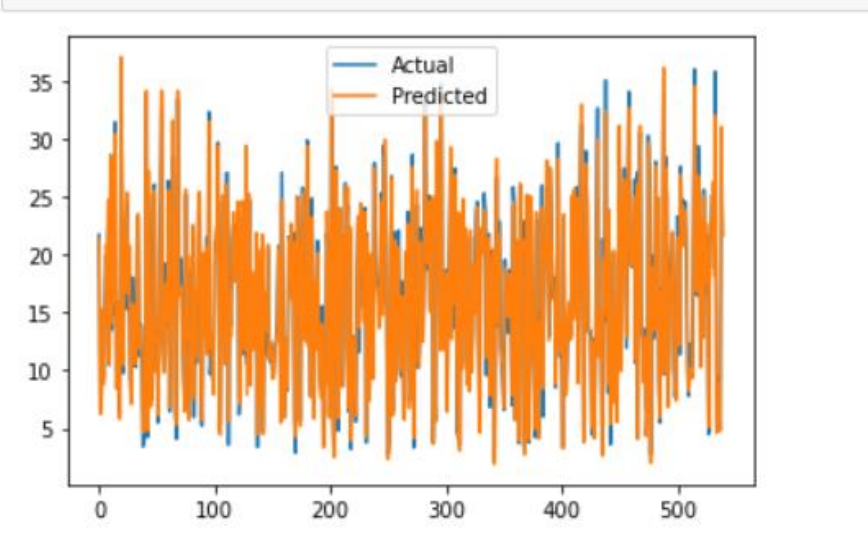


图 7-9 真实值与预测值对比图

通过图像可以看出，因为模型拟合效果较好，预测值基本与真实值一致。

## 7.4 模型优化与调整参数设置

### 7.4.1 优化器的选择

因上述模型拟合效果较好，因此只需更换优化器为 Adam 即可。

1、优化器选择

同线性回归模型一致，改为使用 Adam 优化器，优化器代码如图 7-10 所示。

```
# 使用Adam优化器
optimizer = torch.optim.Adam(model.parameters(), lr)
```

图 7-10 使用 Adam 优化器代码图

2、优化后的模型训练与结果评估（学习率：0.001，迭代次数：10001）

该步骤是训练优化后的模型并记录损失的代码。因为只修改了优化器，对训练过程并未修改，因此可以直接进行模型训练，得到优化后的模型打印结果如图 7-11 所示。

```
Epoch [100/10001], Loss: 37.5211
Epoch [200/10001], Loss: 19.3222
Epoch [300/10001], Loss: 11.2001
Epoch [400/10001], Loss: 8.7405
Epoch [500/10001], Loss: 7.4419
Epoch [600/10001], Loss: 6.3905
Epoch [700/10001], Loss: 5.4939
Epoch [800/10001], Loss: 4.7286
Epoch [900/10001], Loss: 4.0597
Epoch [1000/10001], Loss: 3.4669
Epoch [1100/10001], Loss: 2.9400
Epoch [1200/10001], Loss: 2.5365
Epoch [1300/10001], Loss: 2.2463
Epoch [1400/10001], Loss: 2.0229
Epoch [1500/10001], Loss: 1.8547
Epoch [1600/10001], Loss: 1.7263
Epoch [1700/10001], Loss: 1.6152
Epoch [1800/10001], Loss: 1.5248
Epoch [1900/10001], Loss: 1.4488
Epoch [2000/10001], Loss: 1.3823
```

学习率：0.001
优化器：Adam

```
Epoch [8000/10001], Loss: 0.5822
Epoch [8100/10001], Loss: 0.5782
Epoch [8200/10001], Loss: 0.5751
Epoch [8300/10001], Loss: 0.5721
Epoch [8400/10001], Loss: 0.5691
Epoch [8500/10001], Loss: 0.5661
Epoch [8600/10001], Loss: 0.5626
Epoch [8700/10001], Loss: 0.5590
Epoch [8800/10001], Loss: 0.5558
Epoch [8900/10001], Loss: 0.5557
Epoch [9000/10001], Loss: 0.5522
Epoch [9100/10001], Loss: 0.5470
Epoch [9200/10001], Loss: 0.5441
Epoch [9300/10001], Loss: 0.5415
Epoch [9400/10001], Loss: 0.5387
Epoch [9500/10001], Loss: 0.5365
Epoch [9600/10001], Loss: 0.5337
Epoch [9700/10001], Loss: 0.5332
Epoch [9800/10001], Loss: 0.5286
Epoch [9900/10001], Loss: 0.5270
Epoch [10000/10001], Loss: 0.5277
```

图 7-11 优化后模型训练结果图（前、后 20 行）

通过打印的结果可以看到，优化后的模型损失值随迭代次数下降速度较快，且相比于未优化的模型，模型迭代完成后的损失值从 0.9 附近下降到了 0.52 附近，这说明通过修改优化器使该循环神经网络模型的拟合效果有提升。

5、可视化分析

优化模型后的训练损失曲线图，通过可视化代码，得到优化模型后的训练损失曲线图，如图 7-12 所示。

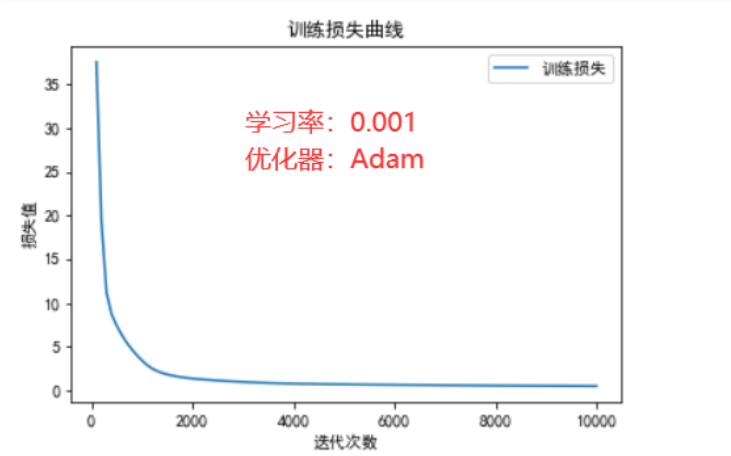


图 7-12 优化模型后的损失值曲线图

通过可视化的结果可以看到，使用 Adam 优化器的模型损失值随迭代次数下降速度相比于使用 SGD 模型的更快一些，且没有波动，更加平稳。

## 7.4.2 调整输入的特征值数量

在本次新冠预测实践开始时，为了排除无关的特征值对模型预测的影响，使线性回归模型与多层感知机模型的拟合效果更好，我们通过使用 Sklearn 库中的 SelectKBest：特征选择库、f_regression：评分函数，以及综合考虑相关系数矩阵热力图，我们从 93 个特征值中选取了 29 个特征值进行模型拟合，而卷积神经网络

在图像分类、目标检测、物体分割、图像生成、自然语言处理（NLP）、推荐系统等领域都有广泛应用。总而言之，卷积神经网络在图像处理和模式识别任务中具有出色的性能，并且也被广泛应用于其他领域的数据分析和模型建设中。而本次新冠预测实践模型对于循环神经网络来说较为简单，因此可以直接使用这 93 个特征值来拟合模型。查看特征值数的结果如图 7-13 所示。

```
(torch.Size([2160, 93]),
 torch.Size([540, 93]),
 torch.Size([2160]),
 torch.Size([540]))
```

图 7-13 查看特征值数量图

由于输入的特征数量由 29 个增加为 93 个，在进行两次卷积以及一次最大化池操作之后，第一个全连接层的输入维度从 160 个扩展成为了 672 个，因此在定义模型时，需要将全连接层的输入维度改为 672，修改操作如图 7-14 所示。

```
self.flatten = nn.Flatten()  # 展平层
# 一个全连接层：输入维度为672，输出维度为
self.fc1 = nn.Linear(672, 1)
self.relu3 = nn.ReLU()  # 激活函数
```

图 7-14 修改全连接层输入维度

输入 93 个特征值后的训练结果如图 7-15 所示。

```
Epoch [100/10001], Loss: 21.7556
Epoch [200/10001], Loss: 9.4668
Epoch [300/10001], Loss: 6.6258
Epoch [400/10001], Loss: 5.1973
Epoch [500/10001], Loss: 4.2632
Epoch [600/10001], Loss: 3.6428
Epoch [700/10001], Loss: 3.1159
Epoch [800/10001], Loss: 2.4829
Epoch [900/10001], Loss: 2.1530
Epoch [1000/10001], Loss: 1.9632
Epoch [1100/10001], Loss: 1.8269
Epoch [1200/10001], Loss: 1.7153
Epoch [1300/10001], Loss: 1.6201
Epoch [1400/10001], Loss: 1.5408
Epoch [1500/10001], Loss: 1.4691
Epoch [1600/10001], Loss: 1.4026
Epoch [1700/10001], Loss: 1.3419
Epoch [1800/10001], Loss: 1.2876
Epoch [1900/10001], Loss: 1.2414
Epoch [2000/10001], Loss: 1.1985
```

学习率：0.001
优化器：Adam
输入的特征值数量：93

```
Epoch [8000/10001], Loss: 0.4709
Epoch [8100/10001], Loss: 0.4675
Epoch [8200/10001], Loss: 0.4643
Epoch [8300/10001], Loss: 0.4610
Epoch [8400/10001], Loss: 0.4583
Epoch [8500/10001], Loss: 0.4548
Epoch [8600/10001], Loss: 0.4734
Epoch [8700/10001], Loss: 0.4490
Epoch [8800/10001], Loss: 0.4448
Epoch [8900/10001], Loss: 0.4417
Epoch [9000/10001], Loss: 0.4386
Epoch [9100/10001], Loss: 0.4354
Epoch [9200/10001], Loss: 0.4325
Epoch [9300/10001], Loss: 0.4299
Epoch [9400/10001], Loss: 0.4265
Epoch [9500/10001], Loss: 0.4235
Epoch [9600/10001], Loss: 0.4234
Epoch [9700/10001], Loss: 0.4204
Epoch [9800/10001], Loss: 0.4163
Epoch [9900/10001], Loss: 0.4162
Epoch [10000/10001], Loss: 0.4107
```

学习率：0.001
优化器：Adam
输入的特征值数量：93

图 7-15 输入 93 个特征值的训练结果图

通过打印的训练结果可以看到，在迭代次数为 10000 时，损失值达到 0.41，与输入 29 个特征值时的 0.52 的差距较小，通过分析卷积神经网络模型得出，可能为全连接层跨度太大导致模型拟合效果较差。

## 7.4.3 通过增加全连接层优化模型

因为输入的特征数量由 29 个增加为 93 个，在进行两次卷积以及一次最大化

池操作之后，第一个全连接层的输入维度从 160 个扩展成为了 672 个，而输出维度仍然是 1，导致跨度较大，因此需要将第一个全连接层的输出维度改为 256，然后需要再添加 2 个全连接层，第二个全连接层的输入维度为 256，输出维度为 64，第三个全连接层的输入维度为 64，输出维度为 1，这样就可以避免因跨度太大导致模型拟合效果变差，添加全连接层的操作如图 7-16、图 7-17 所示。

```
self.flatten = nn.Flatten()  # 展平层
# 第一个全连接层：输入维度为672，输出维度为256
self.fc1 = nn.Linear(672 , 256)
self.relu3 = nn.ReLU()  # 激活函数
# 第二个全连接层：输入维度为256，输出维度为64
self.fc2 = nn.Linear(256 , 64)
# 第三个全连接层：输入维度为64，输出维度为1
self.fc3 = nn.Linear(64 , 1)
```

图 7-16 在初始化方法中添加全连接层

```
x = self.flatten(x)  # 展平
x = self.fc1(x)  # 第一个全连接层
x = self.relu3(x)  # 激活函数
x = self.fc2(x)  # 第二个全连接层
x = self.fc3(x)  # 第三个全连接层
```

图 7-17 在向前传播方法中添加全连接层

得到的训练结果如图 7-18 所示。

```
Epoch [100/10001], Loss: 7.3708
Epoch [200/10001], Loss: 3.6947
Epoch [300/10001], Loss: 2.2684
Epoch [400/10001], Loss: 1.4689
Epoch [500/10001], Loss: 1.1812
Epoch [600/10001], Loss: 1.0327
Epoch [700/10001], Loss: 0.9366
Epoch [800/10001], Loss: 0.8586
Epoch [900/10001], Loss: 0.8013
Epoch [1000/10001], Loss: 0.7523
Epoch [1100/10001], Loss: 0.7173
Epoch [1200/10001], Loss: 0.7291
Epoch [1300/10001], Loss: 0.6588
Epoch [1400/10001], Loss: 0.6282
Epoch [1500/10001], Loss: 0.6054
Epoch [1600/10001], Loss: 0.6084
Epoch [1700/10001], Loss: 0.5625
Epoch [1800/10001], Loss: 0.5569
Epoch [1900/10001], Loss: 0.5307
Epoch [2000/10001], Loss: 0.5210
```

添加全连接层之后的结果

```
Epoch [8000/10001], Loss: 0.2005
Epoch [8100/10001], Loss: 0.1991
Epoch [8200/10001], Loss: 0.3399
Epoch [8300/10001], Loss: 0.2005
Epoch [8400/10001], Loss: 0.2200
Epoch [8500/10001], Loss: 0.2633
Epoch [8600/10001], Loss: 0.2666
Epoch [8700/10001], Loss: 0.2324
Epoch [8800/10001], Loss: 0.1898
Epoch [8900/10001], Loss: 0.2679
Epoch [9000/10001], Loss: 0.1972
Epoch [9100/10001], Loss: 0.2130
Epoch [9200/10001], Loss: 0.1884
Epoch [9300/10001], Loss: 0.2904
Epoch [9400/10001], Loss: 0.2293
Epoch [9500/10001], Loss: 0.3175
Epoch [9600/10001], Loss: 0.1955
Epoch [9700/10001], Loss: 0.1724
Epoch [9800/10001], Loss: 0.1622
Epoch [9900/10001], Loss: 0.1699
Epoch [10000/10001], Loss: 0.1587
```

添加全连接层后的输出结果

图 7-18 添加全连接层后的训练结果图

## 7.4.3 关于卷积神经网络模型的结论

通过上述优化器的选择、输入特征值数量的调整以及全连接层的添加，确定本次新冠预测实践的卷积神经网络模型使用 Adam 优化器，因为该模型拟合效果很好，因此这里再做参数调整并无实际意义，所以确定了该模型在迭代次数为 10001 次，学习率为 0.001 时的效果较好。

## 7.5 生成预测结果并导出文件

1、通过优化后的线性回归模型得到预测结果

使用训练好的模型对测试数据进行预测，并将预测结果转换为 NumPy 数组。

具体来说，代码中的各个步骤如下：

（1）model(test4)：将测试数据 test4 输入到模型中，获取模型的预测输出。

（2）.squeeze()：将预测输出的维度为 1 的维度进行压缩，使得输出结果变为一维数组。

（3）.detach()：从计算图中分离出预测结果，以便后续的操作不会影响梯度计算。

（4）.numpy()：将预测结果转换为 NumPy 数组。

最终，变量 predictions 存储了模型对训练集数据的预测结果，以 NumPy 数组的形式保存。以便下面的文件导出。得到预测结果的代码如图 7-19 所示。

```
# 预测结果
predictions = model(test4).squeeze().detach().numpy()
```

图 7-19 得到预测结果代码图

2、导出文件

（1）创建一个名为 CNN_sampleSubmissionn 的空 DataFrame，并将其保存为 CSV 文件，并保存到桌面的特定文件夹中。

（2）使用 sampleSubmission 文件中的“id”列来填充 CNN_sampleSubmission 的“id”列。

（3）将模型的预测结果数组 predictions 的前 len(sampleSubmission)个值写入到新的样本提交表 CNN_sampleSubmission 的“ested_positive”列中。这里使用了切片操作确保只写入与原始样本提交表相匹配的部分数据。

（4）将更新后的 CNN_sampleSubmission 再次保存为 CSV 文件，覆盖之前创建的同名文件，导出预测结果的代码如图 7-20 所示。

```
#导入 sampleSubmission 表
sampleSubmission = pd.read_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\sampleSubmission.csv', sep = ',')
# 创建新表
CNN_sampleSubmission = pd.DataFrame()
CNN_sampleSubmission .to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\CNN_sampleSubmission.csv', index=False)
# 写入id列
CNN_sampleSubmission["id"] =sampleSubmission["id"]
```

```
# 将数据写入特定一列
CNN_sampleSubmission['tested_positive'] = predictions[:len(sampleSubmission)]  # 只写入与数据帧行数相匹配的部分数组数据
# 写回 CSV 文件
CNN_sampleSubmission.to_csv(r'C:\Users\Administrator\Desktop\3-COVID-19\CNN_sampleSubmission.csv', index=False)
```

图 7-20 导出预测结果代码图

# 第八章 实验结果分析

## 8.1 分析相同模型不同参数的效果

### 8.1.1 基于线性回归模型的实验结果分析

1、模型优化前后的对比

基于线性回归模型的新冠预测实践，在迭代次数 10001，学习率为 0.03 的情况下分析优化前后的模型对比，优化前使用 SGD 优化器、均方误差损失函数，优化后使用 Adam 优化器、和带有 L1 正则化的损失函数，其损失值图像如图 8-1 所示。

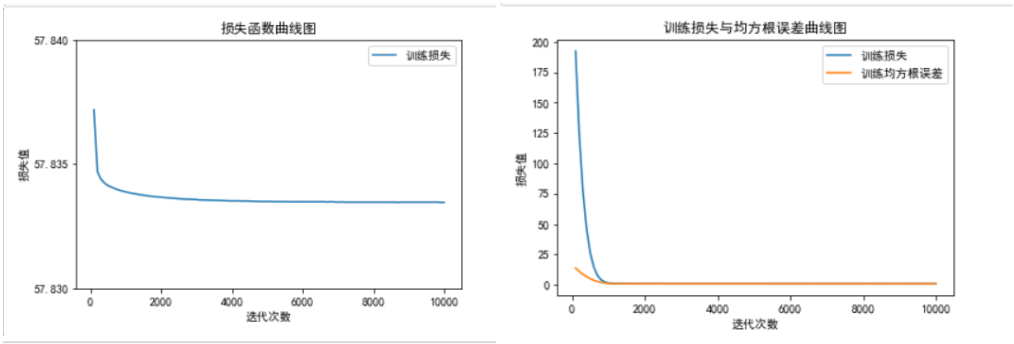


图 8-1 模型优化前后对比图（左：SGD，右：Adam+L1）

通过图像可以看出，优化前的模型损失只在 57.83 到 57.84 之间，下降速度非常慢，拟合效果很差，而优化后的模型损失值下降速度较快，损失值较低，模型拟合效果较好。

2、模型优化后不同参数的对比

基于线性回归模型的新冠预测实践，在使用 Adam 优化器、和带有 L1 正则化的损失函数的情况下，对比不同参数（学习率、迭代次数）的模型拟合效果。

（1）迭代次数：10000，学习率：0.03，其损失值图像如图 8-2 所示。

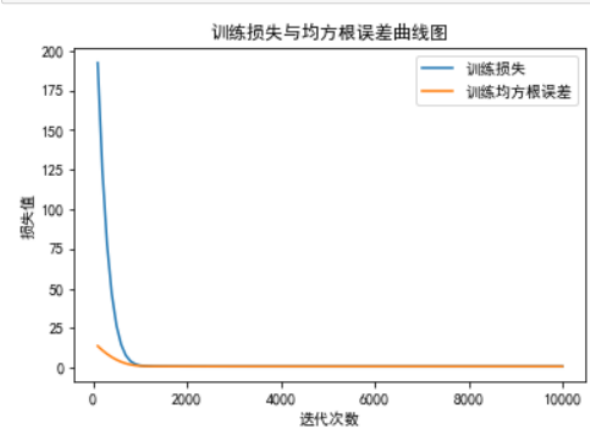


图 8-2 迭代次数：10000，学习率：0.03

（2）迭代次数：1001，学习率：0.05，其损失值图像如图 8-3 所示。

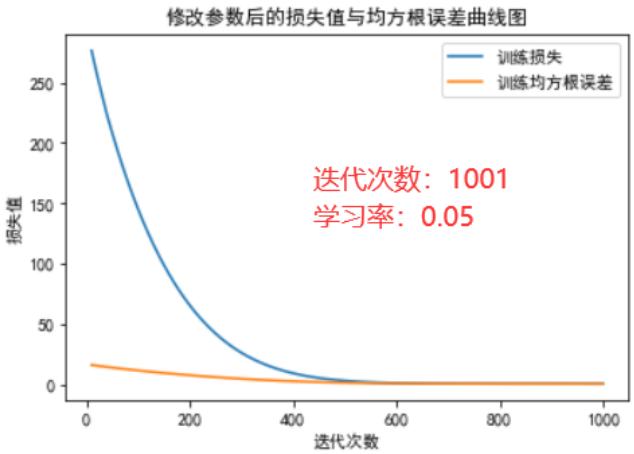


图 8-3 迭代次数：1001，学习率：0.05

（3）迭代次数：1001，学习率：0.3，其损失值图像如图 8-4 所示。

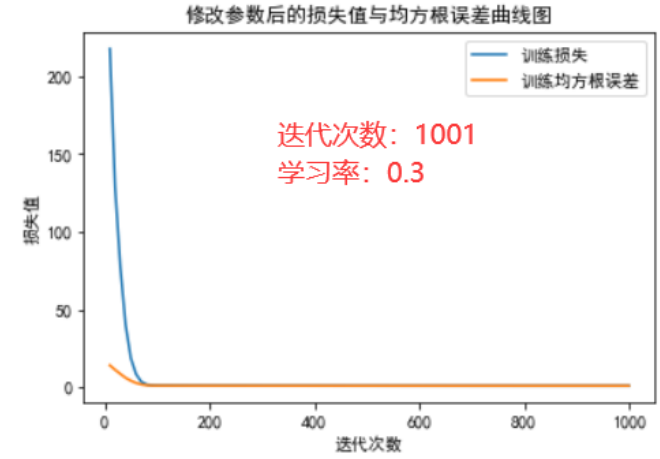


图 8-4 迭代次数：1001，学习率：0.3

（4）迭代次数：200，学习率：3，其损失值图像如图 8-5 所示。

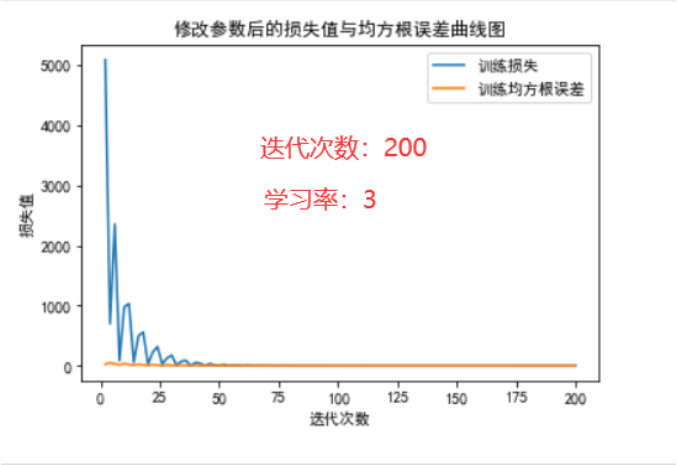


图 8-5 迭代次数：200，学习率：3

（5）迭代次数：500，学习率：0.8，其损失值图像如图 8-6 所示。

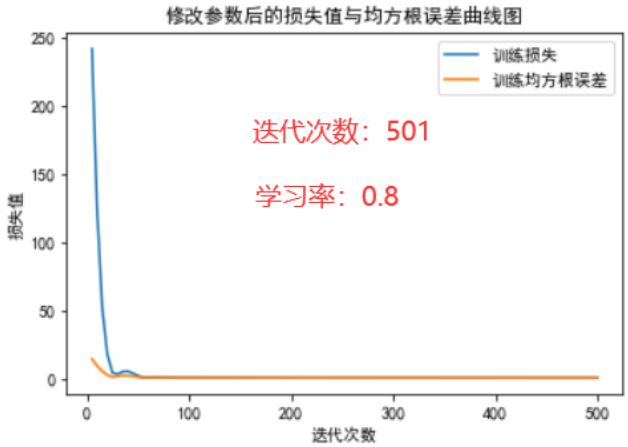


图 8-6 迭代次数：501，学习率：0.8

通过对比图像，可以得到以下结论：

迭代次数：10000，学习率：0.03，模型拟合效果较好，损失值已经收敛，但学习率低导致迭代次数很大，效率不高。

迭代次数：1001，学习率：0.05，损失值平稳下降，但未能收敛。

迭代次数：1001，学习率：0.3，模型拟合效果较好，损失值曲线快速下降，且已经收敛，迭代次数适中，效率较快。

迭代次数：201，学习率：3，学习率过大，导致损失值随迭代次数变化波动较大，效果较差。

迭代次数：501，学习率：0.8，模型拟合效果较好，损失值曲线快速下降，且已经收敛，虽然有较小的波动，但总体来说较为稳定，迭代次数更少，效率更快。

因此确定了该模型在迭代次数为 500 次，学习率为 0.8 时的效果较好，同时效率较高。

## 8.1.2 基于多层感知机模型的实验结果分析

1、模型优化前后的对比

基于多层感知机模型的新冠预测实践，在迭代次数 10001，学习率为 0.03 的情况下分析优化前后的模型对比，优化前使用 SGD 优化器、均方误差损失函数（Mean Squared Error，MSE），优化后使用 Adam 优化器、和带有 L1 正则化的损失函数，其损失值图像如图 8-7 所示。

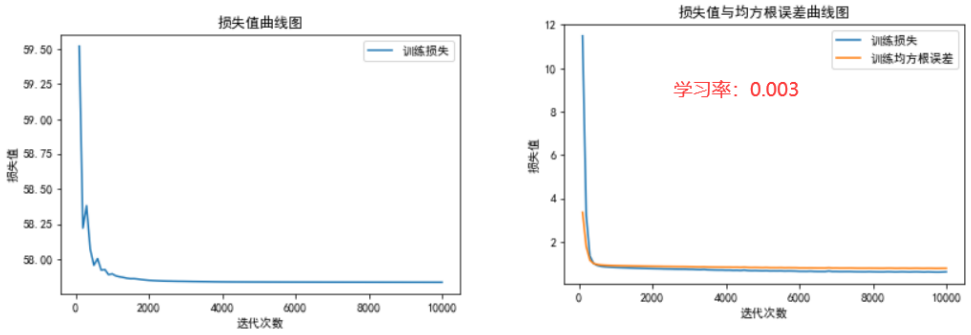


图 8-7 模型优化前后对比图（左：SGD，右：Adam+L1）

通过图像可以看出，优化前的模型损失只下降到了 58，下降速度非常慢，拟合效果很差，而优化后的模型损失值下降速度较快，损失值较低，模型拟合效果较好。

2、模型优化后不同参数的对比

基于线性回归模型的新冠预测实践，在使用 Adam 优化器、和带有 L1 正则化的损失函数的情况下，对比不同参数（学习率、迭代次数）的模型拟合效果。

（1）迭代次数：10001，学习率：0.003，其损失值图像如图 8-8 所示。

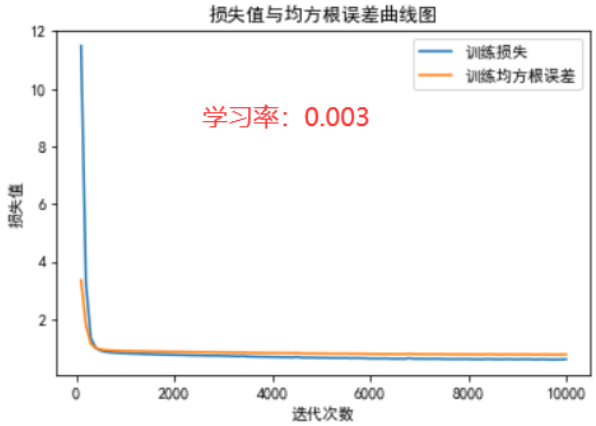


图 8-8 迭代次数：1001，学习率：0.003

（2）迭代次数：1001，学习率：0.03，其损失值图像如图 8-9 所示。

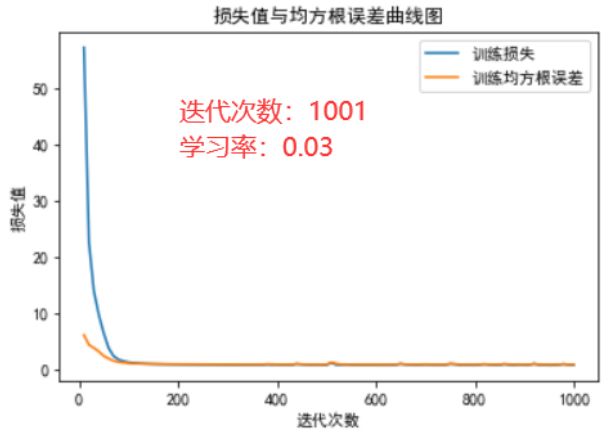


图 8-9 迭代次数：1001，学习率：0.03

（3）迭代次数：1001，学习率：0.3，其损失值图像如图 8-10 所示。

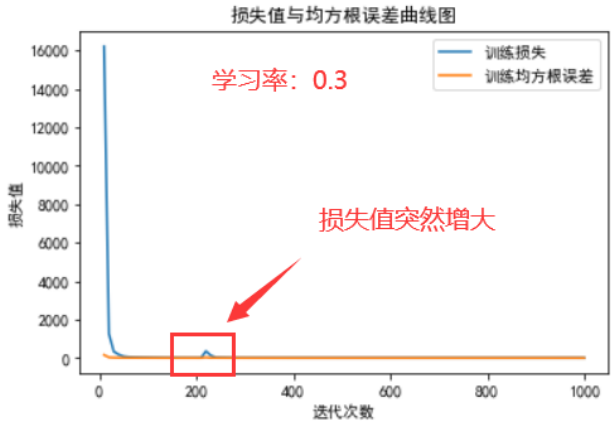


图 8-10 迭代次数：1001，学习率：0.3

（4）迭代次数：500，学习率：0.1，其损失值图像如图 8-11 所示。

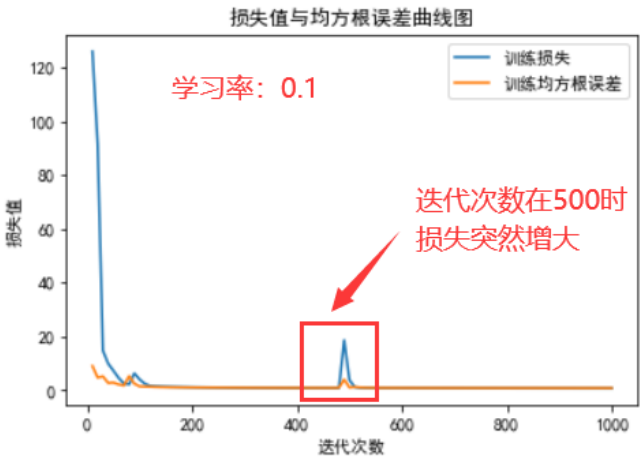


图 8-11 迭代次数：500，学习率：0.1

（5）迭代次数：10000，学习率：0.005，其损失值图像如图 8-12 所示。

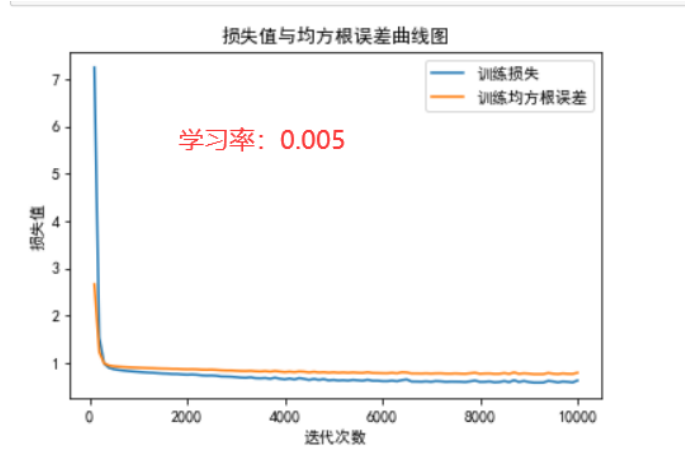


图 8-12 迭代次数：10000，学习率：0.005

通过对比图像，可以得到以下结论：

迭代次数：10000，学习率：0.003，模型拟合效果较好，损失值已经收敛，但学习率低导致迭代次数很大，效率不高。

迭代次数：1001，学习率：0.03，损失值平稳下降，趋于收敛。

迭代次数：1001，学习率：0.3，损失值变大，且存在些许波动，学习率太大。

迭代次数：500，学习率：0.1，学习率过大，导致损失值随迭代次数变化波动较大。

迭代次数：10001，学习率：0.005，模型拟合效果较好，但存在波动，且迭代次数较大，效率较低。

通过对比，确定了该模型在迭代次数为 1001 次，学习率为 0.03 时的效果较好。

### 8.1.3 基于循环神经网络模型的实验结果分析

1、不同优化器对比

基于循环神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001 的情况下分析不同优化器的模型对比，前者使用 SGD 优化器，后者使用 Adam 优化器，其损失值图像如图 8-13 所示。

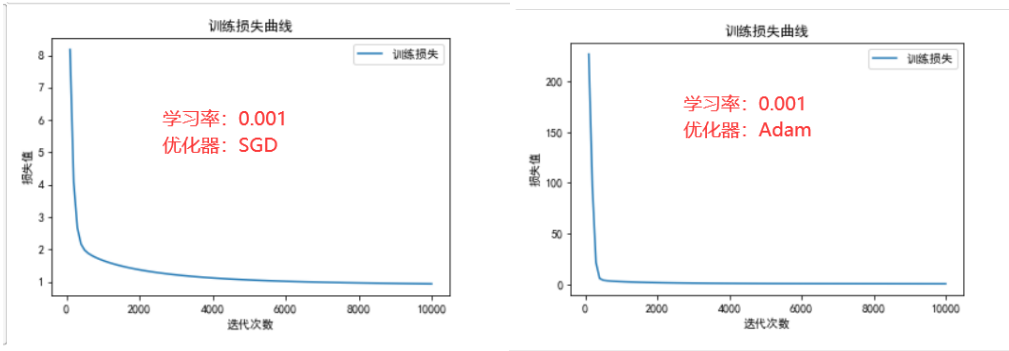


图 8-13 不同优化器对比（左：SGD，右：Adam）

通过图像可以看出，使用 Adam 优化器的损失值下降速度更快，效率更高。

2、调整特征值前后对比

基于循环神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001，优化器为 Adam 的情况下分析不同特征值的模型对比，其损失值情况如图 8-14 所示。

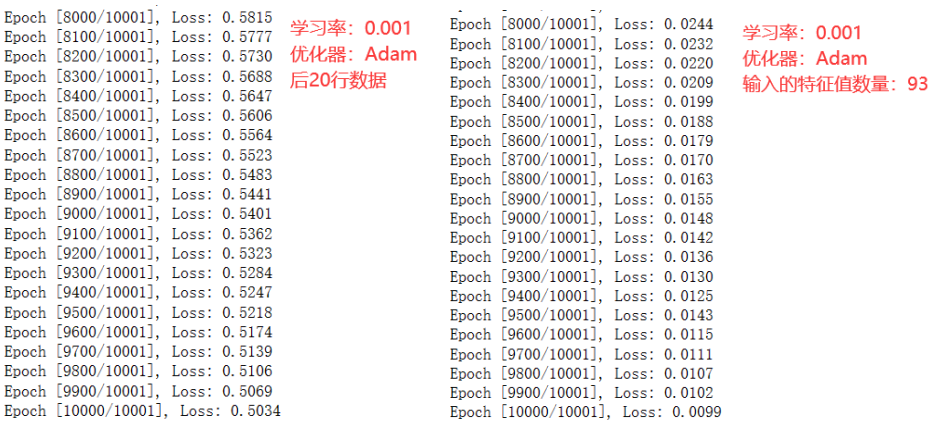


图 8-14 不同特征值数量对比（左：29，右：93）

通过图像可以看出特征值为 93 时的拟合效果更好。

因此本次基于神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001，优化器为 Adam，输入特征值数量为 93 时的模型拟合情况较好。

## 8.1.4 基于卷积神经网络模型的实验结果分析

1、不同优化器对比

基于卷积神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001 的情况下分析不同优化器的模型对比，前者使用 SGD 优化器，后者使用 Adam 优化器，其损失值图像如图 8-15 所示。

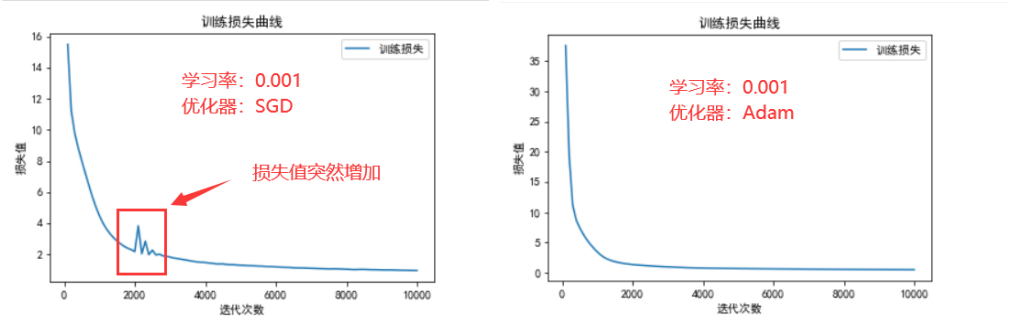


图 8-15 不同优化器对比（左：SGD，右：Adam）

通过图像可以看出，使用 Adam 优化器的损失值下降速度更快，且下降更加平稳，没有波动，效果更好。

2、调整特征值前后对比

基于卷积神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001，优化器为 Adam 的情况下分析不同特征值的模型对比，其损失值情况如图 8-16 所示。

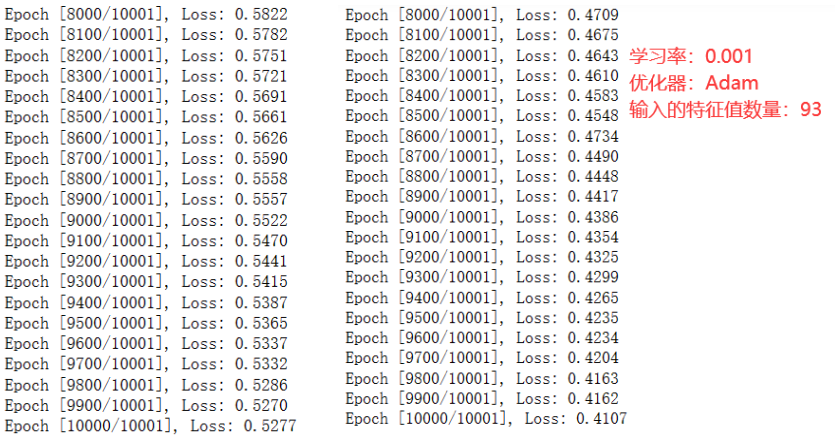


图 8-16 不同特征值数量对比（左：29，右：93）

通过图像可以看出，特征值为 93 时的拟合效果比特征值为 29 时的效果较好一些。

3、添加全连接层前后对比

基于卷积神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001，优化器为 Adam，特征值数量为 93 的情况下，分析不同全连接层数量的模型对比，其损失值情况如图 8-17 所示。


图 8-17 不同特征值数量对比（左：29，右：93）

通过图像可以看出，3 个全连接层的拟合效果比 1 个全连接层的拟合效果更好。

因此本次基于神经网络模型的新冠预测实践，在迭代次数 10001，学习率为 0.001，优化器为 Adam，输入特征值数量为 93，全连接层个数为 3 时的模型拟合情况最好。

## 8.2 对比不同模型的实验结果

本次对比，只对比各模型拟合效果最好情况。

1、线性回归模型

该模型在迭代次数为 500 次，学习率为 0.8，优化器为 Adam，输入特征值数量为 29 时的效果最好，其损失值图像如图 8-18 所示。

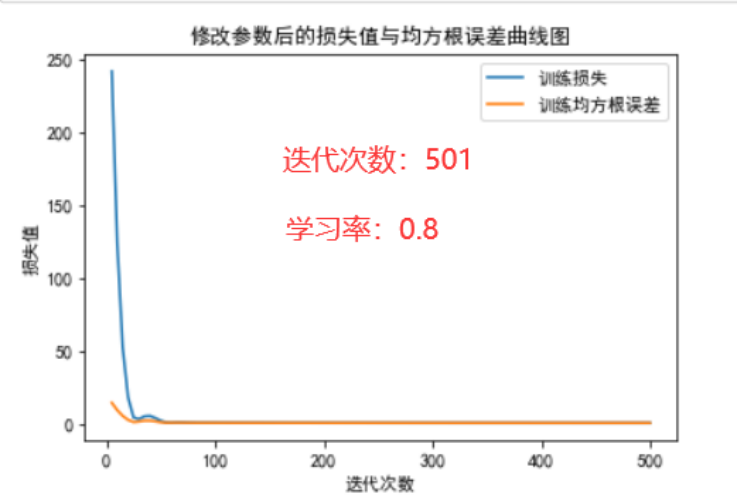


图 8-18 线性回归模型

2、多层感知机模型

该模型在迭代次数为 1000 次，学习率为 0.03，优化器为 Adam，输入特征值数量为 29 时的效果最好，其损失值图像如图 8-19 所示。

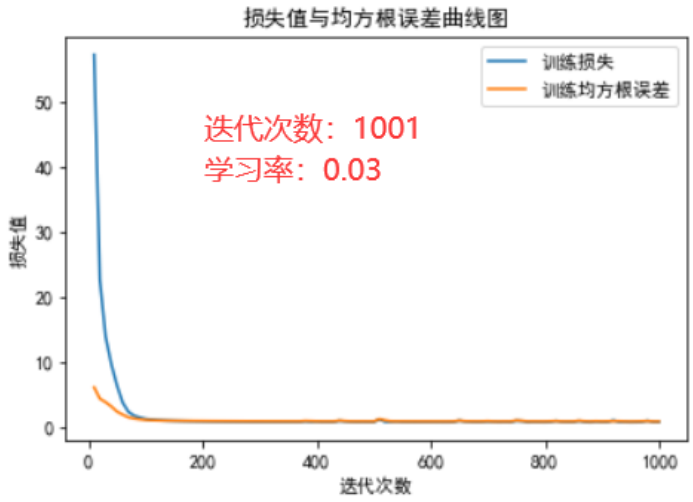


图 8-19 多层感知机模型

3、循环神经网络模型

该模型在迭代次数为 10000 次，学习率为 0.03，优化器为 Adam，输入特征值数量为 93 时的效果最好，其损失值图像如图 8-20 所示。

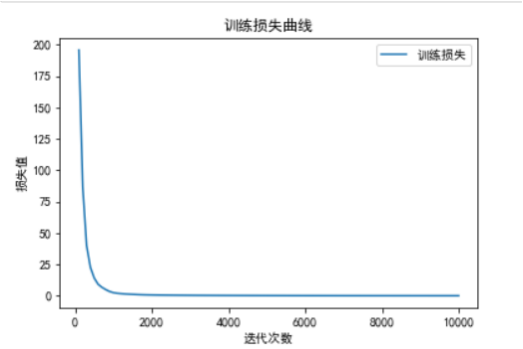


图 8-20 循环神经网络模型

4、卷积神经网络

该模型在迭代次数为 10000 次，学习率为 0.001，优化器为 Adam，输入特征值数量为 93，全连接层为 3 时的效果最好，其损失值图像如图 8-21 所示。

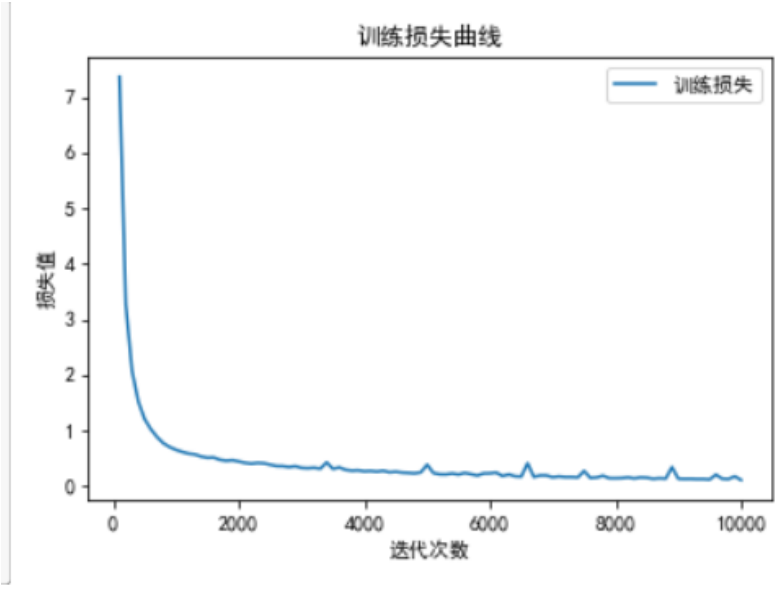


图 8-21 卷积神经网络模型

综上所述，通过对比不同模型，得到拟合效果最好的是循环神经网络模型。

# 结论

通过本次新冠病例预测实践项目，完成了基于线性回归模型的新冠病例预测分析、基于多层感知机模型的新冠病例预测分析、基于循环神经网络模型的新冠病例预测分析和基于卷积神经网络模型的新冠病例预测分析，并在同一模型内选择了拟合效果最好的参数设置，以及拟合效果最好的模型。

1、具体实现内容如下：

（1）在实现线性回归模型时，该项目简述了线性回归模型的原理以及公式，说明了线性回归模型是如何定义的，并详细描述了模型的训练过程，并通过优化器选择和添加 L1 正则化优化了模型，并添加了 RMSE 指标，通过可视化分析进行了参数调整，得到了拟合效果较好的线性回归模型。

（2）在实现多层感知机模型时，该项目简述了多层感知机模型的原理以及公式，定义了多层感知机的模型，包括三个全连接层，并详细描述了模型的训练过程，并通过优化器选择和添加 L1 正则化优化了模型，并添加了 RMSE 指标，通过可视化分析进行了参数调整，得到了拟合效果较好的多层感知机模型。

（3）在实现循环神经网络模型时，简述了循环神经网络模型的原理以及公式，定义了一个简单的循环神经网络模型包括一个 RNN 层、一个全连接层，并详细描述了模型的训练过程，并通过优化器选择和特征值选取优化了模型。

（4）在实现卷积神经网络模型时，简述了卷积神经网络模型的原理以及公式，定义了卷积神经网络模型，包括 2 个卷积层，每个卷积层还包含一个最大化池，以及一个展平层，一个全连接层，并详细描述了模型的训练过程，然后通过优化器选择、特征值选取和添加全连接层优化了模型，得到了拟合效果较好的卷积神经网络模型。

2、通过对比得到以下结论：

（1）本次实践的线性回归模型在迭代次数为 500 次，学习率为 0.8，使用 Adam 优化器，输入特征值个数为 29，L1 正则化权重为 0.001，并计算带有 L1 正则化的损失函数时，该线性回归模型效果最好。

（2）本次实践的多层感知机模型在迭代次数为 1001 次，学习率为 0.03，使用 Adam 优化器，输入特征值个数为 29，L1 正则化权重为 0.001，并计算带有 L1 正则化的损失函数时，该多层感知机模型效果最好。

（3）本次实践的循环神经网络模型在迭代次数为 10001 次，学习率为 0.001，使用 Adam 优化器，输入特征值个数为 93 时，该循环神经网络模型效果最好。

（4）本次实践的卷积神经网络模型在迭代次数为 10001 次，学习率为 0.001，

使用 Adam 优化器，输入特征值个数为 93，全连接层为 3 个时，该循环神经网络模型效果最好。

（5）通过对比不同模型的拟合效果，拟合效果最好的是循环神经网络模型。

# 参考文献


[1] 马龙,张泽宇,高瑞.新冠疫情预测研究综述[J].模式识别与人工智能,2021,144(17):157-167.

[2]Pradhan,P.,Pandey,A.K.,Mishra,A.K.,Gupta,P.,Tripathi,P.K.,Menon,M.B.,&Mohapatra,P.K.J.Forecasting the daily count of COVID-19 cases in India using hybrid machine learning models[J]. Chaos, Solitons & Fractals,2021,152(11):50-51.

[3]王浩,王兆杰,董岩,常锐,徐超.从早期数据预测中国新冠疫情累计病例数[J].混沌、孤立子与分形,2021,143(11):04-11.

[4]胡宇,詹哲,罗琦,李松.采用优化特征选择和机器学习模型预测新冠肺炎每日确诊数[J].混沌、孤立子与分形, 2020,140(11):71-73.

[5]Kiani,M.M.,Khadangi,E.,Barzegari,H.,Shafipour,R.,&Shahmoradi,L. A hybrid model based on clustering and time series analysis to predict the COVID-19 epidemic: A case study of Iran[J]. Chaos, Solitons & Fractals, 2021,151(11)30-34.

[6]黎爽.基于 Python 科学计算包的金融应用实现[J].江西财经大学,2017,6.

[7]聂杲.pandas 大数据技术在央行监管中的应用[J].中国人民银行九江市中心支行，2020,1.

[8]范丽.基于 Python 的数据可视化[J].中国人民银行鞍山市中心支行，2020,4

[9]Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Desmaison, A. PyTorch: An imperative style, high-performance deep learning library[J]. In Advances in neural information processing systems .2019,45(17):8034-8035.

[10] 何小年,段凤华.基于 Python 的线性回归案例分析[J].微型电脑应用,2022,38(11):35-37.

[11]张潇潇,张焕杰.基于机器学习的新冠病例预测研究[J].计算机与数字工程,2020,48(12):2423-2427.

[12]邓华,肖叶青,李林.基于深度学习的新冠病例预测方法研究[J].现代计算机,2021,41(6):170-174.

[13]周志华.机器学习[M].清华大学出版社,2016.

[14]T. O. An Optimized DTW Algorithm Using the RMSE Approach to Classify the Liquids in Ka-Band[J]. IETE Journal of Research,2020,68(4):78-81.

[15]LeCun,Y.,Bengio,Y.,&Hinton,G..Deeplearning.Nature[J], 2015,521(7553):436-444.

[16] [10]Kadek I S,I.P.A. B,Sri M D A. Detection of fake news using deep learning CNN–RNN based methods[J]. ICT Express,2022,8(3):18-20.

[17]LeCun,Y.,Bengio,Y.,& Hinton, G. Deep learning. Nature, 2015,521(7553),:436-444.