模型介绍

吴清柳

Beijing University of Posts and Telecommunications

August 22, 2023

Table of Contentes I

Table of Contentes II

Table of Contentes III

Overview

通过该 ppt，对 9 个实验使用的模型进行介绍．九个实验分别是鸢尾花分类实验，emojify 人脸表情识别实验，贷款预测，房价预测，MNIST 手写数字识别，股票价格预测，泰坦尼克号生还预测，红酒质量预测和假新闻预测．
使用的模型有机器学习方法如决策树，逻辑回归，XGBoost 等；以及深度学习方法如全连接网络，CNN 等．

鸢尾花分类实验

代码和数据集

该论文的复现代码
在https://github.com/Micuks/code/blob/master/gnn/1_iris/iris-classification/train.ipynb 数据集使用了鸢尾花分类数据集.

模型信息

模型使用 PyTorch 编写, 为三层的全连接网络. 前两层使用 ReLU 激活函数, 第三层为完成分类功能, 使用 Softmax 作为激活函数, 输出分类结果.

- 第一层全连接层输入特征数量为 4, 输出特征数量为 25;
- 第二层全连接层输入特征数量为 25, 输出特征数量为 30;
- 第三层全连接层输入特征数量为 30, 输出特征数量为 3, 对应三类鸢尾花;

任务亮点

入门级任务:

1. Iris 数据集简单、维度低，同时提供了一个实际的分类问题，便于初学者学习分类问题;
2. 多类分类：数据集包含三种不同的鸢尾花，目标是根据花的各种特征将其正确分类。
3. 简单的特征维度：Iris 数据集只有四个特征，即萼片和花瓣的长度与宽度，使得数据可视化和理解都相对容易。
4. 明确的类别边界：在 Iris 数据集中，某些类别之间的特征边界非常清晰，便于初学者进行分类.

数据的基本信息

Iris 数据集由英国生物学和统计学家 Ronald A. Fisher 在 1936 年提出. 数据集有 150 个样本, 每类花有 50 个样本.

鸢尾花种类

鸢尾花种类有三种, 分别是:

- Iris-setosa;
- Iris-versicolor;
- Iris-virginica;

数据的基本信息

Iris 数据集由英国生物学和统计学家 Ronald A. Fisher 在 1936 年提出. 数据集有 150 个样本, 每类花有 50 个样本.

鸢尾花特征

每个样本有 4 个特征, 单位为厘米.

- **萼片长度 (sepal length)**
- **萼片宽度 (sepal width)**
- **花瓣长度 petal length**
- **花瓣宽度 (petal width)**

目标变量

目标变量是花的种类, 即前面提到的三种之一.

数据的基本信息

Iris 数据集由英国生物学和统计学家 Ronald A. Fisher 在 1936 年提出. 数据集有 150 个样本, 每类花有 50 个样本.

数据分布

数据集是平衡的, 意味着每个种类都有相同数量的样本 (每个种类 50 个样本);

数据清洁度

鸢尾花数据集是清洁的, 没有任何的缺失值或者显著的离群值, 因此需要很少的预处理.

模型信息

模型搭建代码如下.

```
class Model(nn.Module):
def __init__(self, input_feats=4, hidden_layer1=25,
 ↪    hidden_layer2=30, output_feats=3) -> None:
    super().__init__()
    self.fc1 = nn.Linear(input_feats, hidden_layer1)
    self.fc2=nn.Linear(hidden_layer1, hidden_layer2)
    self.out=nn.Linear(hidden_layer2, output_feats)

def forward(self, x):
    x=F.relu(self.fc1(x))
    x=F.relu(self.fc2(x))
    x=self.out(x)

    return x
```

模型介绍

输入数据为鸢尾花 (Iris) 的特征数据. 鸢尾花有三种, 分别是 Setosa, Versicolour, Virginica. 每个都有 4 个特征: sepal length, sepal width, 以及 petal width.
模型的结构介绍如下.

模型结构

1. **输入层**神经元数量为 4, 代表 4 个输入特征.
2. **第一个隐藏层 'fc1'** 全连接层 (dense layer), 25 个神经元, 激活函数为 ReLU, 当输入为正时输出输入本身, 否则输出 0. 如果没有非线性激活函数, 则模型将等效为线性拟合函数, 拟合性能下降.
3. **第二个隐藏层 'fc2'** 另一个全连接层, 30 个神经元, 激活函数为 ReLU;
4. **输出层 'out'** 一个全连接层, 3 个神经元, 对应鸢尾花的三个分类: Setosa, Versicolour, Virginica.

训练方法

使用交叉熵作为损失函数，Adam 为优化器，学习率为 0.01，在训练集上进行 100 轮训练．对 loss 可视化观察后看到大约 40 轮后模型已经接近收敛．

训练代码

```
epochs=100
losses=[]
for i in range(epochs):
    y_pred=model.forward(X_train)
    loss=criterion(y_pred,y_train)
    losses.append(loss)
    print(f'epoch: {i:2} loss: {loss.item():10.8f}')

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

初始化

- 一轮代表对所有训练样本的一个完全的前向传播和反向传播过程;
- *losses* 列表会存储在每轮中计算的 loss 值，用于追踪模型的表现;

$$epochs = 100$$
$$losses = []$$

训练循环：前向传播和损失计算

Within the loop, the model predicts outcomes and computes the discrepancy between predictions and actual values. 在训练循环中，模型预测输出，计算预测值和实际值之间的差异;

```
for i in range(epochs):
    y_pred = model.forward(X_train)
    loss = criterion(y_pred, y_train)
    losses.append(loss)
```

训练循环：日志和优化

- 通过打印轮数和 loss 值来监控训练进程;
- 计算梯度并更新模型参数来降低 loss;

```
print(f'epoch: {i:2} loss: {loss.item():10.8f}')
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

利用 CNN 进行面部表情识别

在该实验中，利用 Tensorflow 来搭建 CNN 网络进行对摄像头采集的人脸进行表情识别. 模型使用 GPU 进行训练，得到的权重文件作为表情识别的依据. 此处使用 Tensorflow 是为了将 Pytorch 和 Tensorflow 两个常用框架都进行学习.

任务亮点

1. **实时应用场景**借助 Tensorflow 训练完成后，导出的权重文件可以实时对摄像头画面进行标签识别.
2. **深度学习应用**借助卷积神经网络 (CNN) 为面部表情分类这一图像任务提供了强有力的帮助，这是因为 CNN 特别擅长捕捉图像数据中的层次结构和模式.
3. **多分类问题**面部情感识别通常包括多个类别，例如快乐，悲伤，惊讶，愤怒等，增加了任务的复杂型.

数据基本信息

使用 FER2013 数据集.

1. **数据来源** FER2013 是为了 2013 年 ICML 举办的情感识别挑战而创建的. 该数据集在 Kaggle 上发布, 旨在为面部情感识别任务提供一个标准化的挑战.
2. **样本数量**数据集包含 35,887 张图像.
3. **数据类型**灰度图像. 相比于彩色图像, 灰度图像可以减少计算复杂型, 且足够进行表情识别;
4. **图像大小**图像大小为 48×48 像素.

数据基本信息

使用 FER2013 数据集.

5. **类别/标签**数据集包括七个面部表情标签:
   - 0=Angry(愤怒)
   - 1=Disgust(厌恶)
   - 2=Fear(恐惧)
   - 3=Happy(高兴)
   - 4=Sad(悲伤)
   - 5=Surprise(惊讶)
   - 6=Neutral(中性)
6. **数据分割**在本次实验中, FER2013 被分为三个子集:
   - **训练集** 有 28,709 个样本;
   - **验证集** 有 3,589 个样本, 用于训练过程中验证;
   - **测试集** 有 35,89 个样本, 用于最终的测试评估;

数据基本信息

使用 FER2013 数据集.

7. **数据不均衡问题** 数据集中某些情感的样本数量比其他情感多. 例如, “Happy’’ 表情的样本数量远多于 “Disgust”.
8. **挑战性** 由于图像是灰度的, 并且分辨率较低 (48×48), 因此识别模型需要能够从较少的信息中捕获细微的面部特征.

CNN 模型结构

- 目的：从灰度图片进行表情识别;
- 图片输入形状：$48 \times 48 \times 1$;
- 层之间顺序堆叠;

$$\textit{emotion\_model} = \textit{Sequential()}$$

Keras 中的 'Sequential' 是层之间的线性堆叠. 可以允许我们来通过将层逐个堆叠来搭建神经网络.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

第一个卷积层

第一个卷积层有 32 个卷积核，每个尺寸为 3×3，来处理尺寸为 48×48 的输入图片．‘relu’激活函数引入了非线性，让模型可以捕获复杂的特征．

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

第二个卷积层

第二个卷积层有 64 个 3×3 的卷积核，该层进一步处理了来自前一层的特征图，检测了输入图片中的更复杂的特征.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

Max-Pooling 层

该层通过从每个 2×2 窗口中取最大值来对特征图进行降采样. 这减少了计算负担, 并帮助让模型平移不变.

初始化卷积层

```
emotion_model.add(
    Conv2D(32, kernel_size=(3, 3), activation="relu",
    ↪    input_shape=(48, 48, 1))
)
emotion_model.add(Conv2D(64, kernel_size=(3, 3),
 ↪    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

Dropout 层

Dropout 可以用来避免过拟合. 0.25 的 Dropout 率意味着大约 25% 的神经元在前面的层会被随机在训练中关闭, 使得可以获得容错率更高的模型.

多个卷积层

```
emotion_model.add(Conv2D(128, kernel_size=(3, 3),
    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Conv2D(128, kernel_size=(3, 3),
    activation="relu"))
emotion_model.add(MaxPooling2D(pool_size=(2, 2)))
emotion_model.add(Dropout(0.25))
```

更深的卷积层

后面的具有 128 个卷积核的卷积层每个都进一步细化了特征图. 随着模型变深, 这些层会捕获图片中更多的高层特征.

每个卷积层后面都有一个 Max-Pooling 层来降采样, 降低计算负担, 提高平移不变性.

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Flatten 扁平化层

该层将 2D 的特征矩阵映射到 1D 的向量. 在将数据传入全连接层之前, 这是必要的步骤;

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

稠密全连接层

具有 1024 个神经元的全连接层允许模型来基于由卷积层提取的高层次特征来进行决策. ‘ReLU’ 激活函数确保了非线性.

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

输出层

输出稠密层有 7 个神经元，对应在数据集中的七种表情分类．‘softmax’激活函数确保了输出值是和为 1 的概率．

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Softmax

Softmax 函数常常被用在基于神经网络的分类器的最后一层，将一个实数向量转化为一个概率分布.

在数学上，对一个向量 z 的 Softmax 函数 σ 可以被定义为

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{1}$$

其中，

- $\sigma(z)_i$ 是 z 的第 i^{th} 个分量的 Softmax 函数输出;
- K 是分类的数量 (也是向量 z 的长度);

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

Softmax

指数函数确保了输出向量所有的分量都是非负的，除法将所有的分量规范化，让他们的和为 1．结果向量因此可以表示 K 个类别的概率分布．

全连接层

```
emotion_model.add(Flatten())
emotion_model.add(Dense(1024, activation="relu"))
emotion_model.add(Dropout(0.25))
emotion_model.add(Dense(7, activation="softmax"))
```

为什么使用 Softmax

1. **概率解释** softmax 函数被用于分类模型的输出层，因为他提供了多个类之间的概率分布．这意味着对于给定的输入，模型可以提供输入可能属于哪个输出的类别;
2. **处理多个类** 在分类问题中，尤其是对于超过两个类别的情况，分类时需要不仅知道哪个类是最可能属于的，还想预测和其他类的似然成都．Softmax 可以通过将每个类的输出压缩在 0 和 1 之间，并确保他们的和为 1 来帮助实现这一目标．
3. **基于梯度的优化** Softmax 与类别交叉熵作为 loss 函数，提供了一个表现良好的梯度．在训练神经网络的反向传播阶段非常有用．没有良好的梯度，神经网络将不能高效的学习．

Cross-Entropy

Cross-entropy 是衡量两个概率分布之间的差异的方法. 在机器学习和深度学习中, 它常常被用作损失函数来量化预测概率分布和真实分布之间的差异.
对两个离散的概率分布 p 和 q, 交叉熵 $H(p,q)$ 被定义为

$$H(p,q) = -\sum_{x} p(x)\log\left(q(x)\right) \tag{1}$$

其中,

- $p(x)$ 是事件 x 发生的真实概率;
- $q(x)$ 是事件 x 发生的预测概率;
- 对于所有的可能事件 x 进行求和;

对于用于分类任务的神经网络, p 是真实标签的独热编码向量, q 是预测概率 (经常通过 softmax 函数取得).

为什么使用 Cross-entropy

1. **概率解释** Cross-entropy 提供了真实概率分布和预测概率分布之间的差异的度量. 较低的交叉熵值表明模型的预测与真实标签比较相似.
2. **可微分** 对于学习算法, 尤其是神经网络, 具有一个可微分的损失函数是至关重要的. 交叉熵可微分, 使得他可以被用于像梯度下降这样的基于梯度的优化方法.
3. **惩罚自信的错误预测** 交叉熵作为 loss 函数的一个优点是它如何处理预测. 如果一个模型对一个错误的类别预测 0.99 的概率, 对正确的类别预测 0.01 的概率, 交叉熵会变得非常大. 这个特征确保了当模型非常自信得犯错的时候, 会被严重惩罚;
4. **和 Softmax 配合良好**交叉熵损失结合输出层中的 softmax 激活函数是分类问题中流行的组合. 从这个组合中获得的梯度有良好的属性, 使得模型可以更快收敛, 并且经常相较于其他的激活函数-损失函数对具有更好的解决方案.
5. **解决饱和问题** 在神经网络中, sigmoid 激活函数和 quadratic cost 可以导致 ‘‘饱和’’ 问题, 此时神经元的输出和梯度都几乎是 0, 使得网络难以训练. 交叉熵不会受这个问题困扰, 使得训练更加高效.

模型训练

神经网络模型的结构确定后，下一个关键步骤就是在数据上对其进行训练.

1. **编译模型**
   - **损失函数** 模型使用 'categorical_crossentropy' 作为损失函数.
   - **优化器** 使用 'Adam' 作为优化器. Adam 对每个参数调节学习率. 其具有高效率与低内存需求. 'learning_rate' 战术决定了优化器最小化 loss 的步长, 'decay' 可以随时间降低学习率来允许在之后的训练过程中更细化的权重更新.
   - **矩阵** 'accuracy' 是在训练过程中监控的矩阵. 准确率提供了一个清晰的, 直观的对模型执行情况的理解: 表示正确分类的实例在总实例中占的百分比.

模型训练

神经网络模型的结构确定后，下一个关键步骤就是在数据上对其进行训练.

1. **编译模型**
2. **拟合模型**
   - **数据源** 模型借助 'fit_generator' 方法进行训练，该方法允许边读取边数据增强 (data augmentation on-the-fly)，且更加内存高效. 该方法当训练集太大而无法整个放入内存的时候更合适.
   - **轮数** 模型训练了 50 轮. 每轮代表一次对于所有训练样本的前向传播和反向传播.
   - **Batch Size** 每一轮的步数 ('28709 // 64') 表示数据每次被传入大小为 64 个实例的批次. 类似的，对于验证阶段，使用批次大小 '7178 // 64'. 使用批次加速了训练过程，因为模型权重的更新是在每一轮后进行的，而不是在每个数据点之后.
   - **验证数据** 模型的表现在一个单独的验证数据几上并行验证. 这帮助了监控任何的过拟合信号，当模型在训练数据上表现异常良好，而对于新的没有见过的数据非常糟糕的时候就发生了过拟合.

使用逻辑回归模型的贷款预测

借助 scikit-learn 的逻辑回归模型对是否可以给用户贷款进行预测. 该实验主要内容在于数据预处理.

贷款预测任务介绍

金融机构放贷需要评估借款人违约的风险，从而降低坏账损失，确保资金安全.
该任务通过逻辑回归来预测是否可以放贷.

二分类问题

借贷问题是一个二分类问题，目标是预测借款人是否会还款．输出通常是‘yes’或‘no’.

对该问题，在数据预处理上要使用特征工程来判断哪些信息结合可以作为比较良好的放贷参照．例如，将收入和负债比进行组合等.
对该问题使用逻辑回归，是因为逻辑回归不仅可以提供预测结果，还可以为每个预测提供概率评分．这有助于评估违约的风险程度.

贷款预测任务介绍

可解释性

与深度学习模型相比，逻辑回归的一个主要优势是其可解释性．在信贷审批过程中，理解为什么某个请求被拒绝或者批准是重要的，逻辑回归的系数可以提供有关哪些特征最影响预测的见解．

应对不平衡数据

信贷数据通常是不平衡的，因为违约的案例可能远远少于正常偿还的案例．因此需要使用特定的方法，例如过采样，欠采样等方法来解决．

数据集基本信息

数据来源

来源于 kaggle 社区的公开数据集 “Loan Prediction Problem”.

特征

数据集包含多个特征，例如申请人的性别，婚姻情况，教育程度，受雇状态，收入，贷款金额和信贷历史.

目标变量

‘Loan_Status’ 是目标变量，是二元变量. 其中 ‘Y’ 表示贷款获得批准，‘N’ 表示贷款没有获得批准.

数据规模

训练集大小为 614×13，测试集大小为 367×12. 其中测试集少了 ‘Loan_Status’ 一栏.

数据集特征

数据集有如下特征:

- Loan_ID: 一个单独的标识数字, 每个申请人独有;
- Gender: 性别 (男/女);
- Married: 申请人是否结婚 (Yes/No);
- Dependents: 申请人的家人数量, 0/1/2/3+.
- Education: 教育等级 (已毕业/未毕业);
- Self_Employed: 是否申请人是自雇人士 (是/否);
- ApplicantIncome: 申请人收入;
- CoapplicantIncome: 共同申请人收入 (如果没有共同申请人则为 0);
- LoanAmount: 申请的贷款金额 (千为单位);
- Loan_Amount_Term: 贷款期限 (月为单位);
- Credit_History: 借款人负责偿还债务的记录, 为 0 或 1, 表示申请人是否有过贷款偿还历史;
- Property_Area: 申请贷款的房产所在的区域类型 (城市/半城市/农村);

模型训练和预测

```
LR=LogisticRegression()
LR.fit(X_train,y_train)
y_hat=LR.predict(X_test)

# prediction summary by species
print(classification_report(y_test,y_hat))

# accuracy score
LR_SC=accuracy_score(y_hat,y_test)
print('accuracy is',accuracy_score(y_hat,y_test))
```

逻辑回归

'Logistic Regression' 是一个用来进行二元分类问题的统计方法，一个分类问题指预测一个特定数据属于两个中的哪一个类．被叫做 "logistic" 回归因为是基于逻辑函数的，这被用于建模将一个给定的输入点划入两个分类之一的概率．

训练模型

LR=LogisticRegression()
LR.fit(X_train,y_train)

- 首先，借助‘LogisticRegression()’来获取一个逻辑回归模型.
- 下一步，‘fit’方法将模型在训练数据上进行训练. 这里，‘X_train’是具有特征的训练数据，‘y_train’是训练数据对应的标签.

进行预测

y_hat=LR.predict(X_test)

在将模型在训练数据上训练后，下一步就是在新的没有见过的模型上进行预测. ‘predict’ 方法将测试数据 ‘X_test’ 作为参数，返回预测标签 ‘y_hat’.

分类报告

print(classification_report(y_test,y_hat))

'classification_report' 提供了模型表现的准确度，召回度和 F1-score.

- **Precision 准确度**在所有预测为某个特定类的例子中，多少是真阳性;
- **Recall 召回率**在所有属于一个特定类的实例中，哪些被正确预测了，即真阳性与真阴性的和;
- **F1-score** 准确度和召回率的调和平均值，提供了两个之间的平衡指标.

Precision 准确度

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{2}$$

当假阳性的代价高的时候, 准确度是关键的.

Recall 召回率

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{3}$$

当假阴性的代价高的时候, 召回率是重要的.

准确率分数

```
LR_SC=accuracy_score(y_hat,y_test)
print('accuracy is',accuracy_score(y_hat,y_test))
```

准确率分数是一个可以表明在测试数据中正确预测的分类的占比的量. 计算方式为将正确预测的数量除以预测的总数.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{4}$$

对于本处的贷款预测:

- 如果一个贷款请求倾向于被允许, 模型会预测 '1', 否则为 0.
- 准确率通过将预测结果与实际情况对比告诉我们多少的贷款请求被我们的模型正确预测.

模型局限

逻辑回归具有这样的局限:

线性决策边界 逻辑回归假设一个线性决策边界，这可能不能捕获数据中的复杂关系.

特征的独立性 逻辑回归假设输入特征是独立的，意味着他们不会互相影响. 如果一些特征是强相关的，这可能影响模型的参数.

- 改进模型

MNIST 手写数字识别

- 任务亮点
- 数据集基本信息
- 模型结构
- 模型训练

股票价格预测

- 任务介绍
- 数据介绍
- 模型结构
- 训练方法

使用机器学习方法预测从泰坦尼克灾难中生存的可能

- 问题定义和特点
- 数据特点

红酒质量预测

- 任务介绍
- 数据介绍
- 模型介绍
- 训练介绍

在 Boston Hosting 数据集上进行房价预测

借助 PyTorch 搭建全连接模型对 Boston Hosting 数据集进行房价预测.

任务介绍

波士顿房价预测是机器学习领域的经典问题，经常用于回归分析的入门实践.

- **经典的回归问题**波士顿房价预测是经典的监督学习中的回归问题，目标是预测波士顿地区的房屋中位数价格;
- **多元特征**数据集包含了多种可能影响房价的因素，例如犯罪率，居住年代，教师比例等，需要选取合适的因素用于预测，使得特征工程和特征选择变得很重要;
- **理解特征关系**由于特征之间可能存在相关性，例如某个地区的犯罪率可能会影响该地区的居住率，因此探索和理解这些特征之间的关系可以帮助更好地理解数据;
- **模型选择**回归分析可以使用的模型有许多．例如线性回归，决策树回归，随机森林回归，支持向量机等．本实验中使用全链接神经网络作为用于回归的模型;
- **评估指标**为了评估模型的性能，需要选取合适的评估指标．对于回归问题，常用的有均方误差 (MSE)，均方根误差 (RMSE) 等．本次实验使用 MSE.

数据基本信息

波士顿房价数据集 (Boston Housing Dataset) 是机器学习中非常经典的数据集, 来自 1970 年的美国人口普查数据, 主要用于研究房屋价格和各种因素之间的关系. 其详细信息如下.

数据基本信息

特征

- **CRIM** 城镇的人均犯罪率;
- **ZN** 大于 25,000 平方英尺的住宅用地比例;
- **INDUS** 非零售商业用地的比例 (亩);
- **CHAS** 查尔斯河虚拟变量 (如果地块与河流接壤为 1, 否则为 0);
- **NOX** 一氧化氮浓度 (每千万分之一);
- **RM** 每个住宅的平均房间数;
- **AGE** 1940 年以前建造的自住单位的比例;
- **DIS** 到波士顿五个就业中心的加权距离;
- **RAD** 高速公路通达性的指数;
- **TAX** 每 10,000 美元的全额物业税率;
- **PTRATIO** 城镇的学生与教师比例;
- **B** $1000(B_k - 0.63)^2$, 其中 B_k 是城镇中黑人的比例;
- **LSTAT** 人口中地位较低的人的百分比;

数据基本信息

目标变量

- **MEDV** 自住房的中位数价格 (单位: $1000 美元). 即希望预测的房屋价格.

数据集特点和挑战

- **数据规模** 数据集通常包含 506 个观测点, 其中每个观测点都有 13 个特征和 1 个目标变量;
- **缺失值** 波士顿房价数据集没有缺失值, 因此处理起来会相对方便;
- **异常值** 需要检查数据中的异常值或者离群值, 因为他们会影响模型的性能;
- **特征工程** 虽然数据集已经提供了 13 个特征, 但是可能需要进行特征选择或者特征工程来提供模型的性能.

模型结构

```
    # boston housing model
class BostonHousingModel(nn.Module):
    def __init__(self):
        super(BostonHousingModel,self).__init__()
        self.layer1=nn.Linear(13,64)
        self.layer2=nn.Linear(64,64)
        self.layer3=nn.Linear(64,1)

    def forward(self,x):
        x=torch.relu(self.layer1(x))
        x=torch.relu(self.layer2(x))
        x=self.layer3(x)
        return x
```

模型借助 PyTorch 框架, 使用 'nn.Module' 基类来定义神经网络结构.

模型初始化

第一层

全连接层，输入特征为 13，对应 Boston Housing dataset 中的 13 个特征，输出特征数量为 64.

第二层

全连接层，输入特征数量为 64，输出特征数量为 64.

第三层

全连接层，输入特征数量为 64，输出特征数量为 1，即表示预测房价结果.

前向传播

前向传播方法描述了输入数据是如何流过模型的.

- **第一步** 将输入 'x' 通过 'layer1', 应用 ReLU(Rectified Linear Unit) 激活函数. ReLu 函数引入非线性, 允许模型从错误中学习和调整, 对于学习复杂的特征至关重要;
- **第二步** 将前一步的输出传入 'layer2', 再次使用 ReLU 激活函数;
- **最后一步** 将结果传过 'layer3'. 在这一层没有激活函数, 因为这是一个回归问题, 模型被设计为直接输出连续值.

模型训练

- **损失函数** 使用 'nn.MSELoss()' 作为损失函数. 平均平方差损失计算了预测和实际值的平均平方差, 对于房价预测问题是合适的选择.
- **优化器选择** 选择 Adam 优化器来调节模型参数. 学习率为 '0.001'.
- **训练轮数** 训练 500 轮, 并记录日志.

改进模型

在前面基础模型的基础上，使用 Dropout 进行改进．在训练策略上，使用学习率规划下降和提前停止来优化训练．

```
class OptiBostonHousingModel(nn.Module):
    def __init__(self):
        super(OptiBostonHousingModel,self).__init__()
        self.layer1=nn.Linear(13,128)
        self.layer2=nn.Linear(128,64)
        self.dropout=nn.Dropout(0.5)
        self.layer3=nn.Linear(64,1)

    def forward(self,x):
        x=torch.relu(self.layer1(x))
        x=self.dropout(x)
        x=torch.relu(self.layer2(x))
        x=self.dropout(x)
        x=self.layer3(x)
        return x
```

模型结构

仍旧是三层全连接层，但是第二层和第三层之间使用了 Dropout 率为 0.5 的 dropout 层，随机丢弃一半输入数据来帮助避免过拟合.

学习率规划器

使用 'StepLR' 规划器来在训练过程中调节学习率. 特别的，学习率每 100 轮被乘以一个参数 'gamma'(此处为 0.1). 随时间降低学习率可以帮助细化模型的权重，使其更加贴近最优方案.

CNN 进行 MNIST 手写数字识别

借助 PyTorch 框架, 定义一个 CNN 进行 MNIST 手写数字识别.

任务亮点

- **手写数字识别** MNIST 数据集包括了从 0 到 9 的手写数字，是机器学习和计算机视觉领域的一个经典的数据集．这个任务的目标是将每一个手写数字图像正确分类；
- **图片数据处理** 数据是图像形式的，因此这一任务为初学者提供了处理图片数据的经验，例如归一化，数据增强等；
- **学习 CNN** 在本实验中，将使用卷积神经网络 (CNN) 来处理这个任务，对 CNN 进行了解和学习；
- **模型评估** 了解和使用正确的评估指标 (例如准确度，召回率，F1-Score 等) 对模型进行评估，并使用验证集进行调参；
- **模型优化** 了解和实践如何通过各种方法优化模型，例如更深的网络，Dropout 正则化，数据增强等；

数据集基本信息

- **图像信息**
  - 每张图像都是一个手写数字，从 0 到 9;
  - 图像是单通道的灰度图，相比于彩色图，少了许多的混淆信息，降低了处理难度;
- **图像尺寸** 每张图像的尺寸是 28×28 像素;
- **数据规模**
  - 数据集包含 70,00 张图像和对应的标签;
  - 通常被分为 60,000 张训练图像和 10,000 张测试图像;

数据集基本信息

- **标签**
  - 对每张图像，有一个对应的标签；
  - 每个标签都是 0 到 9 之间的整数，对应了图像中的手写数字；
- **数据来源** 图像来自美国高中生和人口普查局的雇员手写并收集；
- **数据格式**
  - 图像数据的像素值在 0 到 255 之间，但是训练神经网络的时候，会归一化到 0 到 1 范围；
  - 标签是单个数字，对应图像中的手写数字；
- **数据挑战性** MNIST 虽然是一个相对简单的数据集，然是仍然包含了各种手写风格，有的数字相比于其他的数字更加模糊或者更加难以识别；

模型结构

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()

        self.conv_layers = nn.Sequential(
            # (n, 1, 28, 28) -> (n,32,28,28)
            nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            # (n,32,28,28)->(n,32,14,14)
            nn.MaxPool2d(kernel_size=2, stride=2),
            # (n,32,14,14)->(n,64,14,14)
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            # (n,64,14,14)->(n,64,7,7)
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
```

模型结构

```
        self.fc_layers = nn.Sequential(
            # (n,64*7*7)->(n,128)
            nn.Linear(64 * 7 * 7, 128),
            nn.ReLU(),
            # (n,128)->(n,10)
            nn.Linear(128, 10),
        )

    def forward(self, x):
        x = self.conv_layers(x)
        x = x.view(x.size(0), -1)  # flatten the tensor
        x = self.fc_layers(x)
        return x
```

模型结构

1. **卷积层 (‘conv_layers’)** 这些层用来处理输入图像的空间信息，检测局部图案，边缘，纹理等．通过级联卷积和池化操作以分层方式转换图像．

   1.1 **第一卷积层**

   - **卷积** 这一层使用单个通道 (因为 MNIST 图片为灰度图片)，将其转化为 32 个通道，借助 3×3 卷积核．大小为 1 的填充确保了输出和输入的宽和高相同，因此输出形状为 $(n, 32, 28, 28)$.
   - **激活函数** ‘ReLU’ 激活函数引入了非线性．
   - **池化** 之后的 ‘MaxPool2d’ 操作使用了步长为 2 的池化核大小为 2×2 的池化操作来降低空间维度大小到原来的一半，使得输出形状为 $(n, 32, 14, 14)$.

   1.2 **第二卷积层**

   - 卷积：将前一层的 32 个通道借助 3×3 的卷积核转化为 64 通道，由于填充为 1，保留了原本的维度．
   - 激活函数：ReLU.
   - 池化：另一个 ‘MaxPool2d’ 操作将其维度降低到原来的一半，输出形状为 $(n, 64, 7, 7)$.

模型结构

2. 全连接层（'fc_layers'）

- 这些层分析卷积层提取和学到的特征，将其分类到 10 个可能的数字类中.

2.1 第一个全连接层

- **扁平化**在将其从卷积层传入全连接层之前，将 3D 张量 $(64, 7, 7)$ 扁平化到 1D 的 3136 个元素的张量.
- **线性变形** 将 3136 个节点降低到 128 个节点.
- **激活函数** 再次使用 'ReLU' 函数来引入非线性.

2.2 第二个全连接层

- 使用了 128 个来自前一层的节点，将其减少到 10 个节点，对应 10 个 MNIST 中可能的数字类 (0 到 9).

模型训练

接下来介绍对模型的训练. 首先将模型和数据移动到 CUDA GPU 上, 然后对模型进行训练.

```
device = torch.device("cuda" if torch.cuda.is_available else "cpu")
# initialize the model
model = CNN().to(device)
```

首先判断 cuda 是否可用, 如果可用则使用其进行加速. 否则在 cpu 上进行训练.

损失和优化器配置

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- Loss 函数使用交叉熵，对于分类任务这是常用的；
- 使用 Adam 优化器，以及 0.001 的学习率来调节模型参数；

训练循环

```
num_epochs = 10
for epoch in range(num_epochs):  # loop over the dataset multiple
↪    times
    running_loss = 0.0
    model.train()  # set model to train mode
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()

        # forward, backward, optimize
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    print(f"Epoch {epoch+1},loss: {running_loss/len(trainloader)}")
```

训练循环

- 模型进行 10 轮训练;
- 在数据集上迭代之前, ‘running_loss’ 被设置为 0 来累加一轮的损失.
- 借助 ‘model.train()’ 将模型设置到训练模式. 该模式影响 dropout 或者 batch normalization 层这种在训练时和测试时表现不同的层.
- 对来自 ‘trainloader’ 数据的每个批次 (‘inputs’ 和对应的 ‘labels’):
  - 数据被转移到配置的装置 (CUDA 或 CPU);
  - 梯度被清零来确保没有来自前一次迭代的累加;
  - 一个前向过程计算模型的预测 (‘outputs’);
  - loss 被在预测和真实标签之间计算出来;
  - 一个反向传播过程计算 loss 与对应的模型参数的梯度值;
  - 优化器基于计算的梯度更新模型的参数;
  - 该批次的 loss 被累加到 ‘running_loss’;
- 在所有的批次被处理后, 该轮次的平均 loss 被打印.

训练循环的验证阶段

```
...
model.eval()  # set the model to evaluation mode
with torch.no_grad():
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0
    for inputs, labels in validloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        running_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total_predictions += labels.size(0)
        correct_predictions += (predicted == labels).sum().item()
    print(
        f"Validation loss: {running_loss/len(validloader)}, Validation
        ↪  accuracy: {correct_predictions/total_predictions*100}%"
    )
```

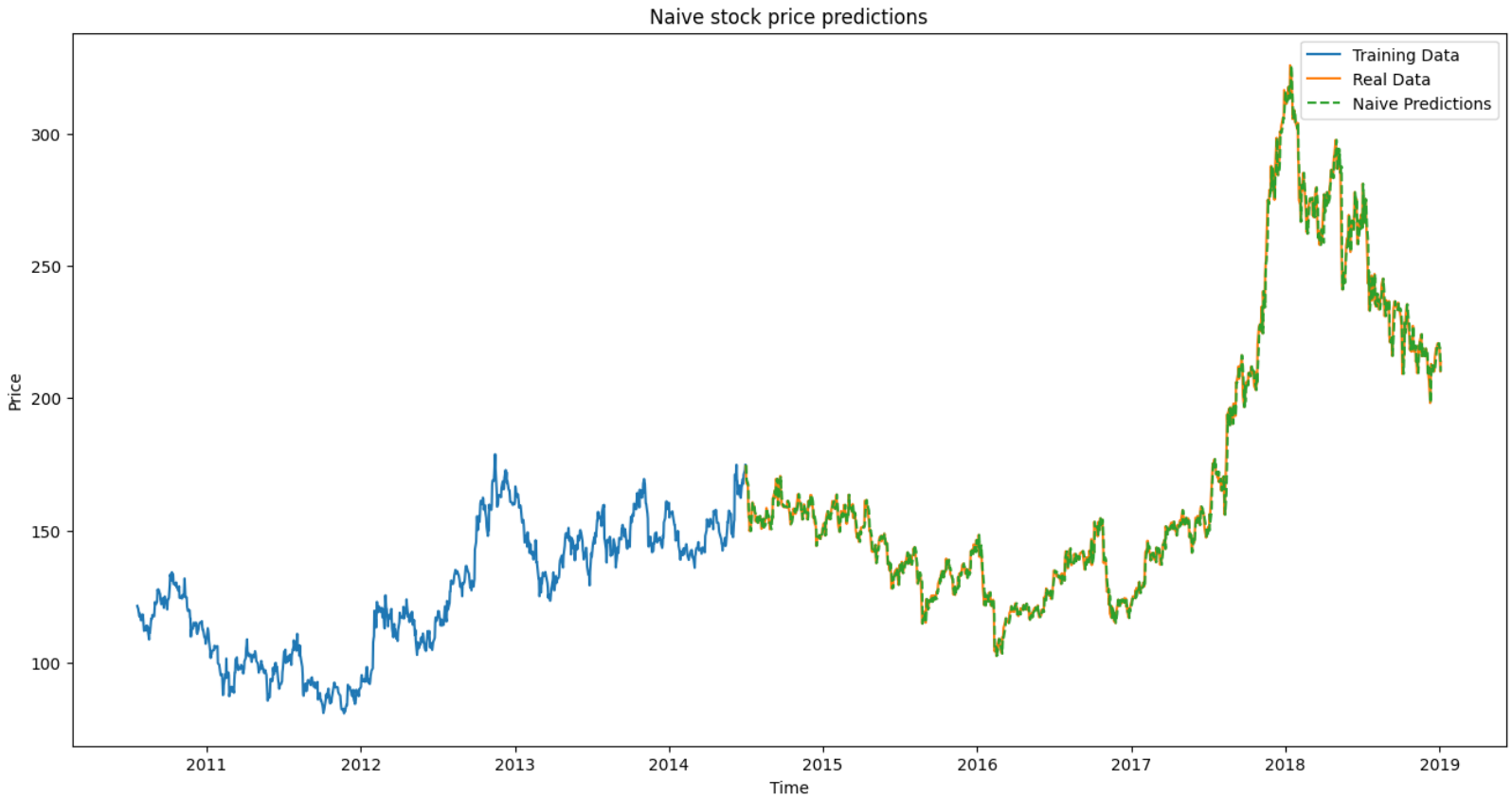
训练循环的验证阶段

- 模型被借助 ‘model.eval()’ 设置为验证模式. 这确保了 dropout 和 batch normalization 等层工作在验证模式;
- ‘torch.no_grad()’ 确保了在这一阶段不会计算梯度, 减少了内存消耗;
- 对来自 ‘validloader’ 的每一批次数据:
  - 数据被转移到 device 上;
  - 一个前向传播过程计算了模型预测;
  - loss 被计算并添加到 ‘running_loss’;
  - 预测值被与真实标签比较来确定准确度.
- 在所有的批次被处理完之后, 平均的验证损失和准确率都被打印;

股票价格预测

使用 LSTM 借助历史收盘股价进行股票价格预测. 实际上如果想要作为投资参考, 对股票的收益率或者涨跌进行预测是更好的方式, 对股票价格进行预测仅仅用来学习.

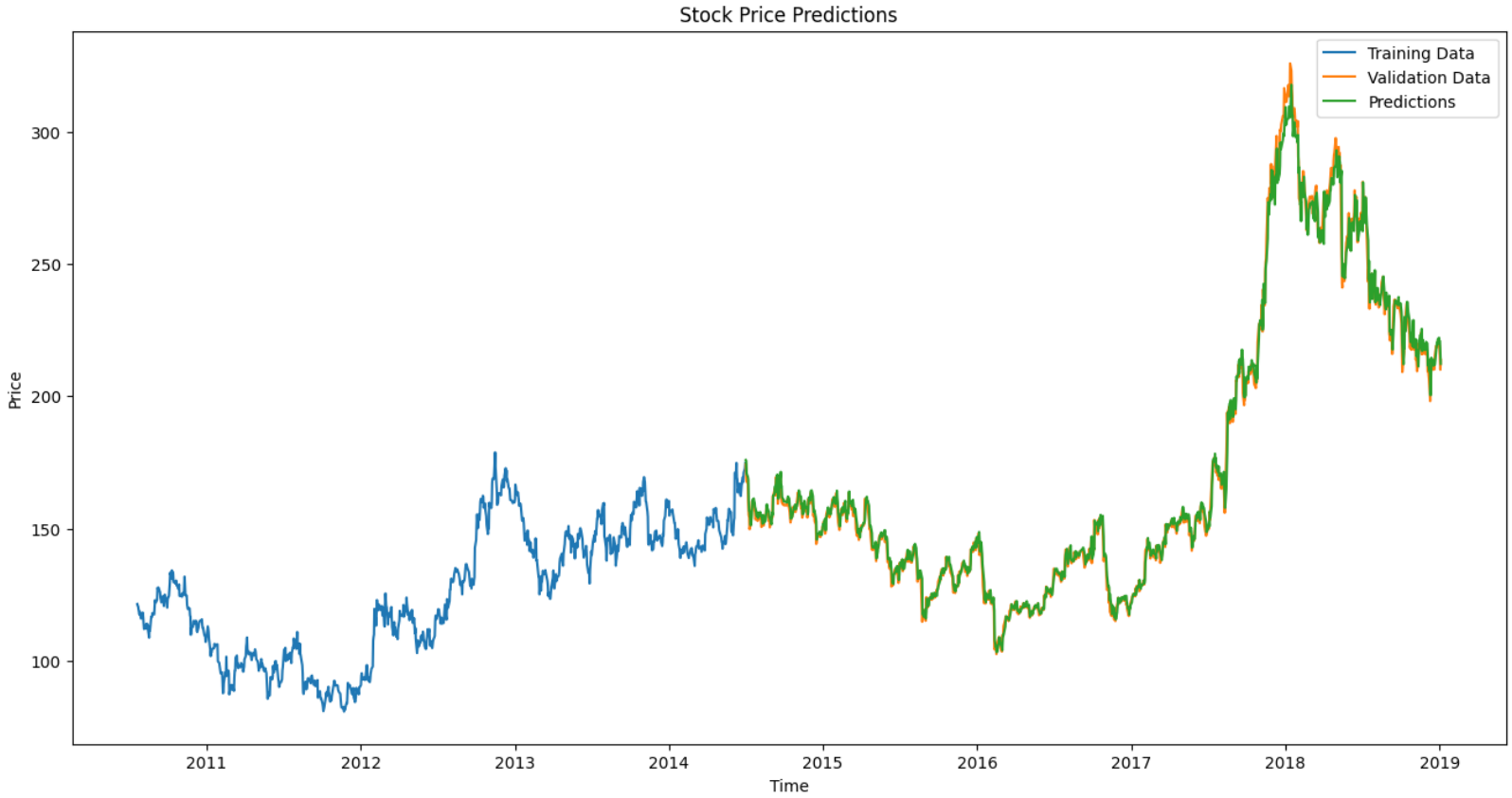
Figure: 如图, 仅仅使用昨天的数据作为今天测预测值也可以获得看起来良好的预测结果, 但是实际上是没有用的



股票价格预测

使用 LSTM 借助历史收盘股价进行股票价格预测. 实际上如果想要作为投资参考, 对股票的收益率或者涨跌进行预测是更好的方式, 对股票价格进行预测仅仅用来学习.

Figure: 如图为 LSTM 的预测结果



任务亮点介绍

- **时间序列预测** 股价预测是时间序列分析的一个典型应用，旨在基于过去的预测数据预测未来的价格走势；
- **利用 LSTM 的长期记忆** LSTM 是一种特殊的循环神经网络 (RNN)，可以捕捉长期的依赖关系．这对于捕捉和记忆股票市场中的长期趋势和周期性非常有用；
- **复杂市场动态** 股票价格受到多种因素影响，包括全球事件，公司新闻和其他宏观经济指标．使用 LSTM 可以帮助模型捕捉这些复杂的关系；

数据介绍

该任务使用印度的 TATAGLOBAL 股价数据集. 该数据集专门针对印度的 Tata Global Beverages Ltd. 公司, 包含了公司过去的股价数据. 其特点如下.

数据集介绍

数据集有如下的信息，借助这些信息来预测未来的股价走势；

- **日期 (Date)** 每个条目都有一个对应的日期，标记了那一天的股票交易情况；
- **开盘价 (Open)** 是股票在特定交易日开始时的价格；
- **最高价 (High) 和最低价 (Low)** 提供了股票在交易日中的最高和最低价格；
- **收盘价 (Close)** 是股票在交易日结束的时候的价格，可认为是股票当天的最终或‘‘官方’’股价；
- **最后交易价格 (Last)** 表示交易日的最后一笔交易的价格．大多数情况下和收盘价接近；
- **总交易量 (Total Trade Quantity)** 表示在特定交易日内交易的股票总数量．高交易量意味着股票有较大的价格变动，低的交易量表示股票价格稳定；
- **换手率 (Turnover in Lacs)** 是当天的总交易额，表示以卢比计价的总价值．

模型结构

```
lstm_model = Sequential()
lstm_model.add(
    LSTM(units=50, return_sequences=True,
    ↪    input_shape=(X_train.shape[1], 1))
)
lstm_model.add(Dropout(0.2))
lstm_model.add(LSTM(units=50))
lstm_model.add(Dropout(0.2))
lstm_model.add(Dense(1))

inputs = new_dataset[len(new_dataset) - len(valid_data) - 60 :].values
inputs = inputs.reshape(-1, 1)
inputs = scaler.transform(inputs)
print(inputs.shape)
```

模型结构

```
lstm_model.compile(loss="mean_squared_error", optimizer="adam")

# patience is the number of epochs to check for improvement
early_stop=EarlyStopping(monitor='val_loss', patience=10,
    verbose=1)

lstm_model.fit(X_train, y_train, epochs=100, batch_size=1, verbose=2,
    validation_split=0.2, callbacks=[early_stop])
```

模型结构

在本实验 vs, 借助 Keras 搭建了一个 LSTM 神经网络. LSTM 是为顺序数据特别设计的, 对于时序数据有良好预测性能, 且相比于 RNN 有效解决了长期记忆衰退问题的模型. 对其结构介绍如下.

输入层

模型期望输入数据有形状*(X_train.shape[1], 1)*. 这意味着每个训练样本应该是一个具有 *X_train.shape[1]*步的序列, 且在每一步只有一个特征.

模型结构

第一个 LSTM 层

- 有 50 个神经元;
- 在 tensorflow 中, 设置 ***return_sequences=True*** 意味着这个 LSTM 层会将整个序列传递给下一层, 对于堆栈的 LSTM 层是必要的.

模型结构

关于 *return_sequences*

LSTM 层中的 ***return_sequences*** 控制了 LSTM 输出的是序列还是整个序列的最后一个输出.

设置 ***return_sequences*** 为 'True' 或者 'False' 有特定的使用场景:

1. ***return_sequences=True***
   - LSTM 层会为输入序列中的每个时间步都产生一个输出, 意味着输出会和输入有相同的长度;
   - 将 LSTM 层堆叠的场景下这是必须的. 因为下一层需要和上一层期望相同长度的序列作为输入.
   - 如果进行的是 seq2seq 的预测, 则也应该使用这种方式.
2. ***return_sequences=False***
   - LSTM 层会对整个输入序列产生一个单独的输出值. 这个只对应了最后一个时间步;
   - 当仅仅对于最后的预测结果感兴趣的时候, 这是合适的. 在许多 seq2val 的预测中采用这种方式.
   - 在一个全连接层之前, 通常也这样使用, 尤其是在分类或回归任务的模型中.

模型结构

第一个 Dropout 层

该层随机设置 20% 的输入单元为 0, 这可以帮助避免过拟合.

为什么在 LSTM 层之间使用 Dropout 层

- Dropout 层放在 LSTM 层之间的时候, 即对 LSTM 层的输出进行了 dropout, 在训练时对其中的部分值随机设置为 0.
- Dropout 可以有效避免过拟合, 尤其是在较小的数据集情况下 (如本实验的数据集).
- 如果模型本身没有过拟合, 添加 dropout 不仅可能不会提供帮助, 反而可能降低模型的表现. 因此, 需要监控验证表现并进行及时的调整.

模型结构

使用 Dropout 可能的缺点

- 在 RNN 中使用 dropout, 尤其是通过 ***recurrent_dropout***, 会延长训练时间, 因为在 Keras 中对于 ***recurrent_dropout*** 的实现关闭了对于 LSTM 层的一些性能优化方法;
- 可能引入了过多的 dropout, 这可能降低模型的学习能力. 在规范化模型和允许他从数据中学习之间应该取得一个平衡.

模型结构

第二个 LSTM 层

- 具有 50 个单元;
- 因为后面没有后继的 LSTM 层, 因此 ***return_sequences*** 没有设置, 默认为 False. 这意味着该层只会输出最后一个时间步的值.

第二个 Dropout 层

仍旧 20%dropout 率的 Dropout 层, 用来避免过拟合;

输出层

全连接层, 仅有一个神经元. 因为这里没有设置激活函数, 默认是线性激活函数, 使其适合回归任务.

模型编译和训练

编译

- 模型使用平均平方差 loss 进行编译，这对于回归任务是常用的选择．使用的优化器是 'Adam'，对于许多深度学习任务这都是一个流行且高效的优化器．

提前停止回调

该回调函数监控了验证损失 (val_loss)．如果在 10 轮训练中验证 loss 没有提升 $(patience=10)$，训练会停止．这可以帮助避免过拟合，并且如果模型已经收敛，可以帮助减少训练时间;

模型训练

- 模型训练了最多 100 轮，批次大小为 1．意味着模型在每个样本之后都会更新参数．
- 20% 的数据用作验证数据．$(validation\_split=0.2)$．
- 使用了提前停止回调函数．

泰坦尼克

根据穿上人员的性别，年龄，职业等信息，设计一种算法来预测泰坦尼克号上乘客的生存概率．借助该问题，学习使用‘scikit-learn’库和 xgboost 等库的各种机器学习方法，例如 SVM 支持向量机，决策树，线性模型，朴素贝叶斯，高斯过程和决策树等．

问题定义和特点

问题描述

根据船上人员的各种特征，如性别、年龄、船票等级（Pclass）、上船地点（Embarked）、家庭成员数量、船票费用（Fare）以及其他相关信息，设计一种算法来预测泰坦尼克号上的乘客的生存概率。

问题定义和特点

问题的特点

对泰坦尼克生存预测任务中，其问题的关键特点分析如下:

1. **数据的复杂性与不完整性** - 数据集中存在大量缺失值，尤其是在'Age'和'Cabin'特征中。处理这些缺失值需要深思熟虑，因为简单的删除或填充可能会导致信息损失或引入偏见。
   - 存在一些杂乱的数据，例如在'Ticket'和'Name'特征中，这要求进行精细的数据清洗和特征工程。
2. **数据的不均衡性** - 大部分乘客在事故中未能生存。因此，我们面临的是一个不平衡分类问题，这可能会影响模型的评估指标和其对于生存和非生存类别的敏感性。
3. **特征的多样性** - 数据集中有数值、分类和文本特征。这意味着我们可能需要使用多种预处理技术来准备数据供模型使用。
   - 某些特征，如'Pclass'和'Sex'，与生存率高度相关。但是，其他特征可能需要更深入的探索和工程化才能发挥其潜在价值。

问题定义和特点

问题的特点

对泰坦尼克生存预测任务中，其问题的关键特点分析如下:

4. **社会经济背景的影响** - 泰坦尼克号的生存率与社会阶层、性别和年龄等因素密切相关。例如，女性和儿童的生存率较高，而一等舱的乘客也有更高的生存机会。
   - 这提示我们，在特征工程和模型建立时，应考虑这些社会和经济背景。
5. **关于家庭的信息** - 'SibSp' 和'Parch' 两个特征提供了关于乘客家庭成员的信息。考虑到家庭成员可能一同行动（如逃生），这些特征可能与生存率有关。
   - 通过这些特征，可以派生出新的特征，如家庭大小或是否独自一人。
6. **隐藏信息的挖掘** - 例如，通过'Name' 特征，可以提取出乘客的头衔，如 'Mr', 'Dr'等. 这与其社会地位、年龄和职业等相关，并间接影响其生存率。

数据介绍

数据介绍

数据中每个乘客有 8 个属性，分别如下:

1. ‘Survived‘代表乘客是否存活
2. ‘PassengerID‘和 ‘Ticket‘被假设为随机独立标识符，对输出没有影响，因此会被从分析中移除
3. ‘Pclass‘代表票型，并映射社会经济状态，表示 1 = 上层阶级，2 = 中层阶级，3 = 下层阶级;
4. ‘Name‘是名字数据类型，可能可以在特征工程中根据 title 判断性别，从 surname 中家庭大小.
8. ‘Cabin‘变量是命名数据类型，可以在特征工程中大致定位事故发生时在船上的位置，以及根据等级判断接机. 然而，由于有许多 Null 值，该变量用处不大，被排除在分析之外;

数据介绍

数据介绍

数据中每个乘客有 8 个属性，分别如下：

5. ‘Sex‘和 ‘Embarked‘变量是命名数据类型．会被转为 dummy 变量来进行数学计算．
6. ‘Age‘和 ‘Fare‘变量是连续量化数据类型；
7. ‘SibSp‘表示同在船上的兄弟姐妹的数量，‘Parch‘表示同在船上的父母孩子．都是离散量化数据类型．可以在特征工程中建立一个家庭大小，是孤立的变量；
8. ‘Cabin‘变量是命名数据类型，可以在特征工程中大致定位事故发生时在船上的位置，以及根据等级判断接机．然而，由于有许多 Null 值，该变量用处不大，被排除在分析之外；

使用的机器学习模型介绍

```
    # machine learning algorithms (MLA) selection and Initialization
MLA = [
    # ensemble methods
    ensemble.AdaBoostClassifier(),
    ensemble.BaggingClassifier(),
    ensemble.ExtraTreesClassifier(),
    ensemble.GradientBoostingClassifier(),
    ensemble.RandomForestClassifier(),
    # Gaussian Processes
    gaussian_process.GaussianProcessClassifier(),
    # GLM
    linear_model.LogisticRegressionCV(),
    linear_model.PassiveAggressiveClassifier(),
    linear_model.RidgeClassifierCV(),
    linear_model.SGDClassifier(),
    linear_model.Perceptron(),
```

使用的机器学习模型介绍

```
    # Naive Bayes
    naive_bayes.BernoulliNB(),
    naive_bayes.GaussianNB(),
    # Nearest Neighbor
    neighbors.KNeighborsClassifier(),
    # SVM
    svm.SVC(probability=True),
    svm.NuSVC(probability=True),
    svm.LinearSVC(),
    # Trees
    tree.DecisionTreeClassifier(),
    tree.ExtraTreeClassifier(),
    # Discriminant Analysis
    discriminant_analysis.LinearDiscriminantAnalysis(),
    discriminant_analysis.QuadraticDiscriminantAnalysis(),
    # XGBoost
    XGBClassifier(),
]
```

使用的机器学习模型介绍

集成方法 (Ensmble methods): 这些方法通常集合了多个模型 (常常称为 ‘‘基础学习者”) 来提升泛化能力和鲁棒性

- AdaBoostClassifier: 一个拟合一系列的弱学习者 (通常是决策树) 的自适应增强算法. 每棵决策树都会纠正其前辈的错误.
- BaggingClassifier: 使用 bagging (Bootstrap Aggregating) 来在数据集的随机子集上拟合一个分类器的多个实例. 子集是通过替换获取的.
- ExtraTreesClassifier: 代表 “Extremely Randomized Trees”. 像一个更加随机版本的随机森林, 其中最好的划分从一个随机的子集和阈值中选取;
- GradientBoostingClassifier: 顺序添加预测器来纠正由前面的预测器做出的错误. 和 AdaBoost 不同, 该分类器专注于直接减少残差.
- RandomForestClassifier: 在训练中建立一个决策树 ‘森林’. 每棵树都在数据和特征的一个随机子集上训练. 预测被整合 (对于回归任务取平均, 对于分类任务取多数投票).

使用的机器学习模型介绍

高斯过程

- GaussianProcessClassifier：一个用于回归和概率分类的非参数贝叶斯方法．他在函数上定义一个分布，借助核技巧在高维空间中进行操作．

泛化线性模型

- LogisticRegressionCV：具有用于自动参数调整的集成交叉验证的逻辑回归 (用于二元分类)．
- PassiveAggressiveClassifier：一种适合大规模学习的在线学习算法，被称为 ‘‘被动攻击”，因为在正确分类时保持被动，并在发生错误时变得攻击．
- RidgeClassifierCV：使用 L2 正则化的线性分类器 (对系数的平方幅度添加惩罚)．该版本包含集成的交叉验证．
- SGDClassifier：线性分类器 (像 SVM 和逻辑回归)，具有随机梯度下降 (SGD) 训练的线性分类器．
- Perceptron：线性分类器的一种，是一种监督二元分类算法．

使用的机器学习模型介绍

朴素贝叶斯

基于应用贝叶斯理论以及特征之间的强独立性假设.

- BernoulliNB: 二元数据使用. 假设特征是二元的.
- GaussianNB: 假设特征有高斯分布.

最近邻居

- KNeighborsClassifier: 基于训练集中他的 'k' 个最近的邻居来分类一个样本.

支持向量机 (SVM)

旨在找到最能将数据集划分为类别的超平面;

- SVC: 使用核技巧来变形输入数据, 然后识别最优的超平面;
- NuSVC: 和 SVC 类似, 但是使用一个参数 'nu' 来控制支持向量的数量.
- LinearSVC: 不是用核技巧的线性 SVM.

使用的机器学习模型介绍

决策树

- DecisionTreeClassifier：构造一棵树，其中每个节点都根据特征阈值作出决策．分割是根据熵或者基尼杂质 (Gini impurity) 等依据做出的；
- ExtraTreeClassifier：类似决策树，但是划分的更加随机．

判别分析：用于分类和降维

- LinearDiscriminantAnalysis，线性判别分析 (LDA)：假设每个类具有高斯分布，并共享相同的协方差矩阵；
- QuadraticDiscriminantAnalysis，二次判别分析 (QDA)：与 LDA 类似，但是允许每个类都有其协方差矩阵；

XGBoost

- XGBClassifier：优化的梯度增强库．代表 eXtreme Gradient Boosting．以其速度和性能闻名．

使用的机器学习模型介绍

```
cv_split = model_selection.ShuffleSplit(
    n_splits=10, test_size=0.3, train_size=0.6, random_state=0
)
# create table to compare MLA metrics
MLA_columns = [
    "MLA Name",
    "MLA Paramaters",
    "MLA Train Accuracy Mean",
    "MLA Test Accuracy Mean",
    "MLA Test Accuracy 3*STD",
    "MLA Time",
]
MLA_compare = pd.DataFrame(columns=MLA_columns)
# create table to compare MLA predictions
MLA_predict = data1[Target]
```

训练并进行预测

```
# index through MLA and save performance to table
row_index = 0
for alg in MLA:
    # set name and parameters
    MLA_name = alg.__class__.__name__
    MLA_compare.loc[row_index, "MLA Name"] = MLA_name
    MLA_compare.loc[row_index, "MLA Parameters"] =
     ↪    str(alg.get_params())

    # score model with cross validation
    cv_results = model_selection.cross_validate(
        alg,
        data1[data1_x_bin],
        data1[Target],
        cv=cv_split,
        return_train_score=True,
    )
```

训练并进行预测

```
MLA_compare.loc[row_index, "MLA Time"] =
    cv_results["fit_time"].mean()
MLA_compare.loc[row_index, "MLA Train Accuracy Mean"] =
    cv_results[
  "train_score"
].mean()
MLA_compare.loc[row_index, "MLA Test Accuracy Mean"] =
    cv_results[
  "test_score"
].mean()
# if this is a non-bias random sample, then +/-3 standard
    deviations(std)
# from the mean, should statistically capture 99.7% of the subsets
MLA_compare.loc[row_index, "MLA Test Accuracy 3*STD"] = (
   cv_results["test_score"].std() * 3
)  # the worst that can happen

# save MLA predictions
```

训练并进行预测

交叉验证划分

借助 'ShuffleSplit' 方法订一个 'cv_split' 这个交叉验证策略. 策略创建 10 个划分, 将数据中的 60% 用于训练, 30% 用于测试. 该方法不是用整个数据集, 因为每个划分中有 10% 的数据没有被使用.

比较表

两个 Pandas DataFrames('MLA_conpare' 和 'MLA_predict') 被初始化来存储每个算法的性能指标和预测.

算法循环

在 'MLA' 列表中的每个算法都被循环经过, 不同的任务被执行:

- 算法名字和参数被保存;
- 算法在数据集上使用前面提到的划分策略进行交叉验证. 计算并存储训练准确度, 测试准确度, 拟合时间等指标.
- 算法在数据的一个子集上进行训练, 在同一个子集上进行预测. 这些预测会被保存, 在之后可视化训练结果的时候使用.

训练并进行预测

结果

'MLA_compare'DataFrame 保存了性能指标，基于测试准确度降序进行排序. 这使得找出表现最好的算法十分简单.

红酒质量预测任务

酒质量预测任务旨在基于物理化学测试的结果来预测红酒和白酒的质量。在该任务中，使用一系列机器学习模型进行预测，模型通过分析红酒或白酒的一系列化学属性来预测其质量。预测的结果通常为 0-10 的得分，代表酒的整体质量。

任务介绍

对该任务，着重于如下方面.

- **模型的比较** 由于数据集具有多个特征，因此使用多种机器学习算法（例如决策树，支持向量机，神经网络等）进行对比，找出表现最好的模型;
- **特征工程**
  - **特征选择的重要性** 葡萄酒的质量可能只与某些关键特征密切相关。通过特征选择，可以将不太相关的特征排除，使模型更加集中地学习关键特征，从而提高预测的准确性。
  - **特征转换与洞察** 某些原始特征可能需要经过转换或组合才能更好地用于模型训练，例如，某两种化学成分的比例可能比它们各自的数值更有预测价值。此外，通过特征工程，还可以深入挖掘和理解这些化学成分如何共同影响葡萄酒的质量。

数据集介绍

任务使用 kaggle 上的 Wine Quality 数据集，其每个样本具有如下特征.

- **固定酸度 (Fixed acidity)** 主要是葡萄酒中的有机酸，如柠檬酸，苹果酸等;
- **挥发性酸度 (Volatile acidity)** 主要是酸和乙醇的量。高的挥发性酸度可能会导致醋酸味。
- **柠檬酸 (Citric acid)** 可以为酒增加新鲜感和风味。
- **残糖 (Residual sugar)** 酒发酵后所剩余的糖分。
- **氯化物 (Chlorides)** 酒中的盐含量。
- **游离二氧化硫 (Free sulfur dioxide)** 游离形式的 SO2。预防微生物生长和葡萄酒氧化的剂量过高可能导致不良的味道。
- **总二氧化硫 (Total sulfur dioxide)** 总的 SO2 量，是游离和结合状态的 SO2 的总和。
- **密度 (Density)** 可以用来估计酒精含量。
- **pH** 表示葡萄酒的酸度或碱度。大多数葡萄酒的 pH 值在 3-4 之间。
- **硫酸盐 (Sulphates)** 一个增加游离二氧化硫的酒添加剂，可能会影响葡萄酒的微生物生长和酒的味道。
- **酒精度 (Alcohol)** 酒的酒精含量。

模型介绍

对该任务，分别使用了多种传统机器学习模型，Optuna 超参数优化的 XGBoost 和神经网络进行预测．下面首先介绍使用的传统机器学习模型．

传统机器学习模型介绍

传统机器学习模型

首先是传统的机器学习模型. 在这一部分, 使用了如下的机器学习模型进行预测:

- **逻辑回归 (Logistic Regression)** 逻辑回归是一种统计方法，主要用于预测某个事件发生的概率。虽然它的名称中包含“回归”，但逻辑回归通常用于分类问题，如二分类。它使用了 Sigmoid 函数来预测概率，该函数输出的值位于 0 到 1 之间。
- **K 最近邻 (K Nearest neighbors, KNN)** KNN 是一个基于实例的学习方法。对于一个新的输入数据，KNN 会在训练数据中查找最接近的 k 个数据点（邻居），并基于这些邻居的分类来预测新数据的分类。
- **支持向量机 (Support Vector Machines, SVM)**SVM 是一种分类和回归方法，它试图找到一个超平面来正确分类数据。对于非线性数据，SVM 使用核函数如径向基函数来转换数据，然后找到一个在新的空间中的超平面来分类数据。
- **决策树 (Decision Trees)** 决策树是一个流程图式结构，其中每个内部节点代表一个属性上的判断，每个分支代表一个判断结果，最终的每个叶节点代表一个决策结果。决策树能够进行可视化，直观地显示决策过程。

传统机器学习模型介绍

传统机器学习模型

首先是传统的机器学习模型. 在这一部分, 使用了如下的机器学习模型进行预测:

- **随机森林 (Random Forest)** 随机森林是一个集成学习方法，它创建多个决策树并进行组合，以提高整体的性能和准确性。每棵树在略微不同的数据子集上进行训练。
- **梯度提升树 (Gradient Boosting Trees)** 这是一个迭代的决策树算法，其中每一次的迭代都试图纠正前一次迭代的错误。每一步都在增加一个新的树，以减少整体的误差。
- **AdaBoost(Adaptive Boosting)**Adaboost 是一个自适应的提升方法，它结合了多个弱学习者来创建一个强学习者。每一个学习者都是在修正前一个学习者的错误上进行训练的。

传统机器学习模型介绍

传统机器学习模型

首先是传统的机器学习模型. 在这一部分, 使用了如下的机器学习模型进行预测:

- **CatBoost**CatBoost 是一种梯度提升算法，特别适用于分类问题。它特别优化了类别特征的处理，可以自动处理类别特征，而无需手动进行独热编码。
- **XGBoost**XGBoost 是 “Extreme Gradient Boosting” 的缩写，是一个高效的梯度提升树算法，旨在提供计算速度和模型性能方面的优势。
- **LightGBM**LightGBM 是微软提供的一个开源的梯度提升框架，与 XGBoost 相似，但更为轻量并且训练速度更快。

使用 Optuna 优化 XGBoost 的超参数选择

Optuna

Optuna 是一个用于机器学习的超参数优化框架。超参数优化对于机器学习模型来说是至关重要的，因为它可以显著提高模型的性能和准确性。Optuna 提供了一个简洁而高效的界面，用于自动搜索超参数的最优值。

使用 Optuna 优化 XGBoost 的超参数选择

在 XGBoost 上的优化

使用 Optuna 对 XGBoost(一种梯度提升树算法) 的参数进行优化。具体涉及的参数如下:

- tree_method: 你选择了"gpu_hist"，意味着算法将使用 GPU 来构建直方图，从而加速训练过程。
-
- colsample_bytree: 这个参数决定了每次树构建时使用的特征的比例。
- subsample: 它决定了每次树的构建所使用的数据样本的比例。
- learning_rate: 这是一个步长，它决定了每一步中梯度下降的大小。
-
- min_child_weight: 它决定了子节点的最小权重，有助于正则化和防止过拟合。

使用 Optuna 优化 XGBoost 的超参数选择

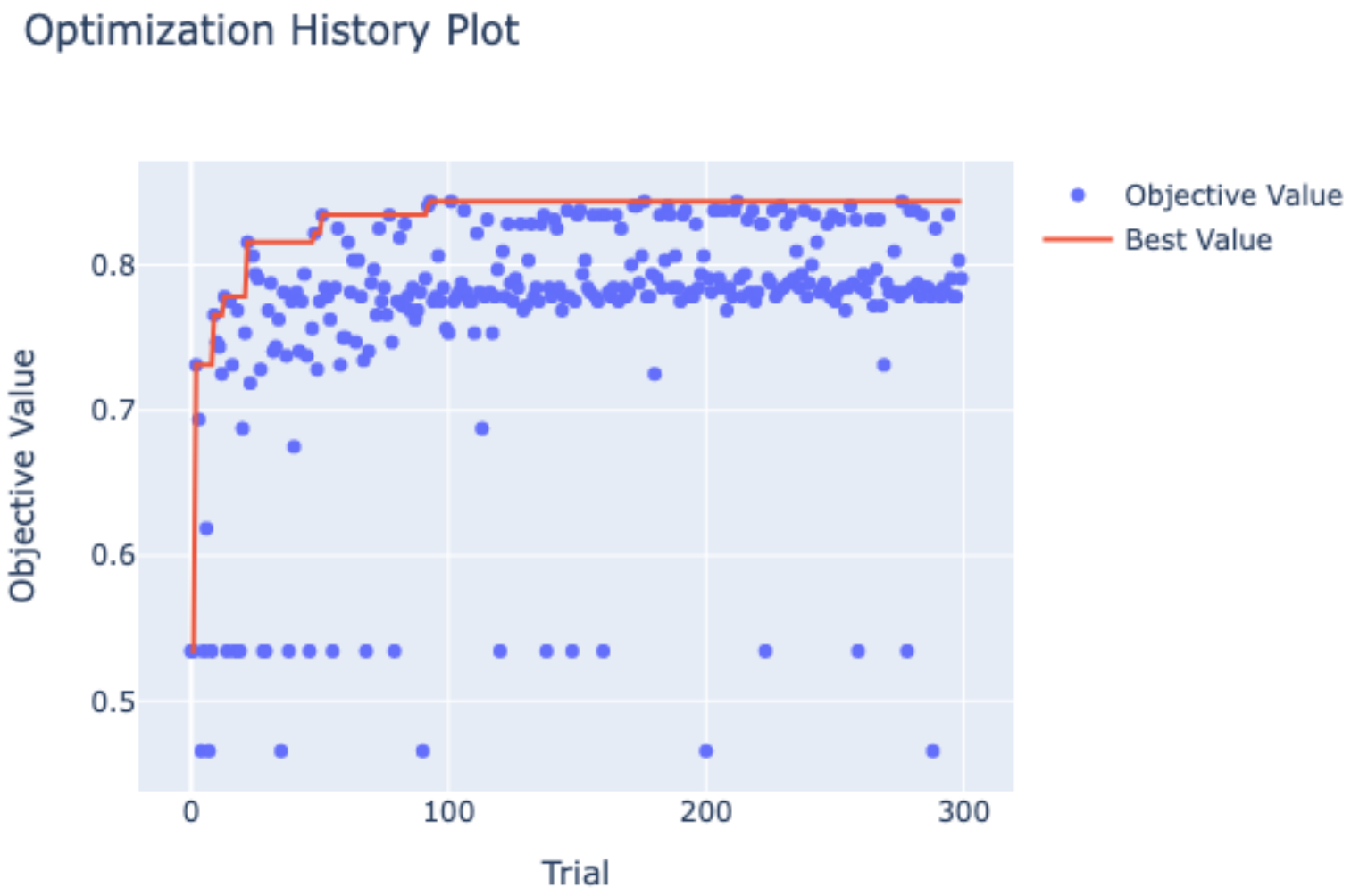
在 XGBoost 上的优化

一旦定义了这些参数范围和空间，Optuna 会进行多次试验，每次都使用不同的参数值，尝试找到可以使 XGBoost 的准确性最大化的最佳参数组合。

本次实验中选择了最多 300 次试验 (“trials”)，这意味着 Optuna 会尝试 300 组不同的参数来找到最佳组合。

使用 Optuna 优化 XGBoost 的超参数选择

Figure: Optuna 的优化参数选择效果



神经网络介绍

模型代码

```
model = Sequential(
[
    Input(shape=(2, 5, 1)),
    Conv2D(32, 3, padding="same", activation="relu"),
    Conv2D(32, 3, padding="same", activation="relu"),
    Conv2D(32, 3, padding="same", activation="relu"),
    MaxPooling2D(),
```

神经网络介绍

模型代码

```
Conv2D(64, 3, padding="same", activation="relu"),
Conv2D(64, 3, padding="same", activation="relu"),
Conv2D(64, 3, padding="same", activation="relu"),
Conv2D(64, 3, padding="same", activation="relu"),
Flatten(),
Dropout(0.2),
```

神经网络介绍

模型代码

```
        Dense(256, input_shape=(2, 5, 1), activation="relu"),
        Dense(128, activation="relu"),
        Dense(32, activation="relu"),
        Dense(1, activation="sigmoid"),
    ]
)
```

神经网络介绍

模型结构

模型借助 Tensorflow 框架搭建.

- **Input Layer** 是神经网络的输入层, 接受形状为 (2,5,1) 的数据;
- **Conv2D Layers** 是卷积层, 用于检测图像或者数据中的局部模式. 模型中共有 3 个这样的层, 前三个有 32 个滤波器, 后面四层有 64 个滤波器, 所有这些层都是用大小为 3 的卷积核, 使用 "same" 填充以及 ReLU 激活函数;
- **MaxPooling2D Layer** 该层用于空间降采样, 帮助减少模型的参数数量, 并防止过拟合;
- **Flatten Layer** 将特征展平, 使其可以被全连接层处理;
- **Dropout Layer** 是一个正则化层, 通过在训练过程中随机丢弃部分节点来放置过拟合. 使用 0.2 的 Dropout Rate, 每次前向传播的时候, 有 20% 的节点被随机关闭.
- **Dense Layers** 是全连接层, 模型中有三个, 分别有 256, 128 和 32 个节点, 使用 ReLU 激活函数. 输出层有 1 个节点, 使用 Sigmoid 激活函数以适应二分类任务.

训练介绍

模型编译

- Loss Function: 使用了 “binary_crossentropy’’ 这一二分类问题常用的损失函数;
- Optimizer: 使用了 Adam 优化器. Adam 是常用的, 自适应学习率的优化方法;
- Metrics: 使用精确度 (‘scc’) 作为评估指标;

训练介绍

学习率规划器

ReduceLROnPlateau 是一个学习率调度策略，当模型的精确度停滞不前的时候，会自动降低学习率:

- **monitor** 它检测模型的精确度;
- **patience** 如果经过 3 个 epoch，精确度没有提高，则学习率将被减少;
- **factor** 学习率将被乘以 0.5;
- **min_lr** 学习率的最小值为 0.00001;

假新闻检测任务

- 任务简介
- 数据集简介
- 模型介绍
- 训练介绍

假新闻检测任务

在本实验中，将借助新闻标题和新闻内容来预测新闻的真实性．考虑到这是一个二分类任务，为了完成该目标，借助 Sklearn 库进行 TfIdf 向量化，并使用 Passive Aggressive Classifier 线性分类模型进行假新闻分类.

任务简介

数据处理: TfIdfVectorizer

为了将新闻文章转化为计算机可以理解的形式, 使用 'TfIdfVectorizer'. Tf-Idf 指 "Term Frequency-Inverse Document Frequency", 是一种统计方法, 用于评估一个字词对于一个文件集或者一个语料库中其中一份文件的重要程度. 其目的是将文本内容转化为向量形式, 同时尽量捕捉文本中的关键信息.

- **Term Frequency(TF)** 指某一个给定的词语在该文件中出现的频率;
- **Inverse Document Frequency(IDF)** 是一个词语普遍重要性的度量. 减少那些经常出现在语料库中的词语的权重;

任务简介

模型：PassiveAggressiveClassifier

‘PassiveAggressiveClassifier’ 是一种线性分类模型，特点是对于错误分类的样本会进行很大幅度的参数更新，对于正确分类的样本则基本上不更新参数。这种方法非常适合大数据流场景，因为它不需要存储以前的数据，可以实时地更新模型。它在文本分类任务中，特别是像假新闻检测这样的任务中，效果很好。

任务简介

评估方式

为了评估模型的性能，使用了如下的方法:

- 准确度分数 (Accuracy Score): 它表示模型正确预测的新闻（无论真实还是假）与总新闻的比例。一个高的准确度通常意味着模型在测试数据上的效果好，但也需要注意其他评估指标，确保没有过拟合。
- 相关性矩阵 (Confusion Matrix): 它显示真正、真负、假正和假负的数量，提供了模型性能的更完整视图。这对于了解模型在哪些类别上出现了困难尤其有价值。

数据集简介

数据集描述

该数据集主要包含了新闻标题和新闻内容，用于判断新闻是真实还是假的．数据集包含了以下特征:

- **Identifier** 是每篇新闻的唯一标识符;
- **title** 新闻的标题;
- **text** 新闻的完整内容;
- **label** 新闻的标签，标记了新闻是真实 (REAL) 还是假的 (FAKE);

数据集简介

数据集的困难

对于该数据集，需要解决如下的困难:

- **不平衡的数据** 假新闻的数量与真实新闻的数量不平衡，需要考虑分类模型可能的偏见;
- **文本长度** 文章的长度可能有很大的变化，这可能会影响模型的效果.
- **文本质量** 由于新闻来源不保证，因此文本的质量可能有所不同. 低质量文本对文本处理技术和模型选择都带来了挑战.

模型介绍

特征提取: TfIdfVectorizer

使用 Tfldf 方法来评估词在文档中的重要性. 对其设置参数来避免常用词的影响:

- *stop_words='english'* 意味着在向量化过程中会排除英语中的常用停用词;
- *max_df=0.7* 意味着当一个词出现在超过 70% 的文档中的时候会被忽略, 这有助于排除太过常见的词汇, 例如 “is”, “the”, “and” 等. 虽然 IDF 已经为罕见词语提供较高值, 这样的整个数据集每个文档都出现的词语提供较低值, 直接去除也会更加方便;

模型介绍

分类模型: PassiveAggressiveClassifier

由于数据集较大, 7796 条样本的总大小为 **29.2MB**, 故使用适用于大型数据集的 Passive Aggressive Classifier. 因为它可以每次处理一个训练样本，而不是批量处理。这也使得它非常适合于文本分类问题，尤其是当面对大量的文本数据时。

其参数***max_iter=50*** 表示模型的最大迭代次数为 50.

性能评估

在数据集上将模型训练后，使用测试集来评估其性能．两个主要的评估指标为：

- **Accuracy** 衡量分类器对于预测正确类别的能力。在这里，准确度达到了 score * 100 的百分比。
- **Confusion Matrix** 它提供了一个关于分类器性能的更全面的视图，显示真实类别与预测类别之间的对应关系。对于这个二分类问题，混淆矩阵会显示四个值：真阳性、假阳性、真阴性和假阴性。