初识机器学习与深度学习

作者：GDUT_Li Wei

支持：Simpletex，通义、Copilot等AI大模型，CSDN、知乎、B站等互联网平台

感谢：广东工业大学数智工作室和RDC工作室提供的学习和编写笔记的动力

本文档版本最后更新时间：2024.4.27

开篇部分内容自2023-2024年的寒假开始编辑，后续内容约从2024.3月开始陆续补充。

P1 机器学习

Ch1 线性回归概述

1.1 本节提要

线性回归是一种监督学习的方法，用于建立一个或多个自变量（特征）和因变量（目标）之间的线性关系模型。线性回归的目的是找到一条直线（或一个平面），使得它能够最好地拟合训练数据，也就是使得预测值和真实值之间的误差（或损失）最小。

梯度下降法和**正规方程法**是线性回归模型中常用的两种方法，两种方法各有优缺点，各有适宜使用的情况。在建立模型前，通常还需要对数据集进行处理，常用的技巧有**最大最小归一化**和**z-score标准化**。

线性回归的一般形式是：

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n + \epsilon$$

符号	符号说明
y	因变量
θ_n	第n个特征的参数
x_n	自变量(特征)
ϵ	随机误差项

若用矩阵和向量的形式表示，则可以写为：

$$y = X\theta + \epsilon$$

符号	符号说明
y	一个$m\times1$的向量，表示m个样本的因变量
X	一个$m\times(n+1)$的矩阵，表示m个样本的n个自变量
θ	一个$(n+1)\times1$的向量，表示模型参数
ϵ	一个$m\times1$的向量，表示随机误差项

1.2 线性回归的损失函数（代价函数）

为了评估线性回归模型的好坏，我们需要定义一个损失函数，用于度量预测值和真实值之间的差异。常用的损失函数是**均方误差**（MSE）：

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

符号	符号说明
m	样本总数
$h_\theta(x)$	假设函数，基于输入特征x和参数θ进行预测
$h_\theta(x^{(i)})$	第i个样本的预测值
$y^{(i)}$	第i个样本的真实值

目标是**找到一组最优的参数θ使得该函数最小。**

1.3 梯度下降法

1.3.1 综述

梯度下降法是一种迭代求解最优参数的方法，它的基本思想是，从一个随机的初始参数（一般是0）开始，沿着损失函数下降最快的方向（也就是负梯度的方向），逐步更新参数，直到达到一个局部最小值或全局最小值。

梯度下降法的关键点是如何确定每次更新的步长，也就是**学习率。**学习率过大，可能会导致参数在最小值附近震荡，甚至跳过最小值；学习率过小，可能会导致参数收敛速度过慢，甚至停滞在一个非最小值的点。因此，选择一个合适的学习率是梯度下降法的重要任务。

梯度下降法的一般公式：

$$\theta := \theta - \alpha\frac{\partial J(\theta)}{\partial\theta}$$

符号	符号说明
θ	参数
α	学习率
$\frac{\partial J(\theta)}{\partial\theta}$	损失函数对参数的偏导数(梯度)

注意，在有多个特征及多个参数的情况下，应注意同步更新θ的值：

$$\begin{aligned}
&\text{temp0} := \theta_0 - \alpha\frac{\partial}{\partial\theta_0}J(\theta_0,\theta_1)\\
&\text{temp1} := \theta_1 - \alpha\frac{\partial}{\partial\theta_1}J(\theta_0,\theta_1)\\
&\theta_0 := \text{temp0}\\
&\theta_1 := \text{temp1}
\end{aligned}$$

1.3.2 多元梯度下降法

当特征数量≥1时，梯度下降法中的偏导数可以这样计算：

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}$$

符号	符号说明
$x_j^{(i)}$	第i个样本的第j个特征值

此时，有如下公式：

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

同样需要注意的是，当j在遍历$j = 1, 2, \ldots, n$的过程中始终要同步更新θ_j。

Examples：

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_1^{(i)}$$
$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_2^{(i)}$$

1.3.3 梯度下降中的技巧1-特征缩放

特征缩放是指在机器学习中对特征数据进行预处理，以确保所有特征都处于相同的尺度。这一步骤对于梯度下降算法特别重要，因为它可以帮助算法更快地收敛。特征缩放最常见的两种技术是：**最大最小归一化**和**z-score标准化**。我们希望尽量使每个特征的值都介于-1到1之间，可以略小于或大于这个范围但不可过大或过小。

最大最小归一化

最大最小归一化将所有特征缩放到$[0, 1]$的范围内。其公式为：

$$X\mathrm{norm} = \frac{X - X\mathrm{min}}{X\mathrm{max} - X\mathrm{min}}$$

z-score标准化

z-score标准化会将特征的均值变为0，标准差变为1。其公式为：

$$X_{\mathrm{std}} = \frac{X - \mu}{\sigma}$$

符号	符号说明
μ	特征的平均值
σ	特征的标准差

Examples

假设我们有以下数据集，里面包含房屋的面积和房价，我们希望使用梯度下降法预测房价：

面积	价格
2104	399.9
1600	329.9
2400	369.0

由于面积的数值比价格的数值大得多，这可能会使得梯度下降的路径变得非常不规则，从而减慢收敛速度。我们可以使用z-score标准化：

```python
import numpy as np
# 假设X是面积，y是房价
X = np.array([2104, 1600, 2400])
y = np.array([399.9, 329.9, 369.0])
# 对X和y进行标准化
X_mean = X.mean()
X_std = X.std()
X_standardized = (X - X_mean) / X_std
y_mean = y.mean()
y_std = y.std()
y_standardized = (y - y_mean) / y_std
# 现在可以使用标准化后的X和y来进行梯度下降算法了
```

标准化后，面积和价格的数值将位于同一个数量级，这样梯度下降算法在搜索参数空间时就更容易找到最优解。在实际应用模型之后，通常需要将预测结果反标准化，使其回到原始的尺度。

1.3.4 梯度下降中的技巧2-学习率

如果学习率过小，收敛过程可能过慢；

如果学习率过大，损失函数可能不是每次迭代都减小，也有可能不收敛。

可以尝试通过绘制损失函数$J(\theta)$随θ的函数曲线来判断（debug）。

尝试如下的学习率：……，0.001，0.003，0.01，0.03，0.1，0.3，1，……

1.4 正规方程法

1.4.1 公式推导

正规方程法是利用矩阵运算，直接求解损失函数的最小值点，也就是最优参数。它的基本思想是，对损失函数求偏导数，令其等于零，然后解出参数。**该方法不需要特征缩放。**

以下是推导过程：

$$
\begin{aligned}
J(\theta) &= \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \\
&= \frac{1}{2m}(X\theta - Y)^T(X\theta - Y) \\
&= \frac{1}{2m}\left(\theta^T X^T - Y^T\right)(X\theta - Y) \\
&= \frac{1}{2m}\left(\theta^T X^T X\theta - \theta^T X^T Y - Y^T X\theta + Y^T Y\right) \\
\frac{\partial J(\theta)}{\partial \theta} &= \frac{1}{2m}\left[\frac{d\theta^T X^T X\theta}{d\theta} - \frac{d\theta^T X^T Y}{d\theta} - \frac{dY^T X\theta}{d\theta} + 0\right] \\
&= \frac{1}{2m}(2X^T X\theta - X^T Y - (Y^T X)^T) \\
&= \frac{1}{2m}(2X^T X\theta - 2X^T Y) = 0 \\
&\therefore X^T X\theta = X^T Y
\end{aligned}
$$

得出最终的公式：

$$\theta = (X^T X)^{-1} X^T Y$$

1.4.2 正规方程法和梯度下降法的对比

特点	梯度下降法	正规方程法
计算复杂度	依赖于迭代次数	高，需要计算$(X^TX)^{-1})$
特征数量影响	适合特征数量较多	当特征数量很多时，计算 $(X^TX)^{-1}$变得非常慢
样本数量影响	大样本下运行较慢	样本数量对速度影响不大
需求内存	较少	较多，需要存储 X^TX和X^Ty
适用情况	特征数量很多时更好	特征数量较少时更好
收敛速度	可调整学习率来加快	不需要迭代，一步到位

1.4.3 正规方程在矩阵不可逆情况下的解决方法

Q：矩阵何时出现不可逆的情况?

1. 学习数据中存在冗余的特征。例如在预测房价时，x_1是以平方米为单位的房屋面积，x_2是以平方英尺为单位的房屋面积，即$x_2 = (3.28)^2x_1$，而**两个特征不可以线性相关。**
2. 样本数量$\leq$特征数量。此时可以尝试删除一些特征，或使用正则化。

Ch2 过拟合与正则化

2.1 本节提要

当我们使用线性回归模型对数据进行拟合时，可能会遇到**过拟合**的问题。过拟合发生在模型对训练数据学得太好，以至于它学习到了训练数据中的**噪声**和**异常点**，导致模型的**泛化能力**下降，对新的、未知的数据预测性能变差。（有时也被称为“高方差”）

正则化是一种避免过拟合的技术。其核心思想是在损失函数中增加一个**惩罚项**，这个惩罚项会限制模型的复杂度。简单来说，正则化通过给模型的参数增加一些约束，迫使模型保持简单，从而提高模型对新数据的泛化能力。

在线性回归中，常用的正则化方法有两种：

1. **L1正则化**，也称为**Lasso回归**，它通过向损失函数增加参数的绝对值之和来惩罚模型的复杂度。这种方法可能会使一些参数变为零，从而实现特征的选择。
2. **L2正则化**，也称为**岭回归**，它通过向损失函数增加参数的平方和来惩罚模型的复杂度。这让模型的参数更加平滑，避免了过大的参数值。

通过调节正则化项前的系数（通常称为λ或α），可以控制正则化的强度，进而找到一个既可以很好地拟合训练数据，又能够很好地泛化到新数据的模型。

2.2 正则化

2.2.1 L2正则化（岭回归）

L2正则化，也被称为岭回归，它的做法是在损失函数后面增加一个惩罚项：

$$J(\theta) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{j=1}^{n}\theta_j^2\right]$$

符号	符号说明
λ	正则化参数
$\lambda\sum_{j=1}^{n}\theta_j^2$	正则化项

正则化参数的作用是平衡两个目标的取舍：目标1是我们想通过这一模型更好的拟合训练数据；目标2是保持参数θ_j尽可能地小，从而保持假设模型的相对简单，避免出现过拟合的情况。

如果将正则化参数λ设置得过大，也即对参数的惩罚程度过大，导致参数趋近于0，使得模型出现了**欠拟合。**（夸张点说就是接近于一条水平的直线）；当λ很小接近于0时，惩罚力度很小，很多自变量特征对应的回归系数绝对值很大，模型自由度很高，处于过拟合的状态。

梯度下降法中的岭回归

易得：

$$\theta_j := \theta_j(1-\alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)}$$

显然，由于α通常很小，而样本数量m却很大，因此$(1-\alpha\frac{\lambda}{m})$通常是一个比1略小的数。这条公式可以理解为将$\theta_j$变小了一点点然后再迭代下降。

正规方程法中的岭回归

我们设计一个$m\times(n+1)$维矩阵X，它的每一行都代表一个单独的训练样本；然后建立一个m维向量，它包含了训练集内的所有标签：

$$X=\begin{bmatrix}(x^{(1)})^T\\ \vdots \\ (x^{(m)})^T\end{bmatrix}\quad Y=\begin{bmatrix}y^{(1)}\\ \vdots \\ y^{(m)}\end{bmatrix}$$

推导如下：

$$\begin{aligned}J(\theta)&=\frac{1}{2m}(X\theta-Y)^T(X\theta-Y)+\frac{1}{2m}\lambda\theta^T\theta\\&=\frac{1}{2m}\left(\theta^TX^T-Y^T\right)(X\theta-Y)+\frac{1}{2m}\lambda\theta^T\theta\\&=\frac{1}{2m}\left(\theta^TX^TX\theta-\theta^TX^TY-Y^TX\theta+Y^TY+\lambda\theta^T\theta\right)\end{aligned}$$

然后求偏导：

$$\frac{\partial J(\theta)}{\partial\theta}=\frac{1}{2m}(2X^TX\theta-2X^TY+\lambda\theta^T\theta)=0$$

得到：

$$\theta=(X^TX+\lambda I)^{-1}X^TY$$

而由于$\theta_0=1$不参与正则，通常会将单位矩阵I的第一个数字1置为0。在正规方程加入正则化项后，$(X^TX+\lambda I)$一定是可逆的，因此**加入正则化项也是解决矩阵不可逆的方法之一。**

Ch3 特征工程

3.1 本节提要

在机器学习中，解决一个问题的一般顺序是：

> **数据预处理(数据清洗)-->算法模型训练--->算法模型验证评估-->获得一个模型(有了适合的参数)**

为了学习**数据预处理**的规范步骤，我通过学习**特征工程**的方法和思路来实现数据预处理。特征工程涉及到从原始数据中提取有用的特征，以便更好地训练模型。我们可以把它比作做菜的过程：

1. **原材料选择**：这就像是从市场上挑选食材。在机器学习中，这意味着从大量的原始数据中选择有潜力的特征（食材）来训练我们的模型（菜肴）。
2. **清洗处理**：就像我们要清洗和切割食材一样，特征工程中也需要对数据进行清洗和预处理。这可能包括填补缺失值、去除异常值或转换数据格式等。
3. **特征构建**：这就像是根据食谱将食材混合在一起。在机器学习中，我们可能会基于现有数据创建新的特征，比如计算两个特征之间的比例，或者将多个特征组合成一个。
4. **调味品调整**：在做菜时，我们会根据口味添加调味品。在特征工程中，这可以比作特征缩放和归一化，确保不同的特征在模型中有相同的影响力。
5. **烹饪方法**：最后，我们选择烹饪方法，比如煎、炒、炖等。在机器学习中，这就是选择合适的算法来训练模型。

通过这个过程，我们可以将原始数据转化为模型能够理解和学习的形式，从而提高模型的性能。~~(以上几条copy于Bing AI)~~

3.2 数据预处理

3.2.1 缺失值处理

直接删除

缺失值最简单的处理方法是**删除**，所谓删除就是删除属性或者删除样本，如果某一个特征中存在大量的缺失值(缺失量大于总数据量的40%~50%及以上)，那么我们可以认为这个特征提供的信息量非常有限，这个时候可以选择删除掉这一维特征；如果整个数据集中缺失值较少或者缺失值数量对于整个数据集来说可以忽略不计的情况下，那么可以直接删除含有缺失值的样本记录。

以下代码提供了一个删除缺失值的方法：

```
def delete_feature(df):
    N = df.shape[0]  # 样本数
    no_nan_count = df.count().to_frame().T  # 统计了每个特征的非缺失值数量，并将其转换成了DataFrame的形式，并且进行了转置操作，使得每个特征的非缺失值数量在一行中
    del_feature, save_feature = [], []
    for col in no_nan_count.columns.tolist():
        loss_rate = (N - no_nan_count[col].values[0])/N  # 缺失率
        # print(loss_rate)
        if loss_rate > 0.5:  # 缺失率大于 50% 时，将这一维特征删除
            del_feature.append(col)
        else:
            save_feature.append(col)
    return del_feature, df[save_feature]
```

统计值填充

统计值一般泛指**平均值、中位数、众数、最大值、最小值**等。

```
# 均值填充
df.fillna(df.mean())
```

其余统计值的函数不做笔记。

统一值填充

对于缺失值，把所有缺失值都使用统一值作为填充词，所谓统一值是指**自定义指定的某一个常数。**常用的统一值有：空值、0、正无穷、负无穷或者自定义的其他值。

```
# 统一值填充
# 自定义统一值常数为 10
df.fillna(value=10)
```

前后向值填充

前后向值填充是指使用缺失值的**前一个或者后一个的值**作为填充值进行填充。

```
df.fillna(method='ffill')  # 前向填充
df.fillna(method='bfill')  # 后向填充
```

3.2.2 连续变量离散化

特征二值化

思想：**设定一个划分的阈值，当数值大于设定的阈值时，就赋值为1；反之赋值为0。**例如对年龄进行二值化，即可以将年龄分为青年、中年、老年三组，每组设置一定的年龄范围，分别将其统一赋值为0，1，2。

```
# 特征二值化
# 自己手写理论公式来实现功能
def Binarizer(ages):
    ages_binarizer = []
    print('>>>原始的定量数据\n', ages)
    for age in ages:
        if (age > 0) and (age <= 18):
            ages_binarizer.append(0)
        elif (age >= 19) and (age <= 40):
            ages_binarizer.append(1)
        elif (age >= 41) and (age <= 60):
            ages_binarizer.append(2)
    print('\n>>>特征二值化之后的定性数据\n', ages_binarizer)
    return ages_binarizer

ages = [4, 6, 56, 48, 10, 12, 15, 26, 20, 30, 34, 23, 38, 45, 41, 18]
Binarizer(ages)
```

3.2.3 独热编码

独热编码是一种常用的分类变量编码方法，用于**将分类变量转换为数值型变量**。它将一个有限个可能的取值集合映射到由0和1组成的向量空间，其中每个可能的取值对应于向量空间中的一个坐标，并且只有一个坐标值为1，其余坐标值为0。

假设一个分类特征有 n 个不同的取值，独热编码将这 n 个取值转换成一个 n 维的向量。对于原始的分类特征中的每一个取值，独热编码的向量中对应该取值的位置标记为1，其余位置标记为0。

假设我们有一个包含性别（Gender）和地区（Region）两个分类特征的数据集，如下所示：

ID	Gender	Region
1	Male	North
2	Female	South
3	Male	East
4	Female	West
5	Male	Central

首先，我们需要导入必要的库并创建数据集：

```
import pandas as pd

# 创建数据集
data = {
    'ID': [1, 2, 3, 4, 5],
    'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
    'Region': ['North', 'South', 'East', 'West', 'Central']
}

df = pd.DataFrame(data)
print("原始数据集：")
print(df)
```

接下来，我们使用独热编码来处理这两个分类特征：

```
# 对 Gender 和 Region 列进行独热编码
df_encoded = pd.get_dummies(df, columns=['Gender', 'Region'])

print("\n经过独热编码后的数据集：")
print(df_encoded)
```

这段代码中，`pd.get_dummies()` 函数会对指定的列进行独热编码处理，将每个分类变量转换为一组二进制特征。然后我们打印出经过独热编码后的数据集，观察转换结果。

3.2.4 异常值处理

箱线图

箱线图是一种常用的可视化工具，用于展示数据的分布情况和识别异常值。它由五个统计量组成：最小值、下四分位数（Q1）、中位数、上四分位数（Q3）和最大值。箱线图能够清晰地显示出数据的中心位置、离散程度以及异常值的存在。

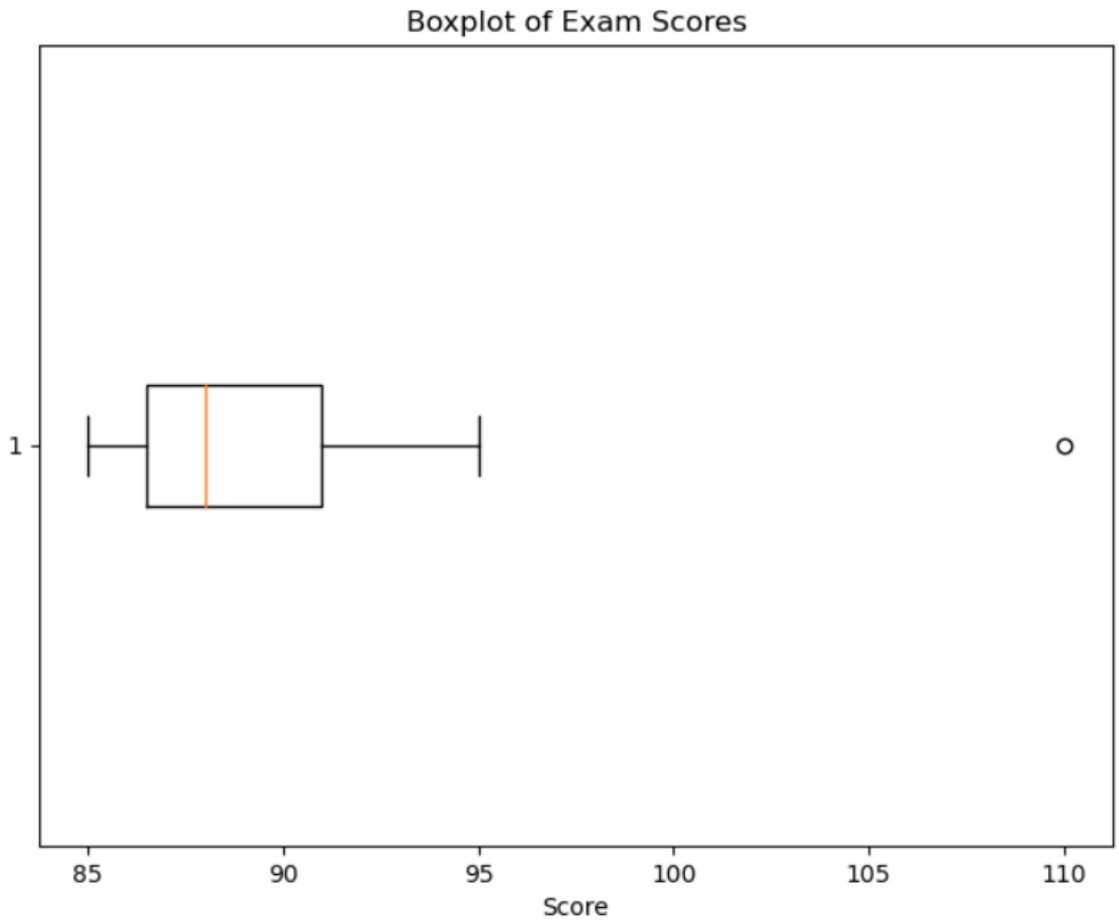
1. 箱体： 箱体显示了数据的四分位数范围，即Q1到Q3。箱体的长度表示数据的离散程度，箱体越长，数据的离散程度越大。
2. 中位数线： 箱体中间的线表示数据的中位数，即第二四分位数（Q2）。
3. 触须：触须延伸出箱体，通常延伸到位于箱体上下边缘的1.5倍的四分位距离（IQR）处。**触须以外的数据点被认为是异常值。**
4. 异常值： 超出触须范围的数据点被认为是异常值，通常用点或者其他符号标识出来。
5. 上下限：在触须延伸的1.5倍IQR之外的点被认为是异常值的临界点。**超过上下限的点被认为是极端异常值。**

假设我们有一个包含学生考试成绩的数据集，我们将使用箱线图来识别可能存在的异常值：

```
import pandas as pd
import matplotlib.pyplot as plt

# 创建示例数据集
data = {
    'Student': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K'],
    'Score': [85, 90, 88, 92, 89, 85, 95, 110, 87, 88, 86]
}

df = pd.DataFrame(data)
# 绘制箱线图
plt.figure(figsize=(8, 6))
plt.boxplot(df['Score'], vert=False)
plt.xlabel('Score')
plt.title('Boxplot of Exam Scores')
plt.show()
```



显然，110分的点远远超过的箱线图的边界，故110为异常值。

df.describe 方法

`df.describe()` 方法用于生成数据集的统计摘要，包括计数、均值、标准差、最小值、25th、50th和75th分位数以及最大值等信息。通过查看描述性统计信息，我们可以初步了解数据的分布情况，并发现一些异常值的迹象。

下面是一个简单的例子，使用 `df.describe()` 方法来查看一个示例数据集中的统计摘要，并尝试通过这些统计信息来识别异常值：

```
import pandas as pd

# 创建示例数据集
data = {
    'Value': [5, 8, 7, 9, 12, 15, 8, 6, 20, 10, 25, 22, 5, 8, 7, 9, 12, 15, 8,
6, 20, 10, 25, 22, 500]
}

df = pd.DataFrame(data)

# 使用 describe() 方法生成统计摘要
summary = df.describe()
print(summary)

# 根据统计摘要查找异常值
Q1 = summary.loc['25%', 'Value']
Q3 = summary.loc['75%', 'Value']
IQR = Q3 - Q1
```

```
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# 输出异常值
outliers = df[(df['Value'] < lower_bound) | (df['Value'] > upper_bound)]
print("\n异常值：")
print(outliers)
```

接下来，根据统计摘要来识别异常值。常用的方法是基于四分位数（Q1、Q3）和四分位间距（IQR）来定义异常值的上下边界。根据 IQR，异常值定义为**小于 Q1 - 1.5 * IQR 或大于 Q3 + 1.5 * IQR 的值。**

以下是程序的运行结果：

	Value
count	25.0000
mean	31.7600
std	97.7617
min	5.0000
25%	8.0000
50%	10.0000
75%	20.0000
max	500.0000

异常值：

	Value
24	500

标准差法（拉依托准则）

标准差法是用来识别异常值的一种方法，它依据的是数据的标准差，是一种基于统计分布的方法。以下是利用标准差法检测数据中的异常值的步骤：

1. 计算数据集的平均值（mean）和标准差（standard deviation，std）。
2. 选择一个阈值，标准的做法通常是选择2个或3个标准差。
3. 根据阈值和标准差计算出异常值的界限，即平均值加上或减去2或3倍的标准差。
4. 数据点如果超出这个界限就可以被认为是异常值。

Ch4 遇到的问题及解决方法

这里的问题记录的是参加数智工作室考核过程中遇到的问题。下P1附录“代码”指的同样是考核的代码。

问题1 第一次提交时的score只有0.02

解决1 问题根源在于我注意到了第9和11个特征（9和11为列名）的特征值显著大于其他特征值，故对且仅对这两列特征做了z-score标准化，使得整个数据集的特征和特征之间标准化不统一，导致拟合性能差。将所有特征都统一进行了z-score标准化后分数提升到0.61。

问题2 频繁提交，频繁负优化

解决2 在提交了多次submission后发现更改迭代次数、学习率、正则化参数等对得分的影响微乎其微，遂将优化的重心放在了数据预处理上（当然对线性回归模型的优化仍有很多学问，但目前来说更改数据预处理的方法的性价比应该是最高的）。最开始，我的数据预处理采用的是以中位数/平均值替换缺失值+IQR方法检测异常值并用平均值来替换+所有特征执行z-score标准化，这样做的话提交的分数在0.60左右。（3.21以后）后来，我继续学习更多的数据预处理方法，尝试了包括但不限于MAD方法、标准差法处理异常值以及用许多不同的统计值来替换，另外使用sklearn库的r2_score方法来评估拟合性能。结合模型对训练集的拟合和测试集的拟合判断，标准差法处理后的数据所用相同方法训练出来的模型的性能提升明显，从0.60提升至0.76。

问题3 有关缺失值的处理方法选择

解决3 一开始我是用中位数替代缺失值，后来尝试直接删除缺失值和用平均数替代缺失值，最终选定直接删除，因为缺失值不算多且这样的打榜分数较高一些。

问题4 梯度下降得到的参数全是NaN

解决4 没有对数据进行归一化。后续采用z-score标准化即解决。

P1附录 线性回归的实际代码中使用到的数学公式

含有L2正则化项的梯度计算公式

$$\nabla_\theta J(\theta) = \frac{1}{m} X^T (X\theta - y) + \frac{\lambda}{m}\theta$$

含有L2正则化项的梯度修正公式

$$\nabla_{\theta_j} R(\theta) = \begin{cases} 0 & \text{if } j = 0 \\ \frac{\lambda}{m}\theta_j & \text{if } j > 0 \end{cases}$$

特征参数更新公式

$$\theta := \theta - \alpha \cdot \nabla_\theta J(\theta)$$

含有L2正则化项的损失函数公式（MSE）

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

P2 从逻辑回归到Softmax回归

Ch5 概述

逻辑回归：从线性回归到二分类问题

现在我们要判断一封邮件是否是垃圾邮件（是/否，二分类问题）。逻辑回归仍然会计算一个类似线性回归的分数，但它不会直接输出结果，而是把这个分数通过一个特殊的S型曲线（Sigmoid函数）转换成0到1之间的概率，用来表示邮件是垃圾邮件的可能性。如果概率超过某个阈值（比如0.5），我们就判定它是垃圾邮件。

公式表达上，逻辑回归会对一系列特征（如邮件中的关键词出现次数等）计算加权总和，然后通过Sigmoid函数得出是垃圾邮件的概率：

$$\text{垃圾邮件概率} = \frac{1}{1 + e^{-(\text{特征}1\times\text{权重}1+\text{特征}2\times\text{权重}2+\ldots)}}$$

Softmax回归：从二分类问题到多分类问题

当你需要预测不止两种可能的结果时，例如识别一张图片是猫、狗还是兔子（多分类问题），就需要用到Softmax回归。它首先对每个类别的可能性计算一个得分（类似于线性回归的分数），然后把这些得分经过一个特定的函数（Softmax函数）处理，使其转化为一组互斥且总和为1的概率分布，最高的那个概率对应的类别就是模型的预测结果。

Softmax函数可以把每个类别的得分变成概率形式：

$$P(\text{类别}_j|\text{特征}) = \frac{e^{\text{得分}_j}}{\sum_{k=1}^{\text{类别总数}} e^{\text{得分}_k}}$$

通过这种方式，Softmax回归允许我们在多个类别中做出选择，并给出了每个类别的相对可信度。

Ch6 二元逻辑回归

6.1 Sigmoid函数与模型的假设函数

对线性回归的结果做一个在函数g上的转换，可以变化为逻辑回归。这个函数g在逻辑回归中我们一般取为Sigmoid函数，形式如下：

$$g(z) = \frac{1}{1 + e^{-z}}$$

其图形呈S形曲线，具有以下关键性质：

- 输入z趋近于正无穷大时，输出接近于1；
- 输入z趋近于负无穷大时，输出接近于0；
- 在$z = 0$处取得最大斜率，使得模型能够很好地处理阈值决策。

我们挪用线性回归中$y = X\theta$的计算公式，令Sigmoid函数中的自变量$z = X\theta$，即得到二元逻辑回归的模型的假设函数（矩阵形式）：

$$h_\theta(X) = \frac{1}{1 + e^{-X\theta}}$$

符号	符号说明
$h_\theta(X)$	一个$m\times1$的向量，表示m个样本的因变量
X	一个$m\times n$的矩阵，表示m个样本的n个自变量
θ	一个$n\times1$的向量，表示模型参数

其中X为样本输入，$h_\theta(X)$为模型输出，可以理解为某一分类的概率大小。而θ为求出的模型参数。对于模型输出：如果$h_\theta(X) > 0.5$，即$X\theta > 0$，y为1。如果$h_\theta(X) < 0.5$，即$X\theta < 0$，则y为0。$y = 0.5$是临界情况，此时$X\theta = 0$，从逻辑回归模型本身无法确定分类。

6.2 对数似然损失（交叉熵损失）

逻辑回归模型常采用的损失函数是对数似然损失，也称为交叉熵损失，对于单个样本而言，其损失函数定义为：

$$L(h_\theta(x), y) = -\Big[y\log(h_\theta(x)) + (1-y)\log(1-h_\theta(x))\Big]$$

对于包含m个样本的数据集，整体损失函数是对每个样本损失函数求和后再取平均：

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log\left(1-h_\theta(x^{(i)})\right)]$$

这个公式用矩阵方法表达更加简洁：

$$J(\theta) = -Y^T\log h_\theta(X) - (E-Y)^T\log(E-h_\theta(X))$$

符号	符号说明
Y	一个$1\times m$的向量，表示m个样本的真实值
E	单位矩阵

6.3 梯度下降法

和线性回归中的迭代公式一样：

$$\theta_j := \theta_j - \alpha\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

也可以用向量形式表示为：

$$\theta := \theta - \alpha X^T(h_\theta(X) - Y)$$

6.4 二元逻辑回归的具体实例

该程序中的数据集是根据人脸外貌分辨性别的数据集，已经被分割为了训练集和数据集。使用accuracy方法对模型的性能进行评估。在多次随机拆分数据集后，准确度达到0.968左右。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt


# Sigmoid 函数实现逻辑回归的假设函数
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


# 交叉熵损失函数
def compute_cost(X, y, theta):
    m = y.size
    h = sigmoid(X.dot(theta))
    # 加入一个小值 epsilon 防止计算 log(0)
    epsilon = 1e-5
    cost = -np.sum(y * np.log(h + epsilon) + (1 - y) * np.log(1 - h + epsilon))
/ m
    return cost
```

```
# 梯度下降函数
def gradient_descent(X, y, theta, alpha, iterations):
    m = y.size  # 样本数量
    cost_history = []  # 记录损失
    for _ in range(iterations):
        predictions = sigmoid(X.dot(theta))  # 假设函数
        error = predictions - y  # 误差
        gradient = X.T.dot(error) / m  # 求出梯度
        theta -= alpha * gradient  # 更新参数
        cost_history.append(compute_cost(X, y, theta))  # 记录损失
    return theta, cost_history


# 准确度计算
def calculate_accuracy(X, y, theta):
    predictions = sigmoid(X.dot(theta)) >= 0.5
    return (predictions == y).mean()


# 加载预处理后的训练集和测试集
train_data = pd.read_csv('train_data.csv')
test_data = pd.read_csv('test_data.csv')
# 准备数据
X_train = np.hstack([np.ones((train_data.shape[0], 1)), train_data.iloc[:,
:-1].values])  # 添加一列全1偏置项
y_train = train_data.iloc[:, -1].values.reshape(-1, 1)  # 将目标变量重塑为2维数组
X_test = np.hstack([np.ones((test_data.shape[0], 1)), test_data.iloc[:,
:-1].values])
y_test = test_data.iloc[:, -1].values.reshape(-1, 1)
# 初始化参数
theta = np.zeros((X_train.shape[1], 1))
alpha = 0.1  # 学习率
iterations = 5000  # 迭代次数
# 训练逻辑回归模型
theta, cost_history = gradient_descent(X_train, y_train, theta, alpha,
iterations)
# 绘制损失函数曲线
plt.plot(cost_history)
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Loss Curve')
plt.show()
# 打印模型参数
print('训练得到的参数 theta: \n', theta)
# 使用测试集评估模型
accuracy = calculate_accuracy(X_test, y_test, theta)
print('测试集准确度: ', accuracy)
```

Ch7 Softmax回归

7.1 logits向量

在softmax回归模型中，logits是一个重要的中间变量，它代表了模型在未进行归一化处理前对每个样本各类别的预测得分，它本质上是**每个类别相对于其他类别的对数几率**。具体来说，给定一个输入样本 $\mathbf{x}_i$，其特征向量表示为 $\mathbf{x}_i \in \mathbb{R}^{n_{\text{features}}}$。在softmax回归模型中，我们会计算该样本对于各个类别的未归一化预测分数，这些分数被组织成一个向量，即所谓的logits向量 $\mathbf{l}_i \in \mathbb{R}^{n_{\text{labels}}}$,其中 n_{labels} 是模型所要区分的类别数。Logits向量的计算公式如下：

$$\mathbf{l}_i = \mathbf{x}_i\mathbf{W} + \mathbf{b}$$

符号	符号说明
$\mathbf{W}$	权重矩阵，维度为$(n_{\text{features}}, n_{\text{labels}})$
$\mathbf{b}$	偏置向量，维度为$(1, n_{\text{labels}})$

通过对输入特征向量 $\mathbf{x}_i$ 与权重矩阵 $\mathbf{W}$ 进行点积，再加上偏置向量 $\mathbf{b}$，得到样本 $\mathbf{x}_i$ 对应的logits向量 $\mathbf{l}_i$。每个元素 l_{ik} 表示样本 $\mathbf{x}_i$ 属于第 k 类的原始预测得分 (未考虑其他类别的相对关系) 。

```
logits = np.dot(X_train, weights) + bias  # logits是下一步被投给softmax的向量
```

7.2 将logits向量应用softmax函数

logits向量中的元素可以是任意实数，没有特定的约束。这些数值通常反映了模型在当前权重和偏置设置下，对于输入样本属于各个类别的初步估计。然而，这些原始得分并不直接适合作为概率输出，因为它们可能为负值、大于1，且各元素之间缺乏概率分布应有的归一化性质（即所有类别概率之和为1）。

为了将这些原始得分转化为概率分布，我们需要对logits向量应用softmax函数：

$$\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{l}_i) = \frac{1}{\sum_{j=1}^{K} \exp(l_{ij})} \begin{bmatrix} \exp(l_{i1}) \\ \exp(l_{i2}) \\ \vdots \\ \exp(l_{iK}) \end{bmatrix}$$

符号	符号说明
$\hat{\mathbf{y}}_i$	第i个样本的预测概率分布，维度为$(1, n_{\text{labels}})$，所有元素之和为1
l_{ij}	第i个样本$logits$向量的第j个元素
K	类别数量(n_{labels})

softmax函数会确保输出向量 $\hat{\mathbf{y}}_i$ 中的所有元素非负且总和为1，从而得到样本 $\mathbf{x}_i$ 对应的类别概率分布。在这个分布中，每个元素 $\hat{y}_{ik}$ 表示样本属于第 k 类的概率。

```
# 定义softmax函数
def softmax(z):
    z -= np.max(z, axis=1, keepdims=True)  # 从z的每一行中减去该行的最大值，防止溢出
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
y_pred = softmax(logits)
```

7.3 带有岭回归的交叉熵损失函数

交叉熵损失

交叉熵损失用于衡量模型预测的概率分布与实际标签分布之间的差异。对于多类别分类问题，假设有一个样本的标签为 $\mathbf{y}_i$ (通常是独热编码形式)，模型预测的概率分布为$\hat{\mathbf{y}}_i$。则该样本的交叉熵损失 L_{CE} 可以表示为：

$$L_{\text{CE}}(\mathbf{y}_i, \hat{\mathbf{y}}_i) = -\sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

符号	符号说明
y_{ik}	样本i实际标签在第k类的值$(\text{独热编码中对应类别为}1, \text{其余为}0)$
$\hat{y}_{ik}$	模型预测的样本i属于第k类的概率

而对于整个训练集，**平均交叉熵损失**$\mathcal{L}_{\text{CE}}$计算如下：

$$\mathcal{L}_{\text{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik}$$

符号	符号说明
N	训练集中的样本数

岭回归（L2正则化）

将交叉熵损失与L2正则化项组合起来，得到带有岭回归的交叉熵损失函数 $\mathcal{L}_{\text{total}}$：

$$\mathcal{L}_{\text{total}}(\mathbf{Y}, \hat{\mathbf{Y}}, \mathbf{W}) = \mathcal{L}_{\text{CE}}(\mathbf{Y}, \hat{\mathbf{Y}}) + L_{\text{Ridge}}(\mathbf{W})$$

即：

$$\mathcal{L}_{\text{total}}(\mathbf{Y}, \hat{\mathbf{Y}}, \mathbf{W}) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log \hat{y}_{ik} + \frac{\lambda}{2N} \sum_{j=1}^{N} \sum_{k=1}^{K} W_{jk}^2$$

```python
# 定义交叉熵损失函数
def cross_entropy_loss(y_pred, y_true):
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)  # 限制范围，防止对0取对数
    loss = -np.mean(np.sum(y_true * np.log(y_pred), axis=1))
    # 岭回归正则化项
    l2_loss = lambda_param * np.sum(weights ** 2)
    return loss + l2_loss / (2 * num_samples)
```

7.4 梯度与参数更新

7.4.1 权重梯度

权重梯度是指损失函数关于权重矩阵$\mathbf{W}$的偏导数，表示了损失函数值随权重矩阵中每个元素微小变动的变化率。对于带有岭回归的交叉熵损失函数 $\mathcal{L}_{\text{total}}$，权重梯度 $\nabla_{\mathbf{W}} \mathcal{L}_{\text{total}}$ 包含两部分：数据项梯度和正则化项梯度。

数据项梯度

数据项梯度源自交叉熵损失 $\mathcal{L}_{\text{CE}}$ 对权重矩阵 $\mathbf{W}$ 的偏导数。在softmax回归中，模型预测$\hat{\mathbf{Y}}$依赖于输入特征与权重的乘积，即$\mathbf{XW}$。根据链式法则，可以计算出数据项梯度：

$$\nabla_{\mathbf{W}}\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}(\hat{\mathbf{y}}_i - \mathbf{y}_i)\mathbf{x}_i^T$$

正则化项梯度

公式为：

$$\nabla_{\mathbf{W}}\text{Ridge} = \frac{\lambda}{N}\mathbf{W}$$

将数据项梯度和正则化项梯度合并后，得到权重梯度的计算公式：

$$\nabla_{\text{W}}\mathcal{L}_{\text{total}} = \frac{1}{N}\left(\sum_{i=1}^{N}(\hat{\mathbf{y}}_i - \mathbf{y}_i)\mathbf{X}_i^T + \lambda\mathbf{W}\right)$$

```
gradient_weights = (1 / num_samples) * np.dot(X_train.T, (y_pred - Y_train)) +
(lambda_param / num_samples) * weights
```

7.4.2 偏置梯度

偏置梯度是指损失函数关于偏置向量$\mathbf{b}$的偏导数，表示了损失函数值随偏置向量中每个元素微小变动的变化率。在softmax回归中，偏置项独立于输入特征，因此偏置梯度只包含数据项梯度，即交叉熵损失 $\mathcal{L}_{\text{CE}}$ 对偏置向量 $\mathbf{b}$ 的偏导数：

$$\nabla_{\text{b}}\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}(\hat{\mathbf{y}}_i - \mathbf{y}_i)$$

```
gradient_bias = np.mean(y_pred - Y_train, axis=0, keepdims=True)
```

7.4.3 参数更新

使用梯度下降法更新权重和偏置：

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\nabla_{\mathbf{W}}\mathcal{L}_{\text{total}}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \eta\nabla_{\text{b}}\mathcal{L}$$

符号	符号说明
η	学习率

```
weights -= learning_rate * gradient_weights
bias -= learning_rate * gradient_bias
```

7.5 accuracy函数

这个太直观了，直接上代码：

```
accuracy = np.mean(predictions == Y_test)
```

这行代码比较预测类别向量 `predictions` 与测试集实际标签向量 `Y_test`，计算两者相等（即预测正确）的元素比例。`predictions == Y_test` 会得到一个布尔型数组，表示每个样本预测是否正确。`np.mean()` 对这个布尔型数组计算平均值，由于布尔值在计算平均时被视为 `True` 为 1，`False` 为 0，所以最终得到的 `accuracy` 值就是正确预测样本数占总样本数的比例，即模型在测试集上的预测准确率。

P3 从机器学习到深度学习

Ch8 感知机与反向传播

8.1 单层感知机

定义与结构

单层感知机是一种**二元分类模型**，其核心是一个计算单元，也称为“感知机单元”。它接收一组输入特征 $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, 并结合对应的权重向量 $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ 和偏置项 b 进行运算。其输出通过激活函数 f 计算得出：

$$z = \mathbf{w} \cdot \mathbf{x} + b = w_1x_1 + w_2x_2 + \ldots + w_nx_n + b$$

$$y = f(z)$$

原始感知机使用的是阶跃函数或者符号函数作为激活函数。

工作原理与训练

单层感知机的目标是寻找一个超平面 $\mathbf{w} \cdot \mathbf{x} + b = 0$ 将数据集中的正例和负例分开。训练过程采用监督学习，通过迭代调整权重向量 w 和偏置项 b 来逐步逼近最优决策边界。当给定一个误分类点时，感知机会更新权重向量以使决策边界向该点移动一定的距离，直到所有训练样本都被正确分类为止。然而，单层感知机只能解决线性可分的问题，对于非线性可分的数据集无能为力。

8.2 多层感知机（MLP）

定义与结构

多层感知机在单层感知机基础上增加了一个或多个隐藏层。每一层由多个神经元组成， 除了输入层和输出层之外，还包括若干中间层——隐藏层。每一层的神经元接收到上一层的输出后，经过加权求和与激活函数的运算产生下一层的输入。

隐藏层神经元的计算表达式为：

$$z_j^{(l)} = \sum_{i=1}^{n^{(l-1)}} w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}$$

$$a_j^{(l)} = g(z_j^{(l)})$$

符号	符号说明
$z_j^{(l)}$	第l层第j个神经元的加权输入
$a_j^{(l)}$	激活值
g	激活函数
$w_{ji}^{(l)}$	从第$(l-1)$层第i个神经元到第l层第j个神经元的权重
$b_j^{(l)}$	第l层第j个神经元的偏置项
$n^{(l-1)}$	第$(l-1)$层神经元的数量

常见的激活函数包括但不限于 sigmoid 函数、tanh 函数、ReLU 函数等。

工作原理与训练

多层感知机通过多层非线性变换实现了对复杂模式的学习和识别。它的训练通常采用**反向传播算法**（BP），这是一种梯度下降法在神经网络中的应用。反向传播过程中，首先计算出输出层误差，然后逐层反向传播误差，更新各层权重和偏置，使得整个网络在训练集上的损失函数逐渐减小。

8.3 反向传播算法

反向传播是一个基于梯度下降法的优化过程，其目的是通过计算损失函数关于网络所有参数（包括输入层至输出层之间所有权重和偏置）的梯度，进而更新这些参数，使得模型能够更准确地拟合训练数据。

反向传播分为以下步骤：

1. 前向传播：输入数据首先通过输入层传入网络，然后在网络中逐层传递并经过激活函数作用，产生每层的输出直到最终到达输出层。
2. 计算损失：输出层的预测结果与实际标签进行比较，计算出损失函数（如交叉熵损失或均方误差）的值，这反映了模型预测的好坏程度。
3. 反向传播阶段：从输出层开始，计算损失函数相对于输出层神经元的权重和偏置的梯度。使用链式法则，将误差逆向传播回网络的每一层，计算每个隐藏层神经元的梯度。对于每一层l,计算其对应的损失函数对权重矩阵$W^{(l)}$和偏置向量$b^{(l)}$ 的梯度，表示为$\frac{\partial L}{\partial W^{(l)}}$ 和$\frac{\partial L}{\partial b^{(l)}}$。
4. 参数更新：利用计算出的梯度，应用优化算法（如梯度下降、动量梯度下降、Adam等）来更新网络中的权重和偏置。更新公式通常是这样的形式：$W^{(l)} = W^{(l)} - \alpha \frac{\partial L}{\partial W^{(l)}}$,其中$\alpha$ 是学习率，用来控制每次迭代中参数更新的步长。
5. 迭代训练：重复以上步骤直至达到预设的停止条件（如达到最大迭代次数、验证集上的损失不再显著降低或满足其他早停条件）。

反向传播的Q&A

在了解这个算法的时候我产生了以下两个问题：

1.为什么要用反向传播?

1-A：参数优化：反向传播提供了一种高效且自动的方法来计算所有这些参数的梯度，这是梯度下降法和其他优化算法所必需的信息，以便迭代地调整参数，最小化训练过程中的损失函数（代价函数）；适应非线性问题：由于神经网络可以表达复杂的非线性关系，因此适用于各种各样的任务，包括分类、回归、聚类等。反向传播正是实现这一复杂优化的关键工具。

2.为什么可以用反向传播?

2-A：链式法则：在网络的每一层，我们可以根据前一层的梯度和该层的激活函数计算出本层参数的梯度。每个神经元的输出不仅取决于自身的权重和偏置，还依赖于前一层神经元的输出。因此，要计算整个网络的全局误差对单个参数的影响，必须将这种依赖关系层层反向传导；可计算性：易于计算机程序实现和高效计算。

Ch9 手搓神经网络

9.1 回归问题

在一轮考核的回归训练集中，由于构建的是一个线性回归模型，故对应的神经网络是一种没有激活函数的单隐层神经网络。以下对代码关键部分的解释：

9.1.1 初始化线性回归神经网络

```
    def __init__(self, input_dim, output_dim, learning_rate, l2_reg):
        """
        初始化线性回归神经网络。

        参数:
        input_dim: 输入维度
        output_dim: 输出维度
        learning_rate: 学习率
        l2_reg: L2正则化系数
        """
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.learning_rate = learning_rate
        self.l2_reg = l2_reg

        # 随机初始化权重和偏置
        self.weights = np.random.randn(input_dim, output_dim)
        self.bias = np.random.randn(1, output_dim)
```

在该任务中，输入维度就是数据集中特征列的列数，输出维度就是1。在后续神经网络实例的代码中有：

```
# 创建神经网络实例并训练
net = LRNN(input_dim=X_train.shape[1], output_dim=1, learning_rate=0.01,
l2_reg=0.1)
```

9.1.2 反向传播

```
    def backward(self, X, y, output):
        """
        反向传播计算梯度。

        参数:
        X: 输入数据
        y: 真实标签
        output: 网络输出值 预测值

        返回值:
        权重梯度，偏置梯度
        """
        m = len(y)  # 样本数量
        d_weights = (1 / m) * X.T @ (output - y) + (self.l2_reg / m) *
self.weights
        d_bias = np.mean(output - y, axis=0, keepdims=True)  # 沿样本维度求和
```

```
        return d_weights, d_bias
```

该代码中分别计算了权重梯度和偏置梯度，公式和附录中的完全一样。

9.1.3 模型训练（梯度下降）

```
    def train(self, X, y, epochs):
        """
        训练模型。

        参数:
        X: 输入数据
        y: 真实标签
        epochs: 训练轮数

        返回值:
        每轮训练的损失历史记录
        """
        loss_history = []
        for _ in range(epochs):
            predictions = self.forward(X)
            d_weights, d_bias = self.backward(X, y, predictions)
            # 更新权重和偏置
            self.weights -= self.learning_rate * d_weights
            self.bias -= self.learning_rate * d_bias

            loss = self.compute_loss(y, predictions)
            loss_history.append(loss)

        return loss_history
```

`train`方法负责模型的训练过程，包括前向传播、反向传播、参数更新，并记录每轮训练的损失值。训练循环进行`epochs`次，每次循环执行以下步骤：

1. 前向传播得到预测值。
2. 反向传播计算梯度。
3. 使用梯度更新权重和偏置。
4. 计算当前批次的损失值，并将其添加到损失历史记录中。

最后返回整个训练过程中的损失历史记录。

9.2 分类问题

神经网络中的分类问题相较于回归问题，本质上多了**隐藏层**，并且引入了ReLU函数作为隐藏层的激活函数，Sigmoid作为输出层的激活函数。这些非线性激活函数引入了模型的非线性能力，使其能够处理复杂的非线性关系。相比之下，回归问题则是输入层直接连接到输出层，没有使用任何激活函数，模型保持完全线性性质。其次，二者区别还在于所选择的损失函数不同以及评估指标不同。以下是分类问题的关键代码解释：

9.2.1 初始化多层神经网络

```
class MLP:
    """
    多层感知器类，用于创建和训练一个基本的多层神经网络。

    参数:
    - input_dim: 输入维度
    - hidden_dim: 隐藏层维度
    - output_dim: 输出维度
    - learning_rate: 学习率
    """

    def __init__(self, input_dim, hidden_dim, output_dim, learning_rate):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.learning_rate = learning_rate

        # 初始化第一层权重和偏置
        self.weights1 = np.random.randn(input_dim, hidden_dim) * 0.01
        self.bias1 = np.zeros((1, hidden_dim))
        # 初始化第二层权重和偏置
        self.weights2 = np.random.randn(hidden_dim, output_dim) * 0.01
        self.bias2 = np.zeros((1, output_dim))
```

在该代码中，输入维度仍为数据集中特征列的列数，而隐藏层维度的选取则较为灵活，通常根据经验和不断试错来选择，输出层的维度仍为1，因为这是一个二分类任务，网络输出的是一个介于0和1之间的值表示样本属于正类的概率。而在多分类任务中，输出维度等于类别总数，网络输出一个概率分布向量，各元素对应各类别的概率。

9.2.2 反向传播

```
def backward(self, X, y, output, hidden_output):
    """
    反向传播算法以计算权重和偏置的梯度

    参数:
    - X: 输入数据
    - y: 真实标签
    - output: 网络的预测输出
    - hidden_output: 隐藏层的输出

    返回值:
    - d_weights1: 第一层(输入层到隐藏层)权重的梯度
    - d_weights2: 第二层(隐藏层到输出层)权重的梯度
    - d_bias1: 第一层偏置的梯度
    - d_bias2: 第二层偏置的梯度
    """
    m = y.size

    # 输出层梯度计算
    delta3 = (output - y) * self.sigmoid_d(output)
```

```
    # 隐藏层梯度计算
    delta2 = delta3.dot(self.weights2.T) * self.relu_d(hidden_output)

    # 更新权重和偏置的梯度
    d_weights2 = (hidden_output.T.dot(delta3)) / m
    d_bias2 = np.sum(delta3, axis=0, keepdims=True) / m  # 沿样本维度求和
    d_weights1 = (X.T.dot(delta2)) / m  #
    d_bias1 = np.sum(delta2, axis=0, keepdims=True) / m  # 沿样本维度求和

    return d_weights1, d_weights2, d_bias1, d_bias2
```

已经定义好的 MLP 网络，其结构为：

- 输入层：特征向量 $X \in \mathbb{R}^{N \times D}$，其中 N 为样本数，D 为输入维度。
- 隐藏层：权重矩阵$W_1 \in \mathbb{R}^{D \times H}$，偏置向量$b_1 \in \mathbb{R}^{1 \times H}$，其中$H$ 为隐藏层神经元数（隐藏层维度）。隐藏层的输出$Z_1 = XW_1 + b_1$经过 ReLU 激活函数得到 $A_1 = \mathrm{ReLU}(Z_1)$。
- 输出层：权重矩阵$W_2 \in \mathbb{R}^{H \times O}$,偏置向量$b_2 \in \mathbb{R}^{1 \times O}$,其中$O$为输出维度。输出层的输出 $Z_2 = A_1W_2 + b_2$,经过Sigmoid 激活函数得到预测输出 $\hat{Y} = \sigma(Z_2)$。

给定真实标签$Y \in \mathbb{R}^{N \times O}$，我们需要计算权重和偏置的梯度：

1. 输出层梯度计算δ_3：
    - 误差信号（或称为损失函数对输出层激活的梯度）：

      $$\delta_3 = (\hat{Y} - Y) \odot \sigma'(Z_2)$$

      其中，$\odot$表示按元素乘法，σ'是 Sigmoid 函数的导数。
2. 隐藏层梯度计算δ_2：
    - 将误差信号δ_3通过 W_2^T 传递到隐藏层，并与隐藏层激活函数的导数相乘：

      $$\delta_2 = \delta_3 W_2^T \odot \mathrm{ReLU}'(Z_1)$$

      其中，ReLU'是 ReLU 函数的导数。
3. 更新权重和偏置的梯度：（其中，L表示二分类交叉熵损失函数）
    - 输入层到隐藏层权重梯度：

      $$\frac{\partial L}{\partial W_1} = \frac{1}{N} X^T \delta_2$$

    - 输入层到隐藏层偏置梯度：

      $$\frac{\partial L}{\partial b_1} = \frac{1}{N} \sum_{i=1}^{N} \delta_2^{(i)}$$

    - 隐藏层到输出层权重梯度：

      $$\frac{\partial L}{\partial W_2} = \frac{1}{N} A_1^T \delta_3$$

    - 隐藏层到输出层偏置梯度：

      $$\frac{\partial L}{\partial b_2} = \frac{1}{N} \sum_{i=1}^{N} \delta_3^{(i)}$$

代码中使用了 `sigmoid_d` 和 `relu_d` 方法分别计算 Sigmoid 和 ReLU 函数的导数，`delta3` 和 `delta2` 分别对应上述公式的 δ_3和 δ_2，`d_weights1`、`d_weights2`、`d_bias1`、`d_bias2` 分别对应 $\frac{\partial L}{\partial W_1}$、$\frac{\partial L}{\partial W_2}$、$\frac{\partial L}{\partial b_1}$、$\frac{\partial L}{\partial b_2}$。代码中的 `m = y.size` 实际上等于样本数 N，用于在计算梯度时除以样本数进行平均。

9.2.3 模型训练（梯度下降）

```
    def train(self, X_train, y_train, iterations):
        """
        训练MLP模型

        参数:
        - X_train: 训练集的输入数据
        - y_train: 训练集的真实标签
        - iterations: 训练迭代次数

        返回值:
        - loss_history: 训练过程中损失函数的历史记录
        """
        loss_history = []

        for _ in range(iterations):
            output, hidden_output = self.forward(X_train)
            cost = self.compute_loss(y_train, output)
            d_weights1, d_weights2, d_bias1, d_bias2 = self.backward(X_train,
y_train, output, hidden_output)
            # 更新权重和偏置
            self.weights1 -= self.learning_rate * d_weights1
            self.bias1 -= self.learning_rate * d_bias1
            self.weights2 -= self.learning_rate * d_weights2
            self.bias2 -= self.learning_rate * d_bias2
            loss_history.append(cost)

        return loss_history
```

`train` 方法接收训练数据 `X_train`、标签 `y_train` 和训练迭代次数 `iterations`。在每次迭代中，执行前向传播、计算损失、反向传播并更新权重和偏置。同时，记录每轮迭代的损失值，返回整个训练过程的损失历史记录。

9.2.4 分类问题的Q&A

1.ReLU函数和Sigmoid函数在该模型中扮演了怎样的角色?

1-A:

- ReLU主要用于隐藏层，引入非线性并保持梯度的有效传播，增强模型的学习能力和表达复杂数据的能力。
- Sigmoid则用于输出层，将模型输出转化为概率形式，适用于二分类问题，并与交叉熵损失函数配合良好，便于模型训练和解释预测结果。

Ch10 循环神经网络

10.1 定义与结构

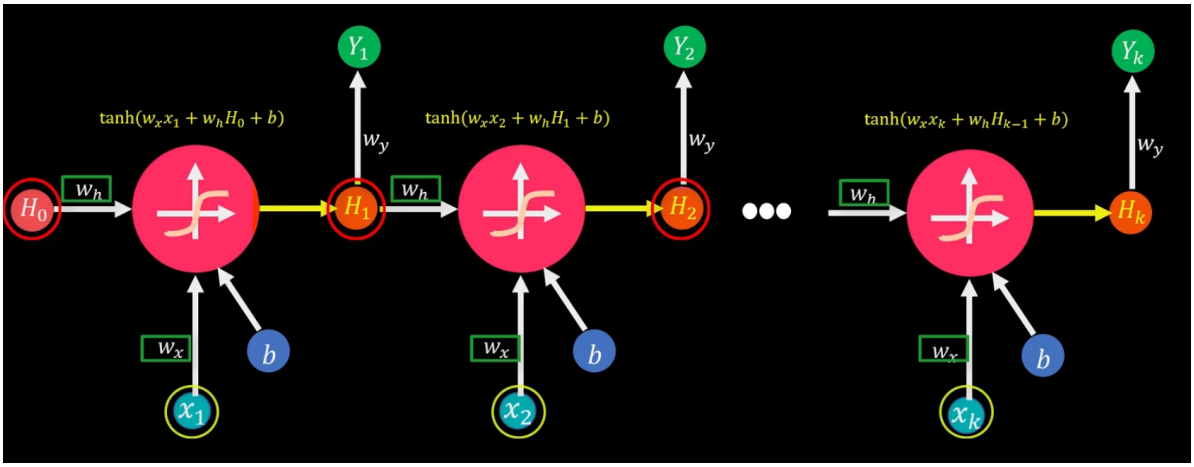
循环神经网络（Recurrent Neural Network, RNN）是一种专为处理序列数据而设计的神经网络模型。与传统的前馈神经网络（如多层感知机）不同，RNN引入了循环结构，允许信息在网络内部沿时间维度进行传递。这种特性使得RNN能够捕捉序列数据中的时间依赖关系，如文本中的语义连贯性、语音信号中的韵律特征等。

RNN的基本单元由输入层、隐藏层(含循环连接) 和输出层构成。在每个时间步 t，网络接收一个输入 x_t，同时结合前一时刻的隐藏状态 h_{t-1} 更新当前隐藏状态 h_t 并可能生成一个输出 y_t。以下是RNN的基本计算流程：

- 输入层：将输入数据x_t转换为适合网络处理的形式。在文本处理中，这通常涉及将词汇编码为**词嵌入向量。**
- 隐藏层：RNN的核心，其计算公式为：

$$h_t = \phi(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

符号	符号说明
h_t	时间步t的隐藏状态，代表网络记忆
h_{t-1}	前一时间步的隐藏状态，通过循环边传递到当前时间步
x_t	当前时间步的输入
W_{hh}	隐藏状态到隐藏状态的权重矩阵
W_{xh}	输入到隐藏状态的权重矩阵
b_h	隐藏层的偏置项
$\phi(\cdot)$	非线性激活函数



- 输出层：根据当前隐藏状态h_t生成输出y_t。对于二分类任务，输出层通常采用Sigmoid函数作为输出函数（多分类任务则采用Softmax函数）：

$$y_t = \sigma(W_{hy}h_t + b_y)$$

符号	符号说明
W_{hy}	隐藏状态到输出的权重矩阵
b_y	输出层的偏置项
$\sigma(\cdot)$	$Sigmoid$函数，将输出向量映射为概率分布

10.2 预处理

10.2.1 词嵌入加载与初始化

- **定义参数**：设置词嵌入维度（`embedding_dim`）为50，这意味着每个词将被映射到一个50维的向量空间。
- **加载GloVe词向量**：使用GloVe预训练词向量（`glove.6B.50d.txt`），为每个词分配一个50维的向量表示。
  - **词到索引映射**：创建两个字典，`word_to_index`和`index_to_word`，分别存储词到索引和索引到词的映射关系。
  - **添加特殊词**：添加`<unk>`（未知词）和`<pad>`（填充词），它们分别对应未在词向量文件中出现的词和用于填充序列的词。
  - **读取词向量**：逐行读取GloVe文件，将词及其对应的向量值存储到`word_to_index`和`word_embeddings`列表中。
  - **添加特殊词向量**：为`<unk>`和`<pad>`添加全零向量作为词嵌入。
  - **构建词嵌入层**：将`word_embeddings`列表转换为张量，使用`torch.nn.Embedding.from_pretrained`创建预训练词嵌入层，并移动到设备上。

```
# 读取GloVe词向量
# 初始化词到索引、索引到词的映射以及词嵌入列表
word_to_index = {}
index_to_word = {}
word_embeddings = []

# 添加特殊词'<unk>'和'<pad>'
word_to_index['<unk>'] = len(word_to_index)
index_to_word[len(word_to_index) - 1] = '<unk>'
word_to_index['<pad>'] = len(word_to_index)
index_to_word[len(word_to_index) - 1] = '<pad>'

# 从文件读取GloVe向量并填充词嵌入列表
with open("E:/二轮考核/glove.6B.50d.txt", 'r', encoding='utf-8') as f:
    for line in f:
        values = line.split()
        word = values[0]
        coefs = torch.tensor([float(x) for x in values[1:]])
        word_to_index[word] = len(word_to_index)
        index_to_word[len(word_to_index) - 1] = word
        word_embeddings.append(coefs)

# 添加'<unk>'和'<pad>'的词嵌入
word_embeddings.append(torch.zeros(embedding_dim))  # '<unk>'词嵌入
word_embeddings.append(torch.zeros(embedding_dim))  # '<pad>'词嵌入
# 将词嵌入列表转换成张量并移动到指定设备上
word_embeddings = torch.stack(word_embeddings)
word_embeddings = torch.nn.Embedding.from_pretrained(word_embeddings).to(device)
```

词嵌入的Q&A

1.词嵌入是什么？为什么需要词嵌入？

1-A：将语言数字化的这个过程叫做 Word Embedding，中文名称叫做 “词嵌入”， 而转化后获得到的向量矩阵就叫做词向量， 其实就是词的数学表示。独热编码是最简单的一种单词表示法，但这种方法不能表示单词间语义的相似度，且具有稀疏性，向量中只有一个位置是1，是有用的，其他位置全是0，十分浪费空间。而代码中所使用到的是Glove预训练集，这种预训练向量为模型提供了一个良好的初始条件，有利于更快收敛并提升模型的性能。如果不采用类似的预训练集，则可以采用随机初始化词向量的方法。

10.2.2 数据集的预处理

读取IMDb数据集

定义了一个名为 read_imdb_data 的函数，用于读取IMDb数据集。该函数接受一个参数 directory，表示数据集的根目录。数据集分为两个子目录：“pos”（正面评价）和“neg”（负面评价）。函数遍历这两个子目录中的每个文件，打开文件并读取其内容（一条影评），存储为字符串 review。同时，根据子目录名称（pos 或 neg），为该影评分配一个标签（1表示正面，0表示负面）。将 (review, label) 元组添加到结果列表 data 中。最后返回包含所有影评及其标签的 data 列表。

```
def read_imdb_data(directory):
    data = []
    for label in ["pos", "neg"]:
        for file in os.listdir(os.path.join(directory, label)):
            with open(os.path.join(directory, label, file), 'r', encoding='utf-
8') as f:
                review = f.read().strip()
                data.append((review, 1 if label == "pos" else 0))
    return data
```

数据预处理

定义了一个名为 preprocess_data 的函数，用于对数据进行预处理。该函数接受三个参数：原始数据列表 data、词到索引的映射 word_to_index 和最大序列长度max_sequence_length。
对于数据列表中的每一条影评（(review, label) 元组）：

- 将影评文本 review 按空格拆分成单词列表 words。
- 截取前 max_sequence_length 个单词（若不足则取全部）。
- 使用 word_to_index 映射将单词列表转换为索引列表 indices。对于不在映射中的单词，使用 的索引代替。
- 根据 max_sequence_length 对索引列表进行填充：在末尾添加足够数量的 索引来使序列长度达到 max_sequence_length，得到 padded_indices。
- 将处理后的索引列表 padded_indices 和标签 label 转换为PyTorch张量。将 (padded_indices_tensor, label_tensor) 元组添加到结果列表 padded_data 中。

```
def preprocess_data(data, word_to_index, max_sequence_length):
    padded_data = []
    for review, label in data:
        words = review.split()
        words = words[:max_sequence_length]
        indices = [word_to_index[word] if word in word_to_index else
word_to_index['<unk>'] for word in words]
        padded_indices = indices + [word_to_index['<pad>']] *
(max_sequence_length - len(indices))
        padded_data.append((torch.tensor(padded_indices).to(device),
torch.tensor(label).to(device)))
    return padded_data
```

10.3 RNN中的前向传播

1. 首先，通过预训练词嵌入层将输入的词索引序列x转换为词嵌入表示。
2. 将嵌入后的词序列输入到 self.rnn 层（GRU层）进行计算。output 是整个序列的输出，hidden 是最后一个时间步的隐藏状态。这里仅使用最后一个隐藏状态作为后续全连接层的输入。
3. 将最后一个RNN隐藏状态通过 self.fc（全连接层）计算得到模型的 logits 输出。Logits 是未经激活函数处理的预测值，用于后续计算损失函数。
4. 最后对 logits 应用 sigmoid 激活函数，得到介于0和1之间的概率值，表示输入文本属于正向情感（标签为1）的概率。返回此概率值作为模型的预测输出 out。

```
    def forward(self, x):
        embedded = self.embedding(x)
        output, hidden = self.rnn(embedded)
        logits = self.fc(hidden[-1])
        return torch.sigmoid(logits)
```

10.4 RNN中的BPTT（随时间反向传播）

在循环神经网络中，随时间反向传播（Backpropagation Through Time, BPTT）是一种专门针对时间序列数据进行梯度计算的方法。BPTT通过展开网络在时间维度上的计算过程，将传统的反向传播算法应用于序列的每一个时间步，以更新网络的权重和偏置。

符号约定

- 时间步t: 表示序列中的某一特定时刻。
- 输入 $\mathbf{x}_t \in \mathbb{R}^{d_x}$: 在时间步$t$的输入向量，维度为$d_x$。
- 隐藏状态$\mathbf{h}_t \in \mathbb{R}^{d_h}$: 在时间步$t$的隐藏层状态，维度为$d_h$。
- 输出$\mathbf{o}_t \in \mathbb{R}^{d_o}$: 在时间步$t$的输出向量，维度为$d_o$。
- 预测值$\hat{\mathbf{y}}_t$: 由输出层计算得到的对真实目标值的预测，如经过激活函数后的结果。
- 真实目标值$\mathbf{y}_t$: 在时间步t的真实标签或目标值。
- 权重矩阵:
  - $\mathbf{U} \in \mathbb{R}^{d_h \times d_x}$: 输入到隐藏层的权重矩阵。
  - $\mathbf{W} \in \mathbb{R}^{d_h \times d_h}$: 前一时刻隐藏状态到当前隐藏状态的权重矩阵（循环权重）。
  - $\mathbf{V} \in \mathbb{R}^{d_o \times d_h}$: 隐藏状态到输出层的权重矩阵。
- 偏置向量:

- $\mathbf{b}_h \in \mathbb{R}^{d_h}$: 隐藏层的偏置向量。
- $\mathbf{b}_o \in \mathbb{R}^{d_o}$: 输出层的偏置向量。

- 激活函数 $\sigma(\cdot)$: 应用于隐藏层的非线性激活函数，如$\tanh$或ReLU。
- 输出激活函数$g(\cdot)$: 应用于输出层的激活函数，如线性激活（恒等映射）、softmax 或 sigmoid。
- $\mathcal{L}$：交叉熵损失函数。

初始化权重梯度

BPTT的核心在于计算损失函数关于所有权重矩阵和偏置向量的梯度，以便通过梯度下降等优化方法更新网络参数。

对于所有权重矩阵 $\mathbf{U}, \mathbf{W}, \mathbf{V}$ 和偏置向量 $\mathbf{b}_h, \mathbf{b}_o$ ，首先要初始化梯度为零：

$$\frac{\partial \mathcal{L}}{\partial \mathbf{U}} \leftarrow \mathbf{0}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \leftarrow \mathbf{0}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{V}} \leftarrow \mathbf{0}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} \leftarrow \mathbf{0}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_o} \leftarrow \mathbf{0}$$

反向传播

从序列末尾$t = T$开始，逆序遍历每个时间步t，执行以下操作：

1. 计算输出梯度（⊙表示逐元素乘法）

$$\frac{\partial \mathcal{L}}{\partial \mathbf{o}_t} = \frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}_t} \odot g'(\mathbf{o}_t)$$

2. 计算隐藏状态梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{o}_t} \cdot \mathbf{V}^\top + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}} \cdot \mathbf{W}^\top$$

   其中 $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t+1}}$ 是下一个时间步的隐藏状态梯度，对于 $t = T$ 初始为零。

3. 计算权重和偏置梯度
   - 输入到隐藏层权重矩阵$\mathbf{U}$的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{U}} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{U}} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \cdot \mathbf{x}_t^\top$$

- 隐藏层循环权重矩阵$\mathbf{W}$的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{W}} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \cdot \mathbf{h}_{t-1}^\top$$

   - 隐藏层到输出层权重矩阵$\mathbf{V}$的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{V}} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{V}} + \frac{\partial \mathcal{L}}{\partial \mathbf{o}_t} \cdot \mathbf{h}_t^\top$$

   - 隐藏层偏置向量$\mathbf{b}_h$的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{b}_h} + \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$$

   - 隐藏层偏置向量$\mathbf{b}_o$的梯度

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_o} \leftarrow \frac{\partial \mathcal{L}}{\partial \mathbf{b}_o} + \frac{\partial \mathcal{L}}{\partial \mathbf{o}_t}$$

4. 参数更新：使用计算出的梯度，根据选定的优化器（如SGD、Adam等）更新权重和偏置。

10.5 F1分数

在IMDb数据集二分类的模型中，我采用了F1分数作为模型的评估指标。

F1分数提供了一种综合评价模型精确率（Precision）和召回率（Recall）的方法，这两个指标分别从不同的角度衡量模型的表现：

精确率

指分类器预测为正类的样本中，实际为正类的比例。精确率反映了模型在识别出的正类样本中，正确分类的程度。其数学公式为：

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

召回率

又称查全率，是指实际为正类的样本中，被模型正确识别出来的比例。召回率体现了模型找出所有真正正类的能力。其数学表达式为：

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

F1分数

F1分数是精确率和召回率的调和平均数，其设计目的是在单个数值上平衡精确率和召回率的权重，以提供一个更为全面的模型评估标准。F1分数的计算公式为：

$$\text{F1 Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

F1分数具有以下几个关键特征：

- 取值范围介于0到1之间。当F1分数等于1时，意味着模型具有完美的精确率和召回率，即所有的正类样本都被正确识别且没有假阳性（False Positive）。反之，当F1分数为0时，表示模型在精确率和召回率上都表现极差。
- 对称性：F1分数对精确率和召回率赋予了相同的权重，这使得它在精确率和召回率同等重要的场景中非常有用。如果需要调整这两者之间的权重关系，可以使用F_β分数，其中β参数允许对召回率赋予相对于精确率的不同权重，当$\beta > 1$时，其更加强调召回率的重要性，适用于那些对漏检（False Negative）惩罚更严厉的应用场景。当$0 < \beta < 1$时，则更加看重精确率，适用于对误报（False Positive）敏感的应用环境。其公式为：

$$\text{F}_\beta \text{ Score} = \frac{(1 + \beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}}$$

- 适用场景：在类别不平衡问题中，单独使用精确率或召回率可能无法全面反映模型的实际性能，因为它们可能受到多数类主导的影响。此时，F1分数由于同时考虑了两类错误（假阳性和假阴性），能够更公正地评估模型在正类识别上的整体效果。

```
# 在测试集上计算F1分数
    model.eval()  # 将模型设置为评估模式

    # 初始化用于计算F1分数的变量
    total_true_positive = 0
    total_predicted_positive = 0
    total_actual_positive = 0
```

```
    # 关闭梯度计算以提高效率
    with torch.no_grad():
        for reviews, labels in test_loader:
            # 使用模型进行预测
            outputs = model(reviews)
            predicted = (outputs > 0.5).float().view(-1)  # 将输出大于0.5的预测为正
类
            true_positive = (predicted * labels).sum().item()  # 计算真正例的数量
            total_true_positive += true_positive
            total_predicted_positive += predicted.sum().item()  # 计算总预测正例的数
量
            total_actual_positive += labels.sum().item()  # 计算总真实正例的数量

    # 计算精确度、召回率和F1分数
    precision = total_true_positive / total_predicted_positive if
total_predicted_positive > 0 else 0
    recall = total_true_positive / total_actual_positive if
total_actual_positive > 0 else 0
    f1_score = 2 * (precision * recall) / (precision + recall) if precision +
recall > 0 else 0

    # 将计算得到的F1分数存储起来
    train_f1_scores.append(f1_score)
```

10.6 RNN的两种重要改进模型直观理解

RNN的两种重要改进模型：长短期记忆网络（Long Short-Term Memory, LSTM）和门控循环单元（Gated Recurrent Unit, GRU）。这两种模型都是为了克服传统RNN在处理长序列时可能出现的梯度消失/爆炸问题，并增强捕捉长期依赖关系的能力。**本小节仅对LSTM和GRU的原理作直观的理解，附有简化版的公式，而不深入到数学公式。**

10.6.1 RNN的问题

在计算损失函数相对于较早时间步隐藏状态的梯度时，需要通过链式法则逐层回溯。对于时间步$t-k\,(k>0)$，其隐藏状态h_{t-k}对损失函数的影响体现在：

$$\frac{\partial \mathcal{L}}{\partial h_{t-k}} = \frac{\partial \mathcal{L}}{\partial h_t} \prod_{j=t-k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}$$

其中$\frac{\partial h_j}{\partial h_{j-1}}$是相邻时间步之间隐藏状态的梯度传递，其具体形式为：

$$\frac{\partial h_j}{\partial h_{j-1}} = W_{hh}^{\top} \sigma'(W_{ih}x_j + W_{hh}h_{j-1})$$

符号	符号说明
W_{ih}	输入到隐藏层的权重矩阵

由于$\sigma'(\cdot)$对于sigmoid或tanh函数在大部分输入值范围内取值接近于0或接近于1的小数值，当序列较长时，上述连乘项中包含多个这样的小数乘积，容易导致整体梯度值迅速减小，即梯度消失现象。特别是对于sigmoid函数，其导数的最大值仅为0.25，这意味着每次时间步间的梯度传递至少会缩小至原来的四分之一。随着序列长度的增长，这种缩放效应呈指数级加剧，最终导致早期时间步的梯度几乎为零，模型无法有效地从这些时间步学到任何信息。

10.6.2 长短期记忆网络

LSTM（Long Short-Term Memory）网络通过引入独特的细胞状态（cell state）和多层门控机制，有效解决了RNN在处理长序列时的梯度消失问题。

细胞状态

LSTM引入了一个名为细胞状态（记作$\mathbf{c}$）的特殊载体，它在时间步间直接线性传递，独立于隐藏状态（$\mathbf{h}$）。细胞状态扮演了长期记忆的角色，能够跨越多个时间步持久地存储和传递重要信息，不受梯度消失的影响。

门控机制

LSTM的关键创新在于使用了三个门控单元来精细控制细胞状态的更新过程，确保信息的有选择性保留和添加。这些门控包括：

- 遗忘门（Forget Gate）：决定细胞状态中哪些信息应当被遗忘。遗忘门的输出（记作$\mathbf{f}$）是一个在[0, 1]之间的向量，通过sigmoid激活函数生成。值接近0表示强烈遗忘，接近1表示强烈保留。遗忘门的计算公式为（省略了权重矩阵和偏置项以简化表述）：$\mathbf{f} = \sigma(\text{当前输入} + \text{前一隐藏状态})$
- 输入门（Input Gate）：决定细胞状态中哪些新信息应当被添加。输入门的输出（记作$\mathbf{i}$）同样是[0, 1]之间的向量，通过sigmoid激活函数生成。值接近0表示阻止新信息加入，接近1表示允许新信息加入。同时，LSTM还计算一个新的候选细胞状态（记作$\tilde{\mathbf{c}}$），通过tanh激活函数生成，表示可能的新信息。输入门和候选细胞状态的计算公式为：$\mathbf{i} = \sigma(\text{当前输入} + \text{前一隐藏状态})$，$\tilde{\mathbf{c}} = \tanh(\text{当前输入} + \text{前一隐藏状态})$
- **更新细胞状态**：基于遗忘门和输入门的输出，LSTM更新细胞状态。遗忘门决定从上一时刻细胞状态中丢弃多少信息，输入门决定从候选细胞状态中添加多少新信息。更新公式为：$\mathbf{c}_{\text{new}} = \mathbf{f} \odot \mathbf{c}_{\text{prev}} + \mathbf{i} \odot \tilde{\mathbf{c}}$。通过这种方式，LSTM能够有选择性地保留旧信息并添加新信息，不受梯度消失影响。

隐藏状态生成与输出

LSTM还引入了一个输出门（记作$\mathbf{o}$），决定细胞状态中哪些信息应作为隐藏状态输出给后续层或作为模型的最终输出。输出门的计算类似于遗忘门和输入门，通过sigmoid激活函数生成[0, 1]之间的向量。隐藏状态（记作$\mathbf{h}$）通过tanh激活函数对细胞状态进行非线性转换，然后与输出门的输出进行逐元素乘法得到：$\mathbf{o} = \sigma(\text{当前输入} + \text{前一隐藏状态})$，$\mathbf{h} = \mathbf{o} \odot \tanh(\mathbf{c}_{\text{new}})$

LSTM如何防止梯度消失

LSTM通过以下方式避免梯度消失问题：

- **细胞状态的直接传递**：细胞状态在时间步间直接线性传递，不受非线性激活函数的影响，因此梯度可以无衰减地回传。
- **门控机制的非饱和非线性激活**：遗忘门、输入门和输出门使用sigmoid激活函数，其导数在全域内非零，避免了梯度消失。同时，候选细胞状态使用tanh激活函数，其导数在大部分输入范围内接近1，也降低了梯度消失的可能性。

10.6.3 门控循环单元

GRU（Gated Recurrent Unit）是一种简化版的LSTM，同样旨在解决RNN在处理长序列时的梯度消失问题。GRU通过合并遗忘门和输入门为单一的更新门，并且将细胞状态与隐藏状态合二为一，实现了一种更为简洁但高效的循环神经网络结构。

更新门

GRU引入了更新门（记作$\mathbf{z}$），它决定了上一时刻的隐藏状态（记作$\mathbf{h}_{\text{prev}}$）中哪些信息应当被保留下来用于更新当前隐藏状态。更新门的输出（在[0, 1]之间）由sigmoid激活函数生成，表示对旧信息的保留程度。计算公式为（省略了权重矩阵和偏置项以简化表述）:$\mathbf{z} = \sigma(\text{当前输入} + \text{前一隐藏状态})$

重置门

GRU还引入了重置门（记作$\mathbf{r}$），它决定了上一时刻的隐藏状态中哪些信息应当被忽略，以便更好地整合当前时刻的新输入。重置门的输出也是[0, 1]之间的向量，由sigmoid激活函数生成。计算公式为：$\mathbf{r} = \sigma(\text{当前输入} + \text{前一隐藏状态})$

候选隐藏状态

基于重置门的输出，GRU计算一个候选隐藏状态（记作$\tilde{\mathbf{h}}$），该状态结合了部分忽略旧信息后的前一隐藏状态和当前时刻的输入。计算过程中通常使用tanh激活函数以获得非线性变换。公式为：$\tilde{\mathbf{h}} = \tanh(\text{当前输入} \odot \mathbf{r} + \text{前一隐藏状态} \odot (1 - \mathbf{r}))$，重置门的作用体现在通过乘以$(1 - \mathbf{r})$部分地“重置”旧隐藏状态，使模型能够更专注于当前时刻的新输入。

更新隐藏状态

最后，GRU结合更新门的输出和候选隐藏状态来更新当前隐藏状态。更新门决定如何以加权的方式混合旧隐藏状态和新的候选隐藏状态。更新公式为：$\mathbf{h}_{\text{new}} = (1 - \mathbf{z}) \odot \mathbf{h}_{prev} + \mathbf{z} \odot \tilde{\mathbf{h}}$

GRU如何防止梯度消失:

GRU通过以下方式避免梯度消失问题：

- **更新门与重置门的非饱和非线性激活**：更新门和重置门均使用sigmoid激活函数，其导数在全域内非零，保证了梯度回传过程中不会因饱和导致消失。
- **候选隐藏状态的tanh激活**：候选隐藏状态使用tanh激活函数，其导数在大部分输入范围内接近1，减少了梯度消失的风险。
- **隐藏状态更新方式**：GRU通过更新门直接控制旧隐藏状态与候选隐藏状态的加权融合，避免了复杂的细胞状态分离操作，使得梯度能够更顺畅地反向传播。

Ch11 卷积神经网络

11.1 定义与结构

卷积神经网络（Convolutional Neural Network, CNN）是一种特别适用于处理具有空间或时间相关性数据（如图像、视频、语音等）的深度学习模型。其核心在于运用卷积运算捕捉数据局部特征，并通过逐层抽象构建更为复杂的高层特征，实现对输入数据的高效识别与分类。

以下是CNN的基本组成单元：

卷积层

卷积层是CNN的基本构建块，其核心是通过可学习的滤波器（或称卷积核、权重矩阵）W与输入数据（如图像）X进行卷积操作。具体而言，对于二维图像，卷积公式如下：

$$s(i,j) = (X * W)(i,j) = \sum_m \sum_n x(i+m, j+n) w(m,n)$$

符号	符号说明
$s(i,j)$	输出特征图在位置(i,j)处的值
$x(i+m,j+n)$	输入图像在相对于当前位置(i,j)偏移量为(m,n)处的像素值
$w(m,n)$	卷积核在对应位置(m,n)的权重值。
m,n	指定卷积核内权重的位置索引

卷积还有其他操作细节：

- 步长（Stride，S）：卷积核在输入图像上滑动的间隔，决定了特征图的分辨率。通常取值为1或大于1的整数。
- 填充（Padding，P）：在输入图像边缘添加额外像素（通常为0），以控制输出特征图的大小，保持空间信息或避免边缘信息丢失。有效填充p的计算公式为：

  $$p = \frac{F-1}{2}$$

  其中，F为卷积核大小。

除此以外，若输入数据有多通道（如RGB图像有三个通道），每个通道对应一个卷积核进行卷积操作，生成相应通道的输出特征图。

激活函数层

紧随卷积层之后，引入非线性变换，使网络能够学习更复杂的特征表示。常见的激活函数包括ReLU（Rectified Linear Unit）、Sigmoid、Tanh等。公式略。

池化层

池化操作（如最大池化Max Pooling）对特征图进行下采样，减小空间维度，降低计算复杂度，同时增强模型对输入数据平移、旋转等变化的鲁棒性。

最大池化公式：

$$p(i,j) = \max_{k,l \in D} s(i \times s + k, j \times s + l)$$

符号	符号说明
$p(i,j)$	池化后的特征图在位置(i,j)处的值
$s(i \times s + k, j \times s + l)$	对应池化窗口内部子区域的最大值
D	池化窗口大小，通常为正方形
s	步长，即池化窗口移动的间隔

全连接层

位于CNN的尾部，将前一层的特征向量映射到分类任务所需的输出维度。每个神经元与前一层所有神经元全连接，实现全局特征的整合与分类。假设前一层输出为H，全连接层神经元个数为N，则权重矩阵W大小为$H \times N$，偏置向量b大小为$N \times 1$。对于每个输出神经元y_i，其计算公式为：

$$y_i = \sigma(\sum_{j=1}^{H} W_{ij} x_j + b_i)$$

其中，σ为激活函数，x_j为前一层第j个神经元的输出，W_{ij}为权重，b_i为偏置。

其他重要概念与技术

- 参数共享：同一卷积核在输入图像的所有位置重复应用，显著减少模型参数数量，有利于模型训练，并增强对平移不变性的学习能力。
- 批量归一化：在卷积层或全连接层之后添加，通过对每一层的输入进行标准化处理（均值为0，方差为1），有效缓解内部协变量偏移问题，加速训练过程，提高模型泛化性能。

11.2 预处理

以下是CNN中常用的图像预处理方法：

1. **图像尺寸调整（Resizing）**
    - **目的**：统一输入图像的尺寸，使之符合网络模型的输入要求。
    - **方法**：
        - **保持纵横比**：按比例缩放图像，保持原始形状，但可能需要填充（padding）或裁剪（cropping）以适应模型尺寸。
        - **固定尺寸**：直接调整图像至指定大小，可能导致图像变形（如拉伸或压缩），或者采用中间裁剪（center cropping）保留图像中心部分。
2. **数据增强（Data Augmentation）**
    - **目的**：增加训练数据的多样性，减少模型对特定图像变换的敏感性，提升模型的泛化能力。
    - **方法**：
        - **几何变换**：随机旋转、翻转（水平或垂直）、缩放、裁剪、平移等。
        - **色彩变换**：亮度调整、对比度变化、色调饱和度变化、随机噪声注入（如椒盐噪声、高斯噪声）等。
3. **归一化（Normalization）**
    - **目的**：将图像像素值调整到相似尺度，加速模型收敛，消除光照、传感器差异等因素的影响。
    - **方法**：
        - **像素值归一化**：将像素值从原始范围（如0-255）缩放到[0, 1]或[-1, 1]之间，通常通过除以255或乘以相应系数实现。
        - **均值减法**（Mean Subtraction）：将每个像素值减去训练集所有图像对应通道的平均值，有时还会除以标准差进行标准化。
        - **Batch Normalization**：虽然通常作为模型内部层的一种正则化技术，但在预处理阶段，有时也会在输入图像上直接应用批量归一化。
4. **灰度化（Grayscale Conversion）**
    - **目的**：简化图像信息，减少模型处理的维度，适用于对颜色不敏感的任务。
    - **方法**：将彩色图像转换为单通道灰度图像，通常通过计算各颜色通道的加权平均。
5. **去噪（Noise Reduction）**
    - **目的**：去除图像中的噪声干扰，提高图像质量。
    - **方法**：应用滤波器，如高斯滤波、中值滤波、双边滤波等。
6. **直方图均衡化（Histogram Equalization）**
    - **目的**：增强图像的整体对比度，尤其对于光照不均匀的图像。
    - **方法**：通过调整像素值分布，使输出图像具有更均匀的直方图。
7. **图像增强（Image Sharpening）**
    - **目的**：提升图像边缘清晰度，增强细节表现。
    - **方法**：应用锐化滤波器，如拉普拉斯算子、Unsharp Mask等。

```
# 数据预处理
transform_train = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

transform_test = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

11.3 CNN中的前向传播

11.3.1 卷积层的前向传播

假设输入图像为X（大小为$H_x \times W_x \times C_x$，其中$H_x, W_x$分别为高度和宽度，$C_x$为通道数），卷积核为$W$（大小为$H_w \times W_w \times C_x \times C_y$，其中$C_y$为输出通道数），步长（Stride）为$S$，填充（Padding）为$P$。对于每个输出通道，卷积操作可表示为：

$$s_{c,y,x} = \sum_{c'=0}^{C_x-1} \sum_{m=0}^{H_w-1} \sum_{n=0}^{W_w-1} x_{c',y\cdot S+m,x\cdot S+n} \cdot w_{c',c,m,n} + b_c$$

符号	符号说明
$s_{c,y,x}$	输出特征图在通道c、位置(y, x)的值
$x_{c',y\cdot S+m,x\cdot S+n}$	输入图像在通道c'、相对于当前位置偏移量为(m, n)的像素值
$w_{c',c,m,n}$	卷积核在输入通道c'、输出通道c、位置(m, n)的权重
b_c	输出通道c对应的偏置项

输出特征图的尺寸计算

$$H_y = \frac{H_x + 2P - H_w}{S} + 1$$
$$W_y = \frac{W_x + 2P - W_w}{S} + 1$$

11.3.2 激活层的前向传播

对于每个卷积层输出特征图的元素，应用选定的激活函数（如ReLU、Sigmoid、Tanh等）。以ReLU为例：

$$a_{c,y,x} = \max(0, s_{c,y,x})$$

11.3.3 池化层的前向传播

池化层对特征图进行下采样，减小空间维度，降低计算复杂度，同时增强模型对输入数据平移、旋转等变化的鲁棒性。以最大池化为例，池化窗口大小为$H_k \times W_k$，步长为S。对于每个输出特征图的位置：

$$p_{c,y,x} = \max_{m=0}^{H_k-1} \max_{n=0}^{W_k-1} a_{c,y\cdot S+m,x\cdot S+n}$$

11.3.4 全连接层的前向传播

全连接层（Fully Connected Layer, FC）接收前一层展平的特征向量作为输入，将其映射到分类任务所需的输出维度。每个神经元与前一层所有神经元全连接，实现全局特征的整合与分类。假设前一层展平输出为$\mathbf{a}$，全连接层神经元个数为N，权重矩阵为$\mathbf{W}$（大小为$N \times L$，其中L为前一层输出长度），偏置向量为$\mathbf{b}$（大小为$N \times 1$）。对于每个输出神经元y_i，其计算公式为：

$$y_i = \sigma\left(\sum_{j=1}^{L} W_{ij} a_j + b_i\right)$$

其中，σ为激活函数（如ReLU、Sigmoid等），a_j为前一层第j个神经元的输出，W_{ij}为权重，b_i为偏置。

11.4 CNN中的反向传播

11.4.1 全连接层的反向传播

全连接层的反向传播基于链式法则，计算损失函数对权重矩阵$\mathbf{W}$和偏置向量$\mathbf{b}$的梯度。

权重梯度

$$\nabla_{\mathbf{W}} L = \frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{W}}$$

符号	符号说明
$\mathbf{a}$	全连接层的激活输出
$\frac{\partial L}{\partial \mathbf{a}}$	损失函数对激活输出的梯度(通常称为误差信号)
$\frac{\partial \mathbf{a}}{\partial \mathbf{W}}$	激活输出对权重矩阵的雅可比矩阵

偏置梯度

$$\nabla_{\mathbf{b}} L = \frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{b}}$$

符号	符号说明
$\frac{\partial \mathbf{a}}{\partial \mathbf{b}}$	激活输出对偏置向量的梯度

11.4.2 激活层的反向传播

激活层的反向传播计算损失函数对上一层输出的梯度，即误差信号通过激活函数的梯度传递。

对于激活函数f(如ReLU)，其输入为z，输出为$\mathbf{a} = f(z)$，损失函数对上一层输出的梯度为：

$$\nabla_z L = \frac{\partial L}{\partial z} = \frac{\partial L}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial z} = \frac{\partial L}{\partial \mathbf{a}} \cdot f'(z)$$

11.4.3 池化层的反向传播

池化层的反向传播计算损失函数对上一层输出的梯度，通常基于最大池化的“最大值指示器”性质或平均池化的简单平均性质。

以最大池化为例，池化窗口大小为$H_k \times W_k$，步长为S，有：

$$\nabla_{a_{c,y,x}} L = \begin{cases} \frac{\partial L}{\partial p_{c,y,x}} & \text{if } a_{c,y\cdot S+m,x\cdot S+n} = \max_{m',n'} a_{c,y\cdot S+m'x\cdot S+n'}, \\ 0 & \text{otherwise} \end{cases}$$

符号	符号说明
$a_{c,y,x}$	上一层输出在通道c、位置(y, x)的值
$p_{c,y,x}$	池化层输出在对应位置的值
$\frac{\partial L}{\partial p_{c,y,x}}$	从池化层输出到损失函数的梯度

11.4.4 卷积层的反向传播

卷积层的反向传播基于卷积运算的反向（互相关）性质，计算损失函数对卷积核和上一层输出的梯度。

卷积核梯度

$$\nabla_{w_{c',c,m,n}} L = \frac{\partial L}{\partial w_{c',c,m,n}} = \sum_{y=0}^{H_y-1} \sum_{x=0}^{W_y-1} \frac{\partial L}{\partial s_{c,y,x}} \cdot x_{c',y\cdot S+m,x\cdot S+n}$$

符号	符号说明
$\frac{\partial L}{\partial s_{c,y,x}}$	从卷积层输出到损失函数的梯度
$x_{c',y\cdot S+m,x\cdot S+n}$	输入图像在对应位置的值

上一层输出梯度

$$\nabla_{x_{c',y',x'}} L = \sum_{c=0}^{C_y-1} \sum_{m=0}^{H_w-1} \sum_{n=0}^{W_w-1} \frac{\partial L}{\partial s_{c,y,x}} \cdot w_{c',c,m,n}$$

符号	符号说明
y'	$y' = \lfloor \frac{y-m}{S} \rfloor$
x'	$x' = \lfloor \frac{x-n}{S} \rfloor$