实 验 报 告

实验名称： 多项式回归与正则化

课程名称： 机器学习实验

院系：人工智能学院

班级：2021 级机器学习 A 班

姓名：

学号：

日期：2024 年 3 月 29 日

中山大学

一、 实验内容

本次实验的任务主要是回顾上次实验线性回归的知识，并且了解多项式回归与正则化的相关知识以及相关使用。本次的实验内容主要有以下三个任务：

(1) 自行完成 MiniBatch 梯度下降方法的代码，得到参数路径 θ，并且在 1.2.4 小节中进行路径对比，思考 1.2.4 中的问题，如何选择 batch 的数量。

(2) 自行完成 2.1 小节中度数为 1，2，100 的设置，绘图观察结果并加以分析。

(3) 分析 2.3 小节，从训练误差、验证误差、误差差异角度分析图，并总结过拟合的标志。

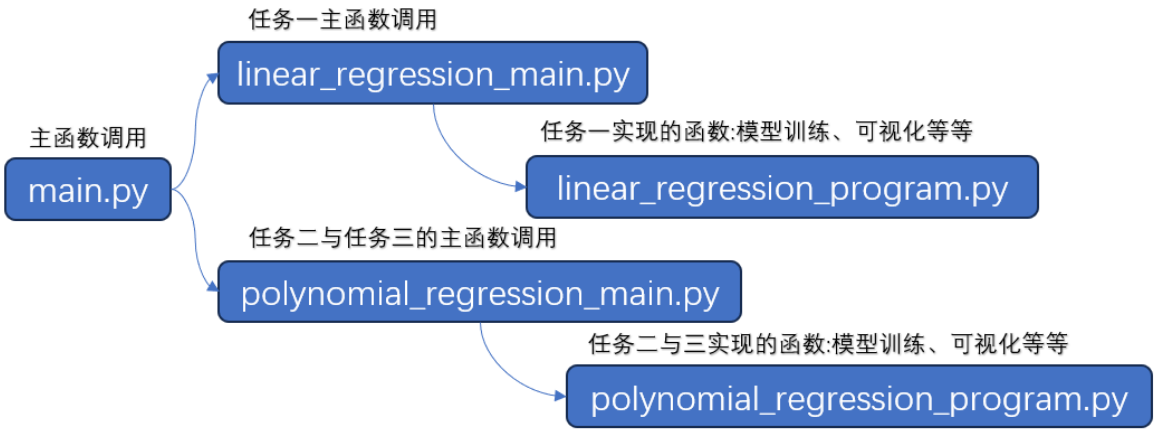
二、 实验环境

操作系统：Windows 11 Version 23H2

Python 版本：Python 3.11.4 ('base':conda)

三、 实验步骤

本次作业文件共有五个 python 程序，其中的逻辑关系如下：



运行

执行所有任务

```
python main.py
```

执行任务一

```
python linear_regression_main.py
```

3.1 MiniBatch 梯度下降法实现

在实现 MiniBatch 之前，我实现了批梯度下降(BGD)和随机梯度下降(SGD)，这两个优化方法由于篇幅原因不展开介绍实现逻辑，主要说明 MiniBatch 的实现逻辑。批梯度下降法、Minibatch 以及随机梯度下降法都是优化算法中用于迭代求解模型参数的方法，它们的主要区别在于每次迭代时使用的样本数量。以下我将对这三种方法及其优缺点进行详细地解释：

批量梯度下降法(Batch Gradient Descent，BGD)，使用全部样本来计算梯度，所以每次迭代的方向都更为准确，能够更稳定地收敛到局部最小值。计算量大，尤其是在处理大规模数据集时，每次迭代都需要计算所有样本的梯度，导致训练过程非常慢。但是对于非凸优化问题，可能会陷入局部最小值，而非全局最小值。

小批量梯度下降法(Mini-Batch Gradient Descent，MBGD)，使用部分的样本来计算梯度，相对于 SGD，每次迭代使用更多的样本，使得梯度的估计更为准确，训练过程更为稳定。通过并行处理多个批次的数据，可以加快模型的训练速度。同时也减少了内存的占用，因为在每次训练中只需要将当前批次的数据加载到内存中。需要选择合适的批次大小，过大可能导致内存溢出或训练速度变慢，过小可能导致模型收敛不稳定或训练效果不佳。

随机梯度下降法(Stochastic Gradient Descent，SGD)，每次的迭代只使用一个样本，计算速度快，适用于大规模数据集和在线学习。由于随机性，SGD 有助于跳出局部最小值，找到更好的全局最小值。但由于每次只使用一个样本进行梯度估计，所以梯度估计的方差较大，导致收敛过程较为震荡，训练过程不够稳定。同时学习率的设置对 SGD 的性能影响很大，需要仔细调整。对噪声和异常值较为敏感，可能会影响模型的准确性。

3.1.1 参数初始化

在主函数中的初始化需要用到的一些学习参数，包括遍历数据样本的次数 n_epochs，以及单次训练中使用的数据样本个数 minibatch，之后是学习率 t 和调节学习率的参数t_0、t_1。

```
print('使用小批量梯度下降法进行线性拟合:')
# 记录更新的 theta 值
theta_path_mgd=[]
# 设置更新参数 theta 的次数
n_epochs = 100
# 设置单次循环训练的数据样本个数
minibatch = 16
# 初始化 theta 值
theta = np.random.randn(2,1)
# 调节学习率的参数，用于逐渐减小学习率
t0, t1 = 200, 1000
t = 0
```

之后我们初始化定义了一个 theta 值，同时也定义了一个路径列表，用于存储优化路径。

3.1.2 训练线性回归模型

在 program 中我实现了一个用于线性拟合的 Mini-Batch 梯度下降算法。程序首先将原始数据 X 增加一列全为 1 的向量，以对应回归模型中的截距项。接着，在每次迭代中，它随机选取一小批样本数据(Mini-Batch)并计算其梯度。然后，程序根据预设的学习率调度策略更新学习率，并使用该学习率与计算出的梯度来更新模型参数 theta。学习率更新的公式为：

$$Learning\ Rate = \frac{t_0}{t_1 + t}$$

在每次迭代过程中，使用当前的 theta 值预测新数据 X_new 的 y 值，并在图上绘制预测线。最后，返回更新后的模型参数 theta。

```
def miniBatch_gradient_descent_linear_fit(n_epochs, minibatch, theta, X, y, X_new, t, t0, t1, theta_path_mgd):
    X_b = np.c_[np.ones((100,1)),X]
    # 这里增加一列是为了对应回归模型中的截距项
    X_new_b = np.c_[np.ones((2,1)),X_new]
    # 先作出原始图像
    plt.plot(X, y,'b.')
    np.random.seed(42)
    # 得到样本数据的总数
    m = len(X_b)
    for epoch in range(n_epochs):
        random_index = [np.random.randint(m) for _ in range(minibatch)]
        Xi = X[random_index]
        yi = y[random_index]
        Xi_b = np.c_[np.ones((minibatch,1)),Xi]

        # 进行一次预测，并且作图
        y_predict = X_new_b.dot(theta)
        plt.plot(X_new,y_predict,'r-')

        # 计算样本的梯度
        gradients = 2/minibatch * Xi_b.T.dot(Xi_b.dot(theta)-yi)

        # 更新学习率 eta
        eta = learning_schedule(t0, t1, epoch*minibatch+t)
        t = t + 1

        # 更新模型的参数 theta
        theta = theta - eta*gradients
        theta_path_mgd.append(theta)
        plt.xlabel('X_1')
        plt.axis([0,2,0,15])
        plt.title('eta = {}'.format(eta))

    return theta
```

3.1.3 传入数据集并可视化结果

我们在前面已经成果初始化了样本的特征值 X，以及对应的目标变量 y，我们将随机创造出来的训练集整个放入 miniBatch 中进行训练，并将训练得到的数据在控制台输出，同时记录下每次更新路径的过程。

```python
def miniBatch_gradient_descent_linear(X, y):
    print('使用小批量梯度下降法进行线性拟合:')
    # 记录更新的 theta 值
    theta_path_mgd=[]
    # 设置更新参数 theta 的次数
    n_epochs = 100
    # 设置单次循环训练的数据样本个数
    minibatch = 16
    # 初始化 theta 值
    theta = np.random.randn(2,1)
    # 调节学习率的参数，用于逐渐减小学习率
    t0, t1 = 200, 1000
    t = 0

    np.random.seed(42)
    X_new = np.array([[0],[2]])
    theta = lr.miniBatch_gradient_descent_linear_fit(n_epochs, minibatch, theta, X, y, X_new, t, t0, t1, theta_path_mgd)

    plt.show()
    print('theta = ')
    print(theta)

    return theta_path_mgd
```

3.2 三种梯度下降法的路径对比的实现

我们要实现的目的是比较批量梯度下降(BGD)、随机梯度下降(SGD)和小批量梯度下降(MBGD)在参数空间中的收敛路径。在函数中我们首先调用三个不同的梯度下降实现函数(这些函数已经定义好)，并分别获取每种方法更新参数 theta 的路径。这些路径是 np 类型的二维数组，其中每一行代表一次迭代后的 theta 值。接下来，函数使用 matplotlib 创建一个图形窗口，并绘制三种方法的参数更新路径。不同方法使用不同的颜色和标记来表示，以便在图形中区分。

最后，函数添加图例(plt.legend)来标识每种方法的路径，设置坐标轴的范围(plt.axis)，并显示图形(plt.show)。这样，通过比较这三种方法在参数空间中的收敛路径，我们可以直观地理解它们的差异和特性。可以预估到的结果是，SGD 的路径可能会比较嘈杂，因为它每次只使用一个样本进行更新；而 BGD 的路径会比较平滑，因为它使用所有样本进行每次更新；MGD 的路径则介于两者之间。

```python
def gradient_descent_path_comparison(X, y):
    theta_path_bgd = np.array(batch_gradient_descent_linear(X, y))
    theta_path_sgd = np.array(stochastic_gradient_descent_linear(X, y))
    theta_path_mgd = np.array(miniBatch_gradient_descent_linear(X, y))

    plt.figure(figsize=(12,6))
    plt.plot(theta_path_sgd[:,0],theta_path_sgd[:,1],'r-s',linewidth=1,label='SGD')
    plt.plot(theta_path_mgd[:,0],theta_path_mgd[:,1],'g-+',linewidth=2,label='MINIGD')
    plt.plot(theta_path_bgd[:,0],theta_path_bgd[:,1],'b-o',linewidth=3,label='BGD')
    plt.legend(loc='upper left')
    plt.axis([3.5,4.5,2.0,4.0])
    plt.show()
```

下面展示批梯度下降法以及随机梯度下降法实现的函数代码。

```python
def batch_gradient_descent_linear(X, y):
    print('使用批梯度下降法进行线性拟合：')
    # 设置学习率为0.1
    eta = 0.1
    # # 设置迭代次数为1000次
    # n_iterations = 1000
    # # 假设我们有100个样本
    # m = 100

    # 随机初始化参数theta
    theta = np.random.randn(2,1)

    # gradients, theta = lr.batch_gradient_descent_linear_fit(n_iterations, X, y, m, theta, eta)
    theta_path_bgd = []
    # 基于之前的参数对数据进行预测
    X_new = np.array([[0],[2]])
    # 这里增加一列是为了对应回归模型中的截距项
    X_new_b = np.c_[np.ones((2,1)),X_new]

    # plt.figure(figsize=(10,4))
    # plt.subplot(131)
    # lr.plot_gradient_descent(X, y, X_new, X_new_b, theta, eta = 0.02)
    # plt.subplot(132)
    # lr.plot_gradient_descent(X, y, X_new, X_new_b, theta, eta = 0.1)
    # plt.subplot(133)
    # lr.plot_gradient_descent(X, y, X_new, X_new_b, theta, eta = 0.5)
    theta = lr.plot_gradient_descent(X, y, X_new, X_new_b, theta, eta, theta_path_bgd)
    print('theta = ')
    print(theta)
    plt.show()

    return theta_path_bgd
```

```python
def stochastic_gradient_descent_linear(X, y):
    print('使用随机梯度下降法进行线性拟合：')
    # 基于之前计算的theta对数据进行预测
    X_new = np.array([[0],[2]])
    X_b = np.c_[np.ones((100,1)),X]
    # 这里增加一列是为了对应回归模型中的截距项
    X_new_b = np.c_[np.ones((2,1)),X_new]

    # 存储优化的 theta 值
    theta_path_sgd=[]

    # 读取样本长度
    m = len(X_b)

    # 设置随机数种子
    np.random.seed(42)

    # 设置遍历整个数据集的次数
    n_epochs = 50

    # 学习调度的参数，用于逐渐减小学习率
    t0 = 5
    t1 = 50

    # 开始训练样本，同时作图可视化
    theta = lr.stochastic_gradient_descent_linear_fit(n_epochs, m, X, y, X_new, t0, t1, theta_path_sgd)
    plt.plot(X,y,'b.')
    plt.axis([0,2,0,15])
    plt.show()
    print('theta = ')
    print(theta)

    return theta_path_sgd
```

3.3 观察多项式拟合中 degree 的影响的实现

3.3.1 构造数据集

首先通过随机数构造出训练集样本的特征以及目标变量，并将训练集可视化。

```
np.random.seed(42)

# 构建数据集
m = 100
X = 6*np.random.rand(m,1) - 3
y = 0.5*X**2+X+np.random.randn(m,1)

# 将训练数据进行可视化
plt.plot(X,y,'b.')
plt.xlabel('X_1')
plt.ylabel('y')
plt.axis([-3,3,-5,10])
# plt.show()
```

3.3.2 设计多项式拟合的函数

这部分的代码实现了一个完整的多项式回归过程，通过将多项式特征转换将原本的非线性问题转化为线性问题，进而利用线性回归模型进行拟合。我们可以根据调整输入参数指定多项式的阶数，会生成相应阶数的多项式特征，并利用这些特征训练线性回归模型。在拟合过程中，代码会输出模型的系数和截距，这些参数描述了多项式拟合曲线的形状。最后，为了更直观地展示拟合效果，通过调用自定义的图表函数，能够看到不同阶数多项式拟合曲线与原始数据点的匹配程度，用于之后对比三种情况下(过拟合、欠拟合和标准拟合)的图像。

```
colors = ['g', 'r-+', 'b--']

def polynomial_regression_fit(X, y, X_new, degree):
    # 生成二次多项式特征，不包含常数项
    poly_features = PolynomialFeatures(degree, include_bias = False)

    # 这里的 X_poly 包含了一次项和二次项的特征，[X] -> [X, X^2]
    # 通过将X转换为X_poly（包含X和X^2），我们实际上是将一个非线性回归问题转化为了一个线性回归问题，从而可以利用线性回归模型的优点来解决它
    X_poly = poly_features.fit_transform(X)

    # 创建实例进行拟合
    lin_reg = LinearRegression()
    lin_reg.fit(X_poly,y)

    # 输出当前的拟合信息
    print(f'当前度数为：{degree}')
    print('coef_ = ')
    print (lin_reg.coef_)
    print('intercept_ = ')
    print (lin_reg.intercept_)

    # 进行预测
    X_new_poly = poly_features.transform(X_new)
    y_new = lin_reg.predict(X_new_poly)

    # 进行可视化操作
    color_index = degree%4
    chart(X_new, y_new, colors[color_index], str(f'degree = {degree}'))
```

3.3.3 调用多项式拟合的函数

我们要观察不同多项式阶数(度数)下的拟合效果，并通过可视化展示欠拟合、标准拟合和过拟合的情况。首先，我们创建了一个数据集 X_new，这是一个在-3 到 3 之间均匀分布的 100 个点的数组，作为输入进行线性回归模型的预测。

接下来，我们分别调用了 3.3.2 中的 函数三次，每次使用不同的多项式阶数来拟合数据：使用一阶多项式(degree＝1)进行拟合，这会导致欠拟合的情况，因为我们的训练集数据是二阶的，由于模型过于简单，会无法捕捉到数据的复杂结构；使用二阶多项式(degree＝2)进行拟合，与训练集的变化特征一致，能够得到一个折中的模型，实现标准拟合；使用一百阶多项式(degree＝100)进行拟合，这会导致过拟合，因为模型过于复杂，不仅拟合了数据的真实结构，还拟合了噪声，导致模型在新数据上的泛化能力下降。每次调用函数后，都会绘制出相应的拟合曲线，并显示当前的拟合信息(包括多项式的阶数、系数和截距)。

```
######################################################################
## 任务二：观察过拟合与欠拟合的图像

# 观察不同度数下的拟合效果
X_new = np.linspace(-3,3,100).reshape(100,1)
plt.title('Underfitting, Overfitting, and Standard Fitting')

# 欠拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 1)
# 标准拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 2)
# 过拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 100)
# 展示图像
plt.show()
```

3.4 观察样本数量对结果的影响的实现

我们的目的是设计一个用于绘制一个模型的学习曲线的函数。学习曲线是一种用来诊断模型是处于欠拟合还是过拟合状态的图形化工具。通过绘制训练误差和验证误差随着训练集大小变化的关系，我们可以观察模型在不同训练数据量下的性能。

3.4.1 数据分割

使用 train_test_split 函数将数据集分割为训练集和验证集，其中验证集占整个数据集的 20%。random_state 参数确保每次分割结果都是一致的。

```
# 数据分割
X_train, X_val, y_train, y_val = train_test_split(X,y,test_size =0.2,random_state=100)
```

3.4.2 循环训练模型并记录误差

创建两个空列表 train_errors 和 val_errors ，用于存储训练误差和验证误差。

对于训练集中的每个样本数 m(从 1 开始到最后一个样本)，使用前 m 个样本作为训练数据来拟合模型。使用模型预测训练集和验证集的结果。计算训练误差(均方误差 RMSE)和验证误差，并将它们分别添加到 train_errors 和 val_errors 列表中。

```
# 空列表用于存储训练误差和验证误差
train_errors,val_errors = [],[]
for m in range(1,len(X_train)):
    model.fit(X_train[:m],y_train[:m])
    y_train_predict = model.predict(X_train[:m])
    y_val_predict = model.predict(X_val)
    train_errors.append(mean_squared_error(y_train[:m],y_train_predict[:m]))
    val_errors.append(mean_squared_error(y_val,y_val_predict))
```

3.4.3 绘制学习曲线

使用 matplotlib 库绘制两条曲线：训练误差曲线(红色虚线)，使用 train_errors 列表中的数据，并将误差值取平方根以得到 RMSE(均方根误差)。验证误差曲线(蓝色实线)，使用 val_errors 列表中的数据，并同样取平方根得到 RMSE。

```
plt.plot(np.sqrt(train_errors),'r-+',linewidth = 2,label = 'train_error')
plt.plot(np.sqrt(val_errors),'b-',linewidth = 3,label = 'val_error')
plt.xlabel('Trainsing set size')
plt.ylabel('RMSE')
plt.legend()
```

3.5 观察过拟合标志的实现

我们定义了一个使用多项式特征的线性回归模型管道 polynomial_reg，然后通过调用之前定义的 plot_learning_curves 函数来绘制该模型的学习曲线。

Pipeline：这是 scikit-learn 中的一个工具，用于将多个估计器串联起来，形成一个单一的估计器，这样可以更方便地处理数据预处理和模型训练的过程。

```
# 使用多项式特征的线性回归模型管道 polynomial_reg
polynomial_reg = Pipeline([
    ('poly_features',PolynomialFeatures(degree = 25,include_bias= False)),
    ('lin_reg',LinearRegression()
    )])

poly.plot_learning_curves(polynomial_reg,X,y)
plt.axis([0,80,0,5])
plt.show()
```

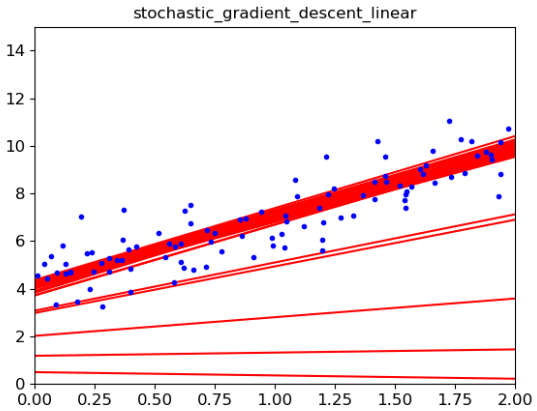
四、 实验结果与分析

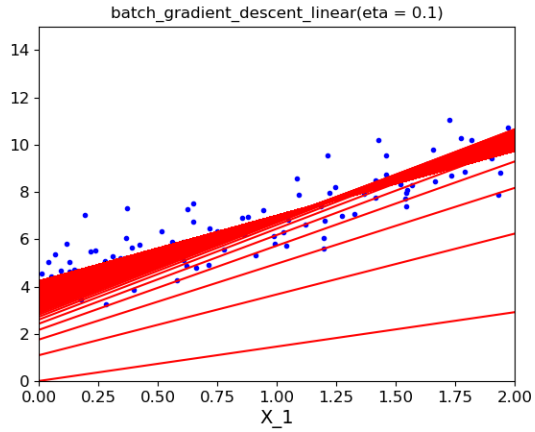
4.1 三种梯度下降法的路径对比

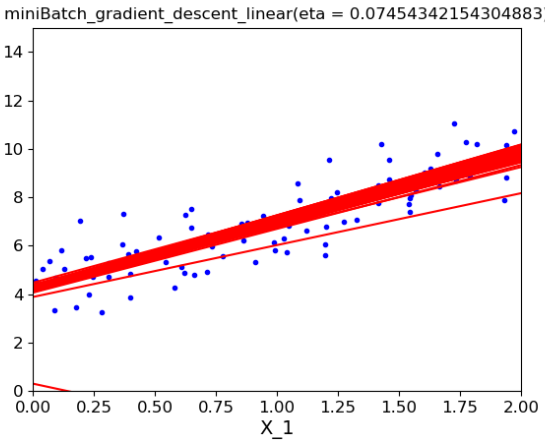
在主函数中，我们调用函数 gradient_descent_path_comparison 进行三种路径的对比。

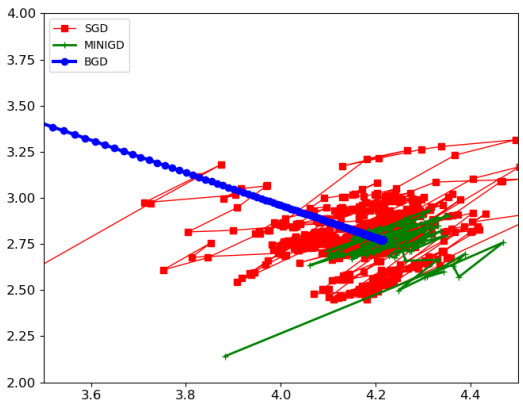
```
# least_squares_linear(X, y)
# gradient_descent_linear(X, y)
# batch_gradient_descent_linear(X, y)
# stochastic_gradient_descent_linear(X, y)
# miniBatch_gradient_descent_linear(X, y)
gradient_descent_path_comparison(X, y)
```

观察得到的图像，其中左上为随机梯度下降法、右上为批梯度下降法、左下为小批量梯度下降法迭代收敛的步骤的展示，右下为三种梯度下降法对应的路径变化情况。通过比较这三种方法在参数空间中的收敛路径，我们可以直观地理解它们的差异和特性。观察到的结果是，SGD 的路径会比较嘈杂，因为它每次只使用一个样本进行更新；而 BGD 的路径会比较平滑，因为它使用所有样本进行每次更新，更新方向比较稳定；而 MGD 的路径则介于两者之间。









Batch 大小的选择会显著影响模型的训练速度、收敛性以及泛化能力。

考虑**收敛速度**的话，通常较大的 batch 大小可以加速训练过程，因为每次更新模型时都使用了更多的数据。然而，这也并不意味着总是应该选择最大的可能 batch 大小。在某些情况下，使用中等大小的 batch 可能会获得更好的收敛速度和更稳定的训练过程。

考虑**泛化能力**的话，较小的 batch 大小通常有助于模型更好地泛化到新数据。这是因为小 batch 在每次更新时引入了更多的随机性，有助于逃离局部最小值并找到更好的全局解。然而，过小的 batch 大小可能导致训练过程不稳定，甚至无法收敛。

考虑**内存限制**的话，batch 大小受到会 GPU 或 CPU 内存的限制，如果在模型或数据集非常大的情况下，我们可能需要使用较小的 batch 大小。

从**操作系统的层面**来说，如果硬件支持并行计算(如多个 GPU 或 CPU 核心)，选择使用较大的 batch 大小可能更能充分利用这些资源，提高训练速度。

综合考虑之下，如何选择 batch size，应该根据训练模型的情况不同而有差异，一般情况下考虑选择中等大小的 batch。此外，还可以使用动态调整 batch 大小的方法，如逐渐增大 batch 大小或在训练过程中根据验证损失调整 batch 大小。

4.2 度数对多项式拟合的影响

欠拟合、过拟合和标准拟合是机器学习领域中关于模型性能的三种状态，它们之间的主要区别体现在模型对训练数据的拟合程度以及对新数据的泛化能力上。

首先是**欠拟合**(Underfitting)，欠拟合是指模型在训练数据上的性能较差，无法充分学习到数据的内在规律和特征。欠拟合可能由多种原因导致，例如模型复杂度过低(如线性模型用于拟合非线性数据)、特征选择不当或训练数据不足等，在我们的实验中，这里体现为模型复杂度过低。模型表现为**在训练集和测试集上的性能都较差，模型不能很好地拟合数据**。解决欠拟合问题的办法主要有增加模型复杂度、增加训练数据量等。

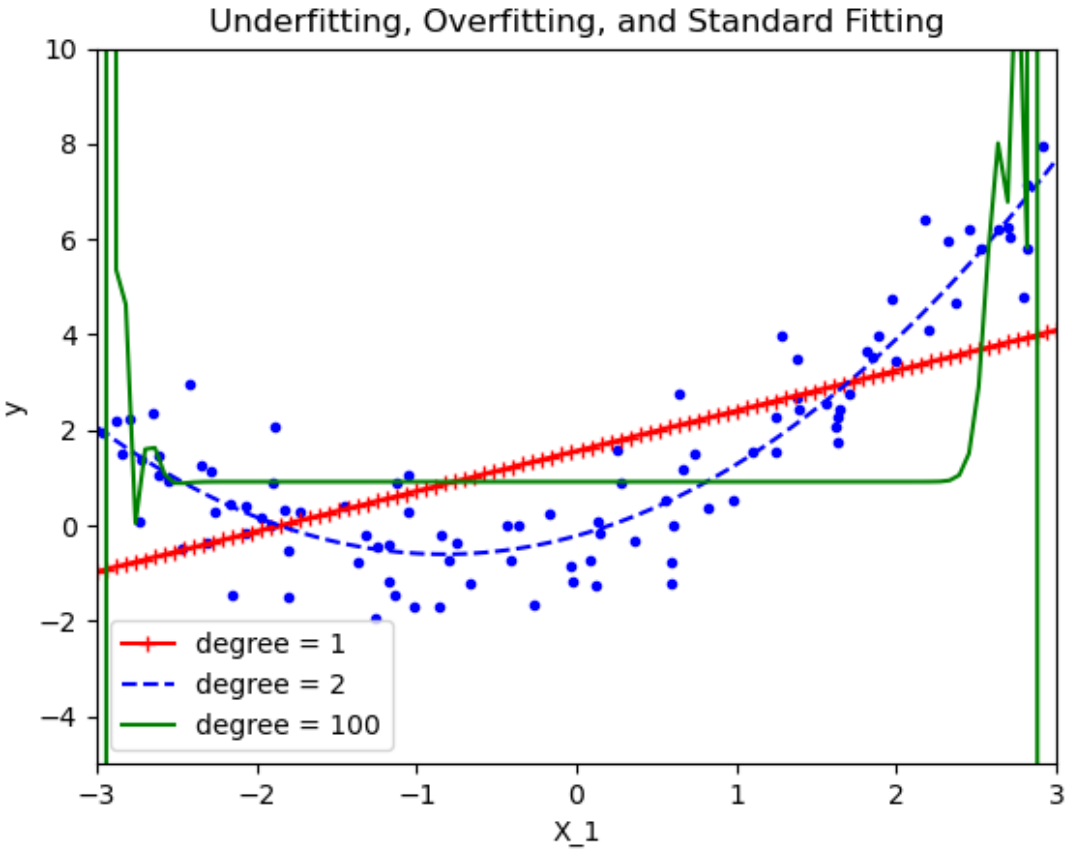
之后是**过拟合**(Overfitting)，过拟合是指模型在训练数据上的性能非常好，但在新数据(测试集)上的性能较差，即模型的泛化能力较弱。过拟合多半是因为模型复杂度过高，导致模型过度拟合了训练数据中的噪声和细节，而忽略了数据的内在规律。在误差性能上的表现主要是**在训练集上的性能很好，但在测试集上的性能较差**，模型对新数据的预测能力不强。主要的解决方法有降低模型复杂度、增加正则化项(如 L1 或 L2 正则化)、采用早停法等等。

调用函数，我们考虑度数为 1、2、100 的情况(训练集设置的阶数为 2)，那么显然，degree=1 为欠拟合状态，degree=2 为标准拟合状态，degree=100 为过拟合状态。下面为该任务下的主函

数代码，若要执行请取消这部分的注释。

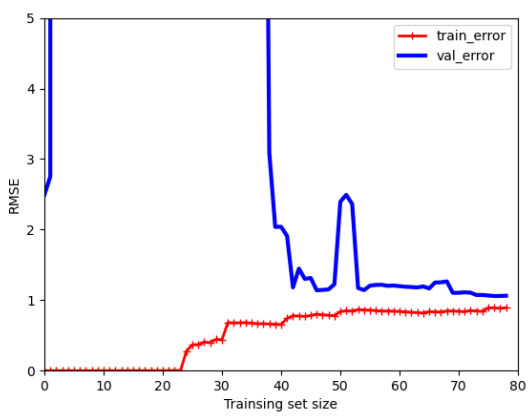
```
## 任务二：观察过拟合与欠拟合的图像

# 观察不同度数下的拟合效果
X_new = np.linspace(-3,3,100).reshape(100,1)
plt.title('Underfitting, Overfitting, and Standard Fitting')

# 欠拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 1)
# 标准拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 2)
# 过拟合
poly.polynomial_regression_fit(X, y, X_new, degree = 100)
# 展示图像
plt.show()
```

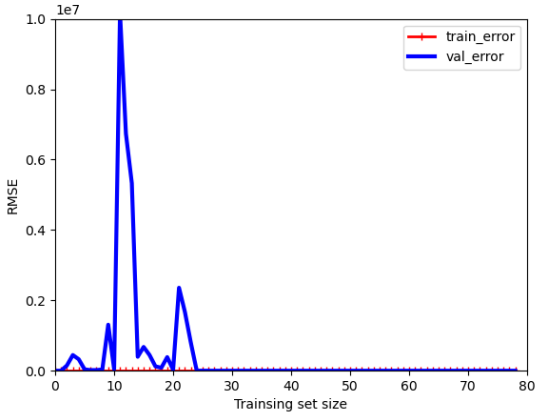
现在来观察结果，可以发现在样本集上，二阶的多项式拟合表现的效果最好，能够较为准确地描述数据的变化特征，而一阶多项式似乎并没有很好地学习到数据的特征，不能准确描述数据的变化，来看三阶多项式，学习到了部分的数据样本集特征，但是学习到了许多误差的特征，过度拟合了训练数据中的噪声和细节，而忽略了数据原来的内在规律。



4.3 从误差分析过拟合的标志

根据示例代码作出如下的图像，这是均方根误差(RMSE)随着训练集数据大小变化的图像。其中作图为原始生成图像，右图为比例缩小后的左图。





模型训练经常出现两类现象：过拟合(训练误差远小于泛化误差)和欠拟合(训练误差较高)，导致这两类现象的两个重要因素是：模型复杂度和训练集大小。

在本任务中我们考虑的是训练集的大小对均方根误差的影响。如果训练集过小，特别是比模型参数数量更小时，过拟合更容易发生，在结果图像中体现为：**在训练数据比较少(小于模型预设阶数 25)时，我们的训练集误差基本上为 0，但是我们的验证集误差极大，这是很明显的过拟合现象。**

但是随着训练集的增大，大到超过我们预设的多项式阶数之后，我们数据的训练集误差有所上升，这是正常的，模型会更难地去拟合我们的数据，这时候验证集误差大幅度的下降，也就是说我们的模型较为有效地学习了数据的特征，并且此时具有了一定的泛化能力，可以一定程度上实现预测。在这个时候，**我们训练集与验证集的误差差异减小**，也就是说泛化能力得到了有效地提升。此时过拟合现象有所缓解，并且随着数据集的不断增加，不论是训练集误差还是验证集误差都比较的接近稳定，过拟合现象消失。

在数据集比较小时，我们更容易出现过拟合的现象。在查阅资料后，我也了解到由于泛化误差不会随训练集的增大而增大，所以**一般情况下我们会希望训练集大一些。**

五、 实验总结

本次实验让我对线性回归、多项式回归以及正则化有了更加深入的理解。通过亲手编写代码和进行数据分析，我体会到了理论知识与实践操作之间的紧密联系。

在实验过程中，我深刻感受到**选择合适的参数和模型复杂度对于机器学习模型性能的重要**

性。特别是在多项式回归部分，当度数过高时，模型虽然能够更好地拟合训练数据，但却容易导致过拟合，使得模型在验证集上的表现不佳。这让我意识到，在构建模型时，我们不仅要追求训练误差的最小化，以及训练时的收敛速度，更要关注模型的泛化能力。

此外，MiniBatch 梯度下降方法的实现也让我对梯度下降算法有了更深刻的认识。通过调整 batch 的大小，我观察到了不同**batch 数量对模型收敛速度和稳定性的影响**。这让我明白了在实际应用中，我们需要根据具体问题和数据集的特点来选择合适的 batch 大小，以达到更好的训练效果。

通过对比训练误差和验证误差的变化趋势，我更加清晰地认识到了过拟合的标志。**在训练数据比较少(小于模型预设阶数 25)时，我们的训练集误差基本上为 0，但是我们的验证集误差极大，这是很明显的过拟合现象**，这意味着模型可能出现了过拟合。这提醒我在以后的实践中，需要时刻关注模型的泛化性能，并采取相应的措施来避免过拟合的发生。

总的来说，这次实验让我更加深入地理解了机器学习中线性回归一些算法的原理和应用，也提升了我的编程能力和数据分析能力。