实验一 基于遗传算法的函数优化

设 $f(x)=-x^2-4x+1$，求 $\max\ f(x), x\in[-2,2]$，要求解的精度保留二位小数。

1）遗传算法主旨思想：（简写）

遗传算法是一种基于自然进化原理的优化算法，其主旨思想是通过模拟自然界中的进化过程，不断地对问题的解进行进化、交叉和变异，以产生更好的解。

具体来说，遗传算法模拟了自然界中的遗传过程，通过选择、交叉和变异等操作，从种群中筛选出优秀的解，并将它们组合生成新的解。这样不断迭代，直到找到满足优化目标的最优解。其中每个解都可以看作是某种基因型，而这些基因型通过遗传算子（如选择、交叉、变异等）进行操作，生成新的基因型，从而不断优化问题的解。

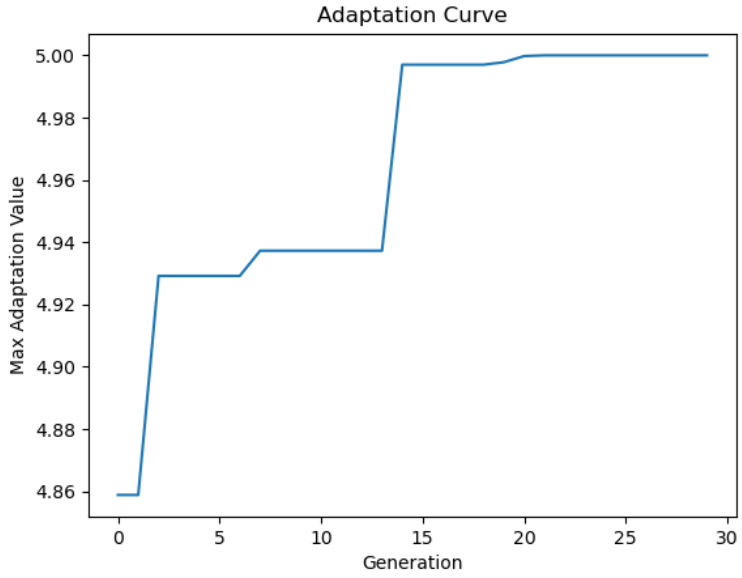
遗传算法一般包括以下步骤：

1. 初始化种群：生成随机的初始种群，每个个体（染色体）代表一个问题的一个可能解。
2. 评估适应度：对每个个体计算其适应度值，即衡量个体优劣的标准。
3. 选择操作：通过轮盘赌从当前种群中选择一定数量的个体作为下一代种群的父代。
4. 遗传操作：对父代进行交叉和子代变异操作，生成新的子代个体。
5. 更新种群：将部分父代（精英）和子代合并成新一代种群，进入下一轮迭代。
6. 终止条件：达到指定的终止条件，如达到最大迭代次数或找到满足要求的最优解。

2）实验结果+分析

输出结果：“ 最大值 f(x) 为 5.0, 当 x = -2.0 时

最佳个体为 000000000”



选用参数：

population_size=10, num_generations=30, crossover_prob=0.8, mutation_prob=0.1, num_bits=9, a=-2, b=2, precision=2, elitism_size=2

程序中最大适应度曲线呈现阶梯状，这是因为在遗传算法中选择交叉变异都具有一定的随机性，可能会出现在连续几代中找不到更好的解的情况，导致最大适应度值在一段时间内保持不变。同时，算法中的精英策略会保留每代中适应度最高的个体。这意味着当前一代的最优个体适应度值不会低于上一代的最优个体适应度值。因此，最大适应度值只会保持不变或增加，从而形成阶梯状曲线

在参数选择上，精度为 2 位小数且解空间一共 400 个，染色体位数为 9，所以可以适当减小种群规模以加快计算速度。交叉概率和变异概率均较高，不同于仅使用单点交叉，而是随机使用单点、双点和均匀交叉中的一种交叉方式，大大增加了收敛速度。实验表明 30 代以内基本均可收敛到最优解

3）附录：代码

```
import numpy as np
import matplotlib.pyplot as plt
# 定义函数 f(x)
def f(x):
    return -x**2 - 4*x + 1
# 将二进制字符串转为浮点数
def bin_to_float(bin_string, a, b, num_bits):
    decimal = int(bin_string, 2)
```

```
    max_decimal = 2**num_bits - 1
    return a + decimal * (b - a) / max_decimal
# 将浮点数转为二进制字符串
def float_to_bin(x, a, b, num_bits):
    max_decimal = 2**num_bits - 1
    decimal = int((x - a) * max_decimal / (b - a))
    return format(decimal, f'0{num_bits}b')
# 定义适应度函数
def adaptation_function(x,min_adaptation_values):
    return f(x)-min_adaptation_values
# 选择算子：轮盘赌
def roulette_wheel(population, fitness):
    total_fitness = np.sum(fitness)  # 总适应度
    r = np.random.rand() * total_fitness     # 生成随机数
    accumulated = 0  # 累计适应度
    for i, individual in enumerate(population):
        accumulated += fitness[i]
        if accumulated >= r:
            return individual
    return population[-1]

# 单点交叉
def single_point_crossover(parent1, parent2):
    crossover_point = np.random.randint(1, len(parent1)-1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2
# 两点交叉
def two_point_crossover(parent1, parent2):
    crossover_points = np.random.choice(range(1, len(parent1)-1), 2, replace=False)  # 生成两个不
重复的交叉
    crossover_points.sort()

    offspring1 = (parent1[:crossover_points[0]] +
                  parent2[crossover_points[0]:crossover_points[1]] +
                  parent1[crossover_points[1]:])
    offspring2 = (parent2[:crossover_points[0]] +
                  parent1[crossover_points[0]:crossover_points[1]] +
                  parent2[crossover_points[1]:])
```

```
    return offspring1, offspring2

# 均匀交叉
def uniform_crossover(parent1, parent2):
    offspring1 = ''
    offspring2 = ''
    for i in range(len(parent1)):
        if np.random.rand() < 0.5:  # 以 50%概率选择一个父代的对应基因
            offspring1 += parent1[i]
            offspring2 += parent2[i]
        else:
            offspring1 += parent2[i]
            offspring2 += parent1[i]
    return offspring1, offspring2

def mutation(individual, mutation_prob):     #位变异
    mutated = ''     # 初始化变异后的个体
    for bit in individual:
        if np.random.rand() < mutation_prob:     # 以 mutation_prob 概率对该基因进行变异
            mutated += '1' if bit == '0' else '0'    #取反
        else:
            mutated += bit    # 未变异，直接复制
    return mutated


def genetic_algorithm(population_size=10, num_generations=30,
                      crossover_prob=0.8, mutation_prob=0.01,
                      num_bits=9, a=-2, b=2, precision=2,
                      elitism_size=2):
    # 初始化种群
    population = [format(np.random.randint(0, 2**num_bits), f'0{num_bits}b')
                  for _ in range(population_size)]
    # 记录每代最大值和最小值
    max_adaptation_values = []
    for generation in range(num_generations):
        float = [f(bin_to_float(individual, a, b, num_bits))
                 for individual in population]

        max_adaptation_values.append(max(float))
        min_adaptation_values = min(float)
```

```
        # 计算适应度
        fitness = [adaptation_function(bin_to_float(individual, a, b, num_bits),
min_adaptation_values)
                   for individual in population]

        # 精英策略 保留最好的 2 个个体
        sorted_population = [x for _, x in sorted(zip(fitness, population), key=lambda pair: pair[0],
reverse=True)]
        new_population = sorted_population[:elitism_size]
        # 生成新的个体，直到种群大小达到 population_size
        while len(new_population) < population_size:
            # 找出两个 parent
            parent1 = roulette_wheel(population, fitness)
            parent2 = roulette_wheel(population, fitness)
            # 若达到交叉概率
            if np.random.rand() < crossover_prob:
                crossover_type = np.random.choice(['single', 'two', 'uniform']) # 随机选择一种交叉方
式（单点，双点和均匀交叉）
                if crossover_type == 'single':
                    offspring1, offspring2 = single_point_crossover(parent1, parent2)
                elif crossover_type == 'two':
                    offspring1, offspring2 = two_point_crossover(parent1, parent2)
                else:
                    offspring1, offspring2 = uniform_crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1, parent2

            offspring1 = mutation(offspring1, mutation_prob)
            offspring2 = mutation(offspring2, mutation_prob)
            # 将新的个体加入种群
            new_population.extend([offspring1, offspring2])

        population = new_population


    best_individual = max(population, key=lambda x: adaptation_function(bin_to_float(x, a, b,
num_bits),min_adaptation_values))    # 找到最好的个体
    best_x = bin_to_float(best_individual, a, b, num_bits)  # 将最好的个体转为对应的浮点数
    max_f = adaptation_function(best_x,min_adaptation_values)+min_adaptation_values # 计算最大
f(x)
```

```
    # 绘制曲线
    plt.plot(max_adaptation_values)
    plt.xlabel('Generation')
    plt.ylabel('Max Adaptation Value')
    plt.title('Adaptation Curve')
    plt.show()

    return round(best_x, precision), round(max_f, precision),best_individual

def main():
    best_x, max_f,best_individual = genetic_algorithm()
    print(f'最大值 f(x) 为 {max_f}，当 x = {best_x}时')
    print('最佳个体为{}'.format(best_individual))

if __name__ == "__main__":
    main()
```

实验二 基于 BP 神经网络的预测

1）BP 网络权重求解（简写公式）

BP 神经网络，即反向传播神经网络，是一种常见的人工神经网络。它由输入层、隐藏层和输出层组成，其中每一层都由若干个神经元（或称为节点）组成。每个神经元接收来自上一层的信号并通过激活函数将其转换为输出信号，输出信号再传递到下一层神经元。

BP 神经网络的训练过程主要包括两个步骤：前向传播和反向传播。前向传播是指将输入信号通过神经网络传递到输出层的过程，计算输出层的输出值。反向传播是指从输出层开始，逆向计算神经网络中每个节点的误差，并根据误差调整每个节点之间的权重，以最小化误差。

1.前向传播：

对于第 l 层的第 i 个神经元，计算加权和 z_i^l：

$$z_i^l = \sum_j w_{ij}^l a_i^{(l-1)} + b_i^l$$

其中，$a_i^{(l-1)}$是上一层（第 l-1 层）第 j 个神经元的输出值，w_{ij}^l是连接第 l-1 层第 j 个神经元和第 l 层第 i 个神经元的权重，b_i^l是第 l 层第 i 个神经元的偏置项。

计算激活值a_i^l：

$$a_i^l = f\left(z_i^l\right)$$

其中，a_i^l是激活函数 sigmoid

使用均方误差损失函数计算损失

$$L = \frac{1}{2}\sum_i \left(y_i - y_i\right)^2$$

其中，y_i是目标值，y_i是输出值。

2.反向传播

计算输出层的误差项δ_{Li}：

$$\delta_{Li} = \left(y_i - y_i\right) \bullet f'\left(z_{Li}\right)$$

计算隐藏层的误差项δ_{i}^l：

$$\delta_{\mathrm{i}}^l = \left(\sum_j \delta_j^{(l+1)} \omega_{ji}^{(l+1)}\right) \bullet f'\left(z_i^l\right)$$

计算权重梯度：

$$\frac{\delta L}{\delta \omega_{ij}^l} = a_j^{(l-1)} \delta_i^l$$

更新权重和偏置项：

$$\omega_{ij}^{l} = \omega_{ij}^{l} - \eta \frac{\delta L}{\delta \omega_{ij}^{l}}$$

$$b_{i}^{l} = b_{i}^{l} - \eta \delta_{i}^{l}$$

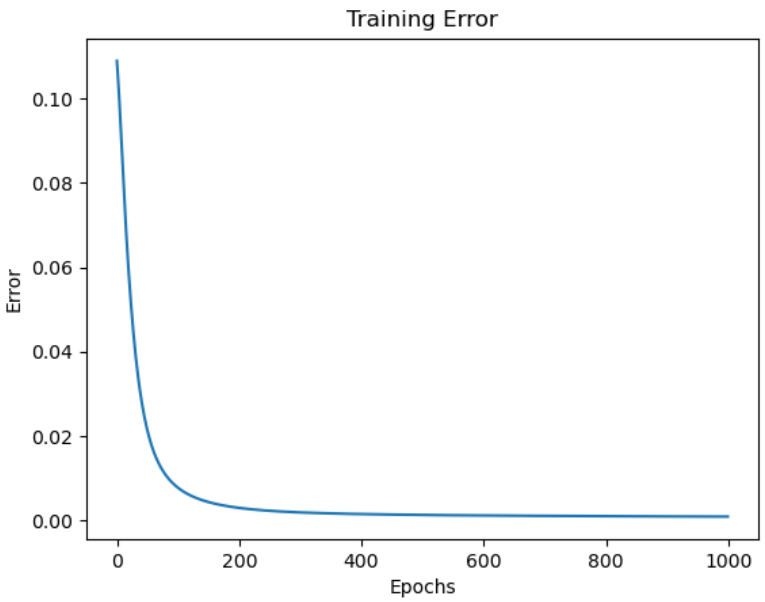
2）两个数据的实验结果+分析（无需数据描述，只分析网络结构和结果图即可）

这个神经网络是一个三层全连接神经网络，具有一个输入层、一个隐藏层和一个输出层。输入层包含 15/17 个节点，输出层包含 1 个节点。隐藏层的节点数量是输入层的两倍再加 1。

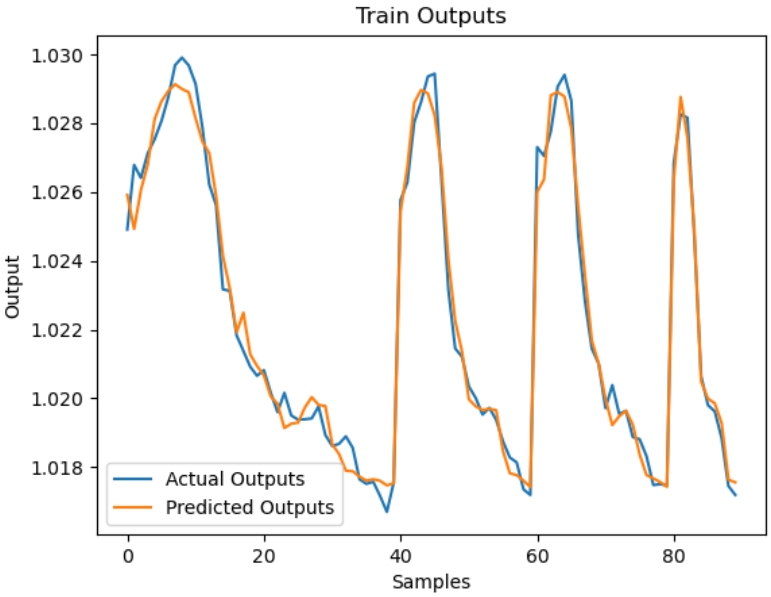
神经网络使用 Sigmoid 作为激活函数，在训练时使用了反向传播算法来更新权重和偏置。其中使用的学习率为 0.01。

在训练和测试之前，还对数据进行了归一化处理。归一化处理可以消除量纲，让各个特征对结果做出的贡献相同，提升模型的收敛速度和精度。

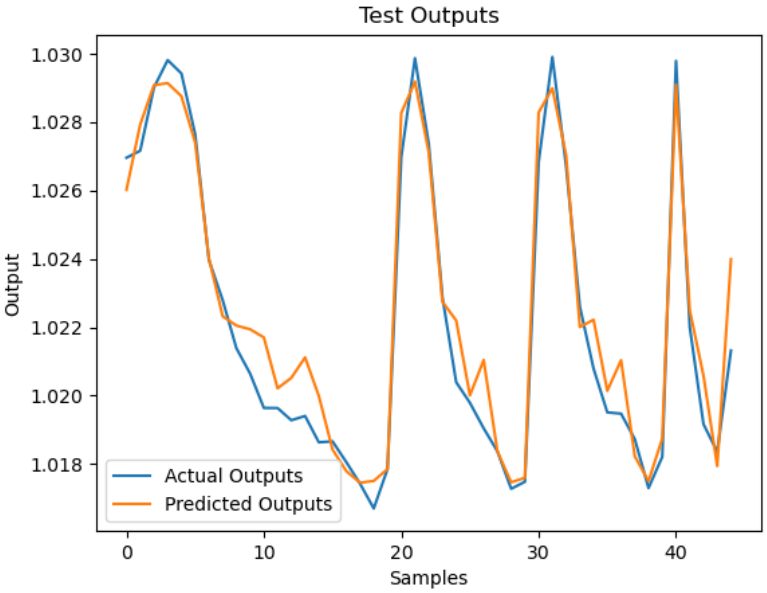
15 数据：



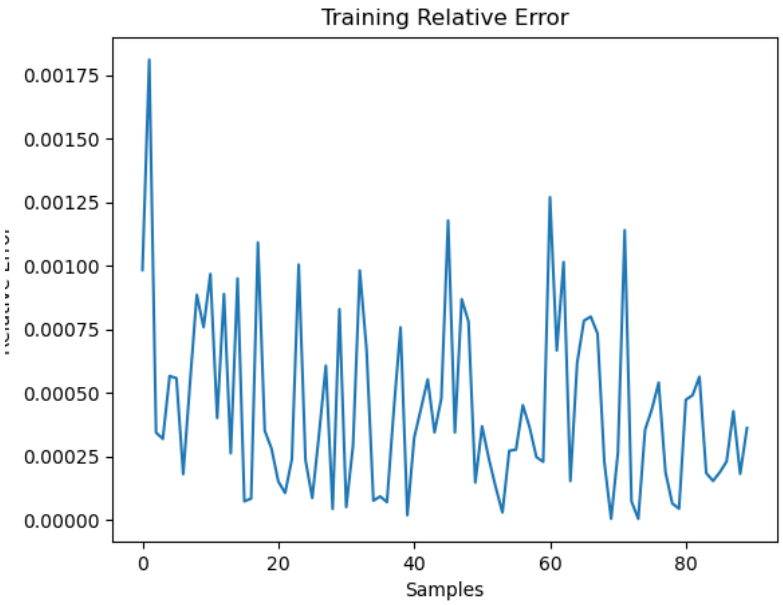
模型收敛速度较快，没有出现过拟合的情况，1000 次迭代足以达到收敛状态



在训练集上预测输出与实际输出紧密匹配，说明模型在训练集上拟合得较好



预测输出与实际输出在测试集上也紧密匹配，说明模型具有较好的泛化能力，预测稳定性较好

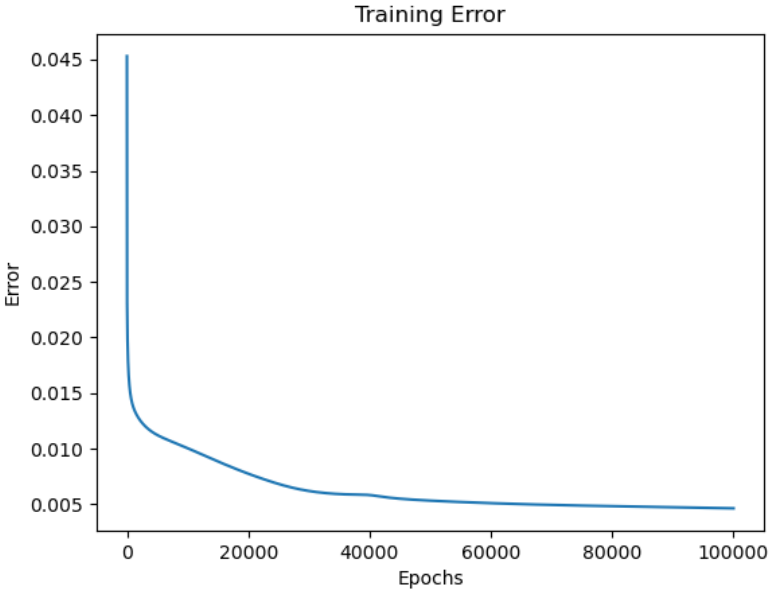


除了个别数据可能存在异常以外其他数据相对误差都较低，训练效果好

17 行数据：

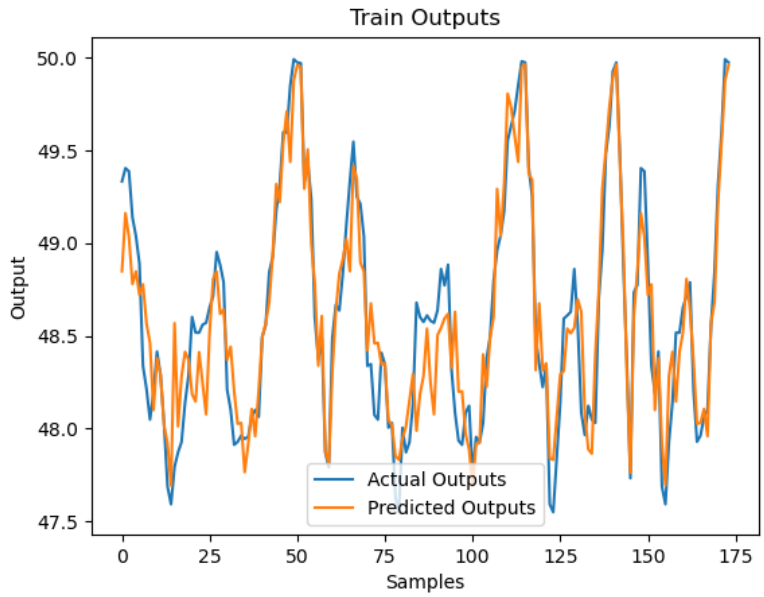
学习率 0.01，训练 10 万代

网络结构不变，但发现模型收敛速度变慢，收敛精度下降。降低学习率后发现收敛精度仍没有提升。升高学习率发现收敛速度变快但是最终收敛误差仍较大且泛化能力差，于是保持学习率不变尽量扩大代数以求取更好的收敛结果

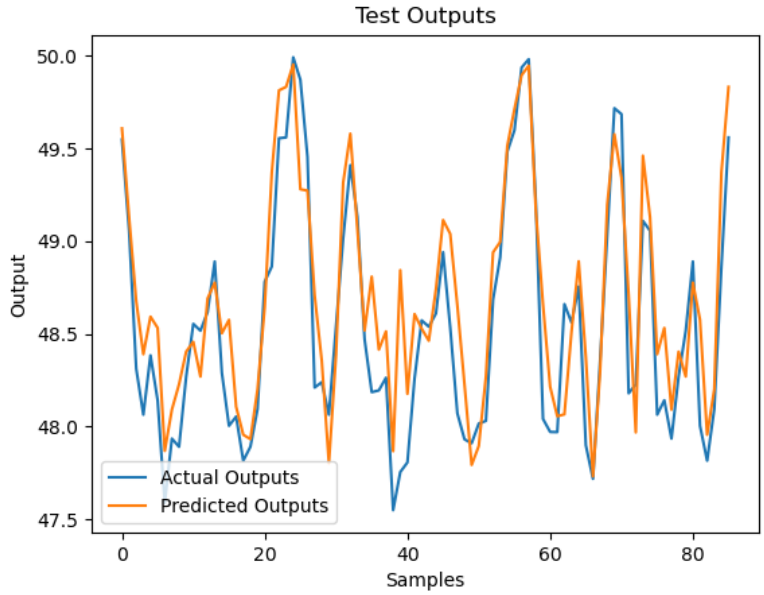


模型最开始收敛速度较快，但在 4 万代后出现明显变慢。没有出现过拟合的情况。达到收敛

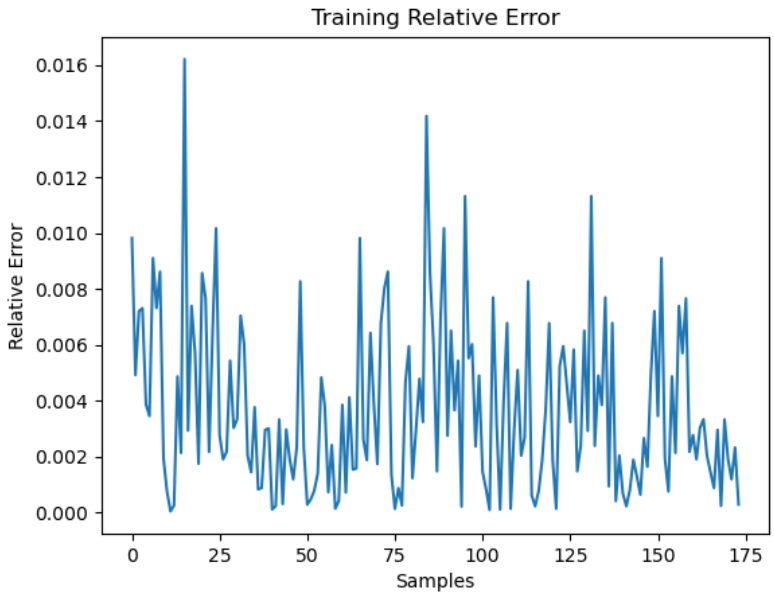
状态但收敛精度不如 15 行数据收敛精度高



在训练集上预测输出与实际输出紧密匹配，说明模型在训练集上拟合得较好。但在个别数据上差别较大



预测输出与实际输出在测试集上较为匹配，说明模型具有不错的泛化能力。且训练 10 万次仍没有出现过拟合的情况



有一些数据相对误差较高，训练效果一般

3）附录：代码

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler

# 从文本文件中读取归一化数据
def read_the_data(file_name):
    with open(file_name, 'r') as file:
        data = []
        for line in file:
            # 按空格拆分行并将数字转换为浮点数
            numbers = list(map(float, line.strip().split(' ')))
            # 将输入（前 15 个数字）和输出（最后一个数字）分开
            inputs, output = numbers[:-1], numbers[-1]
            data.append((inputs, output))
    return data

# 从文本文件中读取原始数据
def read_data(file_path):
    with open(file_path, "r") as f:
        data = [list(map(float, line.strip().split())) for line in f]
    return data
```

```
# 写入归一化数据
def write_data(file_path, data):
    with open(file_path, "w") as f:
        for row in data:
            f.write(" ".join(str(val) for val in row))
            f.write("\n")
# 将数据归一化

def normalize_data(file_path, output_path):
    data = read_data(file_path)
    data_array = np.array(data)

    # 分离输入特征和输出值
    inputs = data_array[:, :-1]
    outputs = data_array[:, -1].reshape(-1, 1)

    # 归一化输入特征和输出值
    input_scaler = MinMaxScaler()
    output_scaler = MinMaxScaler()
    normalized_inputs = input_scaler.fit_transform(inputs)
    normalized_outputs = output_scaler.fit_transform(outputs)

    # 将归一化后的输入特征和输出值合并
    normalized_data = np.column_stack((normalized_inputs, normalized_outputs))

    write_data(output_path, normalized_data)
    return input_scaler, output_scaler # 返回 Scaler，用于之后的反归一化

# 激活函数 sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 激活函数的导数
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

# 使用反向传播算法训练神经网络
def train_network(training_data, epochs, learning_rate):
    # 初始化权重和偏置
```

```
    input_size = 17
    hidden_size = 2*input_size+1
    output_size = 1
    # 两层网络 W：权重 b：偏置
    W1 = np.random.randn(input_size, hidden_size)
    b1 = np.zeros(hidden_size)
    W2 = np.random.randn(hidden_size, output_size)
    b2 = np.zeros(output_size)

    errors = []

    for epoch in range(epochs):
        error = 0

        for inputs, target in training_data:
            # 前向传播
            X = np.array(inputs)
            H = sigmoid(np.dot(X, W1) + b1)
            Y = sigmoid(np.dot(H, W2) + b2)

            # 计算误差
            error += 0.5 * (target - Y[0]) ** 2

            # 反向传播
            dY = (Y - target) * sigmoid_derivative(Y)
            dH = np.dot(dY, W2.T) * sigmoid_derivative(H)

            # 更新权重和偏置
            W2 -= learning_rate * np.outer(H, dY)
            b2 -= learning_rate * dY
            W1 -= learning_rate * np.outer(X, dH)
            b1 -= learning_rate * dH
        errors.append(error / len(training_data))

        # 打印每个 epoch 的误差
        print(f'Epoch {epoch + 1}/{epochs}, Error: {errors[-1]:.6f}')

    return W1, b1, W2, b2, errors
# 对预测的值进行反归一化，测试训练好的神经网络
def test_network(test_data, W1, b1, W2, b2):
```

```
    actual_outputs = []
    predicted_outputs = []

    for inputs, target in test_data:
        X = np.array(inputs)
        H = sigmoid(np.dot(X, W1) + b1)
        Y = sigmoid(np.dot(H, W2) + b2)
        actual_outputs.append(target)
        predicted_outputs.append(Y[0])

    return actual_outputs, predicted_outputs

# 画出训练误差的曲线
def plot_errors(errors):
    plt.plot(errors)
    plt.xlabel("Epochs")
    plt.ylabel("Error")
    plt.title("Training Error")
    plt.show()

# 画出测试集的实际值和预测值
def plot_test_outputs(actual_outputs, predicted_outputs, scaler, train):
    actual_outputs = np.array(actual_outputs)
    predicted_outputs = np.array(predicted_outputs)
    denormalized_actual_outputs = scaler.inverse_transform(actual_outputs.reshape(-1, 1))
    denormalized_predicted_outputs = scaler.inverse_transform(predicted_outputs.reshape(-1, 1))
    plt.plot(denormalized_actual_outputs, label="Actual Outputs")
    plt.plot(denormalized_predicted_outputs, label="Predicted Outputs")
    plt.xlabel("Samples")
    plt.ylabel("Output")
    if train==True:
        plt.title("Train Outputs")
    else:
        plt.title("Test Outputs")
    plt.legend()
    plt.show()

# 画出训练相对误差的曲线
def plot_training_relative_error(actual_outputs, predicted_outputs, scaler):
    actual_outputs = np.array(actual_outputs)
```

```
    predicted_outputs = np.array(predicted_outputs)
    denormalized_actual_outputs = scaler.inverse_transform(actual_outputs.reshape(-1, 1))
    denormalized_predicted_outputs = scaler.inverse_transform(predicted_outputs.reshape(-1, 1))
    E_r = np.abs(denormalized_actual_outputs - denormalized_predicted_outputs) /
denormalized_actual_outputs

    plt.plot(E_r)
    plt.xlabel("Samples")
    plt.ylabel("Relative Error")
    plt.title("Training Relative Error")
    plt.show()

def main():
    # 从文件中读取训练和测试数据
    input_scaler, output_scaler = normalize_data("PTA_T17-1.txt", "normalized_training_data.txt")
    normalize_data("PTA_G17-1.txt", "normalized_test_data.txt")
    training_data = read_the_data("normalized_training_data.txt")
    test_data = read_the_data("normalized_test_data.txt")
    # 训练神经网络
    epochs = 100000
    learning_rate = 0.01
    W1, b1, W2, b2, errors = train_network(training_data, epochs, learning_rate)

    # 测试神经网络
    test_actual_outputs, test_predicted_outputs = test_network(test_data, W1, b1, W2, b2)
    train_actual_outputs, train_predicted_outputs = test_network(training_data, W1, b1, W2, b2)

    # 画出训练误差的曲线
    plot_errors(errors)

    # 画出测试集的实际值和预测值
    plot_test_outputs(test_actual_outputs, test_predicted_outputs, output_scaler,train=False)
    # 画出训练集的实际值和预测值
    plot_test_outputs(train_actual_outputs, train_predicted_outputs, output_scaler,train=True)

    # 画出训练相对误差的曲线
    plot_training_relative_error(train_actual_outputs, train_predicted_outputs, output_scaler)

if __name__ == "__main__":
    main()
```