南京航空航天大學
NANJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS

软件可靠性课程实验报告

题　　目： 基于数据驱动的软件可靠性模型

院　　系： 计算机科学与技术学院/软件学院

专　　业： 软件工程

姓　　名： 陈梓鹏

学　　号： 162230217

2024 年 11 月 24 日

1.引言

1.1 编写目的

针对解析模型的不足，近年来数据驱动的软件可靠性建模方法受到了越来越多的重视，一些专家学者业已提出了一些数据驱动的软件可靠性模型。这类模型通常不需要对软件的失效过程做任何假设，而只是基于观测到的软件失效数据，将其视为一个时间序列进行建模与分析，并对软件将来的失效行为进行预测。由于数据驱动的软件可靠性模型不对软件内部错误及失效过程做任何不符合实际的假设，因而其试用范围较传统的解析模型更广，并且不存在实际应用中的模型选择问题。数据驱动的软件可靠性模型包括基于失效数据驱动的软件可靠性模型和基于软件产品属性数据驱动的软件可靠性模型。其中，基于失效数据的软件可靠性模型又包括基于时间序列分析的模型和智能学习算法（如基于人工神经网络、支持向量机和深度学习算法）的软件可靠性模型等。

2.基于 BP 神经网络的软件可靠性预测模型

2.1 模型背景

人工神经网络具有自组织、自适应、自学习等特点，并且能充分通近任意复杂的非线性关系，因而在模式识别》集精处理；自动轻制、知识工程签力面待到了广泛的回用。在敦件可靠性建模领域，也有学者提出了基于人工神经网络的软件可靠性模型。

BP（Back Propagation）神经网络算法是神经网络中应用牧为广泛的一种，是在 1086 年由 Rumelhart 和 McCelland 领导的科学研究小组提出的基于按误差逆传播算法训练的乡层前馈网络，即 BP 神经网络，最常用的结构是三层 BP 神经网络。BP 神经网络通过记忆功能贮存大量的输入一输出关系，并不需要事前确定该输入一输出关系的数学计算模型。当有同样性质的输入变量输入时，BP 神经网络会利用已经存在的对应关系，自动得出输出的值。

BP 神经网络是通过使用梯度下降算法不断的训练，不断的将训练误差结果前向反馈，不断的调整影响输入输出关系的网络单元的权值和阈值，直到误差在可以接受的范围之内，该对应关系就会贮存在网络中，凭借存在的记忆对再次输入的变量计算出精确的输出值，也因此称之为多层前馈型网络。

BP 神经网络以记忆映射关系代替复杂数学计算模型的优点受到各领域研究者的重视，BP 神经网络最重要的特点是误差反向传播，它能够将网络计算的误差，实时前向反馈，通过分析己知的数据，预测或评估未来结果，整个过程通过不断自动调整内部神经元权值，而无需调整外部结构。

2.2 模型结构

BP 神经网络的基本结构包括输入层、输出层和隐含层。BP 神经网络结构灵活多样，既可以一入多出，又可以多入多出，可以根据实际灵活运用，其基本模型结构如下图所示。

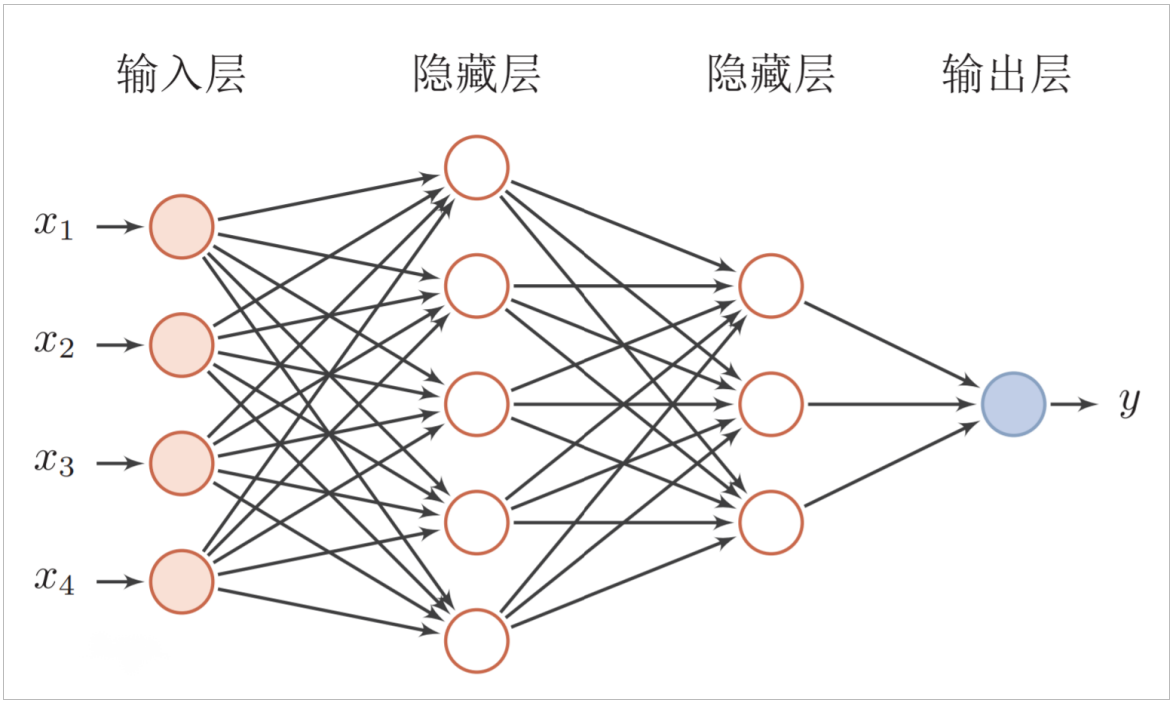

图 1.4 层 BP 神经网络结构图

2.3 算法原理

BP 算法是一种基于δ学习规则的神经网络学习算法，其核心操作是通过预测数据误差的反向传播不断修正网络权值和阈值，以实现模型的快速收敛。

定义输入向量为$\mathbf{X}=(x_1,x_2,\ldots,x_n)^{\mathrm{T}}$,隐层输出向量为$\mathbf{Y}=(y_1,y_2,\ldots,y_m)^{\mathrm{T}}$,输出层输出向量$\mathbf{O}=(O_1,O_2,\ldots,O_m)^{\mathrm{T}}$,期望输出指标向量为：$\mathbf{d}=(d_1,d_2,\ldots,d_m)^{\mathrm{T}}$。

对于隐层，其激活函数一般选用单极性/双极性 sigmoid 函数：

$$\mathrm{f(x)}=\frac{1}{1+\mathrm{e}^{-\mathrm{x}}}\text{或 }\mathrm{f(x)}=\frac{1-\mathrm{e}^{-\mathrm{x}}}{1+\mathrm{e}^{-\mathrm{x}}} \tag{1}$$

对于输出层选用线型函数(pureline 传递函数)，即$y=x$

目标函数取实际输出值与期望输出值的误差函数：

$$E=\frac{1}{2}\sum_{k=1}^{l}\left\{d_k-f\left[\sum_{j=0}^{m}w_{jk}f\left(\sum_{i=0}^{m}v_{ij}x\right)\right]\right\}^2 \tag{2}$$

BP 学习算法改变权值规则如下：

$$\Delta w_{ij}=\tau(d_k-o_k)o_k(1-o_k)y_j \tag{3}$$

$$\Delta v_{ij}=\tau\left(\sum_{k=1}^{l}(d_k-o_k)o_k(1-o_k)w_{jk}\right)y_j(1-y_j)x_i \tag{4}$$

式中:τ为网络学习效率。

输入层输入的是影响评估目标的多个指标：隐含层是调整输入与输出之间关系的桥梁，通过不断的调整神经元的权值和阈值，使输出的误差函数达到理想的范围之内；输出层输出的是评估指标的最终评估值。输入信号正向输入，误差信号反向传播，隐含层不断调整各神经元的阈值和权值，反复进行直到网络达到预期的输出效果，网络训练结束。当有类似的输入变量输入时，会运用已经存储的记忆对样本进行预测或结果修正。

BP 神经网络在训练过程通过误差逆向传播的方式，将网络实际计算误差与设定的期望误差不断比较，对各个神经元权值和阈值不断调整的过程。期望误差的设定是网络训练标准的基本要求，训练过程中反向传播的是网络实际计算误差和期望误差之间的比较结果，结果达到了设定的要求，网络训练结束，否则，不断重复训练过程，直到满足程序设定的要求。

2.4 算法步骤

- 步骤 1：网络的初始化，设定网络的学习效率、误差精度、初始权值和阈值、训练精度等初始参数；
- 步骤 2：设置隐含层节点数量，批量输入明确输入—输出关系的训练样本，并实时记录误差函数；
- 步骤 3：计算输出结果的误差，并与设定的期望误差进行比较；
- 步骤 4：通过反向传播的方式，不断调整神经元的阈值和权值；
- 步骤 5：若输出误差小于设定的误差值，则网络训练结束，否则，将计算的误差反向传播，返回步骤 4。

BP 神经网络结束训练的条件是输出误差小于设定的误差值，误差值越小，表明越精确，否则会不断的调整神经元的权值和阈值，直到输出结果满足条件。误差的设定并非越小越好，误差的设定还要综合考虑网络的收敛速度，网络误差的设定对网络收敛速度产生较大影响，因此在实际应用中要根据实际进行调整。

2.5 算法流程

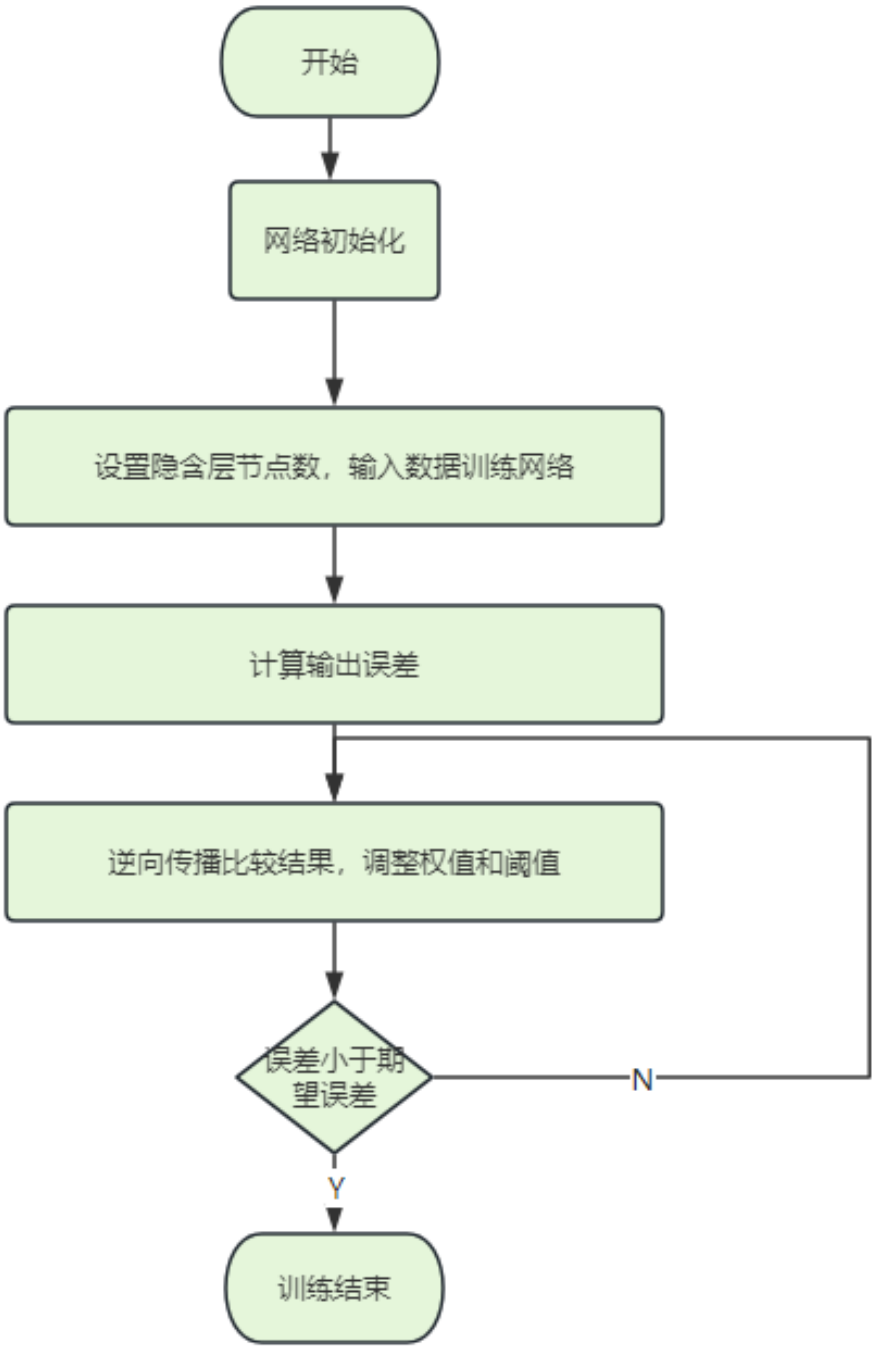

图 2.BP 神经网络算法流程图

2.6 算法实现

初始化：

```
bp1 = BPNN.BPNNRegression([5, 16, 1])
    def __init__(self, sizes):
        # 神经网络结构
        self.num_layers = len(sizes)
        self.sizes = sizes
        # 初始化偏差，除输入层外， 其它每层每个节点都生成一个 biase 值（0-1）
        self.biases = [np.random.randn(n, 1) for n in sizes[1:]]
        # 随机生成每条神经元连接的 weight 值（0-1）
        self.weights = [np.random.randn(r, c) for c, r in zip(sizes[:-1],
sizes[1:])]
```

前向传播：

```
    def feed_forward(self, a):
        """
        前向传输计算输出神经元的值
        """
        for i, b, w in zip(range(len(self.biases)), self.biases,
self.weights):
            # 输出神经元不需要经过激励函数
            if i == len(self.biases) - 1:
                a = np.dot(w, a) + b
                break
            a = sigmoid(np.dot(w, a) + b)
        return a
```

反向传播：

```
    def back_propagation(self, x, y):
        """
        利用误差后向传播算法对每个样本求解其 w 和 b 的更新量
        x: 输入神经元，行向量
        y: 输出神经元，行向量
        """
        delta_b = [np.zeros(b.shape) for b in self.biases]
        delta_w = [np.zeros(w.shape) for w in self.weights]
        # 前向传播，求得输出神经元的值
        a = x  # 神经元输出值
        # 存储每个神经元输出
        activations = [x]
        # 存储经过 sigmoid 函数计算的神经元的输入值，输入神经元除外
        zs = []
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, a) + b
            zs.append(z)
```

```
            a = sigmoid(z)  # 输出神经元
            activations.append(a)
        # -------------
        activations[-1] = zs[-1]  # 更改神经元输出结果
        # -------------
        # 求解输出层δ
        # 与分类问题不同，Delta 计算不需要乘以神经元输入的倒数
        # delta = self.cost_function(activations[-1], y) *
sigmoid_prime(zs[-1])
        delta = self.cost_function(activations[-1], y)  # 更改后
        # -------------
        delta_b[-1] = delta
        delta_w[-1] = np.dot(delta, activations[-2].T)
        for lev in range(2, self.num_layers):
            # 从倒数第 1 层开始更新，因此需要采用-lev
            # 利用 lev + 1 层的 δ 计算 l 层的 δ
            z = zs[-lev]
            zp = sigmoid_prime(z)
            delta = np.dot(self.weights[-lev + 1].T, delta) * zp
            delta_b[-lev] = delta
            delta_w[-lev] = np.dot(delta, activations[-lev - 1].T)
        return (delta_b, delta_w)
```

计算输出误差：

```
    def cost_function(self, output_a, y):
        """
        损失函数
        """
        return output_a - y
    def evaluate(self, train_data):
        test_result = [[self.feed_forward(x), y] for x, y in train_data]
        return np.sum([0.5 * (x - y) ** 2 for (x, y) in test_result])
```

预测：

```
    def predict(self, test_input):
        test_result = [self.feed_forward(x) for x in test_input]
        return test_result
```

小批量训练：

```
    def MSGD(self, training_data, epochs, mini_batch_size, eta,
error=0.01):
        """
        小批量随机梯度下降法
        """
        n = len(training_data)
        for j in range(epochs):
            # 随机打乱训练集顺序
```

```
            random.shuffle(training_data)
            # 根据小样本大小划分子训练集集合
            mini_batchs = [
                training_data[k : k + mini_batch_size]
                for k in range(0, n, mini_batch_size)
            ]
            # 利用每一个小样本训练集更新 w 和 b
            for mini_batch in mini_batchs:
                self.updata_WB_by_mini_batch(mini_batch, eta)

            # 迭代一次后结果
            err_epoch = self.evaluate(training_data)
            print("Epoch {0} Error {1}".format(j, err_epoch))
            if err_epoch < error:
                break
            # if test_data:
            #     print("Epoch {0}: {1} / {2}".format(j,
self.evaluate(test_data), n_test))
            # else:
            # print("Epoch {0}".format(j))
        return err_epoch
    def updata_WB_by_mini_batch(self, mini_batch, eta):
        """
        利用小样本训练集更新 w 和 b
        mini_batch: 小样本训练集
        eta: 学习率
        """
        # 创建存储迭代小样本得到的 b 和 w 偏导数空矩阵，大小与 biases 和
weights 一致，初始值为 0
        batch_par_b = [np.zeros(b.shape) for b in self.biases]
        batch_par_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            # 根据小样本中每个样本的输入 x，输出 y，计算 w 和 b 的偏导
            delta_b, delta_w = self.back_propagation(x, y)
            # 累加偏导 delta_b, delta_w
            batch_par_b = [bb + dbb for bb, dbb in zip(batch_par_b,
delta_b)]
            batch_par_w = [bw + dbw for bw, dbw in zip(batch_par_w,
delta_w)]
        # 根据累加的偏导值 delta_b, delta_w 更新 b, w
        # 由于用了小样本，因此 eta 需除以小样本长度
        self.weights = [
            w - (eta / len(mini_batch)) * dw for w, dw in zip(self.weights,
batch_par_w)
```

```
        ]
        self.biases = [
            b - (eta / len(mini_batch)) * db for b, db in zip(self.biases,
batch_par_b)
        ]
```

数据集划分：

```
df1 = pd.read_excel("train.xls", 0)
df1 = df1.iloc[:, :]
# 进行数据归一化

min_max_scaler = preprocessing.MinMaxScaler()
df0 = min_max_scaler.fit_transform(df1)
df = pd.DataFrame(df0, columns=df1.columns)
x = df.iloc[:, :-1]
y = df.iloc[:, -1]
# 划分训练集测试集
cut = 5  # 取最后 cut=5 天为测试集
x_train, x_test = (
    x.iloc[:-cut],
    x.iloc[-cut:],
)  # 列表的切片操作
y_train, y_test = y.iloc[:-cut], y.iloc[-cut:]
x_train, x_test = x_train.values, x_test.values
y_train, y_test = y_train.values, y_test.values
```

完整代码如下：

Main.py

```
# -*- coding: utf-8 -*-
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import BPNN
from sklearn import metrics
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn import preprocessing


def inverse_transform_col(scaler, y, n_col):
    """scaler 是对包含多个 feature 的 X 拟合的,y 对应其中一个 feature,n_col 为 y
在 X 中对应的列编号.返回 y 的反归一化结果"""
    y = y.copy()
    y -= scaler.min_[n_col]
    y /= scaler.scale_[n_col]
    return y
# 导入必要的库
```

```
df1 = pd.read_excel("train.xls", 0)
df1 = df1.iloc[:, :]
# 进行数据归一化
min_max_scaler = preprocessing.MinMaxScaler()
df0 = min_max_scaler.fit_transform(df1)
df = pd.DataFrame(df0, columns=df1.columns)
x = df.iloc[:, :-1]
y = df.iloc[:, -1]
# 划分训练集测试集
cut = 5  # 取最后cut=30天为测试集
x_train, x_test = (
    x.iloc[:-cut],
    x.iloc[-cut:],
)  # 列表的切片操作，X.iloc[0:2400，0:7]即为1-2400行，1-7列
y_train, y_test = y.iloc[:-cut], y.iloc[-cut:]
x_train, x_test = x_train.values, x_test.values
y_train, y_test = y_train.values, y_test.values
# 神经网络搭建
bp1 = BPNN.BPNNRegression([5, 16, 1])
train_data = [[sx.reshape(5, 1), sy.reshape(1, 1)] for sx, sy in zip(x_train,
y_train)]
test_data = [np.reshape(sx, (5, 1)) for sx in x_test]
# 神经网络训练
bp1.MSGD(train_data, 500000, len(train_data), 0.01)
# 神经网络预测
y_predict = bp1.predict(test_data)
y_pre = np.array(y_predict)  # 列表转数组
y_pre = y_pre.reshape(5, 1)
y_pre = y_pre[:, 0]
# y_pre = min_max_scaler.inverse_transform(y_pre)
y_pre = inverse_transform_col(min_max_scaler, y_pre, n_col=0)  # 对预测值
反归一化
y_test = inverse_transform_col(
    min_max_scaler, y_test, n_col=0
)  # 对实际值反归一化（如果不想用，这两行删除即可）
# 画图 #展示在测试集上的表现
draw = pd.concat([pd.DataFrame(y_test), pd.DataFrame(y_pre)], axis=1)
draw.iloc[:, 0].plot(figsize=(12, 6))
draw.iloc[:, 1].plot(figsize=(12, 6))
plt.legend(("real", "predict"), loc="upper right", fontsize="15")
plt.title("Test Data", fontsize="30")  # 添加标题
plt.show()
# 输出精度指标
print("测试集上的MAE/MSE")
```

```
print(mean_absolute_error(y_pre, y_test))
print(mean_squared_error(y_pre, y_test))
mape = np.mean(np.abs((y_pre - y_test) / (y_test))) * 100
print("=============mape=============")
print(mape, "%")
# 画出真实数据和预测数据的对比曲线图
print("R2 = ", metrics.r2_score(y_test, y_pre))  # R2
```

BPNN.py

```
# encoding:utf-8

"""
BP 神经网络 Python 实现
"""
import random
import numpy as np
def sigmoid(x):
    """
    激活函数
    """
    return 1.0 / (1.0 + np.exp(-x))
def sigmoid_prime(x):
    return sigmoid(x) * (1 - sigmoid(x))
class BPNNRegression:
    """
    神经网络回归与分类的差别在于：
    1. 输出层不需要再经过激活函数
    2. 输出层的 w 和 b 更新量计算相应更改
    """
    def __init__(self, sizes):
        # 神经网络结构
        self.num_layers = len(sizes)
        self.sizes = sizes
        # 初始化偏差，除输入层外， 其它每层每个节点都生成一个 biase 值（0-1）
        self.biases = [np.random.randn(n, 1) for n in sizes[1:]]
        # 随机生成每条神经元连接的 weight 值（0-1）
        self.weights = [np.random.randn(r, c) for c, r in zip(sizes[:-1],
sizes[1:])]
    def feed_forward(self, a):
        """
        前向传输计算输出神经元的值
        """
        for i, b, w in zip(range(len(self.biases)), self.biases,
self.weights):
            # 输出神经元不需要经过激励函数
```

```
            if i == len(self.biases) - 1:
                a = np.dot(w, a) + b
                break
            a = sigmoid(np.dot(w, a) + b)
        return a
    def MSGD(self, training_data, epochs, mini_batch_size, eta,
error=0.01):
        """
        小批量随机梯度下降法
        """
        n = len(training_data)
        for j in range(epochs):
            # 随机打乱训练集顺序
            random.shuffle(training_data)
            # 根据小样本大小划分子训练集集合
            mini_batchs = [
                training_data[k : k + mini_batch_size]
                for k in range(0, n, mini_batch_size)
            ]
            # 利用每一个小样本训练集更新 w 和 b
            for mini_batch in mini_batchs:
                self.updata_WB_by_mini_batch(mini_batch, eta)
            # 迭代一次后结果
            err_epoch = self.evaluate(training_data)
            print("Epoch {0} Error {1}".format(j, err_epoch))
            if err_epoch < error:
                break
            # if test_data:
            #     print("Epoch {0}: {1} / {2}".format(j,
self.evaluate(test_data), n_test))
            # else:
            # print("Epoch {0}".format(j))
        return err_epoch
    def updata_WB_by_mini_batch(self, mini_batch, eta):
        """
        利用小样本训练集更新 w 和 b
        mini_batch: 小样本训练集
        eta: 学习率
        """
        # 创建存储迭代小样本得到的 b 和 w 偏导数空矩阵，大小与 biases 和
weights 一致，初始值为 0
        batch_par_b = [np.zeros(b.shape) for b in self.biases]
        batch_par_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
```

```
            # 根据小样本中每个样本的输入 x, 输出 y, 计算 w 和 b 的偏导
            delta_b, delta_w = self.back_propagation(x, y)
            # 累加偏导 delta_b, delta_w
            batch_par_b = [bb + dbb for bb, dbb in zip(batch_par_b,
delta_b)]
            batch_par_w = [bw + dbw for bw, dbw in zip(batch_par_w,
delta_w)]
        # 根据累加的偏导值 delta_b, delta_w 更新 b, w
        # 由于用了小样本，因此 eta 需除以小样本长度
        self.weights = [
            w - (eta / len(mini_batch)) * dw for w, dw in zip(self.weights,
batch_par_w)
        ]
        self.biases = [
            b - (eta / len(mini_batch)) * db for b, db in zip(self.biases,
batch_par_b)
        ]
    def back_propagation(self, x, y):
        """
        利用误差后向传播算法对每个样本求解其 w 和 b 的更新量
        x: 输入神经元，行向量
        y: 输出神经元，行向量
        """
        delta_b = [np.zeros(b.shape) for b in self.biases]
        delta_w = [np.zeros(w.shape) for w in self.weights]
        # 前向传播，求得输出神经元的值
        a = x  # 神经元输出值
        # 存储每个神经元输出
        activations = [x]
        # 存储经过 sigmoid 函数计算的神经元的输入值，输入神经元除外
        zs = []
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, a) + b
            zs.append(z)
            a = sigmoid(z)  # 输出神经元
            activations.append(a)
        # -------------
        activations[-1] = zs[-1]  # 更改神经元输出结果
        # -------------
        # 求解输出层 δ
        # 与分类问题不同，Delta 计算不需要乘以神经元输入的倒数
        # delta = self.cost_function(activations[-1], y) *
sigmoid_prime(zs[-1])
        delta = self.cost_function(activations[-1], y)  # 更改后
```

```
        # -------------
        delta_b[-1] = delta
        delta_w[-1] = np.dot(delta, activations[-2].T)
        for lev in range(2, self.num_layers):
            # 从倒数第1层开始更新，因此需要采用-lev
            # 利用 lev + 1 层的 δ 计算 l 层的 δ
            z = zs[-lev]
            zp = sigmoid_prime(z)
            delta = np.dot(self.weights[-lev + 1].T, delta) * zp
            delta_b[-lev] = delta
            delta_w[-lev] = np.dot(delta, activations[-lev - 1].T)
        return (delta_b, delta_w)
    def evaluate(self, train_data):
        test_result = [[self.feed_forward(x), y] for x, y in train_data]
        return np.sum([0.5 * (x - y) ** 2 for (x, y) in test_result])
    def predict(self, test_input):
        test_result = [self.feed_forward(x) for x in test_input]
        return test_result
    def cost_function(self, output_a, y):
        """
        损失函数
        """
        return output_a - y
    pass
```

2.7 数据集

表 1.液压控制软件失效数据

失效序号	失效时间	失效序号	失效时间	失效序号	失效时间	失效序号	失效时间
1	500	13	2010	25	3290	37	3830
2	800	14	2100	26	3320	38	3855
3	1000	15	2150	27	3350	39	3876
4	1100	16	2230	28	3430	40	3896
5	1210	17	2350	29	3480	41	3908
6	1320	18	2470	30	3495	42	3920
7	1390	19	2500	31	3540	43	3950
8	1500	20	3000	32	3560	44	3975
9	1630	21	3050	33	3720	45	3982
10	1700	22	3110	34	3750		
11	1890	23	3170	35	3795		
12	1960	24	3230	36	3810		

基于 BP 神经网络的软件可靠性模型可表示为：

$$x_t = f(x_{t-1}.x_{t-2},\ldots,x_{t-d},t)$$

对故障数据进行处理如下：以第 t 次故障之前的第 t-1,t-2,…,t-d 次故障时间数据作为输入，即 d 个故障时间数据x_{t-1},x_{t-2},…,x_{t-d}和第 t 次作为输入变量；以第 t 次失效时间数据x_t作为输出变量。依据工程的实际经验，当 d=5 能达到较好的预计效果。

取前 35 个训练样本对 BP 神经网络可靠性预计模型进行训练，训练达到一定精度后，停止训练。为了检验网络训练的效果，将后 5 个训练样本输入到网络。

2.8 算法结果分析

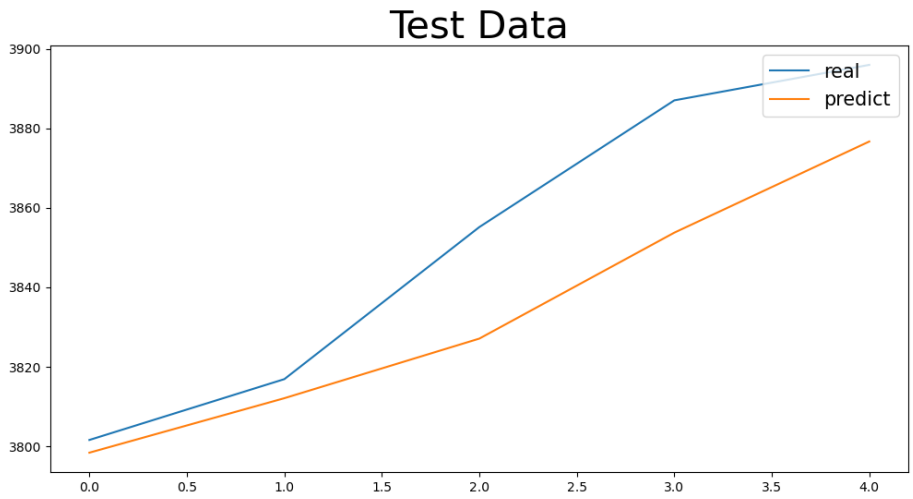

图 3.预测结果与真实结果对比

从输出结果可以看出，预计值与实测值比较接近，但存在一定的误差，这主要是由于样本数量较小的缘故，当样本量较大时，可较好地反映实际情况。

3. 基于灰色理论的可靠性模型

3.1 模型背景

灰色系统模型是利用较少的或不确切的表示灰色系统行为特征的原始数据序列生成变换后建立的，用以描述灰色系统内部事务连续变化过程的模型。灰色模型通过累加操作（AGO）的方法，对无规则的原始数据进行处理，从而弱化原始数据的随机性和波动性，生成有规则的准指数规律的数据，然后对这些生成的数据进行建模、预测。常用的灰色预测模型主要是经典的一阶 GM(1,1)模型。

3.2 算法原理

GM(1,1) 的“1”表示一阶模型，“1”也代表数据处理的次数。在算法中，累加生成操作是关键步骤。通过对原始数据进行累加，消除数据的波动性，使得数据变得平滑，适合进一步

建模。累加后的数据序列比原始序列更容易捕捉到整体趋势，因此更适合建立预测模型。

在得到累加后的数据序列之后，GM(1,1) 假设数据的变化符合一个简单的线性微分方程。具体来说，假设数据序列的变化规律可以通过一个微分方程来描述，方程的解能够提供数据的长期趋势。该微分方程的解形式是指数型的。

GM(1,1) 模型的参数（例如变化速率 a 和常数项 b）是通过最小二乘法来求解的。通过最小化模型的预测值与实际数据之间的误差，找到最符合实际数据的参数。这些参数决定了模型对未来数据的预测能力。

在得到了模型的参数后，可以根据模型对未来的数据进行预测。由于累加操作的使用，预测过程包括两个步骤：（1）通过预测累加序列（通过微分方程）得到未来值；（2）将预测的累加值还原为原始数据序列的预测值。

3.3 算法步骤

GM(1,1)模型由一个单变量和一阶微分方程构成：

$$\frac{dX_1}{dt} + a(t) = b \tag{5}$$

式中，a,b 为待定的参数，X_1为原始数据序列X_0的累加生成值，该式也被称为时间响应函数。设X_0为原始数据的 n 元序列，即$X_0 = [x_0(1), x_0(2), .., x_0(n)]$,做一次累加后得到一次累计（1-AGO）序列:

$$X_1 = [x_1(1), x_1(2), \dots, x_1(n)] \tag{6}$$

其中$x_1(k) = \sum_{i=1}^{k} x_0(i), k = 1,2, \dots, n$。

令Z_1为X_1的紧邻均值生成序列为：

$$Z_1 = [z_1(2), z_1(3), \dots, z_1(n)] \tag{7}$$

其中$z_1(k) = \frac{1}{2}(x_1(k) + x_1(k-1)), k = 2,3, \dots, n$。

若$\hat{a} = [a, b]^T$为待定参数列，记

$$\mathrm{Y} = \begin{bmatrix} \mathrm{x_0(2)} \\ \mathrm{x_0(3)} \\ \vdots \\ \mathrm{x_0(n)} \end{bmatrix}, \mathrm{B} = \begin{bmatrix} -\mathrm{z_1(2)} & 1 \\ -\mathrm{z_1(3)} & 1 \\ \vdots & \vdots \\ -\mathrm{z_1(n)} & 1 \end{bmatrix} \tag{8}$$

则由差分代替微分，可得到 GM(1,1)模型的最小二乘估计参数列满足：

$$\hat{a} = [a, b]^T = (B^T B)^{-1} B^T Y \tag{9}$$

由以上条件可以得到时间响应函数的解为：

$$\widehat{x_1}(k+1) = \left(x_0(1) - \frac{b}{a}\right) e^{-ak} + \frac{b}{a}, k = 1,2, \dots, n \tag{10}$$

又由$\sum_{i=1}^{k} \widehat{x_0}(k) = \widehat{x_1(k)}$,可得

$$\widehat{x_0}(k+1) = (1 - e^a)\left(x_0(1) - \frac{b}{a}\right) e^{-ak}, k = 1,2, \dots, n \tag{11}$$

令 k=1,2,…,n，由上式可得到 GM(1,1)的预测值：

$$\widehat{x^{(0)}} = \left(\widehat{x^{(0)}}(1), \widehat{x^{(0)}}(2), \dots, \widehat{x^{(0)}}(n)\right) \tag{12}$$

3.4 算法流程

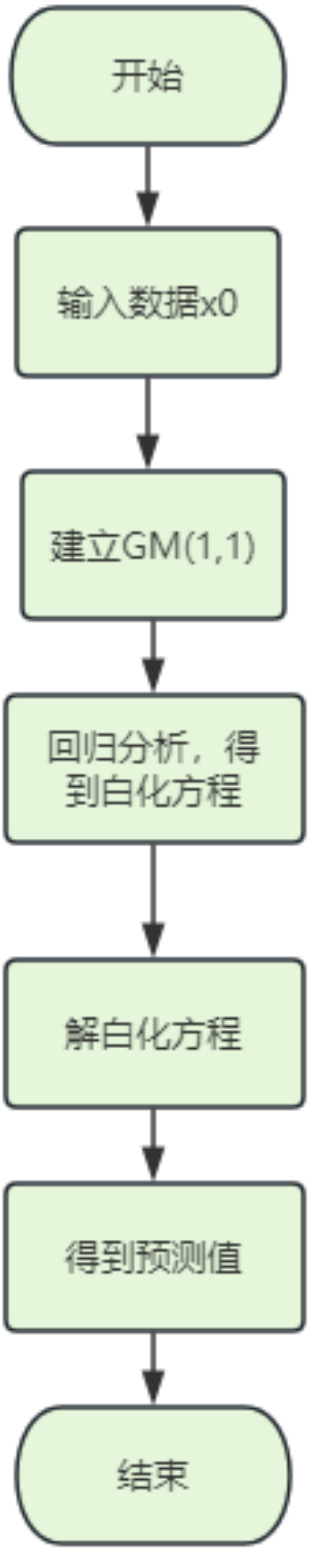

图 4.GM(1,1)算法流程图

3.5 算法实现

初始化：

```
    def __init__(self):
        self.test_data = np.array(())  # 实验数据集
        self.add_data = np.array(())  # 一次累加产生数据
        self.argu_a = 0  # 参数a
        self.argu_b = 0  # 参数b
        self.MAT_B = np.array(())  # 矩阵B
        self.MAT_Y = np.array(())  # 矩阵Y
        self.modeling_result_arr = np.array(())  # 对实验数据的拟合值
        self.P = 0  # 小误差概率
        self.C = 0  # 后验方差比值
```

构建累加数据和矩阵 B 和 Y，矩阵 B 用于线性回归计算参数，矩阵 Y 是目标变量：

```
    def __acq_data(self, arr: list):  # 构建并计算矩阵B和矩阵Y
        self.test_data = np.array(arr).flatten()
        add_data = list()
        sum = 0
        for i in range(len(self.test_data)):
            sum = sum + self.test_data[i]
            add_data.append(sum)
        self.add_data = np.array(add_data)
        ser = list()
        for i in range(len(self.add_data) - 1):
            temp = (-1) * ((1 / 2) * self.add_data[i] + (1 / 2) *
self.add_data[i + 1])
            ser.append(temp)
        B = np.vstack(
            (
                np.array(ser).flatten(),
                np.ones(
                    len(ser),
                ).flatten(),
            )
        )
        self.MAT_B = np.array(B).T
        Y = np.array(self.test_data[1:])
        self.MAT_Y = np.reshape(Y, (len(Y), 1))
```

计算灰参数 a,b

```
    def __compute(self):  # 计算灰参数 a,b
        temp_1 = np.dot(self.MAT_B.T, self.MAT_B)
        temp_2 = np.matrix(temp_1).I
        temp_3 = np.dot(np.array(temp_2), self.MAT_B.T)
        vec = np.dot(temp_3, self.MAT_Y)
```

```
        self.argu_a = vec.flatten()[0]
        self.argu_b = vec.flatten()[1]
```

预测函数：

```
    def __predict(self, k: int) -> float:  # 定义预测计算函数
        part_1 = 1 - pow(np.e, self.argu_a)
        part_2 = self.test_data[0] - self.argu_b / self.argu_a
        part_3 = pow(np.e, (-1) * self.argu_a * k)
        return part_1 * part_2 * part_3
```

完整代码及其它处理函数代码如下：

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


# 从 data.txt 文件中读取数据
def load_data(filename):
    # 读取文件中的每一行，并转换为浮点数
    with open(filename, "r") as file:
        data = file.readlines()
    # 将读取到的数据转换为浮动数组
    data = [float(line.strip()) for line in data]
    return np.array(data)
class GM_1_1:
    """
    使用方法：
    1、首先对类进行实例化：GM_model = GM_1_1()    # 不传入参数
    2、使用 GM 下的 set_model 传入一个一维的 list 类型数据：
GM_model.set_model(list1)
    3、想预测后 N 个数据：GM_model.predict(N)
       想获得模型某个参数或实验数据拟合值，直接访问，如：
GM_model.modeling_result_arr、GM_model.argu_a...等
        想输出模型的精度评定结果：GM_model.precision_evaluation()
    """
    def __init__(self):
        self.test_data = np.array(())  # 实验数据集
        self.add_data = np.array(())  # 一次累加产生数据
        self.argu_a = 0  # 参数a
        self.argu_b = 0  # 参数b
        self.MAT_B = np.array(())  # 矩阵B
        self.MAT_Y = np.array(())  # 矩阵Y
        self.modeling_result_arr = np.array(())  # 对实验数据的拟合值
        self.P = 0  # 小误差概率
        self.C = 0  # 后验方差比值
    def set_model(self, arr: list):
        self.__acq_data(arr)
        self.__compute()
```

```
        self.__modeling_result()
    def __acq_data(self, arr: list):  # 构建并计算矩阵 B 和矩阵 Y
        self.test_data = np.array(arr).flatten()
        add_data = list()
        sum = 0
        for i in range(len(self.test_data)):
            sum = sum + self.test_data[i]
            add_data.append(sum)
        self.add_data = np.array(add_data)
        ser = list()
        for i in range(len(self.add_data) - 1):
            temp = (-1) * ((1 / 2) * self.add_data[i] + (1 / 2) *
self.add_data[i + 1])
            ser.append(temp)
        B = np.vstack(
            (
                np.array(ser).flatten(),
                np.ones(
                    len(ser),
                ).flatten(),
            )
        )
        self.MAT_B = np.array(B).T
        Y = np.array(self.test_data[1:])
        self.MAT_Y = np.reshape(Y, (len(Y), 1))
    def __compute(self):  # 计算灰参数 a,b
        temp_1 = np.dot(self.MAT_B.T, self.MAT_B)
        temp_2 = np.matrix(temp_1).I
        temp_3 = np.dot(np.array(temp_2), self.MAT_B.T)
        vec = np.dot(temp_3, self.MAT_Y)
        self.argu_a = vec.flatten()[0]
        self.argu_b = vec.flatten()[1]
    def __predict(self, k: int) -> float:  # 定义预测计算函数
        part_1 = 1 - pow(np.e, self.argu_a)
        part_2 = self.test_data[0] - self.argu_b / self.argu_a
        part_3 = pow(np.e, (-1) * self.argu_a * k)
        return part_1 * part_2 * part_3
    def __modeling_result(self):  # 获得对实验数据的拟合值
        ls = [self.__predict(i + 1) for i in range(len(self.test_data) -
1)]
        ls.insert(0, self.test_data[0])
        self.modeling_result_arr = np.array(ls)
    def predict(self, number: int) -> list:  # 外部预测接口，预测后指定个数
的数据
```

```
        prediction = [self.__predict(i + len(self.test_data)) for i in
range(number)]
        return prediction
    def precision_evaluation(self):  # 模型精度评定函数
        error = [
            self.test_data[i] - self.modeling_result_arr[i]
            for i in range(len(self.test_data))
        ]
        aver_error = sum(error) / len(error)
        aver_test_data = np.sum(self.test_data) / len(self.test_data)
        temp1 = 0
        temp2 = 0
        for i in range(len(error)):
            temp1 = temp1 + pow(self.test_data[i] - aver_test_data, 2)
            temp2 = temp2 + pow(error[i] - aver_error, 2)
        square_S_1 = temp1 / len(self.test_data)
        square_S_2 = temp2 / len(error)
        self.C = np.sqrt(square_S_2) / np.sqrt(square_S_1)
        ls = [
            i
            for i in range(len(error))
            if np.abs(error[i] - aver_error) < (0.6745 *
np.sqrt(square_S_1))
        ]
        self.P = len(ls) / len(error)
        print("精度指标 P,C 值为: ", self.P, self.C)
    def plot(self):  # 绘制实验数据拟合情况（粗糙绘制，可根据需求自定义更改）
        plt.figure()
        plt.plot(self.test_data, marker="*", c="b", label="row value")
        plt.plot(self.modeling_result_arr, marker="^", c="r", label="fit
value")
        plt.legend()
        plt.grid()
        return plt
if __name__ == "__main__":
    GM = GM_1_1()
    import numpy as np
    # 假设 data.txt 位于当前目录
    filename = "data.txt"
    # 读取数据到 x
    x = load_data(filename)
    print("读取的数据: ", x)
    GM.set_model(x)
    print("模型拟合数据为: ", GM.modeling_result_arr)
```

```
GM.precision_evaluation()
print("后两个模型预测值为: ", GM.predict(5))
p = GM.plot()
p.show()
```

3.6 数据集

表 2.液压控制软件失效数据

失效序号	失效时间	失效序号	失效时间	失效序号	失效时间	失效序号	失效时间
1	500	13	2010	25	3290	37	3830
2	800	14	2100	26	3320	38	3855
3	1000	15	2150	27	3350	39	3876
4	1100	16	2230	28	3430	40	3896
5	1210	17	2350	29	3480	41	3908
6	1320	18	2470	30	3495	42	3920
7	1390	19	2500	31	3540	43	3950
8	1500	20	3000	32	3560	44	3975
9	1630	21	3050	33	3720	45	3982
10	1700	22	3110	34	3750		
11	1890	23	3170	35	3795		
12	1960	24	3230	36	3810		

3.7 算法结果分析

将原始数据与拟合结果数据对比

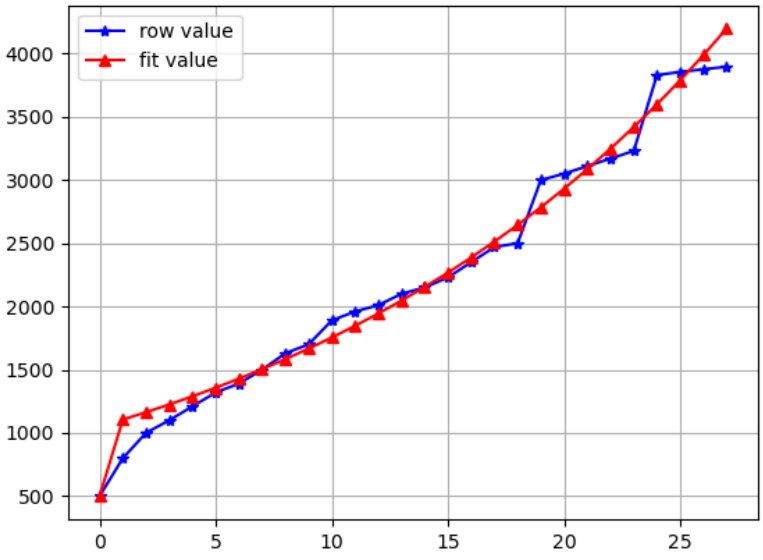

图 5.原始数据与拟合结果对比

预测后五个值

```
后五个模型预测值为： [4419.23871046345, 4651.965906199442, 4896.949047174802, 5154.8335594354, 5426.298858635626]
```

而原始数据为:[3908,3920,3950,3975,3982]，可以看到预测还是不够精准，造成的原因是数据样本过少。

4. 模型对比

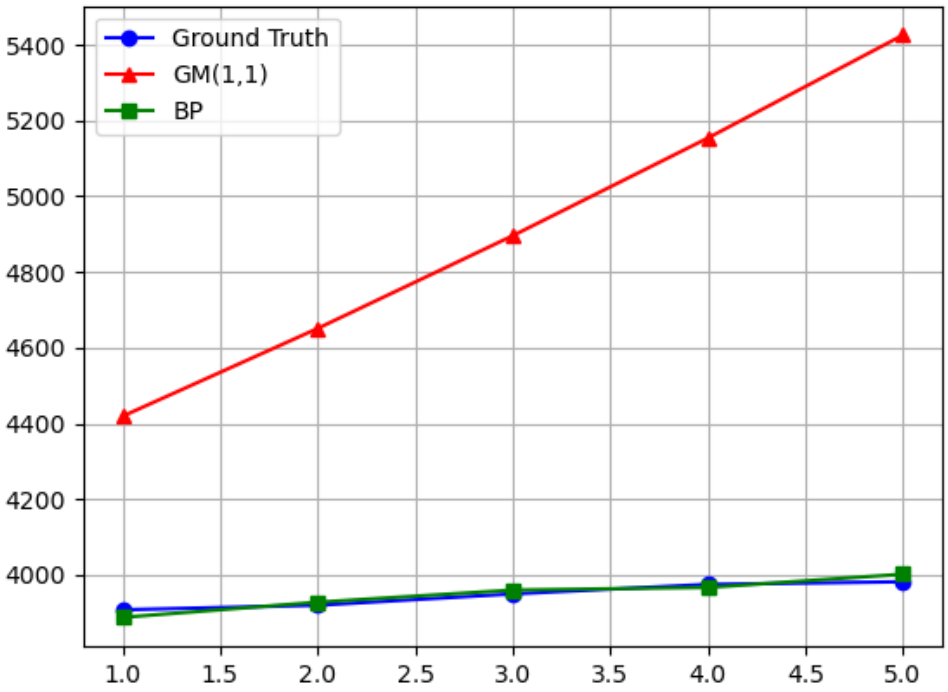

图 6.预测结果与真实数据对比

可以看到，在这一数据上 BP 神经网络的效果明显好于 GM(1,1)模型，我认为可能的原因有以下几点:

1. 数据建模能力

BP 神经网络是一种非线性模型，具有较强的学习能力，可以通过多层网络结构来模拟复杂的非线性关系。它能够通过训练过程自适应调整参数，以拟合数据中的复杂模式和规律。神经网络通过学习大量的样本数据，能够从数据中自动提取特征和规律，从而适应各种不同类型的数据。GM(1,1) 是基于灰色系统理论的模型，主要用于处理少量数据或不完整信息。GM(1,1) 通过一阶累加生成操作 (AGO) 来消除数据的波动性，进而建立线性微分方程模型来进行预测。由于它是基于线性假设和简单的累加过程，它对于复杂的、非线性或波动较大的数据可能不够灵活。

2. 适应性和灵活性

BP 神经网络能够通过多层的神经元结构（多层感知器）来适应不同的输入数据模式，它可以通过反向传播算法逐步调整网络权重，使模型能够自适应学习和拟合复杂的非线性关系。网络的层数和神经元数目可以根据问题的复杂性进行灵活调整，因此它对于复杂的时间

序列数据、季节性波动等问题表现更好。GM(1,1) 的适用性相对较弱，它假设数据的变化趋势是平滑的，并且依赖于数据的累加特性。对于数据波动较大或具有强烈非线性关系的情况，GM(1,1) 可能会无法准确建模。尽管它在某些简单或稳定的数据场景下表现较好，但面对复杂的变化模式时，可能会出现较大的误差。

3. 参数调整

BP 神经网络的训练过程是通过优化算法（如梯度下降、Adam 等）来调整网络的权重，训练数据越多，模型的拟合能力就越强。随着训练样本的增加，神经网络能够不断优化模型，减少误差。因此，在面对大量的数据时，神经网络能够自动适应并找到数据的规律。GM(1,1) 的参数是通过最小二乘法来确定的。虽然这种方法对于简单的线性关系有效，但它的参数调整仅依赖于原始数据的前几个点，并且只能够捕捉到数据的线性趋势。当数据的变化趋势复杂时，GM(1,1) 可能无法很好地拟合数据，导致预测效果较差。

参 考 文 献

[1]（中）张德平编著；软件系统可靠性分析基础与实践（Fundamentals and Practice of Software System Reliablity Analysis）清华大学出版社

[2]朱磊.基于 BP 神经网络的软件可靠性模型选择研究[D].重庆大学[2024-12-20].DOI:10.7666/d.y1017369.

[3]李燕.灰色预测模型的研究及其应用[D].浙江理工大学[2024-12-20].

[4]谢乃明,刘思峰.离散 GM(1,1)模型与灰色预测模型建模机理[J].系统工程理论与实践, 2005, 25(1):93-99.DOI:10.3321/j.issn:1000-6788.2005.01.014.