实验一 基于遗传算法的函数优化

一 实验目标

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

二 遗传算法主旨思想

1. 编码和产生初始群体

保留n位小数时, 设数在区间$[x,y]$ 之间则二进制串的最小位数j 需要满足如下关系式:

$$2^{j-1}<\frac{y-x}{10^n}<2^j$$

在此问题中$j=9$

在实际编程中无需先产生十进制的随机数再通过编码的方式转换为二进制串,而是直接随机生成指定位数的二进制串, 在需要用到数的真值时进行解码, 通过以下公式

$$b_{j-1}b_{j-2}\dots b_1b_0\Leftrightarrow x'=\sum_{i=0}^{j}b_i\cdot 2^i\Leftrightarrow x=a+x'\cdot\frac{y-x}{2^j-1}$$

2. 定义适应函数和适应值

$$g(x)=f(x)-F_{min}$$

此处取F_{min} 为当前种群$f(x)$ 的最小值即可保证适应度函数大于0, $f(x)$越大适应度最高.

3. 更新种群

确定选择标准

用适应值比例来作为入选概率, 适应度越高入选概率越大.

设给定的规模为n 的群体$pop=a_1,a_2,\dots,a_n$,个体 a_i的适应值为则其入选概率为$g(a_i)$

$$P_s(a_i)=\frac{g(a)}{\sum_{i=1}^{n}g(a_i)}(i=1,2,3,\dots,n)$$

交叉与变异

- 交叉也就是将一组染色体上对应基因段的交换得到新的染色体,然后得到新的染色体组,组成新的群体. 其有以下三个作用:
    - 帮助避免早熟收敛, 早熟收敛是指遗传算法在搜索空间中找到一个局部最优解而不是全局最优解的现象.通过交叉操作,遗传算法可以产生更多的多样性和新的组合,从而避免早熟收敛,并增加算法找到全局最优解的可能性.
    - 提高遗传算法的搜索能力: 交叉操作可以产生更多的多样性和新的组合,这些新的个体可以探索搜索空间中的新区域.通过交叉操作,遗传算法可以在搜索空间中更加全面地搜索,从而提高搜索能力.
    - 促进遗传算法的收敛速度: 交叉操作可以将优秀的基因组合在一起,并且丢弃不良的基因.这种操作可以快速地产生优秀的后代个体,并且提高后代个体的质量.这可以促进遗传算法的收敛速度,并加速找到最优解的过程.
- 变异就是通过一个小概率改变染色体位串上的某个基因. 其有以下四个作用:
    - 多样性维持: 变异操作可以在染色体的某个位置随机地改变基因值,从而产生新的个体,增加种群的多样性,避免陷入局部最优解.
    - 搜索空间扩展: 通过变异操作可以增加搜索空间,从而提高全局搜索性能.
    - 收敛速度控制: 变异概率不能太高,否则会导致算法的局部搜索性能降低,收敛速度变慢.一般来说,变异概率通常设置在0.001~0.01之间.
    - 算法鲁棒性提高: 在某些情况下,变异操作可以通过随机性的引入,增强算法的鲁棒性,使得算法更加适用于复杂的优化问题.

三 实验结果与分析

1. 实验结果

```
输出结果为:
DNA_SIZE: 9
最优的基因型:  [0 0 0 0 0 0 0 1 0]
x: -1.984344422700587
y: 4.999754902899422
```

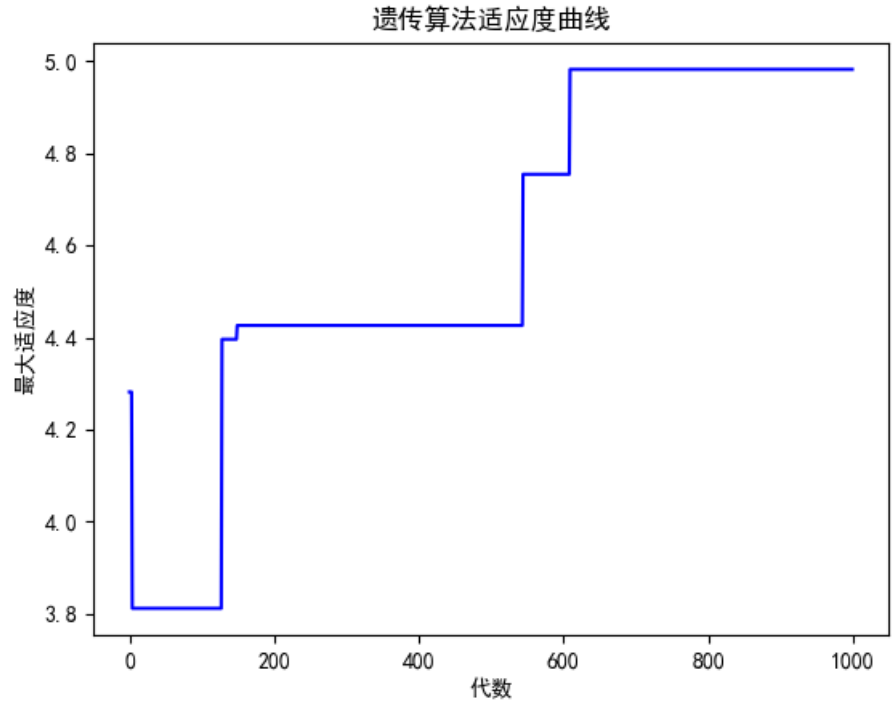

图1. 染色体交换概率: 0.6 变异概率0.01 选择方式: 轮盘赌&不保留精英

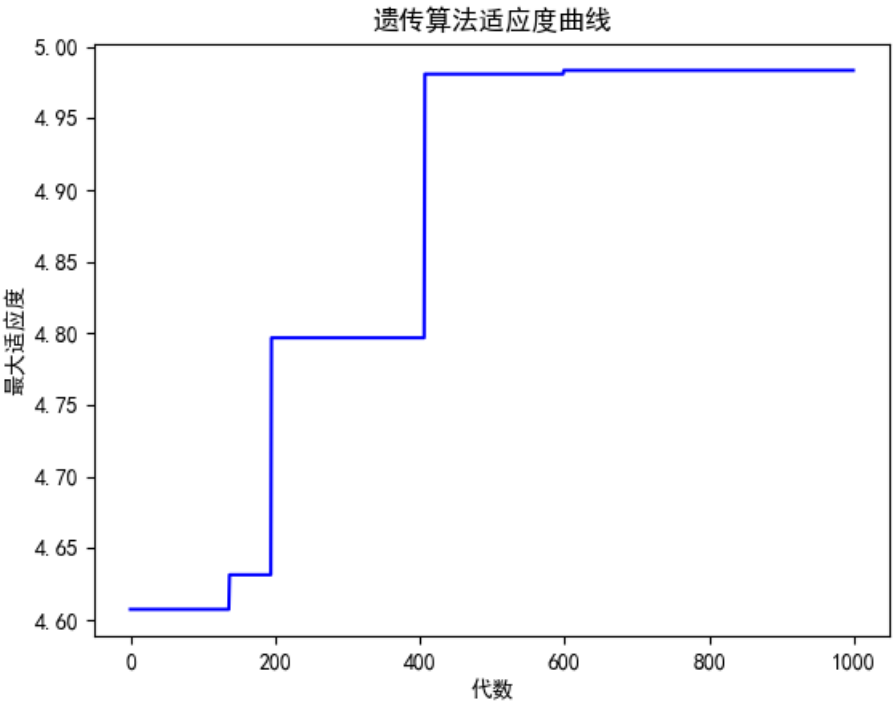

图2. 染色体交换概率: 0.6 变异概率0.01 选择方式: 轮盘赌&保留精英

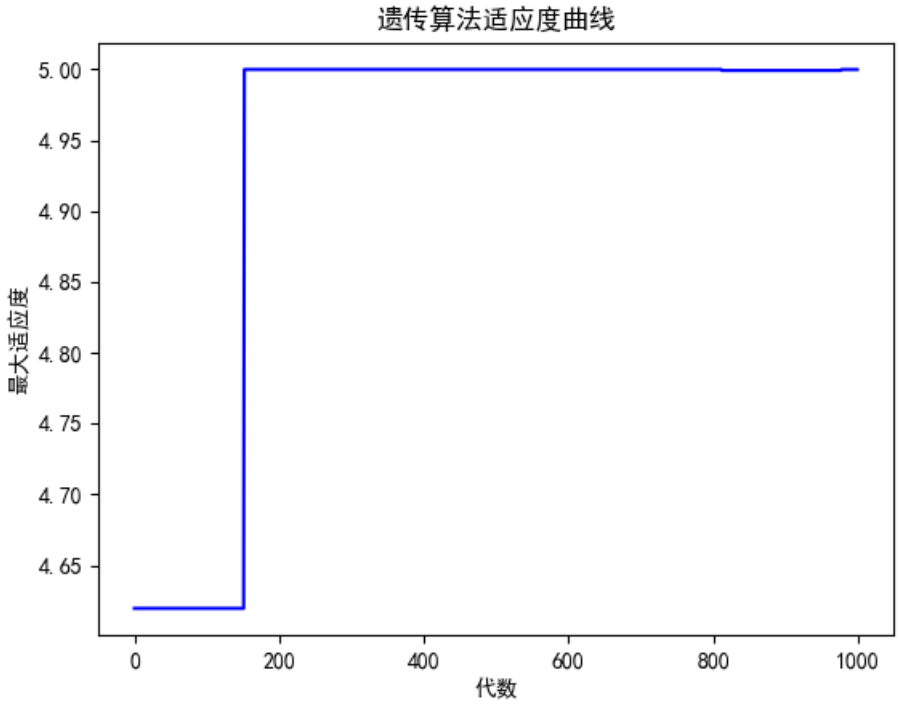

图3. 染色体交换概率: 0.6 变异概率0.5 选择方式: 轮盘赌&不保留精英

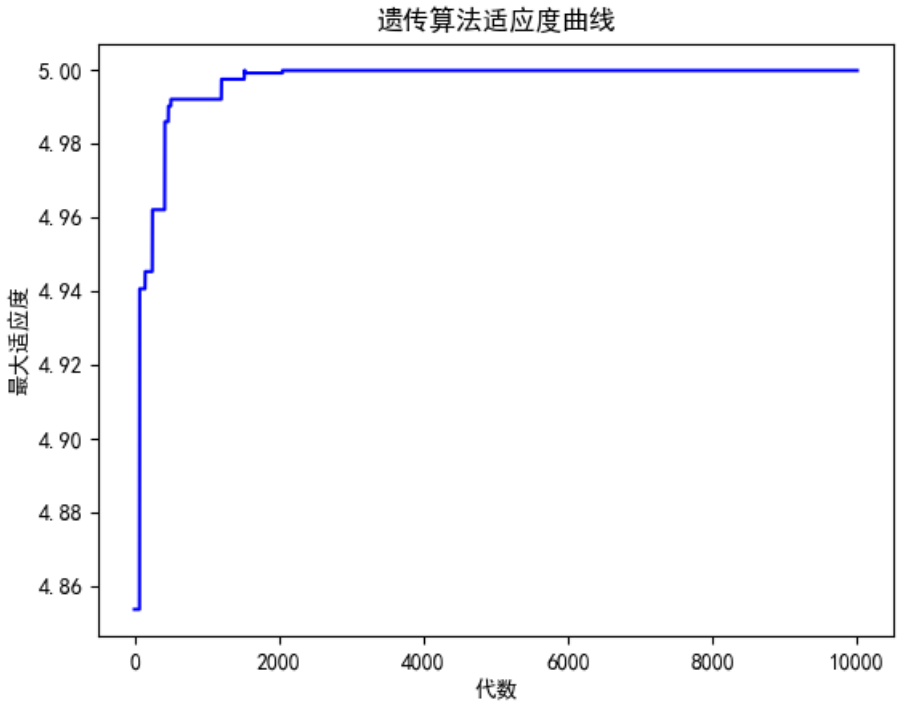

图4. 染色体交换概率: 0.01 变异概率0.01 选择方式: 轮盘赌&不保留精英

2. 结果分析

- 搜索具有随机性, 每次搜索的过程与搜索结果不相同, 但会收敛到最优解附近.
- 由于通过编码的方式对最优值进行求解会损失一定最优性. 例如, 该函数在$x=-2$时取到最大值, 但在算法中得到$x=-1.984$.
- 加入保留精英策略(对比图1, 图2)会有以下优缺点:
  - 优点 :
    1. 保留优秀个体: 精英策略可以保留上一代中的最优解,避免遗传算法的优良特性在进化过程中被稀释或丢失. 得到的结果会呈现持续优化的效果或者效果保持不变, 不会出现最优解变差的情况;
    2. 快速收敛: 由于精英策略的存在,可以加速算法的收敛速度;
    3. 算法鲁棒性: 精英策略可以提高算法的鲁棒性,降低算法陷入局部最优解的概率.
  - 缺点:
    1. 容易陷入局部最优: 如果精英策略过于强调保留历史最优解,有可能导致算法收敛到局部最优解而无法到达全局最优解;
    2. 缺乏多样性: 精英策略可能导致算法过度关注历史最优解,使种群缺乏多样性.
- 适当的变异概率会提高收敛速率(对比图1,图3), 但是变异概率不能太高,否则会导致算法的局部搜索性能降低,收敛速度变慢.变异概率通常设置在0.001~0.01之间.
- 适当的染色体交叉概率会提高收敛效率(对比图1, 图4), 过低的染色体交叉率导致图中经过2000代仍未收敛.

附录: 代码

```
import numpy as np
import matplotlib.pyplot as plt
from pylab import mpl
# 设置显示中文字体
mpl.rcParams["font.sans-serif"] = ["SimHei"]
# 设置正常显示符号
mpl.rcParams["axes.unicode_minus"] = False


DNA_SIZE = 9
POP_SIZE = 10
CROSSOVER_RATE = 0.6
MUTATION_RATE = 0.01
N_GENERATIONS = 1000
X_BOUND = [-2,2]
X_V = []
for j in range(30):
    bit = (X_BOUND[1] - X_BOUND[0])/1e-2
    if (2**j < bit and 2**(j+1) > bit):
        DNA_SIZE = j+1
        break
print("DNA_SIZE:",DNA_SIZE)


def F(x):
    return -x**2 - 4*x + 1

def get_fitness(pop):
    x = translateDNA(pop)
    pred = F(x)
    # return pred
    return pred - np.min(pred)+1e-3  # 求最大值时的适应度
    # fitness = np.max(pred) - pred + 1e-6
    # 求最小值时的适应度,通过这一步fitness的范围为[0, np.max(pred)-np.min(pred)]
    # return fitness

# pop表示种群矩阵,一行表示一个二进制编码表示的DNA,矩阵的行数为种群数目
def translateDNA(pop):
    x_pop = pop.copy()
```

```
    x = x_pop.dot(2 ** np.arange(DNA_SIZE)[::-1]) / float(2 ** DNA_SIZE - 1) * (X_BOUND[1] - X_BOUND[0]) + X_BOUND[0]
    return x


def crossover_and_mutation(pop, CROSSOVER_RATE=0.8):
    new_pop = []
    fitness = get_fitness(pop)
    max_fitness_index = np.argmax(fitness)
    for father in pop:  # 遍历种群中的每一个个体,将该个体作为父亲
        child = father  # 孩子先得到父亲的全部基因
        if np.random.rand() < CROSSOVER_RATE:  # 产生子代时不是必然发生交叉,而是以一定的概率发生交叉
            mother = pop[np.random.randint(POP_SIZE)]  # 再种群中选择另一个个体,并将该个体作为母亲
            cross_points = np.random.randint(low=0, high=DNA_SIZE * 2)  # 随机产生交叉的点
            child[cross_points:] = mother[cross_points:]  # 孩子得到位于交叉点后的母亲的基因
        mutation(child)  # 每个后代有一定的机率发生变异
        new_pop.append(child)
    new_pop[0] = pop[max_fitness_index] # 保留精英

    return new_pop


def mutation(child, MUTATION_RATE=0.003):
    if np.random.rand() < MUTATION_RATE:  # 以MUTATION_RATE的概率进行变异
        mutate_point = np.random.randint(0, DNA_SIZE)  # 随机产生一个实数,代表要变异基因的位置
        child[mutate_point] = child[mutate_point] ^ 1  # 将变异点的二进制位反转


def select(pop, fitness):  # nature selection wrt pop's fitness
    idx = np.random.choice(np.arange(POP_SIZE), size=POP_SIZE, replace=True,
                           p=(fitness) / (fitness.sum()))
    return pop[idx]


def print_info(pop):
    fitness = get_fitness(pop)
    max_fitness_index = np.argmax(fitness)
    print("max_fitness:", fitness[max_fitness_index])
    x = translateDNA(pop)
```

```
    print("最优的基因型: ", pop[max_fitness_index])
    print("x:",x[max_fitness_index])
    print(F(x[max_fitness_index]))


if __name__ == "__main__":
    plt.figure()
    pop = np.random.randint(2, size=(POP_SIZE, DNA_SIZE))  # matrix (POP_SIZE, DNA_SIZE)

    for i in range(N_GENERATIONS):  # 迭代N代
        x = translateDNA(pop)
        pop = np.array(crossover_and_mutation(pop, CROSSOVER_RATE))
        fitness = get_fitness(pop)
        max_fitness_index = np.argmax(fitness)
        x = translateDNA(pop)
        X_V.append(F(x[max_fitness_index]))
        pop = select(pop, fitness)  # 选择生成新的种群
    # X_V
    print_info(pop)
    plt.plot(list(range(0,N_GENERATIONS)),X_V,color='b')
    plt.xlabel('代数')
    plt.ylabel('最大适应度')
    plt.title('遗传算法适应度曲线')
    plt.show()
```

实验二 BP 神经网络

一 公式推导

1. Sigmoid 函数的导数

Sigmoid 函数:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Sigmoid 函数的导数为:

$$\begin{aligned}\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) \\ &= \sigma(1-\sigma)\end{aligned}$$

2. 均方差函数梯度

均方差损失函数表达式为:

$$L = \frac{1}{2}\sum_{k=1}^{K}(y_k - o_k)^2$$

其中y_k为真实值,o_k为输出值. 则它的偏导数$\frac{\partial L}{\partial o_i}$可以展开为:

$$\frac{\partial L}{\partial o_i} = \frac{1}{2}\sum_{k=1}^{K}\frac{\partial}{\partial o_i}(y_k - o_k)^2$$

均方差的导数可以推导为:

$$\frac{\partial L}{\partial o_i} = (o_i - y_i)$$

3. 反向传播算法

- 输出层

$$\frac{\partial L}{\partial w_{jk}} = \delta_k^K \cdot o_j$$

$$\delta_k^K = (o_k - t_k)o_k(1-o_k)$$

  输出层节点数为 `K` ,输出$o^k = [o_1^k, o_2^k, o_3^k, \ldots, o_k^k]$.

- 隐藏层

$$\frac{\partial L}{\partial w_{ij}} = \delta_j^J \cdot o_i$$

$$\delta_j^J = o_j(1 - o_j) \cdot \sum_k \delta_k^K \cdot w_{jk}$$

隐藏层的节点数为 `J` ,输出为 $o^J = [o_1^J, o_2^J, \ldots, o_J^J]$, o_i为输入变量.

二 算法步骤

1. 初始化权值及阈值为小的随机数;
2. 给出输入$x_0, x_1 \ldots x_{n-1}$及期望输出 $t_0, t_1 \ldots t_{n-1}$;
3. 正向传播, 逐层计算输出;
4. 反向传播, 修正权值.
5. 重复3 - 5,直到对所有样本权值不变.

三 实验结果与分析

1. 实验结果:

- 数据集 HDPE_15:

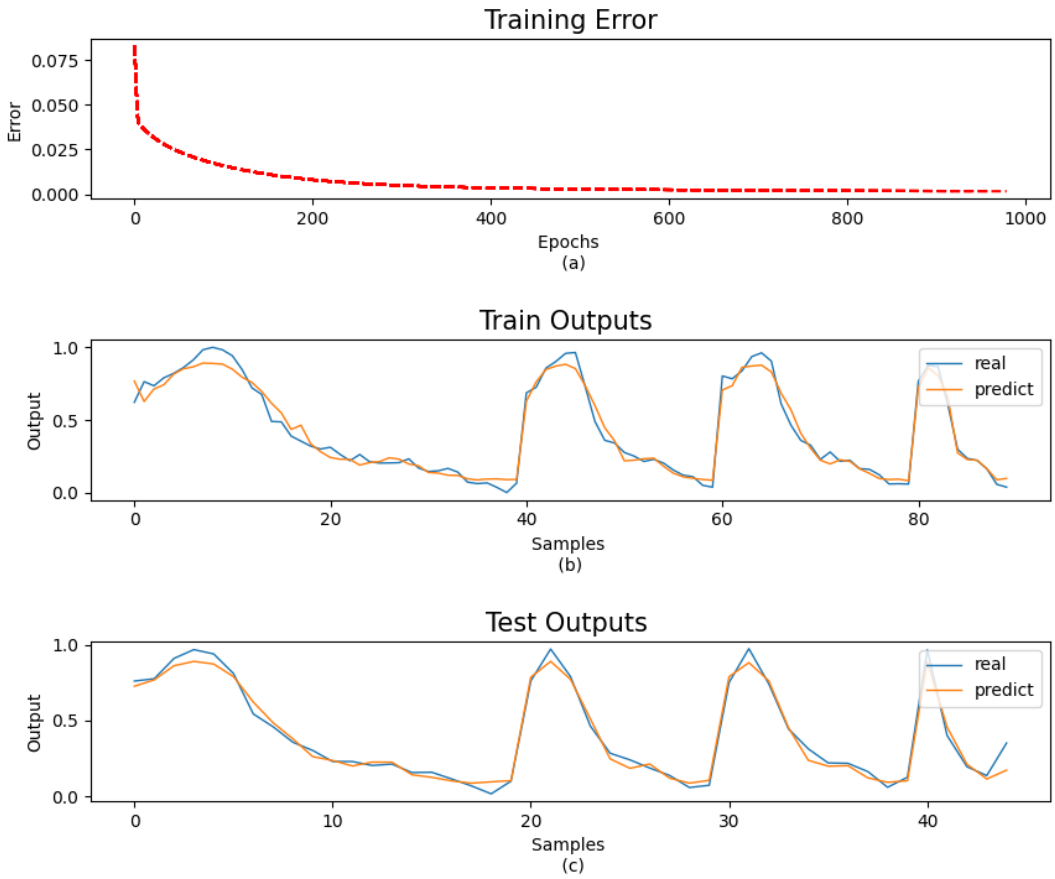

图1. 训练集1, 31个隐藏层节点, 训练1000代, 学习率0.6

- 数据集 PTA_17:

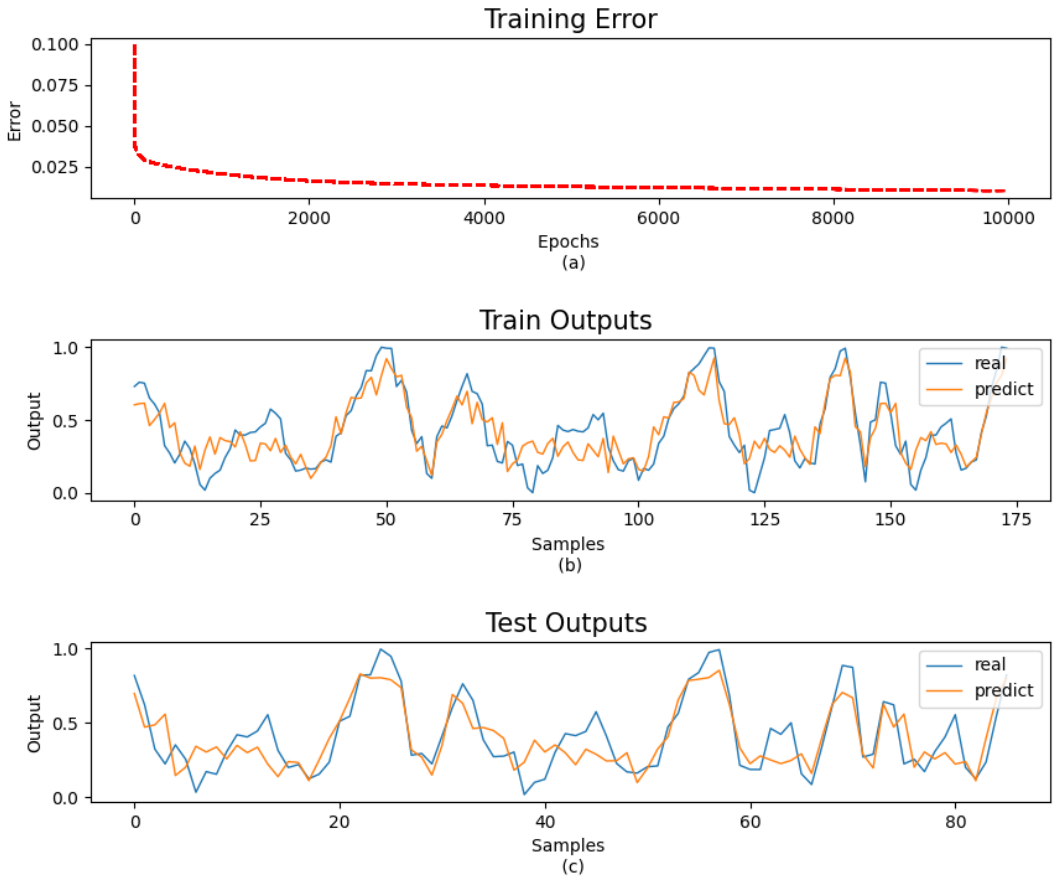

图2. 训练集2, 37个隐藏层节点, 训练10000代, 学习率0.6

其它实验:

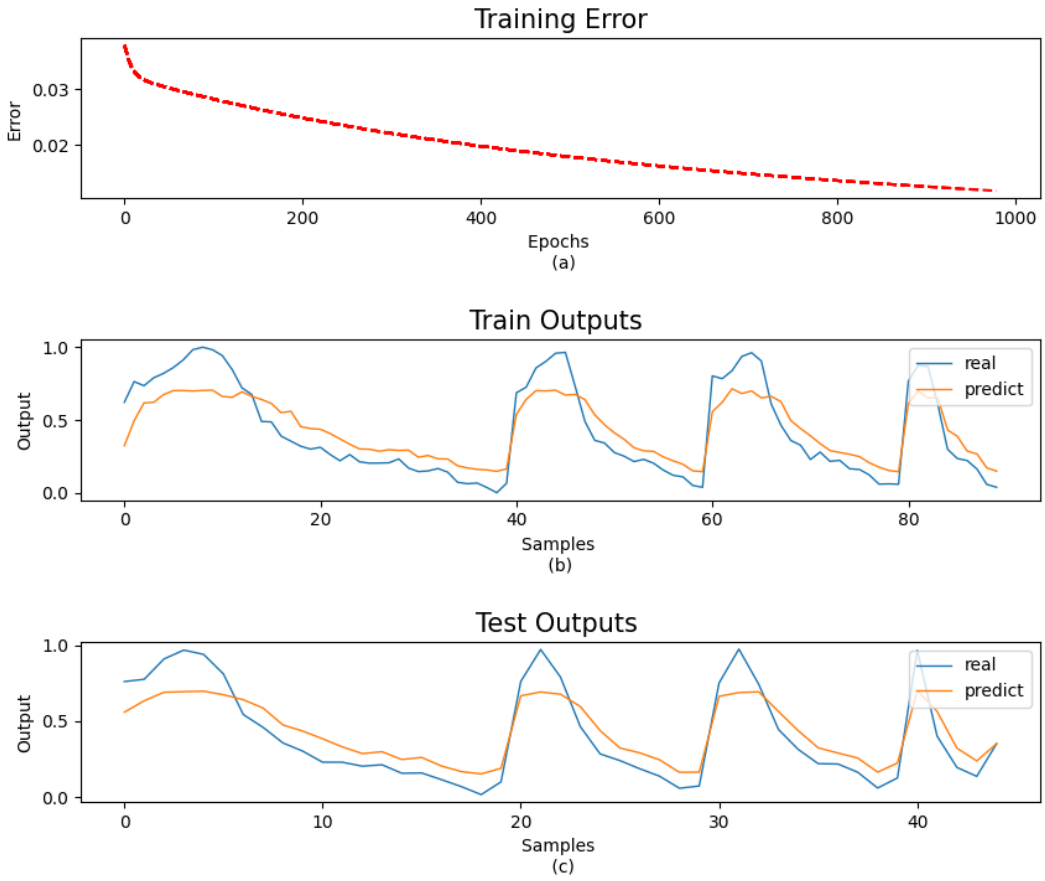

图3. 训练集1, 31个隐藏层节点, 训练1000代, 学习率0.06

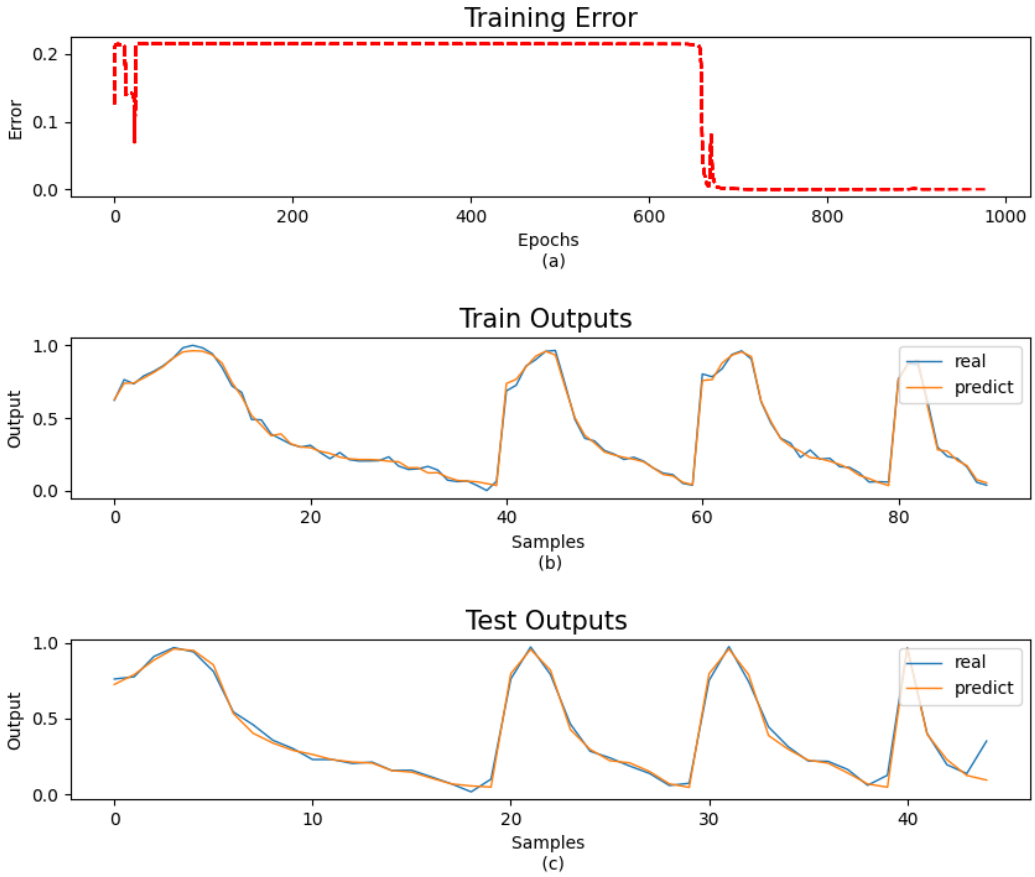

图4. 训练集1, 31个隐藏层节点, 训练1000代, 学习率60

2. 结果分析

本次实验之中, 此神经网络是一个三层全连接神经网络, 包含一个输入层、一个隐藏层和一个输出层. 若输入层有n 个节点, 隐藏层的节点数量为$2n+1$个. 由于$Sigmoid$ 函数具有非线性,光滑可导等优良特性, 此神经网络使用$Sigmoid$作为激活函数,并在训练时使用反向传播算法来更新权重和偏置.

在训练和测试之前,由于原始数据具有量纲, 数值差距大. 如果直接进行数值计算会损失计算结果精度和收敛速率. 所以数据需要经过归一化处理.

在迭代过程中,函数形状类似反比例函数且训练误差不断减小,减小速度不断降低. 改变学习速率和动量因子会影响网络的收敛速度和稳定性. 对比图1,3,4 可发现适中的学习率可有效提高训练收敛的速度, 过大的学习率(图4)会导致模型在梯度方向上来回震荡,甚至可能导致模型无法收敛.

附录: 代码

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


def fun_s(x):
    return 1.0 / (1.0 + np.exp(-x))


df1 = pd.read_excel('C:/Users/HP/Downloads/Bp神经网络/Bp神经网络/1.xlsx', 0)
df1 = df1.iloc[:, :]
# 进行数据归一化
min_max_scaler = preprocessing.MinMaxScaler()
df0 = min_max_scaler.fit_transform(df1)  # 将训练与测试集数据归一化
df = pd.DataFrame(df0, columns=df1.columns)
x = df.iloc[:, :-1]
y = df.iloc[:, -1]

Inum = x.shape[1]
Hnum = 31
Onum = 1

m, n = x.shape

# 划分训练集测试集
cut = 45  # 取最后60行为测试集
m = m-cut
# 列表的切片操作,X.iloc[0:2400,0:7]即为1-2400行,1-7列
x_t, x_gen = x.iloc[:-cut], x.iloc[-cut:]
d_t, d_gen = y.iloc[:-cut], y.iloc[-cut:]

x_t, d_t = x_t.values, d_t.values.reshape(-1, 1)
x_gen, d_gen = x_gen.values, d_gen.values.reshape(-1, 1)


# Initialize weights
Wih = 2 * np.random.rand(Inum, Hnum)-1
Who = 2 * np.random.rand(Hnum, Onum)-1
```

```
# Wih = np.ones((Inum, Hnum))
# Who = np.ones((Hnum, Onum))
dw_wih = np.zeros((Inum, Hnum))
dw_who = np.zeros((Hnum, Onum))

# Train network
E_max = 1e-4
Train_num = 1000
eta = 0.6
aerf = 0.4
E = []
plt.subplots(nrows=3, ncols=1, figsize=(10, 8))
plt.subplots_adjust(hspace=0.886)

for i in range(Train_num):
    Hin = np.dot(x_t, Wih)
    Hout = fun_s(Hin)
    Opin = np.dot(Hout, Who)
    Opot = fun_s(Opin)
    E_p = (d_t - Opot)
    E_train = np.sum(0.5 * E_p ** 2) / m
    E.append(E_train)

    if i % 20 == 0:
        plt.subplot(3, 1, 1)
        plt.title("Training Error", fontsize='15')  # 添加标题
        plt.xlabel("Epochs \n (a)")
        plt.ylabel("Error")
        plt.plot(E[:i], 'r--')
        plt.pause(0.001)
        plt.draw()

    if E_train < E_max:
        flag = 1
        break

    detea_ho = Opot * (1 - Opot) * E_p

    dw_ho = np.zeros((Hnum, m))
    for j in range(m):
        tmp_Hout = Hout[[j]].T
        dw_ho[:, j, np.newaxis] = eta * detea_ho[j] * tmp_Hout
```

```
    dw_who = np.mean(dw_ho, axis=1, keepdims=True) + aerf * dw_who

    detea_ih = np.zeros((m, Hnum))
    for j in range(m):
        detea_ih[j] = Hout[j, :] * (1 - Hout[j, :]) * (detea_ho[j] * Who.T)

    dw_ih = np.zeros((Inum, Hnum, m))
    for j in range(m):
        for k in range(Hnum):
            tmp_x_t = x_t[[j]].T
            dw_ih[:, k, j, np.newaxis] = eta * detea_ih[j, k] * tmp_x_t
    dw_wih = np.mean(dw_ih, axis=2) + aerf * dw_wih
    Wih = Wih + dw_wih  # 更新权重
    Who = Who + dw_who
Hin_train = np.dot(x_t, Wih)
Hout_train = fun_s(Hin_train)
Opin_train = np.dot(Hout_train, Who)
Opot_train = fun_s(Opin_train)

# plt.figure()
# E_r = abs(d_t - Opot_train)
# plt.plot(E_r, linewidth=1, linestyle="solid")

# # plt.legend(('real', 'predict'),loc='upper right',fontsize='15')
# plt.title("Training Error",fontsize='20') #添加标题


plt.subplot(3, 1, 2)
plt.plot(d_t, linewidth=1, linestyle="solid")
plt.plot(Opot_train, linewidth=1, linestyle="solid")
plt.ylabel("Output")
plt.xlabel("Samples \n (b) ")
plt.legend(('real', 'predict'), loc='upper right', fontsize='10')
plt.title("Train Outputs ", fontsize='15')  # 添加标题


Hin_train = np.dot(x_gen, Wih)
Hout_train = fun_s(Hin_train)
Opin_train = np.dot(Hout_train, Who)
Opot_train = fun_s(Opin_train)

plt.subplot(3, 1, 3)
```

```
plt.plot(d_gen, linewidth=1, linestyle="solid")
plt.plot(Opot_train, linewidth=1, linestyle="solid")
plt.ylabel("Output")
plt.xlabel("Samples \n (c)")
plt.legend(('real', 'predict'), loc='upper right', fontsize='10')
plt.title("Test Outputs ", fontsize='15')  # 添加标题
plt.show()

```