第 5 章

粒子群算法

粒子群算法及蚁群算法均属于群智能(Swarm Intelligence, SI)算法。人们将群居昆虫的集体行为称作“群智能”(“群体智能”“群集智能”“集群智能”等),它的特点是个体的行为很简单,但当它们一起协同工作时,却能够凸显出非常复杂(智能)的行为特征,如图 5-1 和图 5-2 所示。



图 5-1　鸟群捕食的群体行为



图 5-2　蚁群

群智能的主要优点是：灵活性，群体可以适应随时变化的环境；稳健性，即使个体失败，整个群体仍能完成任务；自我组织，活动既不受中央控制，也不受局部监管。

群智能作为一种新兴的演化计算技术，已成为研究焦点，它与人工生命，特别是进化策略及遗传算法有着极为特殊的关系。群智能指不够智能的主体通过合作表现出智能行为的特性，在没有集中控制且不提供全局模型的前提下，为寻找复杂的分布式问题求解方案提供了基础。

粒子群(Particle Swarm Optimization，PSO)算法(又称为粒子群优化算法)就是以群鸟觅食的模型寻找最优值。最早是由美国电气工程师 Eberhart 和社会心理学家 Kennedy 在 1995 年基于群鸟觅食提出来的。粒子群算法的主要特点是方法简单、易于编程实现、参数小、收敛速度快。该算法在函数优化、神经网络训练、工业系统优化和模糊系统控制等领域得到了广泛的应用。

5.1 粒子群算法的原理

群鸟在觅食的过程中，每只鸟的初始状态都处于随机位置，且飞翔的方向也是随机的。每只鸟都不知道食物在哪里，但是随着时间的推移，这些初始处于随机位置的鸟类通过群类相互学习、信息共享和个体不断积累寻觅食物的经验，自发组织积聚成一个群落，并逐渐朝位移的目标(食物)前进。每只鸟能够通过一定经验和信息估计目前所处的位置对于能寻找到食物有多大的价值，即多大的适应值；每只鸟能够记住自己所找到的最好位置，称为局部最优(pbest)。此外，每只鸟还能记住群鸟中所有个体所能找到的最好位置，称为全局最优(gbest)，整个鸟群的觅食中心都趋向全局最优移动，这在生物学上称为“同步效应”。通过鸟群觅食的位置不断移动，即不断迭代，可以使鸟群朝食物步步逼近。

在群鸟觅食模型中，每个个体都可以看成一个粒子，则鸟群可以被看成一个粒子群。假设在一个 D 维的目标搜索空间中，有 m 个粒子组成一个群体，其中第 i 个粒子($i=1,2,\cdots,m$)位置表示为 $X_i=(x_i^1,x_i^2,\cdots,x_i^D)$，即第 i 个粒子在 D 维搜索空间中的位置是 X_i。换言之，每个粒子的位置就是一个潜在的解，将 X_i 代入目标函数就可以计算出其适应值，根据适应值的大小衡量其优劣。粒子个体经历过的最好位置记为 $P_i=(p_i^1,p_i^2,\cdots,p_i^D)$，整个群体所有粒子经历过的最好位置记为 $P_g=(p_g^1,p_g^2,\cdots,p_g^D)$。粒子 i 的速度记为 $V_i=(v_i^1,v_i^2,\cdots,v_i^D)$。则粒子群算法采用下列公式对粒子所在的位置不断更新(单位时间 1)：

$$v_i^d=\omega v_i^d+c_1r_1(p_i^d-x_i^d)+c_2r_2(p_g^d-x_i^d) \tag{5-1}$$

$$x_i^d=x_i^d+\alpha v_i^d \tag{5-2}$$

其中，$i=1,2,\cdots,m$；$d=1,2,\cdots,D$；ω 是非负数，称为惯性因子；加速常数 c_1 和 c_2 是非负常数；r_1 和 r_2 是[0,1]范围内变换的随机数；α 称为约束因子，目的是控制速度的权重。此外，$v_i^d\in[-v_{\max}^d,v_{\max}^d]$，即粒子 i 的飞翔速度 V_i 被一个最大速度 $V_{\max}=(v_{\max}^1,v_{\max}^2,\cdots,v_{\max}^D)$ 所限制。如果当前时刻粒子在某维的速度 v_i^d 更新后超过该维的最大飞翔速度 $v_{\max}^d$，则当前时刻该维的速度被限制在 $v_{\max}^d$。另外，$v_{\max}$ 为常数，可以根据不同的优化问题设定。迭代终止条件根据具体问题设定，一般达到预定最大迭代次数或粒子群目前为止搜索到的最优位置满足目标函数的最小容许误差。

5.2　粒子群算法的优缺点及程序设计

5.2.1　粒子群算法的优点及缺点

粒子群算法的优点如下。

(1) 不依赖问题的导数信息,采用目标函数值求解,算法通用性强。

(2) 需要调整的参数少,原理简单,容易实现。

(3) 协同搜索,同时利用个体局部信息和群体全局信息指导搜索。

(4) 收敛速度快,算法对计算机内存和 CPU 要求不高。

(5) 更容易跳出局部最优信息。

粒子群算法的缺点如下。

(1) 算法局部搜索能力较差,搜索精度不够高。

(2) 算法不能绝对保证搜索到全局最优解。

(3) 算法搜索性能对参数具有一定的依赖性。

(4) 粒子群算法是一种概率算法,算法理论不完善,缺乏独特性,理论成果偏少。

(5) 欠完善的生物学背景。

5.2.2　粒子群算法的程序设计

粒子群算法程序设计的主要步骤如下。

(1) 初始化粒子群(速度和位置)、惯性因子、加速常数、最大迭代次数和算法终止的最小允许误差。

(2) 评价每个粒子的初始适应值。

(3) 将初始适应值作为当前每个粒子的局部最优值,并将各适应值对应的位置作为每个粒子的局部最优值所在的位置。

(4) 将最佳初始适应值作为当前全局最优值,并将最佳初始适应值对应的位置作为每个粒子的局部最优值所在的位置。

(5) 依据式(5-1)更新每个粒子当前的飞翔速度。

(6) 对每个粒子的飞翔速度进行限速处理,使之不能超过设定的最大飞翔速度。

(7) 依据式(5-2)更新每个粒子当前所在的位置。

(8) 比较当前每个粒子的适应值是否比历史局部最优值好,如果是,则将当前粒子适应值作为粒子的局部最优值,其对应的位置作为每个粒子的局部最优值所在的位置。

(9) 在当前群中找出全局最优值,并将当前全局最优值对应的位置作为粒子群的全局最优值所在的位置。

(10) 重复步骤(5)～(9),直到满足设定的最小误差或者达到最大迭代次数。

(11) 输出粒子群全局最优值和其对应的位置,以及每个粒子的局部最优值和其对应的位置。

5.2.3　粒子群算法的参数选取

在基本粒子群算法中,粒子数 m、惯性因子 ω、最大飞翔速度 $v_{\max}$ 和加速常数 c_1 与 c_2 等

几个参数对算法寻优性能的影响非常显著。

(1) 粒子数 m,其一般取值为 20～40。粒子数量越多,搜索范围越大,越容易找到全局最优解。

(2) 惯性因子 ω 对粒子群算法的收敛性起到非常大的作用,ω 值越大,粒子飞翔速度越大,容易错失局部寻优能力,而全局搜索能力越强。反之,则局部寻优能力增强,而全局寻优能力减弱。

(3) 加速常数 c_1 与 c_2,一般情况下取 $c_1=c_2=2$。

(4) 最大飞翔速度 $v_{\max}$的选择,需要对问题有一定的先验知识。如果 $v_{\max}$的选择是固定不变的,通常 $v_{\max}$设定为每维变化范围的 10%～20%。

此外,粒子飞翔速度和位置的初始化以及适应度函数的设计也会对算法性能产生一定的影响。目前粒子群算法中的适应度函数的设计主要是借鉴遗传算法适应度函数的设计方法。

用粒子群算法求解约束优化问题主要有两类方法。

(1) 罚函数法。罚函数的目的是将约束优化问题转换为无约束优化问题。

(2) 将粒子群的搜索范围都限制在条件约束族内,即在可行解范围内寻优。

5.3 粒子群算法实例

本节以粒子群算法分别解决一元函数、二元函数、旅行商问题(TSP)、神经网络的学习优化以及带约束的二元函数优化问题。

基于粒子群算法的函数极值寻优的流程如图 5-3 所示。

随机初始化粒子速度和粒子位置,由目标函数计算粒子适应度值,根据初始粒子适应度值确定个体极值和群体极值;根据式(5-1)及式(5-2)更新粒子速度和位置;根据新种群中粒子适应度值更新个体极值和群体极值。

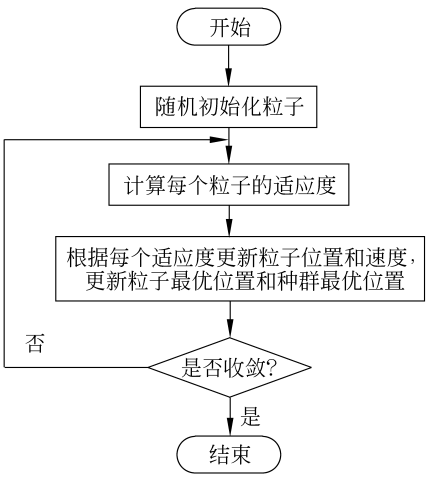


图 5-3 粒子群算法的流程

在例 5-1 中,适应度函数为函数表达式,适应度值为函数值。种群粒子数设置为 50,每个粒子的维数为 1,算法迭代次数定为 100。

【例 5-1】 利用粒子群算法求解如下的一元函数优化问题:

$$\max\ y=f(x)=2.2(2x^2-x+1)\mathrm{e}^{-x^2/2},\quad x\in[-2,2]$$

(1) 问题分析:根据式(5-1)及式(5-2)可以设计粒子群算法求解一元函数的最大值。

(2) Python 代码实现。

程序:

```
import math
from numpy import random
import matplotlib.pyplot as plt
import numpy as np
```

```
E0=0.001                                                    #允许误差
MaxNum=100                                                  #粒子最大迭代次数
narvs=1                                                     #目标函数的自变量个数
particlesize=50                                             #粒子群规模
c1=2                                                #每个粒子的个体学习因子,也称为加速常数
c2=2                                                #每个粒子的社会学习因子,也称为加速常数
w=0.6                                                       #惯性因子
vmax=0.8                                                    #粒子的最大飞翔速度
x=-5+10*random.random(size=(particlesize,narvs))            #粒子所在位置
v=2*random.random(size=(particlesize,narvs))                #粒子的飞翔速度

def fitness(x):
    return 1/(2.2*(1-x+2*x**2)*math.exp(-x**2/2))
f1=[] #记录函数值
for i in range(0,particlesize):
    for j in range(0,narvs):
        f1.append(fitness(x[i,j]))
personalbest_x=x                                            #记录个体
personalbest_faval=f1                                       #记录个体的位置
globalbest_faval=min(personalbest_faval)                    #将个体最优值设置为全局最优值
location=personalbest_faval.index(globalbest_faval) #记录全局最优的位置
globalbest_x=personalbest_x[location,:];
print("全局最优值为:",globalbest_faval)
print("全局最优的位置为:",globalbest_x)

k=1
while k<=MaxNum:
    #更新参数
    for i in range(0,particlesize):
        for j in range(0,narvs):
            f1[i]=fitness(x[i,j])
        if f1[i]<personalbest_faval[i]:          #判断当前位置是否是历史上的最佳位置
            personalbest_faval[i]=f1[i]
            personalbest_x[i,:]=x[i,:]
    globalbest_faval=min(personalbest_faval)    #将个体最优值设置为全局最优值
    location=personalbest_faval.index(globalbest_faval)
#记录全局最优值的位置
    globalbest_x=personalbest_x[location,:]     #记录全局最优值的个体

    for i in range(0,particlesize):              #更新粒子群里每个个体的最新位置
        #random.random()产生[0,1)的随机数

        v[i,:]=w*v[i,:]+c1*random.random()*(personalbest_x[i,:]-x[i,:])+c2
*random.random()*(globalbest_x-x[i,:])
        for j in range(0,narvs):
            if v[i,j]>vmax:
                v[i,j]=vmax
            elif v[i,j]<-vmax:
                v[i,j]=-vmax
        #for i in range(0,len(x)):
        x[i,:]=x[i,:]+0.8*v[i,:]
```

```
        if abs(globalbest_faval)<E0:
            break
        k=k+1
    #输出最大值所在的横坐标位置
    Value2=globalbest_x
    Value2=str(Value2);
    print("最大值横坐标为:",Value2)

    #画图
    x=[5/50*i for i in range(-50,50)]
    y=[2.2*(1-x+2*x**2)*math.exp(-x**2/2) for x in x]
    x1=np.array([-5,-4,-3,-2,-1,0,1,2,3,4,5])
    xlabel=np.array(["-5",'-4','-3','-2','-1','0','1','2','3','4','5'])
    plt.xticks(x1,xlabel)
    plt.plot(x, y, marker='', ms=10)
    plt.plot(globalbest_x,1/globalbest_faval,marker='*', ms=10)
    plt.xlim([-5,5])
    plt.ylim([0,6])
    plt.xlabel("x")
    plt.ylabel("y")
    plt.legend(labels=['目标函数','搜索到的最大值'],loc='upper right', fontsize=13)
    plt.grid()
    plt.show()
```

程序运行结果：

粒子群算法求解得到的优化值如图 5-4 所示。

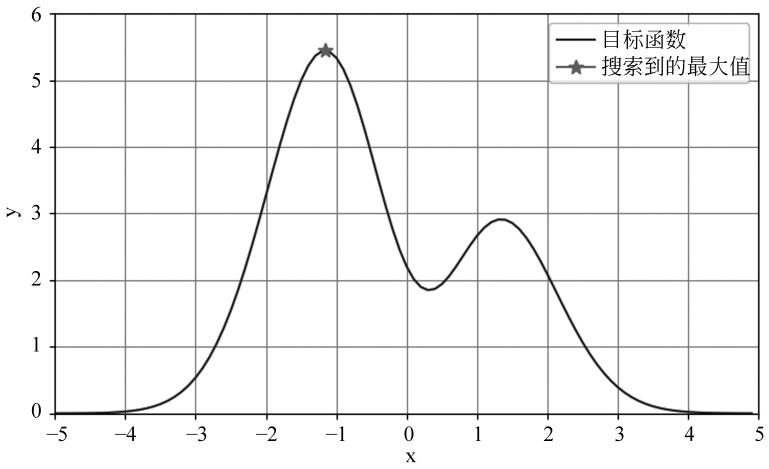


图 5-4　粒子群算法求解一元函数优化问题

粒子群算法求解得到的优化值如下所示：

最优解为 $x=-1.1617022$，最优值为 5.44602

说明：通过实验结果可以发现，最优解的横坐标为 -1.1617022，最大值为 5.44602。通过图 5-4 可以看出，粒子群算法没有陷入局部最优值，而是正确找到了全局最优。

(3) 该问题再分别用梯度下降法、遗传算法求解，比较这几种优化方法的结果。

利用梯度下降法求解一元函数优化问题的程序：

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

def target_function(x):
    return 2.2 * (1 - x + 2 * x**2) * math.exp(-x**2 / 2)

def gradient(x):
    numerator = (8.8 * x - 4.4) * x * math.exp(-x**2 / 2)
    denominator = (1 - x + 2 * x**2)**2
    return -numerator / denominator

def gradient_descent(learning_rate, max_iterations, tolerance):
    x = np.random.uniform(-2, 2)
    iteration = 0
    error_list = []

    while True:
        gradient_value = gradient(x)
        x_new = x - learning_rate * gradient_value

        error = abs(x_new - x)
        error_list.append(error)

        if error < tolerance or iteration >= max_iterations:
            break

        x = x_new
        iteration += 1

    return x, error_list


def plot_error_curve(iterations, errors):
    #处理中文乱码
    plt.rcParams['font.sans-serif'] = ['Microsoft YaHei']
    plt.plot(iterations, errors)
    plt.xlabel('迭代次数')
    plt.ylabel('误差值')
    plt.show()

learning_rate = 0.1                                    #学习率
max_iterations = 500                                   #最大迭代次数
tolerance = 1e-6                                       #停止条件:参数变化小于容差

x_optimal, error_list = gradient_descent(learning_rate, max_iterations,
tolerance)
iterations = range(len(error_list))

print("最优解: x =", x_optimal)
print("最优值: f(x) =", target_function(x_optimal))
```

```
plot_error_curve(iterations, error_list)
```

程序运行结果：

梯度下降法求解一元函数优化问题的迭代过程如图 5-5 所示。

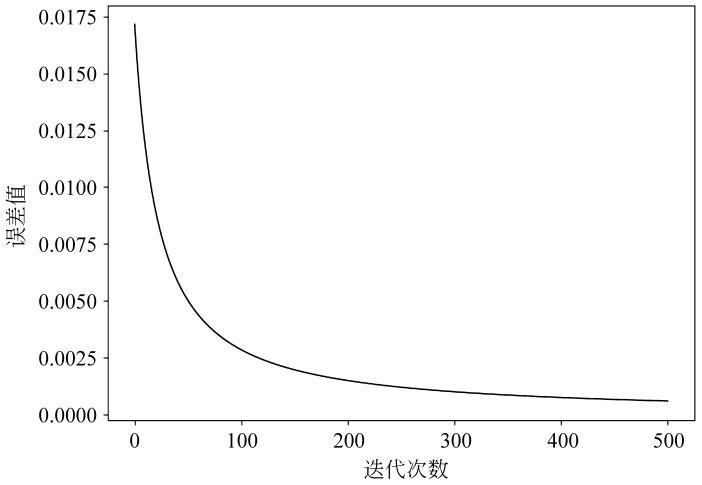


图 5-5　梯度下降法求解一元函数优化问题的迭代过程

梯度下降法求解该优化问题得到的优化解如下：

最优解为 $x=-1.21$，最优值为 5.4112

说明：用粒子群算法求解的最优解与用梯度下降法求解的最优解有比较小的差异。

利用遗传算法求解一元函数优化问题的程序：

```
import math
import random
import numpy as np
from matplotlib import pyplot as plt
#防止中文乱码
plt.rcParams["font.sans-serif"]=["SimHei"] #设置字体
plt.rcParams["axes.unicode_minus"]=False    #该语句解决图像中的"-"负号的乱码问题
#参数定义
a=[1,1]
bn=22;                                      #个体串长度
inn=50;gnmax=100; pc=0.9;pm=0.05;
#随机生成 50 个个体的 22 位编码 0-1 矩阵
s=np.random.randint(0,2,size=(inn,bn))
print()

#计算适应度函数 objf
def objf(s):
    #读取种群大小
    #有 inn 个个体
    #个体长度为 bn
    p=[];                                   #累积概率
    inn=len(s);f=[];ps=[]
```

```
    bn=len(s[0])
    #将种族 s 的所有个体全部转换为编码
    for i in range(len(s)):
        zifu=''
        for j in range(bn):
            zifu1=str(s[i][j])
            zifu=zifu+zifu1
        #将每个个体的二进制转换为十进制
        zifu2=int(zifu,2)
        #转换为[-2, 2]区间的实数
        xx=-2+zifu2*3/(math.pow(2,bn)-1)

        f.append(ft(xx))                          #计算函数值及适应度
    f1=np.transpose(f)#对应 matlabf=f';

    #计算选择概率
    fsum=0;p_tital=0
    for i in range(inn):
        fsum = fsum + f1[i] * f1[i];

    for i in range(inn):
        ps=np.append(ps,f1[i] * f1[i] / fsum);
    #p 为累积概率,最后一个为 1
    for i in range(0,inn):
        p_tital= p_tital+ ps[i];
        p.append(p_tital)
    #返回适应度 f 和累积概率 p
    return p,f

def ft(x):
    y=2.2*(1-x+2*math.pow(x,2))*math.exp(-math.pow(x,2)/2)
    return y

#sel 函数表示选择操作,参数 s 表示种群,p 表示累积概率
#该函数能选出两个适应度高的个体的序号
def sel(s,p_total):
    seln=[];                                      #seln 为选择的两个个体
    for i in range(0,2):                          #循环两次选择两个
        r=np.random.rand()                        #随机产生一个数
        j=1
        for w in range(len(p_total)):
            #选择第一个 p_total[w]-r>0 的个体作为选择个体
            if p_total[w]-r<0:
                j=j+1
        seln.append(j-1);
    return seln;
#cro 函数表示交叉操作,参数 s 为种群,seln 为选择操作选中的两个个体序号,pc 为交叉概率
#返回交换部分染色体后的两个个体
#参数为:原种群,交叉染色体序号,交叉概率
#返回值为交叉后染色体
def cro(s,seln,pc):
```

```
    position=0
    length=22                                   #%取染色体长度为 length
    pcc=IfCroIfMut(pc)                          #%概率决定是否交叉,1 是 0 否
    if pcc==1:
        #随机选取一个交叉节点
        position = int(np.random.uniform(0,1.5) * (length - 2));
    str1,str2=jiaocha(s[seln[0]],s[seln[1]],position)
    return str1,str2

#交叉拼接数组
def jiaocha(str1,str2,n):
    str_11=str1[0:n];
    print(str_11)
    str_12=str1[n:len(str1)]
    str_21=str2[0:n];
    str_22 =str2[n:len(str2)]
    str_new1=np.append(str_11,str_22)
    str_new2 = np.append(str_21, str_12)
    #返回交叉后的两个数组
    return str_new1,str_new2

#轮盘赌:变异或交叉的抽奖函数
def IfCroIfMut(mutORcro):
    z=np.zeros((1,100))                         #建立 1×100 的全零矩阵
    L=random.uniform(0,100*mutORcro)            #取 0~100 内的随机数
    print(int(L))
    for i in range(int(L)):
        z[0][i]=1
    n=int(random.uniform(50,len(z)))
    pcc=z[0][n]
    return pcc

#突变函数 mutation
#返回值为 snew,承载新种群的染色体信息
#进行变异步骤, population 为原种群,pmutation 为变异概率
def mut(s,pm):
    length=len(s[0])                            #获取基因长度
    #snew 为返回值
    n=int(random.uniform(0,50))
    snew = s[n];
    pmm = IfCroIfMut(pm)                        #如果 pmm 为 0 则不变异
    print(pmm)
    if pmm==1:                                  #%抽变异的染色体位置
        n = int(random.uniform(0, length-1))
        snew[n]=abs(snew[n]-1)
        print('变异',snew)
    return snew,n

p_total,f=objf(s)
print('每个种群的适应度函数为:',f)
print('累积概率为:',p_total)
```