文献阅读：
差分进化算法 ——
一种简单高效的连续空间全局优化启发式方法

2022.11.6

摘　要

本文提出了一种新的启发式方法，用于最优化可能的非线性和不可微的连续空间函数。通过一个广泛的测试平台，证明了新方法比许多其他著名的全局优化方法收敛得更快，且更有把握。新方法需要的控制变量少，鲁棒性好，且易于使用，非常适合进行并行计算。

关键字：随机优化；非线性优化；全局优化；遗传算法；进化策略

第一章 引言

涉及连续空间的全局优化问题在科学界中普遍存在。一般来说，我们需要通过适当地选择系统参数来优化系统的某些特性。为方便起见，系统参数用向量形式表示。优化问题的基本方法为首先设计一个目标函数，该目标函数可以模拟问题的目标，同时包含所有约束。特别是在电路设计领域，使用的方法不需要目标函数，而是要在可行域内进行操作：Brayton 等人（1981），Lueder（1990），Storn（1995）。尽管这些方法可以使问题的表述更加简单，但它们的表现通常比使用目标函数的方法要差。因此，我们将只关注使用目标函数的最优化方法。在大多数情况下，目标函数将优化问题定义为一个最小化问题。为此，下面的研究将进一步聚焦在最小化问题上。对于此类问题，目标函数被更准确地称为“成本函数”。

当成本函数是非线性的、不可微时，首选方法是直接搜索方法。其中最有名的是 Nelder–Mead 算法：Bunday 等人（1987），Hooke-Jeeves 算法：Bunday 等人（1987），遗传算法（GA）：Goldberg(1989)，以及进化策略（ES）：Rechenberg（1973），Schwefel（1995）。每种直接搜索方法的核心是生成参数调整策略。结果一旦产生变化，就必须决定是否接受新的参数。大多数基本直接搜索方法使用贪婪标准来做出这个决定。在贪婪准则下，当且仅当一个新的参数使成本函数值减小时，才接受新的参数。虽然贪婪决策的收敛速度较快，但它有陷入局部最优的风险。遗传算法和进化策略等固有的并行搜索技术有一些内置的方法来避免算法出现不收敛的现象。通过同时运行几个向量，好的参数设置可以帮助其他向量跳出局部最优。另一种可以使参数跳出局部最优的方法是模拟退火算法。Ingber(1992), Ingber(1993), Press 等人(1992)。退火过程放松了贪婪标准，偶尔允许上坡动作。这种动作有机会使得参数跳出局部最优。随着迭代次数的增加，接受一个很大的上坡动作的概率会降低。从长远来看，这就导致了贪婪准则的出现。虽然所有的直接搜索方法都可以用于退火过程，但它主要是用于随机游走，而随机游走本身就是进化算法最简单的案例：Rechenberg（1973）。然而，人们已经尝试对其他直接搜索方法进行退火，Nelder–Mead 法：Press 等人（1992）和遗传算法：Ingber（1993），Price（1994）。用户通常要求实用的最优化技术应满足五个要求：

（1）具有处理不可微、非线性和多模态成本函数的能力；

（2）可对计算复杂的成本函数进行并行处理；

（3）易于使用，即最优化问题的控制变量较少。这些变量应该是鲁棒性好，易于选择的；

（4）良好的收敛性，即在连续独立试验中一致收敛到全局最优。

如下所述，新的最优化方法差分进化（DE）算法被设计成满足所有上述要

求。

为了满足要求（1），DE 被设计成一种随机直接搜索方法。直接搜索法还具有易于应用于实验最优化的优点，其中成本函数值来自于物理实验而非计算机仿真。物理实验确实是 Rechenberg 等人开发 ES 的动机。

要求（2）对计算要求高的优化很重要，例如，成本函数的一次评估可能需要几分钟到几小时，这在集成电路设计或有限元仿真中经常发生。为了在合理的时间内获得可用的结果，唯一可行的方法是求助于并行计算机或计算机网络。DE 通过使用一个向量种群来满足要求（2），其中种群向量的随机扰动可以独立完成。

为了满足要求（3），最优化方法应该是自组织的，且需要用户输入的参数较少。Nelder–Mead 算法：Bunday 等人（1987）的方法是一个自组织最优化的优秀案例。如果当前的成本函数有 D 个参数，Nelder–Mead 算法使用一个有 $D+1$ 个顶点的多面体来定义当前的搜索空间。每个顶点由对成本函数进行采样的 D 维参数表示。新的参数是通过对成本函数高的向量的反射和对成本函数低的向量的收缩生成的。如果新的向量与之前的向量相比成本函数值降低，则新的向量将取代之前的向量。这种策略允许搜索空间，即多面体，在没有用户进行人为控制变量设置的情况下进行扩展和收缩。不幸的是，Nelder–Mead 方法基本上是一种局部最优化方法，我们的实验表明，即使引入了退火的概念，Nelder–Mead 方法对于全局最优化问题来说也不够强大。然而，DE 借用了 Nelder–Mead 的想法，即利用向量种群内的信息来改变搜索空间。DE 的自组织策略采用两个随机选择的种群向量的差值来对现有向量进行扰动。扰动是针对每个种群向量进行的。这个关键思想与传统 ES 使用的方法不同，后者是由预定的概率分布函数决定向量的扰动。

最后不得不提，要求（4）中要求的良好收敛性是一个好的最优化算法所必须的。尽管有许多方法可以从理论上描述全局最优化方法的收敛性，但只有在各种条件下的广泛测试才能显示出最优化方法是否能实现其承诺。在这方面，DE 的成绩非常好，我们将在第三章详细解释。

第二章 差分进化算法

差分进化（DE）算法一种并行直接搜索方法，它利用 NP 个 D 维的参数向量

$$x_{i,G}, i = 1, 2, ..., NP \qquad (1)$$

作为每一代 G 的种群。在最优化过程中， NP 值不变。初始种群是随机选择的，应覆盖整个参数空间。通常，除非另有说明，我们假设所有随机决策变量的概率分布是均匀的。在初始解可行的情况下，初始种群通过向标称解 $x_{nom,0}$ 添加正态分布的随机偏差来生成。DE 通过将两个种群向量之间的加权差添加到第三个向量中来生成新的参数。这个操作被称为变异。然后，将变异向量的参数与另一个预定义的向量（即目标向量）进行参数混合，生成试验向量。参数混合在 ES 领域中通常被称为“交叉”，后面将详细解释。如果试验向量产生的成本函数值比目标向量低，则试验向量就会在下一代中取代目标向量。这最后一步操作叫做选择。每个种群向量必须充当一次目标向量，以便在一代中发生 NP 次竞争。

更具体地说，DE 的基本策略可以描述如下：

变异

对于每个目标向量 $x_{i,G}, i=1,2,...,NP$，根据以下方法生成一个变异向量

$$v_{i,G+1}=x_{r_1,G}+F\cdot\left(x_{r_2,G}-x_{r_3,G}\right) \tag{2}$$

随机指数 $r_1,r_2,r_3\in\{1,2,...,NP\}$ 为整数，且互不相等，$F>0$。随机选择的整数 r_1,r_2,r_3 不能等于运行指数 i，因此 NP 必须大于等于 4。F 是一个实数常数因子 $\in[0,2]$，它控制差分变化 $(x_{r2,G}-x_{r3,G})$ 的放大倍数。图 1 显示了一个二维示例，该示例说明了在 $v_{i,G+1}$ 的生成中起作用的不同向量。

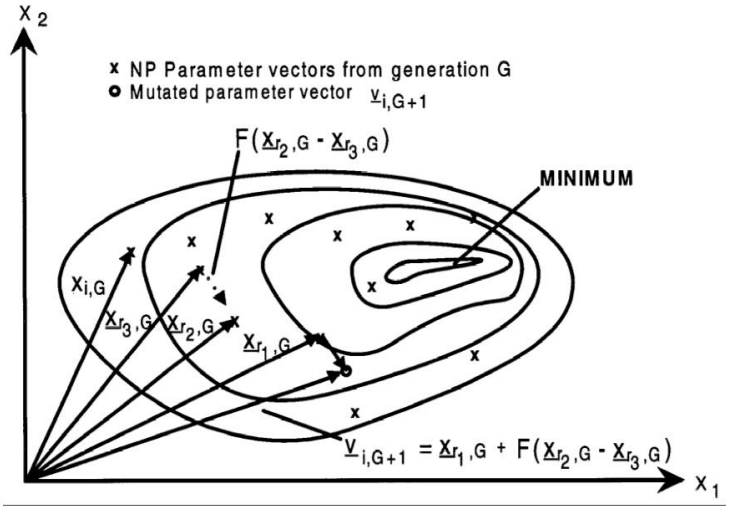

图 1 一个二维成本函数的例子，显示其轮廓线和生成 $v_{i,G+1}$ 的过程

交叉

为了增加被扰动参数的多样性，引入了交叉操作。为此，生成试验向量：

$$u_{i,G+1}=\left(u_{1i,G+1},u_{2i,G+1},\ldots,u_{Di,G+1}\right) \tag{3}$$

其中，

$$u_{ji,G+1}=\begin{cases}v_{ji,G+1} & if\ \ (randb(j)\le CR)\ \ or\ \ j=rnbr(i)\\ x_{ji,G} & if\ \ (randb(j)>CR)\ \ and\ \ j\ne rnbr(i)\end{cases} \tag{4}$$
$$j=1,2,\ldots,D$$

在（4）中，$randb(j)\in[0,1]$ 为均匀随机数发生器的第 j 次评估。$CR\in[0,1]$ 为交叉概率，必须由用户指定。$rnbr(i)\in 1,2,...,D$ 为随机选择的索引，确保 $u_{i,G+1}$ 从 $v_{i,G+1}$ 获得至少一个参数。图 2 给出了一个 7 维向量的交叉操作实例。

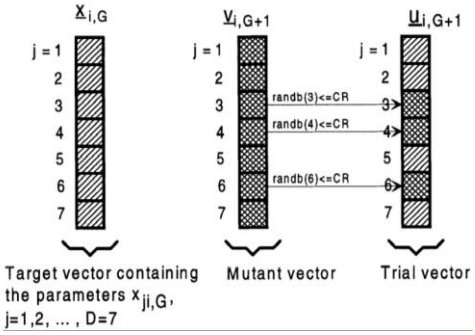

图 2 D=7 个参数的交叉过程图解

选择

为了判断它是否应该成为$G+1$代的个体，试验向量$u_{i,G+1}$与目标向量$v_{i,G+1}$使用贪婪准则进行比较。如果向量$u_{i,G+1}$产生的成本函数值比$v_{i,G+1}$小，那么$x_{i,G+1}$就被设置为$u_{i,G+1}$；否则，保留上一代的值$x_{i,G}$。

伪代码

DE 的简易性最好通过一些 C 语言风格的伪代码来解释，图 3 中给出了这一代码。

DE 的其他变体

上述方案并不是唯一被证明是有效的DE变体。为了对不同的变体进行分类，记为

$$DE/x/y/z$$

其中，x表示目标向量的选择方式，目前可以是“rand”（随机选择的种群向量）或“best”（当前种群中成本函数值最小的向量）。

y表示执行差分操作的向量的个数。

z表示执行交叉操作的方式。目前的变体为“bin”（由于独立二项式实验而产生的交叉，如第二章中所述）

使用这种标记，上一章中描述的基本 DE 策略可以写成：DE/rand/1/bin。这是我们后来用于所有性能比较的DE变体。尽管如此，有一个值得一提的好方案，那就是 DE/best/2/bin 方法。Price（1996），其中

$$v_{i,G+1} = x_{best,G} + F \cdot \left(x_{r_1,G} + x_{r_2,G} - x_{r_3,G} - x_{r_4,G} \right) \qquad (5)$$

如果种群向量的数量NP足够多，使用两个差向量似乎可以改善种群的多样性。

```
/*---------------Main loop------------------------------------------*/
while (count < gen_max)         /* Halt after gen_max generations. */
{
   for (i=0; i<NP; i++)  /*-------Start loop through population.---*/
   {
                            /********** Mutate/recombine **********/

      do a=rnd_uni()*NP; while (a==i); /* Randomly pick 3 vectors, */
      do b=rnd_uni()*NP; while (b==i || b==a);    /* all different */
      do c=rnd_uni()*NP; while (c==i || c==a || c==b);  /* from i. */
      j=rnd_uni()*D;          /* Randomly pick the first parameter. */
      for (k=1; k<=D; k++)       /* Load D parameters into trial[]. */
      {                             /* Perform D-1 binomial trials. */
         if (rnd_uni() < CR || k==D)     /* Source for trial[j] is */
         {           /* a random vector plus weighted differential */
            trial[j]=x1[c][j]+F*(x1[a][j]-x1[b][j]);      /* or... */
         }          /* trial parameter comes from target vector    */
         else trial[j]=x1[i][j];               /* x1[i][j] itself. */
         j=(j+1)%D;                 /* get next parameter, modulo D. */
      }    /* Last parameter (k=D) comes from noisy random vector. */

                             /********** Evaluate/select **********/

      score=evaluate(trial); /* Evaluate trial with your function. */
      if (score<=cost[i])       /* If trial[] improves on x1[i][], */
      {                         /* move trial[] to secondary array */
         for (j=0; j<D; j++) x2[i][j]=trial[j];
         cost[i]=score;                 /* and store improved cost */
      }             /* otherwise, move x1[i][] to secondary array. */
      else for (j=0; j<D; j++) x2[i][j]=x1[i][j];
   }                /* Mutate/recombine next primary array vector. */

          /*----------End of population loop; swap arrays----------*/

   for (i=0; i<NP; i++)                 /*  After each generation, */
   {                    /* move secondary array into primary array. */
      for (j=0; j<D; j++) x1[i][j]=x2[i][j];
   }                     /* ...or just swap pointers (not shown). */
   count++;             /* End of generation...increment counter. */


}
/*-------End of main loop---------------------------------------- */
```

图 3 DE 的 19 行 C 语言风格的伪代码