Otimização

Gustavo Alves Pacheco*

11821ECP011

1 Introdução

Até o momento, entre as técnicas utilizadas, pouco se falou sobre a otimização de funções. Otimização, eu seu significado, representa a seleção das melhores alternativas. Diversas técnicas existem, as quais buscam minimizar ou maximizar uma função objetivo através da escolha de certos valores de entradas (variáveis de projeto) dentro de um conjunto possível de soluções (espaço de busca), que garante saída mínima/máxima (valor ótimo) [1].

Esta busca pode ser determinística ou estocástica. No método determinístico, conhecendo o ponto de partida, é possível determinar a resposta do algoritmo, visto que o mesmo sempre leva à mesma resposta, em condições iguais, exemplo método Gradiente Descendente. Já no estocástico, um caráter aleatório é introduzido ao algoritmo, advindo de uma chamada para uma função geradora de números aleatórios em algum ponto de sua execução.

Neste trabalho, a técnica de Evolução Diferencial será tratada. É um método estocástico, que possui grande capacidade de encontrar soluções de otimização global, podendo trabalhar com funções custo não-lineares e não diferenciáveis, além de permitir facilmente a paralelização e ser acessível.

Nessa abordagem, são utilizados vetores n-dimensionais, dependendo do número de variáveis da função de custo. Inicialmente, a população (o conjunto) desses vetores é escolhida de forma aleatória (ou em alguns casos, distribuída uniformemente pelo espaço de busca). A partir dela, cada indivíduo é colocado em xeque, para compor a nova geração. Esse indivíduo, chamado de target vector, é comparado com um outro indivíduo, gerado através da combinação de três vetores aleatórios, e do próprio target, gerando o trial vector. O que possuir maior fitness, dentre os dois, é incorporado à nova geração.

Para aplicação da evolução diferencial, deve-se minimizar a função de Rosenbrock (eq. 1), para n variáveis.

*gap1512@gmail.com

$$f(x_1, x_2, ..., x_{D-1}) = \sum_{i=1}^{D-1} (1 - x_i)^2 + 100(x_{i+1} - x_i^2)^2 \quad (1)$$

Com $x_i \in [-1, 2]$.

2 Objetivos

- Aprimorar o conhecimento sobre Algoritmos de Otimização e obter experiência prática na implementação dos mesmos.
- Minimizar a função de Rosenbrock utilizando Evolução Diferencial

3 Materiais e Métodos

Para implementação do algoritmo de otimização foi utilizada a linguagem de programação Common Lisp, compilando-a com o SBCL (Steel Bank Common Lisp). Como interface de desenvolvimento, foi utilizado o Emacs em Org Mode, configurado com a plataforma SLIME (The Superior Lisp Interaction Mode for Emacs) para melhor comunicação com o SBCL. Foi utilizada uma abordagem bottom-up para o desenvolvimento. O código produzido segue majoritariamente o paradigma funcional, sendo este trabalho como um todo uma obra de programação literária. Parte das funções já foram implementadas em Regra de Hebb, Perceptron e Adaline, Regressão Linear, Multilayer Perceptron, Feature Engineering e Clustering.

4 Desenvolvimento

A implementação se inicia pela definição da função de Rosenbrock para n variáveis, da seguinte forma:

```
(defun rosenbrock (lst)
  (labels ((rec (lst nxt res)
                  (if nxt
                      (let ((x (car lst)))
                        (rec (cdr lst)
                             (cadr lst)
                             (+ res
                                (+ (expt (- 1 x) 2)
                                   (* 100 (expt (- nxt (* x x)) 2))))))
                      res)))
    (rec lst (cadr lst) 0)))
```

Essa equação, possui como gráfico (fig. 1):

```
(defun plot-3d-from-top (output fn min max step &optional (color-min
↪  "blue") (color-max "red"))
  (scatter-plot output (do ((x min (+ step x))
                            res)
                           ((> x max) res)
                         (do ((y min (+ step y)))
                             ((> y max))
                           (let ((point (list x y)))
                             (push (append point (list (funcall fn
                              ↪  point)))
                                   res))))
                nil 0 0 color-min color-max))
```

```
(plot-3d-from-top "plots/rosenbrock.png" #'rosenbrock -1 2 0.025
↪  "black" "red")
```

Além disso, é necessário implementar uma função que gera a população inicial aleatoriamente. Esta função é construida tendo como base a `random-weights`, definida anteriormente:

```
(defun initial-population (size n-variables min max)
  (loop repeat size
        collecting (random-weights n-variables min max)))
```

Assim, é possível criar populações de vários tamanhos, com várias variáveis em cada indivíduo:

```
(initial-population 3 2 -1 2)
```

1.9004931	0.64767957
1.4909754	-0.3765416
0.24269068	1.1978741

Necessário também implementar funções de `mutation` e `crossover`. Nesse método, a mutação acontece primeiro, e é a responsável pela criação do *donor vector*, a partir da relação:

$$v = x_1 + F * (x_3 - x_2) \tag{2}$$

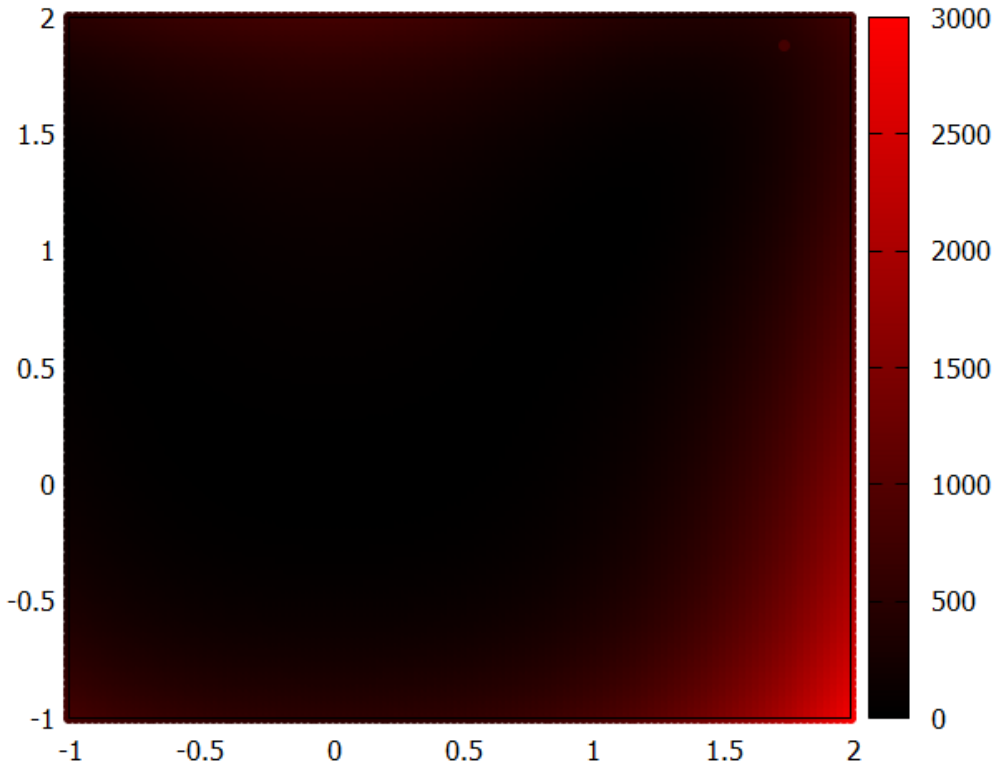

Figure 1: Função de Rosenbrock em 3 dimensões, vista de cima

Sendo x_1, x_2 e x_3 vetores escolhidos aleatoriamente da população atual e F uma constante real entre 0 e 2, definida pelo usuário.

Então, em Lisp, essa função seria diretamente definida:

```
(defun mutation (f x-1 x-2 x-3)
  (mapcar #'(lambda (a b c)
              (+ a (* f (- c b))))
          x-1 x-2 x-3))
```

Esse vetor encontrado é combinado com o *target vector* (vetor atualmente em cheque da população) para gerar o *trial vector*, no processo conhecido como `crossover`. Essa combinação é feita escolhendo aleatoriamente posições ou do *target vector* ou do *donor vector*, segundo a relação:

$$\begin{cases} v_i, \text{ se } r_i \leq CR \text{ ou } i = I \\ x_i, \text{ se } r_i > CR \text{ e } i \neq I \end{cases} \tag{3}$$

Sendo x o *target vector*, v o *donor vector*, r_i um número real aleatório entre 0 e 1, gerado para cada posição do vetor, CR uma constante entre 0 e 1, definida pelo usuário, i o número da posição atual e I um número aleatório gerado apenas uma vez, entre 1 e D, sendo D o número de dimensões do vetor. Isso é feito para que pelo menos uma componente de v esteja em u. Ao final, deve-se verificar se o mesmo se encontra dentro dos intervalos especificados. Assim, tem-se:

```
(defun crossover (target donor c-rate min max)
  (labels ((choose (x v r i index)
             (let ((c (if (and (> r c-rate) (not (eq i index)))
                          x v)))
               (cond
                 ((< c min) min)
                 ((> c max) max)
                 (t c)))))
    (do* ((i (length target) (1- i))
          (x target (cdr x))
          (v donor (cdr v))
          (index (random i))
          res)
         ((or (not x) (not v))
          (nreverse res))
      (push (choose (car x) (car v) (random 1.0) i index)
            res))))
```

Assim, finalmente o vetor após o cruzamento (*trial vector*) é comparado com o *target*. Essa comparação é feita ao aplicar a função de aptidão (nesse caso, Rosenbrock) em ambos e verificar quem apresenta menor resultado (para minimização).

```
(defun trial (target trial fitness-fn comparison-fn)
  (if (funcall comparison-fn (funcall fitness-fn trial)
               (funcall fitness-fn target))
      trial target))
```

Então, o algoritmo é uma combinação dessas funções:

```
(defun differential-evolution (fn n-variables comparison-fn
                               population-size n-generations c-rate f
                               min max)
  (do ((i 0 (1+ i))
       (population (initial-population population-size n-variables min
       ↪  max)
                   (mapcar
                    #'(lambda (vec)
                        (trial
                         vec
                         (crossover
                          vec
                          (apply #'mutation f
                                 (loop repeat 3
                                    collect (nth
                                             (random population-size)
                                             population)))
                          c-rate min max)
                         fn comparison-fn))
                    population))
       result)
      ((>= i n-generations) (values (best population fn comparison-fn)
      ↪  result))
    (push population result)))

(defun best (lst fn comparison-fn)
  (let ((res (list (car lst) (funcall fn (car lst)))))
    (dolist (item (cdr lst) res)
      (let ((fit (funcall fn item)))
        (when (funcall comparison-fn fit (second res))
          (setf res (list item fit)))))))
```

A chamada do mesmo, para a função $y = x^2$ é:

```
(values (differential-evolution #'(lambda (ind)
                                    (let ((x (car ind)))
                                      (* x x)))
                                  1 #'< 100 6000 0.9 0.5 -1 1))
```

```
(-9.967971e-024)  0.0
```

Sendo o primeiro elemento o valor de x e o segundo, o de y correspondente. Verificada a validade da função, a mesma é aplicada para `Rosenbrock`, com 2 variáveis:

```
(values (differential-evolution #'rosenbrock 2 #'< 100 1000 0.9 0.5 -1
↪  2))
```

```
(1.0 1.0)  0.0
```

A título de curiosidade, faz-se essa mesma otimização para a `Rosenbrock` com 10 variáveis:

```
(values (differential-evolution #'rosenbrock 10 #'< 100 1000 0.9 0.5 -1
↪  2))
```

```
(1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)  0.0
```

5 Conclusão

O algoritmo de Evolução Diferencial se mostrou extremamente eficaz. A resposta foi confiável, precisa e exata. A execução do mesmo foi rápida e o tempo de implementação foi reduzido. A ED foi capaz de encontrar com sucesso o mínimo da função de Rosenbrock tanto para 2 quanto para 10 variáveis, em 1000 gerações, com 100 indivíduos, com precisão e exatidão, em um tempo muito curto.

References

[1] K. Yamanaka. Aprendizagem de máquina (machine learning - ml). Universidade Federal de Uberlândia.