Instituto Tecnológico de Aeronáutica – ITA
Inteligência Artificial para Robótica Móvel – CT-213
Professor: Marcos Ricardo Omena de Albuquerque Maximo

Laboratório 5 – Estratégias Evolutivas

Observação: antes de começar a fazer esse laboratório, é necessário instalar o CMA-ES no seu Anaconda. Para isso, faça o seguinte:

- Vá até o seu ambiente no Anaconda Navigator e abra um terminal.
- Então, instale com: `pip install cma`

1. Introdução

Nesse laboratório, seu objetivo é implementar uma estratégia evolutiva simples e comparar seu desempenho com o CMA-ES em funções usadas como benchmark para algoritmos de otimização. A Figura 1 ilustra uma estratégia evolutiva sendo aplicada para otimizar a função de Rastrigin.

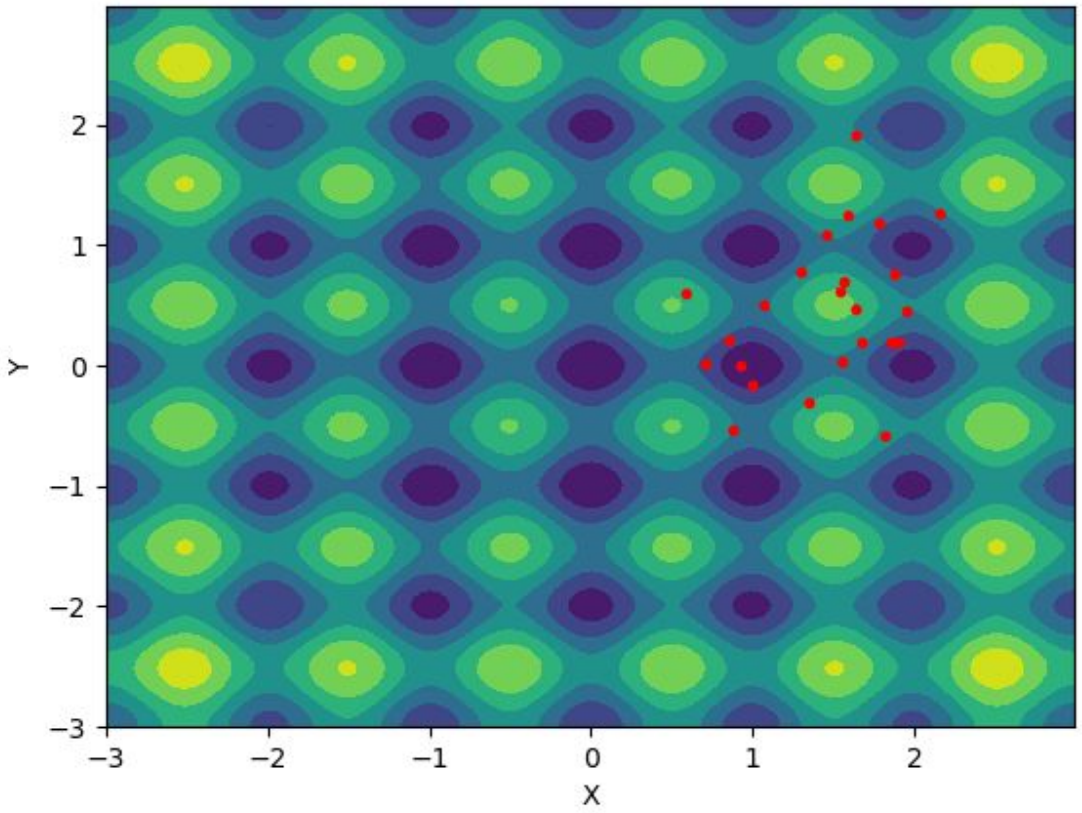

Figura 1: otimização da função de Rastrigin usando estratégia evolutiva. As amostras são os pontos vermelhos.

2. Descrição do Problema

O problema a ser resolvido é a otimização de funções *benchmark* através de estratégias evolutivas. A saber, as seguintes funções serão usadas:

Translated Sphere (com centro em $(1, 2)$) - `translated_sphere`:

$f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 2)^2$

Ackley - `ackley`:

$f(x_1, x_2) = -20exp(-0.2\sqrt{-0,5(x_1^2 + x_2^2)}) - exp(0,5(cos(2\pi x_1 + 2\pi x_2)) + exp(1) + 20$

Schaffer function N. 2 - `schaffer2d`:

$f(x_1, x_2) = 0,5 + \frac{sin^2(x_1^2 - x_2^2) - 0,5}{(1 + 0,001(x_1^2 + x_2^2))^2}$

Rastrigin (2D) - `rastrigin`:

$f(x_1, x_2) = 2A + \sum_{i=1}^{2}(x_i^2 - Acos(2\pi x_i))$, em que $A = 10$

A estratégia evolutiva que será implementada é bem simples e evolui média e covariância de uma distribuição gaussiana através das seguintes equações:

$$m^{(g+1)} = \frac{1}{\mu}\sum_{i=1}^{\mu} s_{i:\lambda}^{(g+1)}$$

$$C^{(g+1)} = \frac{1}{\mu}\sum_{i=1}^{\mu}(s_{i:\lambda}^{(g+1)} - m^{(g)})(s_{i:\lambda}^{(g+1)} - m^{(g)})^T$$

em que $m^{(g+1)}$ e $C^{(g+1)}$ são a média e a covariância da distribuição gaussiana na geração $g + 1$, $s_{i:\lambda}^{(g+1)}$ é $i - \acute{e}sima$ melhor amostra (de um total de λ amostras em cada geração) e as μ melhores amostras são escolhidas para evoluir a distribuição gaussiana para a próxima geração. Perceba que $m^{(g+1)}$ e $s_{i:\lambda}^{(g+1)}$ são vetores, enquanto $C^{(g+1)}$ é uma matriz.

Então, será realizado um *benchmark* através de simulações de Monte Carlo para comparar os desempenhos dessa estratégia evolutiva simples (SES - Simple Evolution Strategy) e do CMA-ES.

3. Código Base

O código base já implementa teste e *benchmark* das estratégias evolutivas. Além disso, ele usa a implementação oficial do CMA-ES na comparação. Segue uma breve descrição dos arquivos fornecidos:

- `simple_evolution_strategy.py`: implementação da estratégia evolutiva simples.
- `benchmark.py`: implementação das funções de *benchmark*.
- `test_evolution_strategy.py`: implementação de teste das estratégias evolutivas. Apresenta uma animação do progresso da evolução.
- `benchmark_evolution_strategy.py`: implementação de um *benchmark* entre as estratégias evolutivas.

O foco da sua implementação nesse laboratório é apenas o método `tell()` da classe `SimpleEvolutionStrategy`. Perceba que embora a classe `SimpleEvolutionStrategy`

tente seguir mais ou menos a interface da implementação oficial do CMA-ES, fez-se algumas modificações, de modo a deixar a implementação mais didática e mais simples quando se usa os recursos do NumPy.

4. Tarefas

4.1. Implementação da Estratégia Evolutiva Simples

Sua primeira tarefa consiste em implementar e testar a estratégia evolutiva simples (SES). Para isso, basta implementar o método `tell()` da classe `SimpleEvolutionStrategy`. Uma otimização usando um objeto dessa classe segue o seguinte pseudocódigo:

```
for i in range(num_iterations):
   samples = es.ask()
   for j in range(np.size(samples, 0)):
      fitnesses[j] = function(samples[j, :])
   es.tell(fitnesses)
```

4.2. Teste das Estratégias Evolutivas

Então, teste sua implementação usando o arquivo `test_evolution_strategy.py`. As variáveis que podem ser alteradas nesse script são as seguintes:
`algorithm`: escolhe entre o SES (`'ses'`) e o CMA-ES (`'cmaes'`). Obs.: você deve usar uma `string` com o nome do algoritmo.
`function`: escolher a função de teste (`translated_sphere`, `ackley`, `schaffer2d` ou `rastrigin`). Obs.: usa-se "ponteiro de função" aqui.

Por padrão, o script usa o (12,24)-SES, i.e. $\mu = 12$ e $\lambda = 24$. Além disso, usa-se a estratégia padrão do CMA-ES, que escolhe $\mu = 3$ e $\lambda = 6$ para problemas 2D.

Como resultado do teste, é mostrada uma animação. Teste os dois algoritmos para as diferentes funções algumas vezes para pegar intuição. No seu relatório, comente de forma sucinta sobre os resultados para cada um dos algoritmos e das funções, principalmente sobre questões como convergência, incluindo sobre convergência para mínimo local. Além disso, o script salva uma figura (arquivo `evolution_strategy.png`) com o resultado final da otimização. Inclua figuras de um exemplo de execução para cada algoritmo e cada função no seu relatório.

4.3. Benchmark

Finalmente, faremos um *benchmark* usando simulações de Monte Carlo para comparar os dois algoritmos. Esse *benchmark* é realizado usando o arquivo `benchmark_evolution_strategy.py`. Em cada simulação de Monte Carlo, $m^{(0)}$ é amostrado uniformemente num quadrado de lado 6 e centrado em $(0, 0)$. Com isso, executa-se

uma quantidade considerável de simulações de Monte Carlo para garantir uma convergência adequada dos resultados. As métricas usadas são:

1. *Fitness* médio das amostras em cada geração (calcula-se uma média entre todas as simulações de Monte Carlo). Seja M o número de execuções do Monte Carlo, essa métrica é calculada por:

$$f_{mean}^{(g)} = \sum_{k=1}^{M} \sum_{i=1}^{\lambda} f(s_i^{(g)})$$

2. Melhor *fitness* entre todas as amostras em cada geração (calcula-se uma média entre todas as simulações de Monte Carlo). Essa métrica é calculada por:

$$f_{best}^{(g)} = \sum_{k=1}^{M} f(s_{1:\lambda}^{(g)})$$

Portanto, compara-se o desempenho entre:

1. (3,6)-SES: estratégia evolutiva simples com $\mu = 3$ e $\lambda = 6$.
2. (6,12)-SES: estratégia evolutiva simples com $\mu = 6$ e $\lambda = 12$.
3. (12,24)-SES: estratégia evolutiva simples com $\mu = 12$ e $\lambda = 24$.
4. CMA-ES: CMA-ES com estratégia padrão, que usa $\mu = 3$ e $\lambda = 6$.

No script `benchmark_evolution_strategy.py`, tem-se as seguintes variáveis:

- `num_trials`: número de execuções de Monte Carlo.
- `num_iterations`: número de gerações da estratégia evolutiva.
- `function`: função usada para *benchmark*.

Perceba que como está indicado em comentários no código, recomenda-se usar um número de execuções e de iterações maior para a função Schaffer2D, pois com os valores iniciais os resultados ficam muito ruidosos para essa função. Como isso faz o *benchmark* demorar mais, recomenda-se mudar os valores apenas para essa função.

No final, o script plota gráficos das métricas de *benchmark*. Analise estes gráficos e discuta brevemente no relatório suas conclusões (comente também porque você acha que os resultados são diferentes para cada função). Adicione também os gráficos no seu relatório (que são salvos em arquivos com nomes `mean_fitness.png` e `best_fitness.png`).

5. Entrega

A entrega consiste do código e de um relatório, submetida através do Google Classroom. Modificações nos arquivos do código base são permitidas, desde que o nome e a interface dos scripts “main” não sejam alterados. A princípio, não há limitação de número de páginas para o relatório, mas pede-se que seja sucinto. O relatório deve conter:

- Breve descrição em alto nível da sua implementação.
- Figuras que comprovem o funcionamento do seu código.

Por limitações do Google Classroom (e por motivo de facilitar a automatização da correção), entregue seu laboratório com todos os arquivos num único arquivo **.zip** (**não** utilize outras tecnologias de compactação de arquivos) com o seguinte padrão de nome: “**<login_email_google_education>_labX.zip**”. Por exemplo, no meu caso, meu login Google Education é **marcos.maximo**, logo eu entregaria o lab 5 como “**marcos.maximo_lab5.zip**”.

Não crie subpastas para os arquivos da sua entrega, **deixe todos os arquivos na "raiz" do .zip**. Os relatórios devem ser entregues em formato **.pdf**.

6. Dicas

- Para criar um vetor de zeros usando NumPy que tenha o mesmo número de elementos que outro vetor, faça:

```
array = np.zeros(np.size(other_array))
```

- Operações normais de vetor, como soma, subtração e multiplicação por escalar, funciona como esperado em NumPy:

```
sum = a + b
sub = a - b
mul_scalar = scalar * a
```

- Para amostrar `population_size` amostras de uma gaussiana multivariada com média `m` e covariância `C`, use:

```
x = np.random.multivariate_normal(m, C, population_size)
```

Nesse caso, as amostras ficam numa matriz de dimensão `(population_size, n)`, em que `n` é a dimensão do problema.

- Para pegar as μ melhores amostras de um certo vetor, pode-se usar:

```
indices = np.argsort(fitnesses)
best_samples = self.samples[indices[0:mu], :]
```

- Para transpor um vetor em NumPy, é necessário primeiro transformar esse vetor em matriz usando:

```
matrix = np.matrix(array)
transpose = matrix.transpose()
```

- **Na dica anterior, perceba ainda que a matriz será um vetor linha, que é diferente da notação usual usada em matemática, em que os vetores são coluna por padrão.**