

# Cours Machine Learning

## Chapitre 10

### Algorithmes Génétiques et Optimisation Évolutionnaire

#### Objectifs d'apprentissage :

- Comprendre les algorithmes génétiques (AG)
- Maîtriser sélection, crossover, mutation
- Appliquer les AG à des problèmes d'optimisation
- Découvrir les variantes modernes

**Durée estimée : 4-6 heures**

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Inspiration biologique . . . . .	2
1.2	Quand utiliser les AG ? . . . . .	2
<b>2</b>	<b>Algorithme Génétique Standard</b>	<b>2</b>
2.1	Représentation . . . . .	2
2.2	Algorithme principal . . . . .	2
2.3	Opérateurs de sélection . . . . .	3
2.4	Crossover (Recombinaison) . . . . .	3
2.5	Mutation . . . . .	3
<b>3</b>	<b>Implémentation</b>	<b>3</b>
<b>4</b>	<b>Applications</b>	<b>5</b>
4.1	Problème du voyageur de commerce (TSP) . . . . .	5
4.2	Optimisation de fonctions . . . . .	5
4.3	Hyperparameter Tuning . . . . .	5
4.4	Neuroévolution . . . . .	5
<b>5</b>	<b>Variantes modernes</b>	<b>6</b>
5.1	CMA-ES (Covariance Matrix Adaptation) . . . . .	6
5.2	NSGA-II (Non-dominated Sorting GA) . . . . .	6
5.3	Differential Evolution . . . . .	6
<b>6</b>	<b>Résumé</b>	<b>6</b>
6.1	Points Clés . . . . .	6
6.2	Avantages et Limites . . . . .	6
<b>7</b>	<b>Pour Aller Plus Loin</b>	<b>6</b>
7.1	Bibliothèques Python . . . . .	6
7.2	Lectures . . . . .	6
<b>8</b>	<b>Notebooks Pratiques</b>	<b>7</b>

# 1 Introduction

## 1.1 Inspiration biologique

Les **algorithmes génétiques (AG)** s'inspirent de l'évolution naturelle :

- **Population** : Ensemble de solutions candidates
- **Fitness** : Qualité d'une solution
- **Sélection** : Les meilleurs survivent
- **Crossover** : Reproduction (combinaison de solutions)
- **Mutation** : Variation aléatoire

## 1.2 Quand utiliser les AG ?

- Espace de recherche discret, combinatoire
- Fonction objectif non différentiable
- Optimisation multi-objectifs
- Pas de méthode analytique disponible

# 2 Algorithme Génétique Standard

## 2.1 Représentation

**Chromosome** : Encodage d'une solution

### Exemple : Encodages courants

- **Binaire** :  $[1, 0, 1, 1, 0]$
- **Réel** :  $[2.5, -1.3, 0.8]$
- **Permutation** :  $[3, 1, 4, 2]$  (TSP)

## 2.2 Algorithme principal

---

**Algorithm 1** Algorithme Génétique

---

```
1 : Initialiser population  $P$  aléatoirement
2 : Évaluer fitness de chaque individu
3 : while critère d'arrêt non atteint do
4 :   Sélection : Choisir parents selon fitness
5 :   Crossover : Créer enfants par recombinaison
6 :   Mutation : Appliquer mutations aléatoires
7 :   Évaluation : Calculer fitness des enfants
8 :   Remplacement : Nouvelle génération
9 : end while
10 : return Meilleur individu
```

---

## 2.3 Opérateurs de sélection

### 1. Roulette Wheel (Proportionnelle)

$$P(\text{sélectionner } i) = \frac{f_i}{\sum_{j=1}^N f_j} \quad (1)$$

### 2. Tournament Selection

- Choisir  $k$  individus aléatoirement
- Sélectionner le meilleur

### 3. Rank-Based

- Trier par fitness
- Probabilité basée sur le rang

## 2.4 Crossover (Recombinaison)

**One-Point Crossover :**

Parent 1: [1 1 0 | 1 0 1]

Parent 2: [0 1 1 | 0 1 0]

-----+-----

Enfant 1: [1 1 0 | 0 1 0]

Enfant 2: [0 1 1 | 1 0 1]

**Uniform Crossover :** Chaque gène a 50% de chance de venir de chaque parent.

**Arithmetic (réels) :**

$$\text{enfant} = \alpha \cdot \text{parent}_1 + (1 - \alpha) \cdot \text{parent}_2 \quad (2)$$

## 2.5 Mutation

**Bit Flip (binaire) :**

$$[1, 0, 1, 0] \xrightarrow{\text{mutation}} [1, 1, 1, 0] \quad (3)$$

**Gaussian (réel) :**

$$x' = x + \mathcal{N}(0, \sigma^2) \quad (4)$$

**Taux de mutation :** Typiquement  $p_m = 1/L$  où  $L$  est la longueur du chromosome.

## 3 Implémentation

Listing 1 – AG simple en Python

```

1 import numpy as np
2
3 class GeneticAlgorithm:
4     def __init__(self, fitness_func, pop_size=100, chrom_length=10,
5                   crossover_rate=0.8, mutation_rate=0.01):
6         self.fitness_func = fitness_func
7         self.pop_size = pop_size

```

```

8         self.chrom_length = chrom_length
9         self.crossover_rate = crossover_rate
10        self.mutation_rate = mutation_rate
11
12        def initialize_population(self):
13            return np.random.randint(0, 2, (self.pop_size, self.chrom_length
14                                           ))
15
16        def evaluate(self, population):
17            return np.array([self.fitness_func(ind) for ind in population])
18
19        def selection(self, population, fitness):
20            # Tournament selection
21            selected = []
22            for _ in range(self.pop_size):
23                i, j = np.random.choice(self.pop_size, 2, replace=False)
24                winner = i if fitness[i] > fitness[j] else j
25                selected.append(population[winner].copy())
26            return np.array(selected)
27
28        def crossover(self, parent1, parent2):
29            if np.random.rand() > self.crossover_rate:
30                return parent1.copy(), parent2.copy()
31
32            point = np.random.randint(1, self.chrom_length)
33            child1 = np.concatenate([parent1[:point], parent2[point:]])
34            child2 = np.concatenate([parent2[:point], parent1[point:]])
35            return child1, child2
36
37        def mutate(self, chromosome):
38            for i in range(len(chromosome)):
39                if np.random.rand() < self.mutation_rate:
40                    chromosome[i] = 1 - chromosome[i]
41            return chromosome
42
43        def run(self, generations=100):
44            population = self.initialize_population()
45            best_fitness_history = []
46
47            for gen in range(generations):
48                fitness = self.evaluate(population)
49                best_fitness_history.append(fitness.max())
50
51                if gen % 10 == 0:
52                    print(f"Gen_{gen}: Best_fitness={fitness.max():.4f}")
53
54                # Selection
55                parents = self.selection(population, fitness)
56
57                # Crossover + Mutation
58                offspring = []

```

```

58         for i in range(0, self.pop_size, 2):
59             child1, child2 = self.crossover(parents[i], parents[i
60                 +1])
61             offspring.append(self.mutate(child1))
62             offspring.append(self.mutate(child2))
63
64         population = np.array(offspring[:self.pop_size])
65
66         # Résultat final
67         final_fitness = self.evaluate(population)
68         best_idx = final_fitness.argmax()
69         return population[best_idx], final_fitness[best_idx],
70             best_fitness_history
71
72 # Exemple : Maximiser nombre de 1
73 def fitness(chromosome):
74     return chromosome.sum()
75
76 ga = GeneticAlgorithm(fitness, pop_size=50, chrom_length=20)
77 best, best_fit, history = ga.run(generations=50)
78 print(f"\nMeilleure solution: {best}")
79 print(f"Fitness: {best_fit}")

```

## 4 Applications

### 4.1 Problème du voyageur de commerce (TSP)

Trouver le chemin le plus court visitant toutes les villes.

- **Chromosome** : Permutation des villes
- **Fitness** : 1/distance totale
- **Crossover** : Order Crossover (OX)
- **Mutation** : Swap, Inversion

### 4.2 Optimisation de fonctions

Minimiser  $f(x_1, x_2) = x_1^2 + x_2^2 - 10 \cos(2\pi x_1) - 10 \cos(2\pi x_2)$  (Rastrigin)

### 4.3 Hyperparameter Tuning

Optimiser hyperparamètres d'un ML model (alternative à Grid/Random Search).

### 4.4 Neuroévolution

Optimiser poids et architecture de réseaux de neurones.

## 5 Variantes modernes

### 5.1 CMA-ES (Covariance Matrix Adaptation)

Algorithme évolutionnaire pour optimisation continue, très performant.

### 5.2 NSGA-II (Non-dominated Sorting GA)

Pour optimisation multi-objectifs (Pareto-optimal).

### 5.3 Differential Evolution

Mutation basée sur différences entre individus :

$$\mathbf{v} = \mathbf{x}_r + F(\mathbf{x}_a - \mathbf{x}_b) \quad (5)$$

## 6 Résumé

### 6.1 Points Clés

- AG = Métaheuristique inspirée de l'évolution
- Opérateurs : Sélection, Crossover, Mutation
- Bon pour optimisation combinatoire, non différentiable
- Variantes : CMA-ES, NSGA-II, Differential Evolution

### 6.2 Avantages et Limites

**Avantages :**

- Pas besoin de gradient
- Exploration globale
- Flexible (encodages variés)

**Limites :**

- Pas de garantie de convergence
- Lent comparé à méthodes gradient
- Nombreux hyperparamètres

## 7 Pour Aller Plus Loin

### 7.1 Bibliothèques Python

- DEAP : Framework AG complet
- PyGAD : Simple et pédagogique
- Optuna : Hyperparameter tuning

### 7.2 Lectures

- Goldberg - *Genetic Algorithms in Search, Optimization and Machine Learning*
- Eiben & Smith - *Introduction to Evolutionary Computing*

## 8 Notebooks Pratiques

Ce chapitre est accompagné des notebooks suivants :

- `10_demo_ag_base.ipynb` : Introduction aux algorithmes génétiques
  - Implémentation AG from scratch
  - Optimisation de fonctions mathématiques
  - Résolution du problème du voyageur de commerce (TSP)
  - Visualisation de l'évolution des populations
- `10_demo_applications.ipynb` : Applications pratiques des AG
  - Hyperparameter tuning avec AG
  - Feature selection
  - Comparaison avec Grid Search et Random Search
  - Cas d'usage sur datasets réels