utad UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

Computação Natural

Mestrado em Engenharia Informática - 1.º ano

José Paulo Barroso de Moura Oliveira
Eduardo José Solteiro Pires

Trabalho Prático

Otimização de Redes Neuronais usando Algoritmos Genéticos

Diogo Medeiros (70633) João Santos (68843)

maio 2023

Resumo

Neste trabalho, será desenvolvido um algoritmo genético, utilizando a linguagem de programação Python, para otimizar os parâmetros de uma rede neuronal artificial. Pretende-se entender melhor a eficácia e utilidade dos algoritmos genéticos na otimização de modelos de aprendizagem de máquina. Para analisar o desempenho do algoritmo, será utilizado um modelo capaz de classificar espécies de flores da base de dados IRIS, e os resultados serão comparados com uma implementação já existente em Python, recorrendo à biblioteca PyGAD.

Através deste estudo, espera-se demonstrar a importância e desafios da otimização de parâmetros de redes neuronais utilizando algoritmos genéticos. A complexidade e o vasto espaço de soluções envolvidos na pesquisa pelos melhores conjuntos de parâmetros destacam a necessidade de abordagens eficazes para melhorar o desempenho e adaptabilidade das redes neuronais em tarefas de classificação e reconhecimento de padrões. O trabalho realizado servirá como base para futuras investigações e melhorias nessa área, visando o desenvolvimento de modelos mais refinados e eficazes, capazes de otimizar os resultados em diversas aplicações, contribuindo para avanços significativos no campo da inteligência artificial e da aprendizagem de máquina.

Palavras-chave – algoritmos de otimização, otimização de parâmetros, redes neuronais, algoritmos genéticos, operadores genéticos, espaço de pesquisa, classificação, reconhecimento de padrões, aprendizagem de máquina

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Excertos de Código

Capítulo 1

Introdução

No âmbito da Unidade Curricular de Computação Natural, foi solicitado um trabalho prático que consiste no desenvolvimento de um algoritmo genético na linguagem de programação Python, para otimizar os parâmetros de uma Rede Neuronal Artificial.

1.1 Algoritmos Genéticos

Os algoritmos genéticos (GA) são algoritmos de pesquisa baseados na mecânica da seleção natural e nos princípios da genética, que combinam a sobrevivência dos indivíduos mais aptos numa população de entidades binárias - cromossomas, com a troca estruturada e aleatória de informação (Fig. 1.1).

Em cada geração, um novo conjunto de indivíduos (sequências) é criado usando pedaços dos elementos mais aptos. Ainda que aleatórios, os algoritmos genéticos não são uma simples "caminhada" estocástica, explorando eficientemente a informação histórica para especular novos pontos de pesquisa com um desempenho esperado melhorado [1].

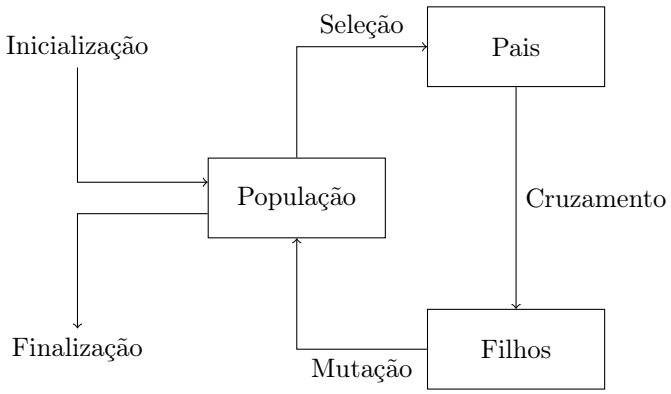

Figura 1.1: Processo geral de um algoritmo genético

1.2 Redes Neuronais

As Redes Neuronais Artificiais (ANNs) são modelos computacionais inspirados no funcionamento do cérebro humano, compostos por unidades de processamento, chamadas neurónios, responsáveis por receber entradas, processar informação e gerar saídas. As ANNs utilizam conexões sinápticas ajustáveis para aprender com os dados de entrada [2].

Essas redes possuem a capacidade de aprender por meio de exemplos e generalizar esse conhecimento para novos dados. Por essa razão, elas são amplamente utilizadas em diversas áreas, como reconhecimento de padrões, classificação, previsão, processamento de imagens, entre outras [3].

Atualmente, as ANNs são consideradas ferramentas fundamentais para aprendizagem de máquina (ML) e inteligência artificial (AI), com inúmeras aplicações práticas em saúde, finanças, indústria e outros setores.

1.3 Otimização de Redes Neuronais

A crescente popularidade das ANNs na década de 80 potenciou o surgimento de novas técnicas para o seu treino e otimização, com particular atenção para os algoritmos genéticos. A retropropagação (BP), apesar da sua utilidade e simplicidade, apresentava limitações, entre as quais:

- a pobre escalabilidade face ao aumento da complexidade do problema (maior dimensionalidade e/ou complexidade dos dados), resultando numa degradação de desempenho;

- a necessidade de diferenciabilidade no cálculo dos gradientes, que impede o seu uso em certos tipos de nós e critérios de otimização não diferenciáveis.

Para superar essas limitações, Montana e Davis [4] propuseram um algoritmo genético capaz de codificar os pesos duma rede neuronal *feed-forward* (FNN) numa lista de números reais, utilizando uma função de *fitness* para avaliar o desempenho da rede, e incorporando operadores de mutação, cruzamento e gradiente para gerar novas soluções.

Montana e Davis realizaram múltiplas experiências usando uma base de dados de imagens de energia acústica, e compararam o desempenho do algoritmo genético com o da retropropagação. Os resultados mostraram que o GA superou o BP em termos de velocidade de treino e capacidade de lidar com tipos de nós e critérios de otimização, provando os GAs como uma alternativa eficaz à retropropagação, especialmente em problemas complexos e não diferenciáveis.

Capítulo 2

Problema

O treino de redes neuronais é um processo fundamental para obter um desempenho adequado em tarefas de classificação, reconhecimento de padrões e outras aplicações. No entanto, encontrar os parâmetros ideais para uma rede neural pode ser um desafio devido à sua complexidade e ao vasto espaço de busca envolvido.

Nesse contexto, técnicas como os algoritmos genéticos (GAs) têm mostrado eficácia na otimização dos parâmetros das redes neuronais. Os GAs combinam conceitos da teoria da evolução biológica, como a seleção natural, o cruzamento e a mutação, para explorar o espaço de pesquisa de forma eficiente. Estas técnicas permitem encontrar conjuntos de parâmetros que maximizem o desempenho da rede neuronal, melhorando a precisão das classificações e tornando-a mais adaptável a diferentes conjuntos de dados.

O objetivo do presente trabalho consiste em desenvolver um algoritmo genético capaz de encontrar os pesos ideais de uma rede neuronal, de modo a maximizar o seu desempenho na classificação de flores do género Iris *L.*, recorrendo ao dataset Iris [5].

A rede neuronal em questão deverá ter a seguinte estrutura:

- 4 neurónios na camada de entrada
- 10 neurónios na camada oculta
- 3 neurónios na camada de saída

O algoritmo desenvolvido deverá ser igualmente capaz de inibir conexões a certos neurónios da rede durante a sua execução. Essa modificação visa explorar a capacidade da rede de desativar neurónios irrelevantes para a tarefa de classificação, visando otimizar a sua eficiência computacional.

Capítulo 3

Metodologia

A escolha da linguagem Python para este trabalho deve-se a várias vantagens que a mesma oferece. Python é uma linguagem de programação de alto nível, reconhecida pela sua sintaxe simples e legibilidade, facilitando o desenvolvimento e a manutenção do código. Por outro lado, Python possui uma ampla gama de bibliotecas e 'frameworks' para aprendizagem de máquina e computação científica, como NumPy, scikit-learn e PyTorch, que facilitam a implementação de redes neuronais e algoritmos genéticos.

De modo a poder comparar os resultados obtidos, o algoritmo genético foi implementado de raiz e com recurso à biblioteca PyGAD [6]. PyGAD é uma biblioteca 'open-source', compatível com Keras e PyTorch, utilizada para construir algoritmos genéticos e otimizar algoritmos de ML. Este oferece diferentes tipos de operadores de cruzamento, mutação e seleção, permitindo a otimização de inúmeros problemas, através da personalização da função de aptidão.

3.1 Modelo de Classificação

Tal como foi referido em 2, e respeitando as características do dataset IRIS, a rede neuronal tem 4 neurónios na camada de entrada, 10 neurónios na camada oculta, e 3 neurónios na camada de saída, de modo a classificar as 3 espécies.

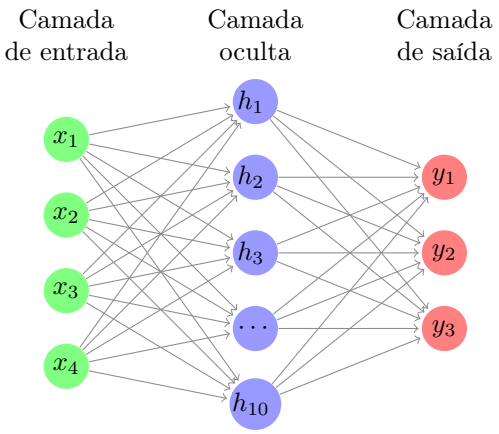

Figura 3.1: Estrutura do modelo de classificação

O modelo em questão é uma "Fully Connected Neural Network" (FCN), também conhecida por Rede Neuronal Totalmente Conectada, como se poder ver pela Fig. 3.1. Trata-se de uma arquitetura de rede neuronal artificial na qual cada neurónio numa camada está conectado a todos os neurónios da camada seguinte - conexões totalmente conectadas ou conexões densas.

Camada	**Pesos**	**Bias**	**Parâmetros**
Entrada (X)	40	10	50
Oculta (H)	30	3	33
Saída (Y)	-	-	-
Total	50	33	83

Tabela 3.1: Composição da rede neuronal

Assim, e segundo a informação disponível na Tabela 3.1, o modelo em questão contém um total de 83 parâmetros que podem ser otimizados pelo algoritmo genético descrito na secção seguinte.

3.2 Algoritmo Genético

O algoritmo genético desenvolvido, intitulado `GeneticAlgorithm` e descrito em anexo A, apresenta os seguintes atributos configuráveis:

Nome	**Tipo**	**Por omissão**
`model`	`nn.Module`	-
`population_size`	`int`	10
`mutation_rate`	`float`	0.05
`neuron_off_rate`	`float`	10^{-3}
`crossover_rate`	`float`	0.95
`elitism`	`bool`	`True`
`num_generations`	`int`	100
`on_generation_interval`	`int`	10
`best_score`	`float`	0.0
`best_solution`	`np.ndarray`	`None`
`fitness_scores`	`list[float]`	-
`fitness_fn`	`Callable`	`None`
`on_generation`	`Callable`	`None`

Tabela 3.2: Parâmetros do algoritmo genético

O método `run`, detalhado em Alg. 1, é responsável pela execução do algoritmo genético por um determinado número de gerações, de acordo com os parâmetros fornecidos na instanciação do mesmo.

Inicialmente, a população de pesos é inicializada aleatoriamente de acordo com o tamanho desejado e o número de parâmetros do modelo a otimizar. De seguida, é executado o ciclo principal, que inclui as seguintes etapas:

- Seleção dos pais: Os pais são selecionados a partir da população atual, usando o método `select_parents`;

- Cruzamento: A informação genética dos pais é combinada usando o operador de cruzamento definido no método `crossover`;
- Mutação: A informação genética dos filhos sofre alterações usando o método `mutate`, podendo ser modificada ou anulada (desligar a conexão);
- Atualização da população: A população atual é substituída pelos filhos;
- Preservação do melhor indivíduo: O melhor indivíduo é preservado usando elitismo, substituindo o pior indivíduo na população;
- Armazenamento da aptidão: A aptidão do melhor indivíduo é armazenada na lista de melhores aptidões para posterior análise;
- Chamada da função `on_generation`: Se a geração for múltipla do intervalo de gerações especificado, a função `on_generation` é chamada para fornecer informações sobre o estado atual do algoritmo genético.

Algoritmo 1: Execução do algoritmo genético (`run`)

```
population ← random(populationSize, model.numParams);
for generation ← 0 to numGenerations do
    parents ← selectParents(population);
    children ← crossover(parents);
    children ← mutate(children);
    population ← children;
    scores ← calculateScores(population);
    (maxScore, maxScoreIdx) ← max(scores);
    if maxScore > bestScore then
        bestScore ← maxScore;
        bestSolution ← population[maxScoreIdx];
    end
    if elitism then
        (_, minScoreIdx) ← min(scores);
        population[minScoreIdx] ← bestSolution;
    end
    fitnessScores.add(bestScore);
    if generation % onGenerationInterval == 0 then
        onGeneration(generation, scores);
    end
end
```

O operador de seleção, definido em Alg. 1, utiliza um processo chamado seleção proporcional à aptidão (ou seleção por roleta) para escolher os pais da próxima geração, no qual é atribuída uma probabilidade de seleção a cada indivíduo com base na sua aptidão relativa. Quanto maior a aptidão de um indivíduo, maior a probabilidade de ele ser selecionado. Este processo é repetido até que pares suficientes de pais sejam selecionados para a reprodução.

```
def select_parents(self, population: Population) -> Population:
    scores = self.calculate_scores(population)
    normalized_scores = scores / np.sum(scores)
    parent_indices = np.random.choice(
        range(self.population_size),
        size=(self.population_size // 2, 2),
        replace=True,
        p=normalized_scores
    )
    parents = population[parent_indices]
    return parents
```

Excerto de código 1: Operador de seleção

```
def crossover(self, parents1: Population, parents2: Population):
    sh = parents1.shape
    cross_pts = np.random.randint(1, sh[1], size=sh[0])
    mask = np.random.random(size=sh) < self.crossover_rate
    crossover_mask = np.arange(sh[1]) < cross_pts[:, np.newaxis]
    mask = np.logical_and(mask, crossover_mask)
    child1 = parents1 * np.logical_not(mask) + parents2 * mask
    child2 = parents2 * np.logical_not(mask) + parents1 * mask
    children = np.concatenate((child1, child2), axis=0)
    return children
```

Excerto de código 2: Operador de cruzamento

No caso do operador de cruzamento, definido em Alg. 2, é utilizada a técnica de cruzamento num único ponto, ou "single point crossover". Nessa técnica, um ponto de cruzamento é escolhido aleatoriamente e o material genético é trocado entre os pais até esse ponto, resultando nos filhos. A probabilidade de ocorrer cruzamento é determinada pelo valor de `crossover_rate`. Esse processo é repetido para cada par de pais, gerando o par correspondente de filhos.

```
def mutate(self, children: Population) -> Population:
    mask_off = (np.random.random(size=children.shape)
                < self.neuron_off_rate)
    mask_mutate = (np.random.random(size=children.shape)
                   < self.mutation_rate)
    children[mask_off] = 0.0
    children[mask_mutate] += np.random.normal(scale=0.1,
        size=np.sum(mask_mutate))
    return children
```

Excerto de código 3: Operador de mutação

Por fim, o operador de mutação, definido em Alg. 3, atua sobre a população de filhos, onde alguns neurónios são desativados (definidos como 0), e alguns parâmetros sofrem mutação adicionando valores aleatórios extraídos de uma distribuição normal com $\mu = 0$ e $\sigma = 0.1$. Isso introduz variabilidade e diversidade genética na população de filhos.

Capítulo 4

Resultados

Neste capítulo serão apresentados e analisados os resultados da execução do algorithmo genético, descrito na Secção 3.2, para a otimização do modelo de classificação, descrito na Secção 3.1, bem como os resultados obtidos com a biblioteca PyGAD.

Com o auxílio da biblioteca `scikit-learn`, foi possível carregar o dataset IRIS usando a função `load_iris`, normalizá-lo com a classe `StandardScaler`, e dividi-lo em conjuntos de treino e teste, com a função `train_test_split`, segundo a proporção 90/10. Após a divisão, os conjuntos de treino e teste foram convertidos em `Tensors` do PyTorch, de modo a poderem ser utilizados pelo modelo de classificação.

```
def load_params(model: nn.Module, params: ModelParams) -> None:
    model_params = model.state_dict()
    model_params['0.weight'] =
    ↪  torch.FloatTensor(params[:40].reshape(10, 4))
    model_params['0.bias'] = torch.FloatTensor(params[40:50])
    model_params['2.weight'] =
    ↪  torch.FloatTensor(params[50:80].reshape(3, 10))
    model_params['2.bias'] = torch.FloatTensor(params[80:])
    model.load_state_dict(model_params)
```

Excerto de código 4: Função para carregar os parâmetros do modelo

A seguir, foi criado um modelo de classificação, com a estrutura descrita na Secção 3.1, com a função de ativação `ReLU` na camada oculta, e definido o critério de erro como `CrossEntropyLoss`, dado se tratar de um problema de classificação multiclasse.

```
def fitness_fn(solution: ModelParams) -> float:
    load_params(iris_model, solution)
    with torch.no_grad():
        outputs = iris_model(X_train)
        loss = criterion(outputs, y_train)
        fitness = 1.0 / (loss.detach().item() + 1e-8)
    return fitness
```

Excerto de código 5: Função de aptidão

```python
def on_generation(generation: int, scores: list[float]) -> None:
    print(
        f"Generation: {generation:0=3} "
        f"Best fitness: {np.max(scores):.10f} "
        f"Average fitness: {np.mean(scores):.10f} "
        f"Worst fitness: {np.min(scores):.10f}"
    )
```

Excerto de código 6: Função de 'callback´ chamada em cada N gerações

Foram definidas as funções `load_params` (Alg. 4), responsável por atualizar os pesos do modelo com os valores passados como argumento, a função de aptidão (Alg. 5), que recebe como argumento um vetor de parâmetros, e retorna o inverso da função de erro, calculada com base no conjunto de treino, e a função `on_generation`, chamada a cada geração, que imprime os valores da melhor, pior e aptidão média da população.

Parâmetro	**GeneticAlgorithm**	**PyGAD**
Tamanho da população	30	10
Número de gerações	600	200
Intervalo de gerações	50	-
Probabilidade de mutação	0.05	-
Probabilidade de *crossover*	0.95	-
Probabilidade de desconexão	10^{-4}	-
Elitismo	`True`	`True`

Tabela 4.1: Valores dos parâmetros do algoritmo genético

Por fim, o algoritmo genético foi instanciado com os parâmetros apresentados na Tabela 4.1, que também contém os valores utilizados no algoritmo da biblioteca PyGAD.

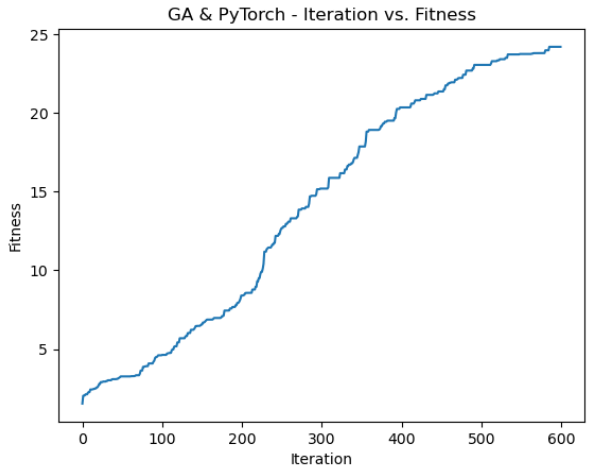

Figura 4.1: Evolução da aptidão da melhor solução - GeneticAlgorithm

O algoritmo genético implementado com a classe `GeneticAlgorithm` foi executado durante 600 gerações, com uma população de 30 soluções. A incorporação de elitismo provou ser eficaz, permitindo uma evolução gradual da aptidão, como é possível verificar na Fig. 4.1. Inicialmente, a aptidão apresenta um aumento prolongado, ainda que esse acelere por volta das 200 gerações. Esse aumento torna-se menos evidente após 400 gerações, acabando por estagnar, sensivelmente, a partir de 500 iterações.

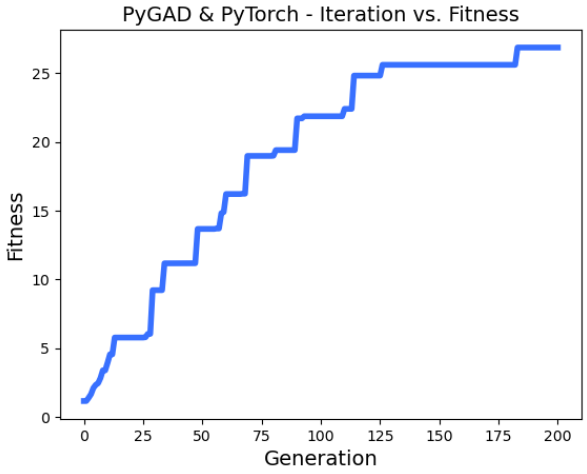

Figura 4.2: Evolução da aptidão da melhor solução - PyGAD

No caso do algoritmo genético implementado com a biblioteca PyGAD, este foi executado durante 200 gerações, com uma população fixa de 10 indivíduos, ou seja, apenas 1/3 do algoritmo original. Dado tratar-se de uma biblioteca altamente otimizada, este foi capaz de atingir uma aptidão semelhante ao anterior em apenas 115 gerações, ainda que tenha estagnado no final da sua execução, como é evidente pela Fig. 4.2.

Métricas	**GeneticAlgoritm**	**PyGAD**
Melhor aptidão	24.1857	26.8775
Perda de treino	0.0414	0.0372
Acurácia de treino	98.52%	98.52%
Perda de teste	0.0112	0.0120
Acurácia de teste	100%	100%

Tabela 4.2: Métricas dos algoritmos

Apesar do algoritmo PyGAD ter sido capaz de atingir uma aptidão final superior de 26.8775, face à aptidão do algoritmo GeneticAlgorithm de 24.1857, ambos conseguiram obter excelentes resultados na classificação das três espécies do género Iris, como é possível observar na Tabela 4.2, com uma exatidão de treino e teste de, respetivamente, 98.52% e 100%.

Capítulo 5

Conclusões

Neste trabalho, desenvolveu-se um algoritmo genético na linguagem de programação Python, para otimizar os parâmetros de uma rede neuronal artificial. A rede neuronal era uma Rede Neuronal Totalmente Conectada, capaz de classificar as três espécies de flores do género Iris, que compõem o dataset IRIS [5].

O algoritmo GeneticAlgorithm, desenvolvido de raiz, mostrou um desempenho consistente ao longo das 600 gerações. A incorporação do elitismo contribuiu para um aumento gradual na aptidão durante as primeiras centenas de gerações, resultando num modelo bem ajustado. No entanto, observou-se uma estagnação da aptidão após aproximadamente 500 iterações, indicando que o algoritmo atingiu o seu limite de otimização.

Por outro lado, a biblioteca PyGAD, dada a sua alta otimização, conseguiu alcançar uma aptidão semelhante em apenas 115 gerações. Embora também tenha mostrado uma estagnação no final da execução, o desempenho geral foi notável considerando o número reduzido de gerações e a menor população.

Apesar da ligeira diferença na aptidão final, ambos os algoritmos apresentaram resultados impressionantes na tarefa de classificação, tendo alcançado altas taxas de acurácia quer nos dados de treino, quer nos dados de teste, com uma exatidão de 98.52% e 100%, respetivamente.

Como trabalho futuro, seria relevante explorar diferentes operadores genéticos, como seleção, cruzamento e mutação. Diferentes estratégias de seleção, como a seleção por torneio, pode permitir uma exploração mais eficiente do espaço de pesquisa. Além disso, a utilização de diferentes técnicas de cruzamento, como cruzamento de múltiplos pontos ou cruzamento uniforme, e técnicas de mutação mais sofisticadas, podem fornecer uma maior diversidade genética e, potencialmente, melhorar a busca por soluções ótimas.

Em suma, ambos os algoritmos genéticos mostraram-se eficientes na otimização dos parâmetros da rede neuronal para a tarefa de classificação especificada. Embora apresentassem diferenças em termos do desempenho e número de gerações necessárias para atingir a aptidão final, ambos alcançaram resultados altamente precisos na classificação, demonstrando a relevância e utilidade dos algoritmos genéticos neste tipo de aplicações.

Apêndice A

Anexos dos ficheiros fonte

A.1 Especificação do algoritmo genético

```
from dataclasses import dataclass, field
from typing import Callable

import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn

ModelParams = np.ndarray[tuple[int], np.dtype[np.float64]]
Population = np.ndarray[tuple[int, int], np.dtype[np.float64]]


def compute_num_params(model: nn.Module) -> int:
    return sum(p.numel() for p in model.parameters() if
    ↪  p.requires_grad)


@dataclass
class GeneticAlgorithm:
    model: nn.Module
    population_size: int = 10
    mutation_rate: float = 0.05
    neuron_off_rate: float = 1e-3
    crossover_rate: float = 0.95
    elitism: bool = True
    num_generations: int = 100
    on_generation_interval: int = 10
    best_score: float = 0.0
    best_solution: ModelParams = None
    fitness_scores: list[float] = field(default_factory=list)
    fitness_fn: Callable[[ModelParams], float] = None
    on_generation: Callable[[int, list[float]], None] = None

    def crossover(self, parents1: Population, parents2:
    ↪  Population) -> Population:
        num_parents, num_params = parents1.shape
```

```
        crossover_points = np.random.randint(1, num_params,
        ↪  size=num_parents)
        mask = np.random.random(size=parents1.shape) <
        ↪  self.crossover_rate
        crossover_mask = np.arange(num_params) <
        ↪  crossover_points[:, np.newaxis]
        mask = np.logical_and(mask, crossover_mask)
        child1 = parents1 * np.logical_not(mask) + parents2 *
        ↪  mask
        child2 = parents2 * np.logical_not(mask) + parents1 *
        ↪  mask
        children = np.concatenate((child1, child2), axis=0)
        return children

    def mutate(self, children: Population) -> Population:
        mask_off = np.random.random(size=children.shape) <
        ↪  self.neuron_off_rate
        mask_mutate = np.random.random(size=children.shape) <
        ↪  self.mutation_rate
        children[mask_off] = 0.0
        children[mask_mutate] += np.random.normal(scale=0.1,
        ↪  size=np.sum(mask_mutate))
        return children

    def select_parents(self, population: Population) ->
    ↪  Population:
        scores = self.calculate_scores(population)
        normalized_scores = scores / np.sum(scores)
        parent_indices = np.random.choice(
            range(self.population_size),
            size=(self.population_size // 2, 2),
            replace=True,
            p=normalized_scores
        )
        parents = population[parent_indices]
        return parents

    def calculate_scores(self, population: Population) ->
    ↪  list[float]:
        return [self.fitness_fn(individual) for individual in
        ↪  population]

    def run(self) -> None:
        # Initialize population
        num_params = compute_num_params(self.model)
        population: Population = np.random.uniform(low=-1,
        ↪  high=1, size=(self.population_size, num_params))
        scores: list[float] = []
        # Run for num_generations
        for gen in range(self.num_generations):
```

```
            # Select parents
            parents = self.select_parents(population)
            # Vectorized crossover and mutation
            children = self.crossover(parents[:, 0], parents[:,
            ↪  1])
            children_mutated = self.mutate(children)
            # Update population
            population = children_mutated
            # Preserve the best individual using elitism
            scores = self.calculate_scores(population)
            max_score = np.max(scores)
            # Update the best solution and score
            if max_score > self.best_score:
                self.best_score = max_score
                self.best_solution =
                ↪  population[np.argmax(scores)]
            # Preserve the best solution using elitism
            if self.elitism:
                worst_idx = np.argmin(scores)
                population[worst_idx] = self.best_solution
            # Save fitness score of best solution
            self.fitness_scores.append(self.best_score)
            # Print generation info
            if gen % self.on_generation_interval == 0:
                self.on_generation(gen, scores)
        self.on_generation(self.num_generations - 1, scores)

    def plot_fitness(self) -> None:
        """ Plots the fitness of the best solution over time """
        plt.plot(self.fitness_scores)
        plt.title("GA & PyTorch - Iteration vs. Fitness")
        plt.xlabel("Iteration")
        plt.ylabel("Fitness")
        plt.show()

    def print_summary(self) -> None:
        """ Prints a summary of the best solution """
        print(f"Best solution fitness: {self.best_score}")
        print(f"Best solution: {self.best_solution}")
```

Bibliografia

[1] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[2] R. P. Lippmann, "An introduction to computing with neural nets," *ACM SIGARCH Computer Architecture News*, vol. 16, no. 1, pp. 7–25, 3 1988. [Online]. Available: https://dl.acm.org/doi/10.1145/44571.44572

[3] A. Jain, Jianchang Mao, and K. Mohiuddin, "Artificial neural networks: a tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, 3 1996. [Online]. Available: http://ieeexplore.ieee.org/document/485891/

[4] D. J. Montana and L. Davis, "Training Feedforward Neural Networks Using Genetic Algorithms," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, pp. 762–767.

[5] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1988. [Online]. Available: http://archive.ics.uci.edu/ml/datasets/Iris

[6] A. F. Gad, "PyGAD: An Intuitive Genetic Algorithm Python Library," 2021.