Tech Challenge 2: Utilização de Algoritmos Genéticos para Otimização no ambiente 'LunarLander-v2' do toolkit Gymnasium

Aluno: Leandro Cordeiro David
RM: 356103

Introdução

Neste trabalho, abordaremos o uso de Algoritmos Genéticos (AGs) para otimização de um problema específico no framework Gymnasium. Escolhemos o environment **LunarLander-v2** como o alvo de nossa solução, um desafio interessante que simula o pouso de uma nave na Lua. A escolha se deve à complexidade do problema, que combina elementos de física e controle, tornando-o um bom candidato para estudo de AGs. Além disso, o Gymnasium oferece uma plataforma robusta para experimentação e testes, permitindo a aplicação de soluções a diversos problemas de otimização.

Framework Gymnasium

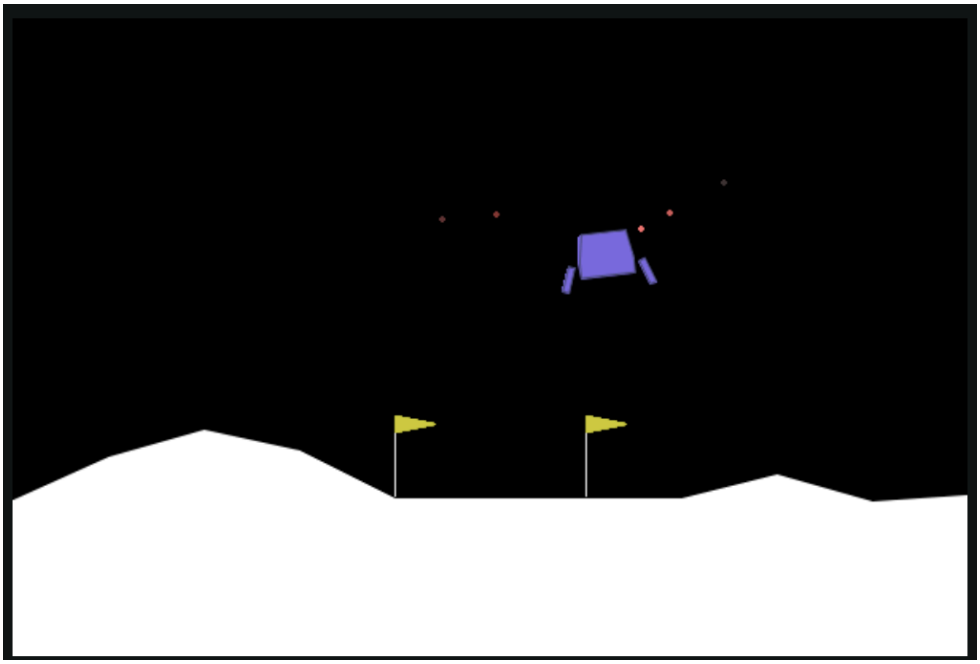
O Gymnasium é um toolkit desenvolvido para fornecer uma ampla gama de simulações (*environments*) que facilitam o desenvolvimento e a comparação de algoritmos de aprendizado por reforço. Ele é amplamente utilizado na comunidade de inteligência artificial por sua simplicidade e flexibilidade, permitindo que pesquisadores e desenvolvedores testem algoritmos em cenários padronizados. Mais informações estão disponíveis no site do projeto: https://gymnasium.farama.org/

Definição do Problema

O problema escolhido para este desafio é a simulação **LunarLander-v2** do Gymnasium. Este ambiente simula o pouso de uma nave em uma superfície lunar, onde o objetivo é pousar a nave suavemente no local designado. O agente deve controlar os motores da nave para ajustar sua trajetória e velocidade, evitando colisões e economizando combustível. Os comandos possíveis para a nave são 0:nenhuma ação, 1:Acionar motor esquerdo, 2:Acionar motor principal, 3:Acionar motor direito. A cada comando executado um novo estado da nave é calculado e uma pontuação é gerada. O score máximo da simulação é de 200 pontos e a simulação é encerrada quando se atinge esta pontuação.Os detalhes técnicos do ambiente LunarLander-v2 estão disponíveis em: https://gymnasium.farama.org/environments/box2d/lunar_lander/

Objetivo: Pousar a nave no local designado com a maior pontuação possível, maximizando a suavidade do pouso e minimizando o consumo de combustível.

Critérios de Sucesso: A solução deve demonstrar uma pontuação média elevada em múltiplas execuções, sendo capaz de pousar a nave consistentemente no local designado.



Tela da simulação LunarLander-v2 do gymnasium

Implementação do Algoritmo Genético

O código a seguir foi desenvolvido em Python, utilizando o Gymnasium para simulação e Numpy para operações numéricas.

O código também está disponível publicamente no repositório github em https://github.com/leandrodvd/fiap-postech-ia4devs/blob/main/techchallenge2_geneticalgorithm/moon_lander.py

```
import gymnasium as gym
import matplotlib.pyplot as plt
import numpy as np
import random


# Define the fitness function
def evaluate_fitness(individual, env):
    state = env.reset(seed=seed_value)
    total_reward = 0
    for i in range(len(individual)):
        # get next action to execute from individual
        action = individual[i]
        # execute the action and get the reward
        observation, reward, terminated, truncated, info = env.step(action)
        total_reward += reward
        # check if execution has terminated
        if terminated or truncated:
            break
    return total_reward
```

```
# Initialize the population
def initialize_population(pop_size, individual_length, env):
    population = []
    for _ in range(pop_size):
        # create an individual codified as an array of random actions
        individual = [env.action_space.sample() for _ in
range(individual_length)]
        # append new individual to the population under cosntruction
        population.append(individual)
    return population

# Sort the population to simplify selection
def sort_population(population, fitnesses):
    # Combine the population and fitness lists usinf zip and sort it based on
the fitness
    zipped = zip(fitnesses, population)
    sorted_pairs = sorted(zipped, reverse=True, key=lambda x: x[0])

    # Separate the sorted lists
    sorted_fitnesses, sorted_individuals = zip(*sorted_pairs)
    sorted_fitnesses = list(sorted_fitnesses)
    sorted_individuals = list(sorted_individuals)

    return sorted_individuals, sorted_fitnesses

# select random parents for crossover
def select_parents(sorted_population, selected_parents):
    # choose 2 random individuals from the top 10 of the sorted population
    return random.choices(sorted_population[:selected_parents], k=2)

# Perform crossover
def crossover(parent1, parent2):
    # Select a random index to split the parents
    idx = np.random.randint(1, len(parent1))
    # generate 2 children by combining the parents
    child1 = np.concatenate((parent1[:idx], parent2[idx:]))
    child2 = np.concatenate((parent2[:idx], parent1[idx:]))
    return child1, child2

# Perform mutation
def mutate(env, individual, mutation_rate):
    # decide if mutation will occure based on the mutation rate~
    if random.random() < mutation_rate:
        # mutate using the mutation rate to mutate cromossomes randonly
        for i in range(len(individual)):
            if random.random() < mutation_rate:
                individual[i] =  env.action_space.sample()
    return individual

# Function to update the plot
def update_plot(gen, fitness_scores):
    line.set_data(range(len(fitness_scores)), fitness_scores)
    ax.relim()  # Recalculate limits
    ax.autoscale_view()  # Rescale view
    plt.draw()
    plt.pause(0.1)  # Pause to update the plot
    # clear_output(wait=True)
    plt.show()



# Set the random seed
seed_value = 42
```

```
# Initialize the environment and set the seed
env = gym.make('LunarLander-v2', render_mode=None)
env.action_space.seed(seed_value)
np.random.seed(seed_value)
random.seed(seed_value)

# Main Genetic Algorithm loop
population_size = 100
mutation_rate = 0.4
selected_parents = 60
generations = 1000
individual_length = 200

# initialize chart to visualize fintness evolution
plt.ion()  # Turn on interactive mode
fig, ax = plt.subplots()
line, = ax.plot([], [], 'b-')  # Initialize an empty line
ax.set_xlim(0, generations)
ax.set_ylim(-300, 300)  # Set appropriate y-axis limits
ax.set_xlabel('Generation')
ax.set_ylabel('Fitness/Score')

# initialize a random population to be the first generation
population = initialize_population(population_size, individual_length, env)
best_individuals = []
fitness_scores = []
best_individual = []
best_fitness = 0
for generation in range(generations):
    # calculate fitness for each individual in the population
    fitnesses = [evaluate_fitness(ind, env) for ind in population]
    # sort the population based on fitness
    sorted_population, sorted_fitness = sort_population(population, fitnesses)
    # save best individual of this generation for later comparison
    best_individual = sorted_population[0]
    best_fitness = sorted_fitness[0]
    best_individuals.append(best_individual);
    fitness_scores.append(best_fitness);
    update_plot(generation, fitness_scores)
    print("Generation:", generation," Best fitness:", sorted_fitness[0])
    # Initialize next generation using elitism - keep the best 2 individual in
the next generation
    new_population = [sorted_population[0], sorted_population[1]]
    while len(new_population) < population_size:
        # select parents for crossover
        parent1, parent2 = select_parents(sorted_population, selected_parents)
        # perform crossover and generate 2 new childs for the new generation
        child1, child2 = crossover(parent1, parent2)
        # mutate the childs with a given mutation rate
        new_population.extend([mutate(env, child1, mutation_rate), mutate(env,
child2, mutation_rate)])
    population = new_population

print("Best fitness:", best_score)
print("Best individual:", best_individual)
# visualize/render best individual
env = gym.make('LunarLander-v2', render_mode="human")
evaluate_fitness(best_individual, env)

plt.ioff()  # Turn off interactive mode
plt.show()
```

Explicação do Código:

Função de avaliação de aptidão:
Calcula a aptidão (fitness) de um indivíduo executando as ações codificadas utilizando o ambiente gymnasium. Cada ação executada gera uma recompensa de acordo com o resultado da ação. No caso do ambiente LunarLander-v2, a recompensa considera a proximidade do alvo de pouso, a velocidade da nave, uso de combustível, o contato com o solo e ângulo de inclinação da nave.

```
# Define the fitness function
def evaluate_fitness(individual, env):
    state = env.reset(seed=seed_value)
    total_reward = 0
    for i in range(len(individual)):
        # get next action to execute from individual
        action = individual[i]
        # execute the action and get the reward
        observation, reward, terminated, truncated, info = env.step(action)
        total_reward += reward
        # check if execution has terminated
        if terminated or truncated:
            break
    return total_reward
```

Função de inicialização da população:
Inicializa a população codificando cada indivíduo com ações aleatórias dentre as possíveis ações disponibilizadas para o ambiente. No caso do ambiente LunarLander-v2 as ações possíveis são codificadas como: 0:nenhuma ação, 1:Acionar motor esquerdo, 2:Acionar motor principal, 3:Acionar motor direito. O código '`env.action_space.sample()`' seleciona uma ação aleatória.

```
# Initialize the population
def initialize_population(pop_size, individual_length, env):
    population = []
    for _ in range(pop_size):
        # create an individual codified as an array of random actions
        individual = [env.action_space.sample() for _ in
range(individual_length)]
        # append new individual to the population under construction
        population.append(individual)
    return population
```

Função de ordenação da população com base na aptidão de cada indivíduo:
Ordena a população com base na lista de aptidão calculada

```
# Sort the population to simplify selection
def sort_population(population, fitnesses):
    # Combine the population and fitness lists usinf zip and sort it based on
the fitness
    zipped = zip(fitnesses, population)
    sorted_pairs = sorted(zipped, reverse=True, key=lambda x: x[0])

    # Separate the sorted lists
    sorted_fitnesses, sorted_individuals = zip(*sorted_pairs)
    sorted_fitnesses = list(sorted_fitnesses)
    sorted_individuals = list(sorted_individuals)

    return sorted_individuals, sorted_fitnesses
```

Função de seleção de 2 genitores para realizar o crossover
Recebe a população ordenada e seleciona 2 genitores aleatoriamente dentre os melhores da população

```
# select random parents for crossover
def select_parents(sorted_population, selected_parents):
    # choose 2 random individuals from the top sorted population
    return random.choices(sorted_population[:selected_parents], k=2)
```

Função de crossover para gerar 2 filhos a partir de 2 genitores
Realiza o cruzamento ordenado de 2 genitores e gera 2 novos filhos. Um índice aleatório é gerado para dividir os genitores, o primeiro filho é gerado a partir da primeira parte do genitor 1 e a seginda parte do genitor 2 e o segundo filho é gerado com a primeira parte do genitor 2 e a segunda parte do genitor 1.

```
# Perform crossover
def crossover(parent1, parent2):
    # Select a random index to split the parents
    idx = np.random.randint(1, len(parent1))
    # generate 2 children by combining the parents
    child1 = np.concatenate((parent1[:idx], parent2[idx:]))
    child2 = np.concatenate((parent2[:idx], parent1[idx:]))
    return child1, child2
```

Função de mutação de um indivíduo considerando uma taxa de mutação fornecida
Utiliza uma taxa de mutação para primeiramente decidir se um indivíduo deve ou não sofre mutação e, em caso positivo, utiliza a mesma taxa de mutação para decidir quais cromossomos de um indivíduo sofrerão mutação. A mutação altera uma ação codificada para uma nova ação aleatória.

```
# Perform mutation
def mutate(env, individual, mutation_rate):
    # decide if mutation will occure based on the mutation rate
    if random.random() < mutation_rate:
        # mutate using the mutation rate to mutate cromossomes randonly
        for i in range(len(individual)):
            if random.random() < mutation_rate:
                individual[i] =  env.action_space.sample()
    return individual
```

Função de atualização do gráfico que exibe o melhor resultado de cada geração

Atualiza o gráfico com o melhor score de uma geração.

```
# Function to update the plot
def update_plot(gen, fitness_scores):
    line.set_data(range(len(fitness_scores)), fitness_scores)
    ax.relim()  # Recalculate limits
    ax.autoscale_view()  # Rescale view
    plt.draw()
    plt.pause(0.1)  # Pause to update the plot
    # clear_output(wait=True)
    plt.show()
```

Inicialização do gráfico que exibe a evolução dos melhores resultados de cada geração

```
# initialize chart to visualize fitness evolution
plt.ion()  # Turn on interactive mode
fig, ax = plt.subplots()
line, = ax.plot([], [], 'b-')  # Initialize an empty line
ax.set_xlim(0, generations)
ax.set_ylim(-300, 300)  # Set appropriate y-axis limits
ax.set_xlabel('Generation')
ax.set_ylabel('Fitness/Score')
```

Inicialização do ambiente gymnasium com uma semente pré definida para garantir execuções iguais

```
# Set the random seed
seed_value = 42

# Initialize the environment and set the seed
env = gym.make('LunarLander-v2', render_mode="none")
env.action_space.seed(seed_value)
np.random.seed(seed_value)
random.seed(seed_value)
```

Loop Principal do Algoritmo Genético:

Após a inicialização da primeira geração codificando individuos com ações aleaatórias, O loop principal do algoritmo genético segue os passos:

1. Cálculo da aptidão de todos os indivíduos da geração
2. Ordenação dos indivíduos com base na aptidão calculada
3. Inicialização de uma nova geração com elitismo, mantendo os indivíduos mais aptos da geração anterior
4. Seleção de genitores dentre os mais aptos
5. Mutação ordenada para criar uma nova geração a partir dos genitores selecionados

```
# Main Genetic Algorithm loop
population_size = 50
mutation_rate = 0.4
selected_parents = 30
generations = 1000
individual_length = 200

# initialize a random population to be the first generation
population = initialize_population(population_size, individual_length, env)
best_individuals = []
fitness_scores = []
best_individual = []
best_fitness = 0
for generation in range(generations):
    # calculate fitness for each individual in the population
    fitnesses = [evaluate_fitness(ind, env) for ind in population]
    # sort the population based on fitness
    sorted_population, sorted_fitness = sort_population(population, fitnesses)
    # save best individual of this generation for later comparison
    best_individual = sorted_population[0]
    best_fitness = sorted_fitness[0]
    best_individuals.append(best_individual);
    fitness_scores.append(best_fitness);
    update_plot(generation, fitness_scores)
    print("Generation:", generation," Best fitness:", sorted_fitness[0])
    # Initialize next generation using elitism - keep the best 2 individual in
the next generation
    new_population = [sorted_population[0], sorted_population[1]]
    while len(new_population) < population_size:
        # select parents for crossover
        parent1, parent2 = select_parents(sorted_population, selected_parents)
        # perform crossover and generate 2 new childs for the new generation
        child1, child2 = crossover(parent1, parent2)
        # mutate the childs with a given mutation rate
        new_population.extend([mutate(env, child1, mutation_rate), mutate(env,
child2, mutation_rate)])
    population = new_population

print("Best fitness:", best_score)
print("Best individual:", best_individual)
# visualize/render best individual
env = gym.make('LunarLander-v2', render_mode="human")
evaluate_fitness(best_individual, env)

plt.ioff()  # Turn off interactive mode
plt.show()
```

Resultados e Conclusões

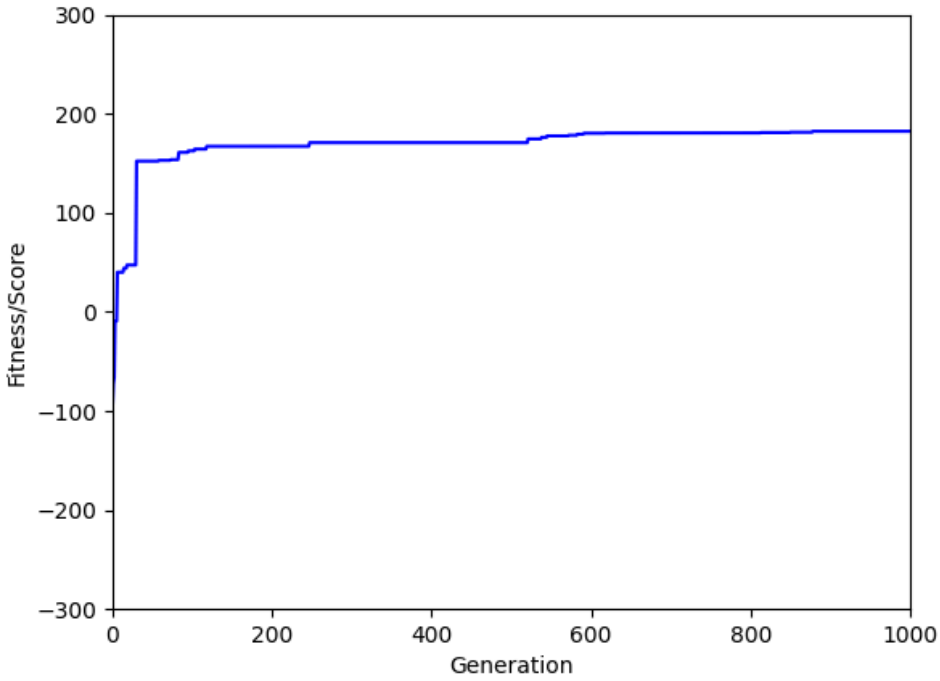
Foram feitas diversas execuções do algoritmo genético para 1000 gerações alterando os parâmetros de inicialização do algoritmo. Foram experimentadas variações no tamanho da população, taxa de mutação e quantidade de indivíduos considerados na etapa de seleção de genitores. A tabela a seguir mostra a melhor pontuação obtida em cada execução.

Tamanho da população	Taxa de mutação	Quantidade de indivíduos selecionados	Melhor pontuação
10	20%	5	40,73
10	30%	5	157,92
10	40%	5	169,97
50	40%	5	93,38
50	20%	30	56,6
50	30%	30	172,63
50	40%	30	182,26
100	30%	30	119,1
100	40%	30	164,85
100	40%	60	169,19

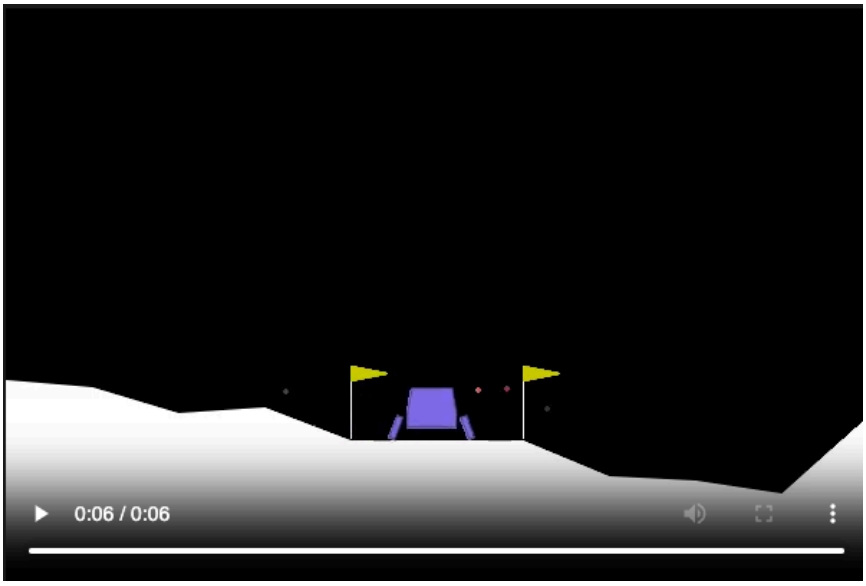
A melhor pontuação foi de 182,26, obtida simulando 1000 gerações com população de 50 indivíduos, taxa de mutação de 40% e considerando os 30 melhores indivíduos de cada geração como genitores na etapa de cruzamento.

O score máximo da simulação é de 200 pontos e a simulação é encerrada quando se atinge esta pontuação. Dessa forma, um score de 182,26 pode ser considerado um ótimo resultado.

O gráfico a seguir mostra a evolução da melhor pontuação de cada geração na melhor execução que obteve os 182,26 pontos:



A imagem a seguir mostra a posição final da nave para o resultado acima:



O video exibindo a execução completa do pouso está disponível para download em https://github.com/leandrodvd/fiap-postech-ia4devs/raw/main/techchallenge2_geneticalgorithm/best_result.mp4

Conclusão

A solução desenvolvida demonstra a eficácia dos Algoritmos Genéticos em resolver problemas complexos de otimização e pode ser facilmente adaptada para outros environments do Gymnasium ou problemas do mundo real.

Os resultados de múltiplas execuções com parâmetros diferentes demonstra o quanto tais parâmetros podem influenciar drasticamente o resultado final sendo necessária a experimentação com diferentes valores para que se encontre os melhores resultados.

Os Algoritmos Genéticos se mostraram uma abordagem robusta e flexível, capaz de encontrar soluções eficientes em um espaço de busca complexo, confirmando sua aplicabilidade em diversos cenários de otimização.

Este trabalho demonstra não apenas a aplicação prática de AGs, mas também a importância de frameworks como o Gymnasium para a pesquisa e desenvolvimento em inteligência artificial, fornecendo uma base sólida para futuras investigações e melhorias.