ENGENHARIA

Machine Learning

(6° semestre)

PROJETO DE MACHINE LEARNING

Gustavo Morin de Sousa

Fabio Monteiro

Yuri Tabacof

Prof. Fábio José Ayres

São Paulo

26/11/2024

1. OBJETIVO

O objetivo do projeto foi desenvolver e aplicar técnicas de *Reinforcement Learning* para resolver o desafio de pousar uma nave espacial na Lua de maneira eficiente. Para isso, criamos um ambiente simulado utilizando a biblioteca Pygame, no qual uma nave espacial deve ser controlada para pousar em uma área demarcada entre duas bandeiras, quais representam a zona de pouso ideal.

Esse objetivo incluiu a construção de um ambiente interativo no estilo de um jogo, integrando as leis da física simplificadas e restrições reais, como combustível limitado e gravidade constante. Para que o pouso seja bem-sucedido, a nave precisa estar dentro da zona de pouso demarcada e atingir o solo com uma velocidade vertical baixa, exigindo precisão e controle cuidadoso do movimento. O sistema foi projetado para que o agente pudesse aprender autonomamente as melhores ações a serem realizadas com base em um modelo de recompensas e penalidades, utilizando o algoritmo PPO (Proximal Policy Optimization) para treinar o agente.

O foco esteve em explorar como o *Reinforcement Learning* pode ser aplicado em cenários de otimização contínua e tomada de decisão em tempo real. O ambiente também foi expandido para permitir interatividade, permitindo que usuários testem o sistema manualmente, comparando suas ações com as decisões tomadas pelo modelo treinado.

2. CÓDIGO

```
import gymnasium as gym
from gymnasium import spaces
import numpy as np
import pygame
import random
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv
```

Figura [1] - Imports realizados

2.1 Bibliotecas utilizadas

- **Gymnasium**: O Gymnasium é uma biblioteca que fornece uma interface padronizada para criar e interagir com ambientes de simulação. Em um contexto de aprendizado por reforço, esses ambientes permitem que um agente aprenda interagindo por meio de ações, observações e recompensas.
- **Stable-Baselines3 (PPO)**: É um algoritmo de aprendizado por reforço baseado em políticas, projetado para ser estável e robusto. Ele utiliza uma abordagem de otimização proximal, limitando grandes atualizações na política do agente para evitar oscilações

instáveis durante o treinamento. Essa técnica é especialmente útil em ambientes com dinâmica complexa.

- **NumPy :** NumPy é uma biblioteca poderosa para manipulação de arrays e cálculos matemáticos. No contexto de aprendizado por reforço, é usada para processar e armazenar dados, como estados e recompensas.
- **Pygame:** PyGame permite a criação de interfaces gráficas, tornando a simulação mais visual e interativa. Ele pode ser usado para renderizar o ambiente e observar em tempo real como o agente toma decisões.

2.2 Explicação do Código

- Este código é uma base para treinar agentes em ambientes customizados. O PPO será usado para:

1. Explorar o ambiente e aprender padrões de comportamento.
2. Ajustar sua política iterativamente com base nas recompensas obtidas.
3. Garantir que as mudanças na política sejam estáveis e otimizadas.

A integração com PyGame permite visualizar a simulação, enquanto NumPy auxilia no processamento de dados, e o Stable-Baselines3 executa o treinamento, tornando assim possível com que a máquina entenda o funcionamento da e os padrões e assim consiga jogar o jogo individualmente .

```python
MAX_VEL_X = 5.0
MAX_VEL_Y = 5.0

class LunarLandingEnv(gym.Env):
    metadata = {'render_modes': ['human'], 'render_fps': 60}

    def __init__(self):
        super(LunarLandingEnv, self).__init__()

        # Define action and observation space
        # Actions: 0 - Do nothing, 1 - Thrust Up, 2 - Thrust Left, 3 - Thrust Right
        self.action_space = spaces.Discrete(4)

        # Observations: x, y, velocity_x, velocity_y, fuel, landing_center_norm
        low = np.array([-1, -1, -1, -1, 0, -1], dtype=np.float32)
        high = np.array([1, 1, 1, 1, 1, 1], dtype=np.float32)
        self.observation_space = spaces.Box(low, high, dtype=np.float32)

        # Game parameters
        self.gravity = 0.05
        self.thrust_vertical = 0.1
        self.thrust_horizontal = 0.05
        self.max_fuel = 1000  # Increased fuel

        # Pygame initialization
        self.screen_width = 800
        self.screen_height = 600
        self.screen = None
        self.clock = None
        self.is_open = True
```

Figura [2] - Código de Criação do Jogo

```
        # Game outcome
        self.last_outcome = None  # Will be 'victory' or 'defeat' when the game ends

        self.reset()

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)
        self.x = self.screen_width / 2  # Start at the center
        self.y = 50
        self.vel_x = 0.0
        self.vel_y = 0.0
        self.fuel = self.max_fuel
        self.landing_zone = self._choose_landing_zone()
        self.current_step = 0  # Step counter
        self.max_steps = 1000  # Max steps per episode
        self.last_outcome = None  # Reset outcome
        return self._get_obs(), {}

    def step(self, action):
        # Apply action
        if action == 1 and self.fuel > 0:  # Thrust Up
            self.vel_y -= self.thrust_vertical
            self.fuel -= 1
        elif action == 2 and self.fuel > 0:  # Thrust Left
            self.vel_x -= self.thrust_horizontal
            self.fuel -= 1
        elif action == 3 and self.fuel > 0:  # Thrust Right
            self.vel_x += self.thrust_horizontal
            self.fuel -= 1

        # Update physics
        self.vel_y += self.gravity
        self.y += self.vel_y
        self.x += self.vel_x
```

Figura [3] - Código de Criação do Jogo

```
        # Impose horizontal limits
        self.x = max(20, min(self.x, 780))

        # Increment step counter
        self.current_step += 1

        # Initialize reward and constants
        reward = 0.0
        k_y = 5  # Weight for vertical velocity adjustment
        k_x = 4   # Weight for horizontal velocity adjustment
        k_dir = 3  # Weight to penalize wrong direction
        k_dist = 1  # Weight for distance from center

        # Calculate landing zone center
        landing_center = (self.landing_zone[0] + self.landing_zone[1]) / 2
        distance_to_center_x = self.x - landing_center

        # Reward for low and positive Y velocity
        if 0.5 <= self.vel_y <= 1:
            reward += k_y  # Max reward for ideal velocity
        else:
            reward -= k_y * abs(self.vel_y - 0.75)  # Penalty proportional to deviation from 0.75

        if self.x < landing_center:  # Lander is to the left
            if self.vel_x < 0.4:
                if action == 3:  # Pushing to the right
                    reward += k_x
                elif action == 2:  # Pushing to the left
                    reward -= k_dir
```

Figura [4] - Código de Criação do Jogo

```
        elif self.x > landing_center:  # Lander is to the right
            if self.vel_x > -0.4:
                if action == 2:  # Pushing to the left
                    reward += k_x
                elif action == 3:  # Pushing to the right
                    reward -= k_dir

        # Normalize reward between -10 and 10
        reward = max(-10, min(10, reward))

        # Check for termination
        terminated = False
        if self.y >= 530:  # Lander has reached the platform height
            self.y = 530  # Adjust position to exactly on platform
            # Determine if landing is successful
            landing_velocity_threshold = 0.8  # units
            if self.x > self.landing_zone[0] and self.x < self.landing_zone[1] and abs(self.vel_y) <= landing_velocity_threshold:
                # Successful landing
                self.last_outcome = 'victory'
            else:
                # Unsuccessful landing
                self.last_outcome = 'defeat'
            self.vel_y = 0  # Stop vertical movement
            self.vel_x = 0  # Stop horizontal movement

            terminated = True

        elif self.current_step >= self.max_steps:
            terminated = True
            self.last_outcome = 'defeat'
        elif self.y > self.screen_height or self.x <= 20 or self.x >= 780:
            terminated = True
            reward = -10  # Max penalty for going out of bounds
            self.last_outcome = 'defeat'
```

Figura [5] - Código de Criação do Jogo

```
        # Return observations, reward, and episode info
        return self._get_obs(), reward, terminated, False, {}

    def render(self):
        if self.screen is None:
            pygame.init()
            pygame.display.init()
            self.screen = pygame.display.set_mode(
                (self.screen_width, self.screen_height)
            )
        if self.clock is None:
            self.clock = pygame.time.Clock()

        if self.last_outcome is None:
            # Normal game rendering
            self.screen.fill((0, 0, 0))  # Clear screen

            # Draw game elements
            self._draw_stars()
            self._draw_terrain()
            self._draw_lander()

            # Show game state information
            landing_center = (self.landing_zone[0] + self.landing_zone[1]) / 2
            distance_to_center = self.x - landing_center
```

Figura [6] - Código de Criação do Jogo

```
            # Display rocket coordinates and state
            font = pygame.font.Font(None, 36)
            info_text = [
                f"Rocket X: {self.x:.2f}",
                f"Rocket Y: {self.y:.2f}",
                f"Distance to Center: {distance_to_center:.2f}",
                f"Velocity X: {self.vel_x:.2f}",
                f"Velocity Y: {self.vel_y:.2f}",
                f"Fuel: {self.fuel}",
                f"Landing Center X: {landing_center:.2f}",
            ]
            for i, text in enumerate(info_text):
                surface = font.render(text, True, (255, 255, 255))
                self.screen.blit(surface, (10, 10 + i * 30))

            # Visual marker for landing platform center
            pygame.draw.circle(
                self.screen, (0, 255, 0), (int(landing_center), 550), 5
            )

            # Display rocket's current coordinates next to it
            coord_text = f"({int(self.x)}, {int(self.y)})"
            coord_surface = font.render(coord_text, True, (255, 255, 255))
            self.screen.blit(coord_surface, (int(self.x) + 20, int(self.y) - 40))

            # Update display
            pygame.display.flip()
            self.clock.tick(self.metadata["render_fps"])
        else:
            # Draw victory or defeat screen
            self._draw_end_screen()
```

Figura [7] - Código de Criação do Jogo

```
    def _draw_end_screen(self):
        # Clear screen
        self.screen.fill((0, 0, 0))

        # Victory or defeat text
        font = pygame.font.Font(None, 74)
        if self.last_outcome == 'victory':
            text = font.render("YOU WIN!", True, (0, 255, 0))
        else:
            text = font.render("GAME OVER", True, (255, 0, 0))
        text_rect = text.get_rect(center=(self.screen_width // 2, self.screen_height // 2))
        self.screen.blit(text, text_rect)

        # Update display
        pygame.display.flip()

    def close(self):
        if self.screen is not None:
            pygame.display.quit()
            pygame.quit()
            self.is_open = False

    # Helper methods
    def _choose_landing_zone(self):
        start = random.randint(100, 600)
        width = random.randint(50, 150)
        end = start + width
        return (start, end)
```

Figura [8] - Código de Criação do Jogo

```
        )  # Left wing
        pygame.draw.polygon(
            self.screen,
            (150, 150, 150),
            [(x + 10, y + 20), (x + 20, y + 30), (x + 10, y + 30)],
        )  # Right wing
        pygame.draw.polygon(
            self.screen,
            (255, 0, 0),
            [(x, y - 40), (x - 10, y - 30), (x + 10, y - 30)],
        )  # Top cone

        # Draw fuel level indicator
        fuel_percentage = self.fuel / self.max_fuel
        fuel_bar_height = 50
        fuel_bar_width = 10
        pygame.draw.rect(
            self.screen,
            (255, 255, 255),
            (x - 25, y - 30, fuel_bar_width, fuel_bar_height),
            1,
        )  # Fuel bar border
        pygame.draw.rect(
            self.screen,
            (0, 255, 0),
            (
                x - 25,
                y - 30 + fuel_bar_height * (1 - fuel_percentage),
                fuel_bar_width,
                fuel_bar_height * fuel_percentage,
            ),
        )  # Fuel bar
```

Figura [9] - Código de Criação do Jogo

```
        # Draw landing zone
        start, end = self.landing_zone
        pygame.draw.rect(
            self.screen, (150, 150, 150), (start, 550, end - start, 50)
        )
        # Flags
        pygame.draw.line(self.screen, (255, 255, 255), (start, 550), (start, 520), 2)
        pygame.draw.polygon(
            self.screen,
            (255, 255, 0),
            [(start, 520), (start + 15, 527), (start, 534)],
        )
        pygame.draw.line(self.screen, (255, 255, 255), (end, 550), (end, 520), 2)
        pygame.draw.polygon(
            self.screen,
            (255, 255, 0),
            [(end, 520), (end - 15, 527), (end, 534)],
        )

    def _draw_lander(self):
        x = int(self.x)
        y = int(self.y)

        # Draw the lander
        pygame.draw.rect(
            self.screen, (200, 200, 200), (x - 10, y - 30, 20, 60)
        )  # Main body
        pygame.draw.polygon(
            self.screen,
            (150, 150, 150),
            [(x - 10, y + 20), (x - 20, y + 30), (x - 10, y + 30)],
        )  # Left wing
```

Figura [10] - Código de Criação do Jogo

```python
    def _get_obs(self):
        x_norm = (self.x - self.screen_width / 2) / (self.screen_width / 2)
        y_norm = (self.y - self.screen_height / 2) / (self.screen_height / 2)
        vel_x_norm = self.vel_x / MAX_VEL_X
        vel_y_norm = self.vel_y / MAX_VEL_Y
        fuel_norm = self.fuel / self.max_fuel
        landing_center_norm = (self.landing_zone[0] + self.landing_zone[1]) / 2 / self.screen_width
        return np.array(
            [x_norm, y_norm, vel_x_norm, vel_y_norm, fuel_norm, landing_center_norm],
            dtype=np.float32,
        )

    def _draw_stars(self):
        # Draw static stars
        if not hasattr(self, "stars"):
            self.stars = [
                (random.randint(0, 800), random.randint(0, 600)) for _ in range(50)
            ]
        for star in self.stars:
            pygame.draw.circle(self.screen, (255, 255, 255), star, 2)

    def _draw_terrain(self):
        # Draw terrain and landing zone
        pygame.draw.polygon(
            self.screen,
            (100, 100, 100),
            [
                (0, 550),
                (200, 450),
                (400, 500),
                (600, 450),
                (800, 550),
                (800, 600),
                (0, 600),
            ],
```

Figura [11] - Código de Criação do Jogo

Nesta parte do código é a primeira parte de um projeto que cria um jogo baseado em um simulador de pouso lunar utilizando Python, ou seja, trata-se de um jogo que ao iniciado um foguete começa a cair sobre a tela e o objetivo é fazer com que ele pouse entre as bandeiras. Vale lembrar que para isso o código emprega as bibliotecas Gymnasium e Pygame para estruturar o ambiente e visualizar a interação do jogador com o cenário, como já dito anteriormente. A classe `LunarLandingEnv` herda de `gym.Env`, a estrutura base para criar ambientes personalizados no Gymnasium. Este ambiente simula o controle de um módulo lunar em direção a uma zona de pouso.

O espaço de ação, definido como `spaces.Discrete(4)`, permite ao agente ou jogador escolher entre quatro ações: não realizar nenhuma ação, ativar o propulsor vertical para subir (apertar a seta para cima), ativar o propulsor esquerdo para mover-se para a esquerda (apertar a seta para esquerda) ou ativar o propulsor direito para mover-se para a direita (apertar seta para direita). Já o espaço de observação é configurado com `spaces.Box` e inclui variáveis como as coordenadas `x` e `y` do módulo lunar, as velocidades horizontal e vertical, o nível de

combustível restante e a distância normalizada até o centro da zona de pouso. Essas informações ajudam o agente a compreender o estado atual do jogo para tomar decisões.

Os parâmetros do ambiente modelam a física do jogo, incluindo a gravidade, a força dos propulsores e o limite máximo de combustível disponível. O método `reset` é chamado para reiniciar o ambiente a cada novo episódio, posicionando o módulo lunar no centro da tela, zerando as velocidades, configurando o combustível para o valor máximo e escolhendo aleatoriamente a zona de pouso. Este método retorna uma observação inicial que descreve o estado do jogo.

O método `step` processa as ações realizadas pelo jogador ou agente, reduzindo o combustível, alterando as velocidades conforme o propulsor acionado e aplicando a gravidade para atualizar a posição do módulo. Ele também calcula recompensas e penalidades com base nas ações realizadas, atribuindo pontos positivos para movimentos que levam o módulo ao centro da zona de pouso e penalizando desvios do objetivo. Este método também verifica condições para finalizar o episódio, como pouso bem-sucedido, ultrapassagem dos limites do mapa ou esgotamento do número máximo de passos.

O método `render`, utilizando Pygame, é responsável por desenhar graficamente o estado do jogo. Ele representa o módulo lunar e o terreno por meio de formas geométricas, exibe informações como coordenadas, combustível restante e velocidade, e destaca eventos de vitória ou derrota com mensagens na interface. Além disso, o método `_choose_landing_zone` cria variabilidade entre os episódios, definindo aleatoriamente a posição da zona de pouso.

Este código estabelece a estrutura básica do jogo, incluindo os elementos de física, lógica e renderização. A integração com Pygame torna o ambiente visual e interativo, permitindo observar o comportamento do agente ou jogador em tempo real. Agora com o código do jogo já criado, torna-se possível o realizar o treinamento do bot para que ele jogue individualmente da maneira correta.

```python
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv

# Callback para registrar recompensas
from stable_baselines3.common.callbacks import BaseCallback
import numpy as np

class RewardLoggerCallback(BaseCallback):
    def __init__(self, verbose=0):
        super(RewardLoggerCallback, self).__init__(verbose)

    def _on_step(self) -> bool:
        if len(self.locals["infos"]) > 0 and "episode" in self.locals["infos"][0]:
            episode_rewards = [info["episode"]["r"] for info in self.locals["infos"] if "episode" in info]
            if episode_rewards:
                mean_reward = np.mean(episode_rewards)
                self.logger.record("reward/mean", mean_reward)
        return True

from stable_baselines3.common.vec_env import SubprocVecEnv
```

Figura [12] - Código de Treinamento do Bot

```python
env = SubprocVecEnv([lambda: LunarLandingEnv() for _ in range(4)])
# Criar o modelo
model = PPO(
    "MlpPolicy",
    env,
    verbose=1,
    tensorboard_log="./ppo_lunar_landing_tensorboard/",
    learning_rate=3e-4,
    n_steps=2048,
    batch_size=64,
    gae_lambda=0.95,
    gamma=0.99,
    ent_coef=0.0,
    clip_range=0.2,
)
# Treinar o agente
model.learn(total_timesteps=500_000, callback=RewardLoggerCallback())
# Salvar o modelo
model.save("ppo_lunar_landing")
```

Figura [13] - Código de Treinamento do Bot

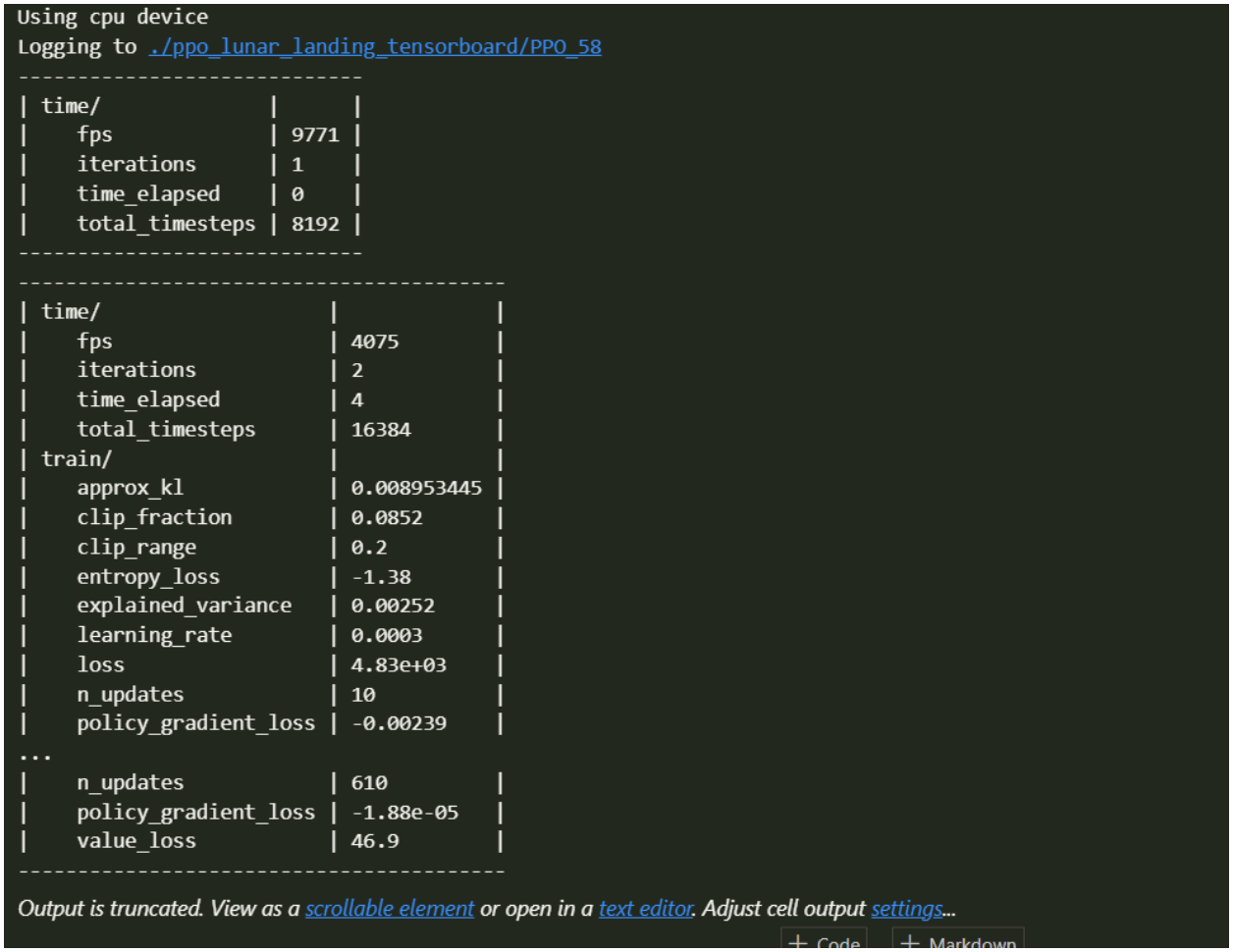


Figura [14] - Resposta do Código de Treinamento do Bot

Na segunda parte do projeto, o algoritmo Proximal Policy Optimization (PPO), da biblioteca Stable-Baselines3, é integrado ao ambiente `LunarLandingEnv` para treinar o agente no objetivo de pousar o módulo lunar com eficiência. Essa etapa envolve a configuração do modelo de aprendizado por reforço, a implementação de um callback para monitorar as recompensas durante o treinamento e a definição de ambientes paralelos para otimizar a coleta de dados.

O código começa com a criação de múltiplas instâncias do ambiente usando o `SubprocVecEnv`, que executa quatro cópias paralelas do ambiente. Essa abordagem permite que o agente colete mais dados em menos tempo, tornando o treinamento mais eficiente e permitindo maior exploração do espaço de estados.

Em seguida, o modelo PPO é configurado com a definição de importantes hiperparâmetros que têm um impacto direto na eficiência e na estabilidade do treinamento. A política utilizada é definida como `MlpPolicy`, que utiliza uma rede neural perceptron multicamada para processar

as entradas do ambiente e determinar as ações do agente. Outros hiperparâmetros configurados incluem:

- **`learning_rate`** (3e-4): A taxa de aprendizado controla a velocidade com que os pesos da rede neural são ajustados durante a otimização. Uma taxa muito alta pode causar instabilidade no treinamento, enquanto uma taxa muito baixa pode tornar o processo lento ou até impedir a convergência.
- **`n_steps`** (2048): Representa o número de interações que o agente coleta antes de atualizar a política. Valores maiores permitem maior estabilidade, mas aumentam o tempo de processamento, enquanto valores menores favorecem atualizações mais frequentes, mas podem resultar em aprendizado ruidoso.
- **`batch_size`** (64): Define o tamanho dos lotes usados para calcular os gradientes. Um tamanho de lote adequado garante que as atualizações sejam baseadas em informações representativas dos dados coletados.
- **`gamma`** (0.99): O fator de desconto que controla a importância das recompensas futuras. Valores próximos de 1 favorecem estratégias de longo prazo, enquanto valores menores priorizam recompensas imediatas.
- **`gae_lambda`** (0.95): Parâmetro que ajusta o cálculo da vantagem generalizada estimada (GAE). Valores menores tornam o GAE mais próximo das recompensas imediatas, enquanto valores maiores consideram mais informações temporais.
- **`clip_range`** (0.2): Define o limite de clipping para restringir grandes mudanças nos gradientes. Esse parâmetro é crucial para a estabilidade do PPO, pois impede que o modelo sofra alterações excessivas na política em cada atualização.

Esses hiperparâmetros são fundamentais para equilibrar a exploração e a exploração do agente durante o treinamento, garantir estabilidade nas atualizações e acelerar a convergência para uma política ótima. Pequenas mudanças nesses valores podem alterar drasticamente o desempenho do agente, tornando sua configuração uma etapa essencial do processo.

A implementação do callback, através da classe `RewardLoggerCallback`, permite monitorar e registrar as recompensas médias acumuladas durante o treinamento. A cada passo, o callback verifica se há informações sobre episódios finalizados, calcula a média das recompensas acumuladas e registra esses valores no TensorBoard. Isso facilita o acompanhamento do desempenho do agente e possibilita ajustes nos hiperparâmetros caso os resultados não estejam satisfatórios.

O treinamento do modelo é realizado por meio do método `learn`, configurado para executar 500.000 passos de interação com o ambiente. Durante o treinamento, o callback é utilizado para registrar métricas como recompensas médias, que permitem monitorar o progresso do agente em tempo real. Após o término do treinamento, o modelo é salvo em um arquivo chamado `ppo_lunar_landing`, permitindo reutilizá-lo posteriormente para testes ou ajustes adicionais.

O monitoramento do treinamento com o TensorBoard exibe métricas importantes, como a frequência de quadros por segundo (FPS), recompensas médias, perda de entropia e a fração de gradientes cortados. Essas métricas ajudam a identificar possíveis problemas no treinamento, como taxas de aprendizado inadequadas, tamanho de passos insuficiente ou variação excessiva nos gradientes. A análise dessas métricas também pode guiar ajustes finos nos hiperparâmetros para melhorar a performance do agente.

Em resumo, esta etapa conecta o ambiente ao algoritmo PPO, criando um agente que aprende a maximizar as recompensas ao pousar o módulo lunar. A utilização de ambientes paralelos, a configuração detalhada do modelo, o monitoramento via callback e o uso do TensorBoard são cruciais para garantir eficiência e clareza durante o processo de treinamento. A escolha e o ajuste dos hiperparâmetros desempenham um papel central no sucesso do treinamento, afetando diretamente a velocidade de aprendizado, a estabilidade da política e o desempenho final do agente. O próximo passo seria testar o modelo treinado no ambiente para avaliar sua performance.

```python
# Load the model
model = PPO.load("ppo_lunar_landing")

# Create a new environment for testing
env = LunarLandingEnv()

# Run the agent
obs, _ = env.reset()
done = False
while not done:
    action, _ = model.predict(obs)
    obs, reward, done, _, _ = env.step(action)
    env.render()
env.close()
```

Figura [15] - Código de visualização do Treinamento do Bot

Esta parte do código é responsável por testar o modelo treinado em um ambiente simulado, permitindo que o agente execute as ações aprendidas de forma visual. O objetivo é observar, em tempo real, como o agente utiliza a política aprendida para tomar decisões no jogo. Inicialmente, o modelo treinado é carregado a partir do arquivo salvo, contendo os pesos da rede neural e as configurações otimizadas durante o treinamento. Em seguida, um novo ambiente é instalado, configurando as mesmas condições utilizadas no treinamento.

O ambiente é reiniciado utilizando o método `reset`, que retorna a observação inicial representando o estado do jogo, enquanto a variável `done` é inicializada como `False` para indicar que o episódio ainda está em andamento. Dentro do loop principal, o modelo prevê a

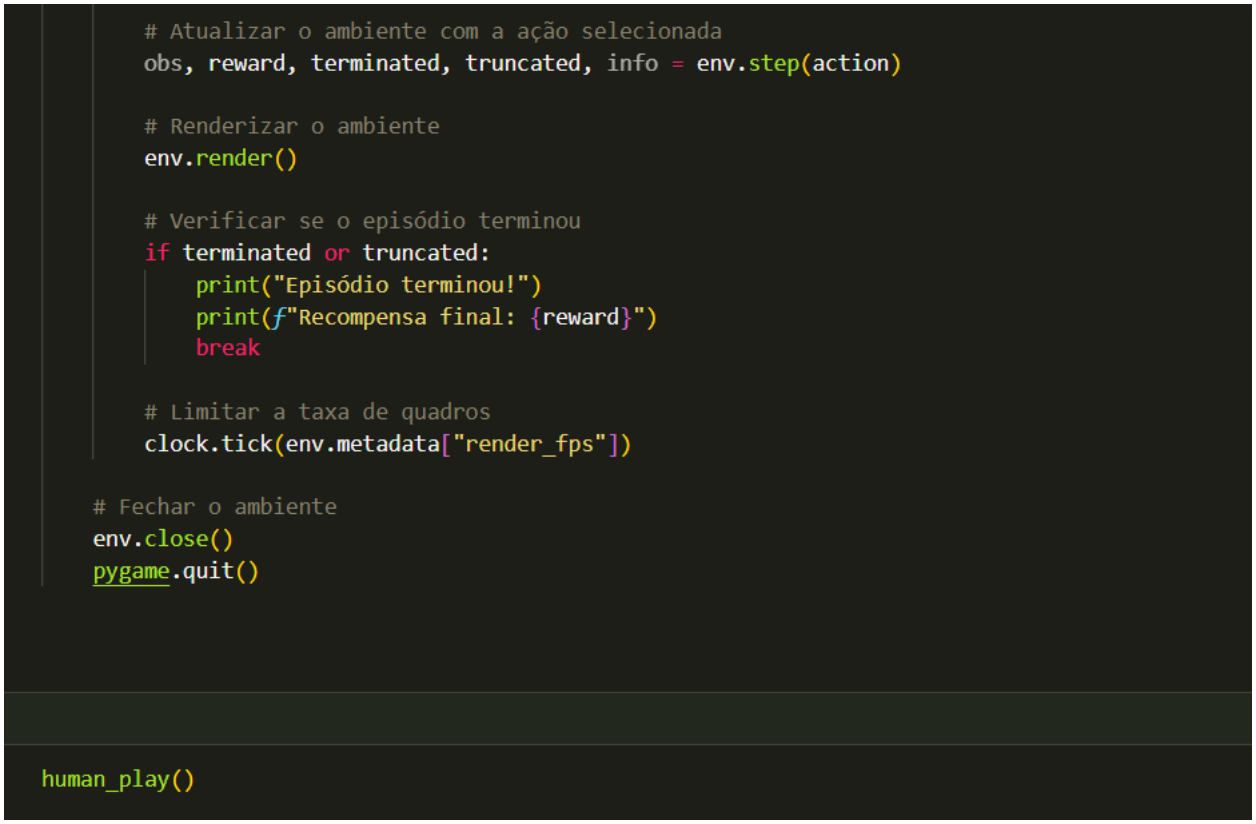
próxima ação com base na observação atual, utilizando o método `predict`. Essa ação é aplicada no ambiente por meio do método `step`, que atualiza o estado do ambiente e retorna a nova observação, a recompensa obtida pela ação, e um indicador de término do episódio. O método `render` é utilizado em cada passo para gerar uma visualização gráfica do estado atual do jogo, permitindo acompanhar como o agente ajusta suas estratégias para pousar o módulo lunar.

O loop continua até que a variável `done` indique o fim do episódio, seja por um pouso bem-sucedido, falha ou esgotamento dos passos permitidos. Após isso, o ambiente é fechado para liberar os recursos alocados. Esse processo permite validar o desempenho do modelo treinado de forma interativa, facilitando a análise do comportamento do agente e sua capacidade de aplicar o que foi aprendido durante o treinamento.

Essa abordagem é fundamental para identificar se o modelo aprendeu a alcançar o objetivo desejado, como ajustar o uso dos propulsores para pousar o módulo lunar com precisão. Além disso, a visualização do jogo permite detectar possíveis falhas ou comportamentos inesperados, auxiliando na necessidade de ajustes no treinamento ou nos hiperparâmetros. Por fim, essa funcionalidade também serve como uma ferramenta prática para demonstrar os resultados do aprendizado por reforço em um cenário interativo e visualmente compreensível.

```python
def human_play():
    import pygame
    from pygame.locals import K_UP, K_LEFT, K_RIGHT, K_ESCAPE, QUIT

    env = LunarLandingEnv()
    env.reset()

    # Inicializar Pygame
    pygame.init()
    screen = pygame.display.set_mode((env.screen_width, env.screen_height))
    pygame.display.set_caption("Lunar Landing - Human Play")
    clock = pygame.time.Clock()

    done = False
    action = 0  # Ação inicial: Fazer nada

    while not done:
        for event in pygame.event.get():
            if event.type == QUIT:
                done = True
            elif event.type == pygame.KEYDOWN:
                if event.key == K_ESCAPE:  # Sair do jogo
                    done = True
                elif event.key == K_UP:  # Empuxo para cima
                    action = 1
                elif event.key == K_LEFT:  # Empuxo para esquerda
                    action = 2
                elif event.key == K_RIGHT:  # Empuxo para direita
                    action = 3
            elif event.type == pygame.KEYUP:
                if event.key in [K_UP, K_LEFT, K_RIGHT]:  # Parar de aplicar força
                    action = 0
```

Figura [16] - Código que dá possibilidade do Humano Jogar



```
        # Atualizar o ambiente com a ação selecionada
        obs, reward, terminated, truncated, info = env.step(action)

        # Renderizar o ambiente
        env.render()

        # Verificar se o episódio terminou
        if terminated or truncated:
            print("Episódio terminou!")
            print(f"Recompensa final: {reward}")
            break

        # Limitar a taxa de quadros
        clock.tick(env.metadata["render_fps"])

    # Fechar o ambiente
    env.close()
    pygame.quit()


human_play()
```

Figura [17] - Código que dá possibilidade do Humano Jogar

Este trecho de código implementa a funcionalidade de permitir que um jogador humano controle o módulo lunar diretamente no ambiente do jogo, substituindo o agente treinado. Essa interação é feita através de entradas de teclado capturadas pela biblioteca **Pygame**, proporcionando uma experiência interativa em tempo real.

A função `human_play` inicia instanciando o ambiente `LunarLandingEnv` e reiniciando-o com o método `reset`, definindo o estado inicial do jogo. A biblioteca Pygame é inicializada, configurando uma janela para renderizar o ambiente com o título "Lunar Landing - Human Play". Um objeto `clock` é criado para controlar a taxa de quadros por segundo, garantindo uma execução suave do jogo. A variável `action` é inicialmente definida como `0`, representando a ação de "fazer nada", e a variável `done` é configurada como `False` para indicar que o jogo ainda não terminou.

Dentro de um loop principal, os eventos do teclado são capturados com `pygame.event.get()`. Se o jogador pressionar a tecla **ESC** ou fechar a janela, o loop é encerrado e o jogo termina. Caso o jogador pressione as teclas direcionais, as ações correspondentes são atribuídas:

- **Seta para cima (`K_UP`)**: Ativa o propulsor vertical, ação 1.
- **Seta para esquerda (`K_LEFT`)**: Ativa o propulsor para mover o módulo à esquerda, ação 2.
- **Seta para direita (`K_RIGHT`)**: Ativa o propulsor para mover o módulo à direita, ação 3. Quando as teclas são liberadas, a ação retorna para `0`, indicando que nenhuma força será aplicada.

Em cada iteração do loop, a ação selecionada é aplicada ao ambiente com o método `step`, que atualiza o estado do jogo e retorna a nova observação, a recompensa obtida pela ação, e indicadores que verificam se o episódio foi encerrado. O ambiente é renderizado visualmente com `env.render()`, permitindo que o jogador veja o estado atual do módulo lunar, incluindo sua posição e velocidade.

Caso o episódio termine, seja por pouso bem-sucedido, falha ou esgotamento do número de passos, uma mensagem é exibida no console indicando o término do episódio e a recompensa final acumulada. O loop é então encerrado. A cada iteração, a taxa de quadros é limitada pelo método `clock.tick`, utilizando o valor de quadros por segundo definido no metadado `render_fps` do ambiente. Por fim, quando o jogador encerra o jogo, os recursos do ambiente e da biblioteca Pygame são liberados com os métodos `env.close()` e `pygame.quit()`.

Esta funcionalidade oferece aos usuários a oportunidade de experimentar o controle do módulo lunar manualmente, permitindo comparar seu desempenho com o do agente treinado. É uma forma prática de validar o comportamento do ambiente e de criar uma experiência mais interativa para os usuários.

3. CONCLUSÃO

Este projeto destacou a eficácia do uso do algoritmo PPO para treinar um agente em um ambiente de simulação baseado em recompensas e penalidades. O sistema de recompensas foi essencial para guiar o agente a aprender o comportamento ideal, penalizando ações que o afastavam do objetivo e premiando movimentos que favoreciam um pouso seguro e preciso na zona demarcada. A escolha e o ajuste dos hiperparâmetros do PPO, como a taxa de aprendizado, o fator de desconto (gamma), e o limite de clipping (clip_range), desempenharam um papel central no sucesso do treinamento, garantindo estabilidade no aprendizado e permitindo que o modelo convergisse para uma política eficiente.

Os objetivos do projeto foram plenamente atendidos, com a criação de um agente capaz de aprender autonomamente a resolver o problema de pouso lunar e a demonstração de como um sistema bem estruturado de recompensas aliado a configurações otimizadas pode resolver problemas complexos em ambientes simulados. Além disso, a integração de um sistema interativo permitiu comparar as decisões do modelo treinado com o controle manual, validando os resultados obtidos.

Todo o código, juntamente com um vídeo de demonstração, está disponível no GitHub: https://github.com/yuritaba/projeto_final/tree/main.