Laboratorio 6 Grupal "Aprendizaje por Refuerzo".

Descripción del Trabajo.

Nombres:

- Aramayo Valdez Joaquin.

- Piza Nava Vladimir.

- Viza Hoyos Maria Belen.

- Mendoza Ovando Carlos Saul.

- Solorzano Diego.

Link GitHub:

1. **Importación de Librerías**

```
import gymnasium as gym
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import DummyVecEnv,
VecNormalize
from stable_baselines3.common.callbacks import EvalCallback
import os
import time
import matplotlib.pyplot as plt
import numpy as np
```

En este primer bloque, importamos las librerías necesarias para el entrenamiento de un modelo de aprendizaje por refuerzo utilizando Stable Baselines3 que es una biblioteca de Python que implementa algoritmos avanzados de **aprendizaje por refuerzo** (RL) como PPO, A2C, DQN, entre otros. Está diseñada para facilitar el entrenamiento de agentes en entornos simulados, proporcionando implementaciones optimizadas y listas para usar.

2. **Configuración del Entorno y del Modelo**

```
# Definir el nombre del entorno y la ruta para guardar el modelo
entrenado
ENV_NAME = "BipedalWalker-v3"
MODEL_PATH = "ppo_bipedalwalker"

# Crear el entorno con la opción de renderizar en modo humano para
observar el entrenamiento en tiempo real
env = DummyVecEnv([lambda: gym.make(ENV_NAME,
render_mode="human")])  # Vectorizar y renderizar
```

```
env = VecNormalize(env, norm_obs=True, norm_reward=True)  # Normalizar observaciones y recompensas
```

Aquí definimos el entorno **BipedalWalker-v3** de OpenAI Gym, que simula un agente que debe aprender a caminar en un terreno complejo. Se establece la ruta para guardar el modelo entrenado como **ppo_bipedalwalker**. Luego, el entorno se vectoriza utilizando **DummyVecEnv** para manejarlo de manera eficiente, incluso si se usaran múltiples instancias, y se habilita la visualización en tiempo real mediante el parámetro **render_mode="human"**. Finalmente, se normalizan las observaciones y recompensas del entorno con **VecNormalize**, lo que ayuda a mejorar la estabilidad y rendimiento del entrenamiento.

3. **Carga del Modelo**

```
model = PPO("MlpPolicy", env, verbose=1, learning_rate=0.0003, n_steps=2048, batch_size=64, n_epochs=10, gamma=0.99)
```

Creamos el modelo de aprendizaje por refuerzo utilizando el algoritmo **PPO (Proximal Policy Optimization)** de la librería **Stable Baselines3**. El modelo se configura con una política **MlpPolicy**, que indica que el agente utilizará una red neuronal de perceptrón multicapa (MLP) para tomar decisiones. Se ajustan varios parámetros importantes: **learning_rate=0.0003** define la tasa de aprendizaje, **n_steps=2048** especifica el número de pasos por actualización de política, **batch_size=64** define el tamaño del lote para cada actualización, **n_epochs=10** establece el número de veces que se actualizará el modelo por paso, y **gamma=0.99** determina el factor de descuento para las recompensas futuras, lo que influye en cómo el agente valora las recompensas a largo plazo.

4. **Configuración de Callbacks para guardar el mejor modelo**

```
eval_env = DummyVecEnv([lambda: gym.make(ENV_NAME)])
eval_env = VecNormalize(eval_env, norm_obs=True, norm_reward=True)
eval_callback = EvalCallback(eval_env, best_model_save_path=MODEL_PATH, log_path='./logs/', eval_freq=5000, deterministic=True)
```

Ahora creamos un entorno de evaluación **eval_env** utilizando **DummyVecEnv**, similar al entorno de entrenamiento, pero sin renderización. A continuación, se normalizan las observaciones y recompensas del entorno de evaluación con **VecNormalize**. Luego, se define un **EvalCallback**, que permite evaluar el rendimiento del modelo durante el entrenamiento. El **EvalCallback** guarda el mejor modelo en el directorio especificado por **best_model_save_path=MODEL_PATH**, guarda los registros en

log_path='./logs/', y realiza evaluaciones cada **5000 pasos**. Además, se establece **deterministic=True** para que las evaluaciones se realicen con acciones deterministas, es decir, sin exploración aleatoria.

5. **Entrenamiento del modelo**

```
total_timesteps = 100000  # Aumentar el número de timesteps para un
entrenamiento más prolongado
rewards = []

print("Entrenando el modelo y registrando recompensas por
episodio...")
obs = env.reset()
episode_reward = 0

for timestep in range(total_timesteps):
    action, _ = model.predict(obs, deterministic=False)
    obs, reward, done, info = env.step(action)
    # Acumular recompensa y renderizar el entorno
    episode_reward += reward[0] if isinstance(reward, np.ndarray)
else reward
    env.render()  # Renderizar el entorno en cada paso para ver la
ventana de pygame
    time.sleep(0.01)  # Controla la velocidad del renderizado

    if done:
        rewards.append(episode_reward)  # Almacenar la
recompensadel episodio
       obs = env.reset()
        episode_reward = 0

    # Ejecutar el aprendizaje y actualizar los parámetros
    model.learn(total_timesteps=1, reset_num_timesteps=False,
callback=eval_callback)

# Guardar el modelo entrenado
model.save(MODEL_PATH)
print("Modelo entrenado y guardado.")
```

Lo que procede es entrenar al modelo, en este caso lo haremos usando un total de **100,000 pasos de tiempo** (timesteps). Durante cada paso, el modelo predice una acción basada en el estado actual del entorno, luego el entorno ejecuta esa acción y retorna el nuevo estado, la recompensa, y si el episodio ha finalizado. Las recompensas obtenidas por episodio se acumulan en **episode_reward** y se almacenan en la lista **rewards** cuando el episodio termina. Además, se renderiza el entorno en cada paso para visualizar el entrenamiento en tiempo real y se

controla la velocidad de renderización con **time.sleep(0.01)**. El modelo se actualiza después de cada paso, y el entrenamiento continúa hasta que se alcanzan los timesteps definidos. Al final, el modelo entrenado se guarda en el directorio especificado por **MODEL_PATH**.}

6. **Evaluacion del modelo**

```
print("Evaluando el modelo...")
obs = env.reset()
total_reward = 0

for _ in range(10000):
    action, _ = model.predict(obs, deterministic=True)
    obs, reward, done, info = env.step(action)
    total_reward += reward[0] if isinstance(reward, np.ndarray)
else reward

    env.render()  # Renderizar el entorno
    time.sleep(0.01)  # Controla la velocidad del renderizado

    if done:
        obs = env.reset()
        break

print(f"Evaluación completada. Recompensa total acumulada:
{float(total_reward):.2f}")
```

Una vez ya entrenado tenemos un modelo ya entrenado, ahora haremos una evaluación de este, durante **10,000 pasos de tiempo**. El modelo predice la acción a tomar en cada paso, y el entorno retorna el nuevo estado, la recompensa y si el episodio ha finalizado. La recompensa acumulada durante el episodio se suma a **total_reward**. Además, se renderiza el entorno en cada paso para visualizar el comportamiento del agente durante la evaluación, controlando la velocidad de renderización con **time.sleep(0.01)**. Si el episodio termina, se reinicia el entorno. Al final de la evaluación, se imprime la recompensa total acumulada del episodio evaluado.

7. **Registro de Recompensas Durante el Entrenamiento**

```
# Asegurarse de que `rewards` contenga solo valores escalares
rewards = [reward.item() if isinstance(reward, np.ndarray) else
reward for reward in rewards]

def plot_rewards(rewards, window=10):
    plt.figure(figsize=(12, 6))
```

```
    # Graficar recompensas acumuladas (sin promedio móvil)
    plt.subplot(1, 2, 1)
    plt.plot(rewards, color='blue')
    plt.title('Recompensas Totales por Episodio')
    plt.xlabel('Episodios')
    plt.ylabel('Recompensa Total')

    # Graficar promedio móvil para suavizar la curva
    rolling_mean = np.convolve(rewards, np.ones(window) / window,
mode='valid')
    plt.subplot(1, 2, 2)
    plt.plot(rolling_mean, color='orange')
    plt.title(f'Recompensas Promedio (window={window})')
    plt.xlabel('Episodios')
    plt.ylabel('Recompensa Promedio')

    plt.tight_layout()
    plt.show()

# Graficar las recompensas registradas durante el entrenamiento
plot_rewards(rewards)
```

Por último, graficamos las recompensas obtenidas durante el entrenamiento. La función que creamos genera dos subgráficas: la primera muestra las **recompensas totales por episodio** y la segunda presenta el **promedio móvil de las recompensas** para suavizar la curva, usando una ventana de tamaño configurable (por defecto 10). Finalmente, se utiliza la función **plot_rewards** para graficar las recompensas acumuladas durante todo el proceso de entrenamiento, proporcionando una visión visual del rendimiento del agente a lo largo del tiempo.

8. Imágenes del entrenamiento y resultados.