Proyecto 2

Borge Chacon Brainer Antonio, Medinila Robles Pedro

I. 3.1 Primera Etapa: DQN

I-A. Atari Space Invaders

En este caso se eligió el juego Space Invaders ya que es un juego con puntajes y movimiento del personaje, el código a explicar es SpaceInvaders_WandB.

Para poder correr todo se necesitan las librerías en la sección III de Anexos.

```
1   import tensorflow as tf
2   from tensorflow import keras
3   import numpy as np
4   from collections import deque
5   import wandb
6
7   from ale_py import ALEInterface
8   from ale_py.roms import SpaceInva
9   import pathlib
10  import gymnasium as gym
11  import os.path
```

Figura 1: Librerías para Space Invaders

A continuación, se explica cada una de ellas:

- ‘tensorflow‘ y ‘keras‘: Estas son bibliotecas populares de aprendizaje automático y redes neuronales. ‘TensorFlow‘ es una plataforma de código abierto para construir y entrenar modelos de aprendizaje automático, mientras que ‘Keras‘ es una API de alto nivel para construir y entrenar modelos de aprendizaje profundo.

- ‘numpy‘: Es una biblioteca para el lenguaje de programación Python que agrega soporte para matrices y funciones matemáticas de alto nivel para operar con ellas. Es ampliamente utilizado en el procesamiento numérico y científico.

- ‘collections‘: Es un módulo de la biblioteca estándar de Python que proporciona tipos de datos adicionales para manejar colecciones de elementos. En este caso, se utiliza la clase ‘deque‘ (double-ended queue) para almacenar y manipular una cola de elementos de manera eficiente.

- ‘wandb‘: Es una biblioteca de registro y seguimiento de experimentos. Se utiliza para realizar un seguimiento de métricas, hiperparámetros y resultados de experimentos de aprendizaje automático.

- ‘ale_py‘ y ‘ale_py.roms‘: Estas bibliotecas están relacionadas con el entorno del juego utilizado en el proyecto. ‘ale_py‘ proporciona una interfaz para interactuar con el emulador de juegos Atari Learning Environment (ALE), mientras que ‘ale_py.roms‘ contiene definiciones de juegos específicos, como el juego Space Invaders.

- ‘pathlib‘: Es una biblioteca para manipular rutas de archivos y directorios de manera eficiente. En este caso, se utiliza para trabajar con rutas de archivos en el sistema de archivos.

- ‘gymnasium‘: Es una biblioteca de código abierto que proporciona una colección de entornos de aprendizaje automático. Se utiliza para cargar el entorno del juego y permitir la interacción con él.

- ‘os.path‘: Es un módulo de la biblioteca estándar de Python que proporciona funciones para trabajar con rutas de archivos y directorios. Se utiliza en este código para verificar la existencia de archivos en el sistema de archivos.

Para la siguiente parte se inicializan algunas variables y se configuran los entornos de juego:

```
13  print(tf.__version__)
14  tf.config.list_physical_devices('GPU')
15  print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))
16
17  ale = ALEInterface()
18  ale.loadROM(SpaceInvaders)
19
20  env = gym.make('ALE/SpaceInvaders-v5')
21
22  n_inputs = env.observation_space.shape[0]
23  n_outputs = env.action_space.n
```

Figura 2: Inicialización del entorno de Space Invaders

Las primeras líneas de la Figura 2 lista los dispositivos físicos disponibles en el sistema para ejecutar cálculos en la GPU y muestra la cantidad de GPUs disponibles, esto requiere más software y en este caso se implementó como prueba.

Luego se crea una instancia de la clase ‘ALEInterface‘ llamada ‘ale‘, esta clase proporciona una interfaz para interactuar con el emulador de juegos Atari Learning Environment (ALE).

La siguiente línea carga el juego Space Invaders en el emulador de juegos utilizando el método ‘loadROM()‘ de ‘ale‘. ‘SpaceInvaders‘ es una clase proporcionada por ‘ale_py.roms‘ que contiene la definición específica del juego y se crea un entorno de juego utilizando ‘gym.make()‘. El argumento ‘’ALE/SpaceInvaders-v5’‘ especifica el entorno del juego Space Invaders en la versión 5 de ALE.

Para ‘n_inputs‘ se establece como la dimensión de la forma del espacio de observación del entorno de juegom, representa la cantidad de entradas que el modelo recibirá para realizar predicciones y ‘n_outputs‘ se establece como la cantidad de acciones posibles en el entorno de juego, representa la cantidad de salidas posibles que el modelo puede generar como acciones.

```
25  main_nn = keras.Sequential([
26      keras.layers.Conv2D(32, (8, 8), strides=4, activation='relu', input_shape=(210, 160, 3)),
27      keras.layers.Conv2D(64, (4, 4), strides=2, activation='relu'),
28      keras.layers.Conv2D(64, (3, 3), strides=1, activation='relu'),
29      keras.layers.Flatten(),
30      keras.layers.Dense(512, activation='relu'),
31      keras.layers.Dense(n_outputs)
32  ])
33
34  target_nn = keras.models.clone_model(main_nn)
35
36  optimizer = keras.optimizers.Adam(lr=0.01)
37  loss_fn = keras.losses.mean_squared_error
38
39  replay_buffer = deque(maxlen=10000)
```

Figura 3: Modelo de entrenamiento para Space Invaders

Con respecto a la Figura 3 en este fragmento de código se definen y se inicializan los componentes principales del modelo de aprendizaje automático:

El modelo ‘main_nn‘ es el que se utilizará para tomar decisiones basadas en las observaciones del entorno de juego, es una secuencia de capas (‘Sequential‘) de la biblioteca Keras.

- La primera capa es una capa convolucional (‘Conv2D‘) con 32 filtros de tamaño 8x8 y una función de activación ReLU, recibe una entrada de forma (210, 160, 3), que representa una imagen de color de 210x160 píxeles.
- La segunda capa es otra capa convolucional con 64 filtros de tamaño 4x4 y una función de activación ReLU.
- La tercera capa es una capa convolucional adicional con 64 filtros de tamaño 3x3 y una función de activación ReLU.
- Después de las capas convolucionales, se agrega una capa ‘Flatten‘ para aplanar los datos en un vector.
- Luego, se agrega una capa densa (‘Dense‘) con 512 unidades y una función de activación ReLU.

- Finalmente, se agrega una capa densa sin función de activación para generar las salidas del modelo, que corresponden a las posibles acciones en el entorno de juego.

En el caso de ‘target_nn‘ es una copia del modelo principal ‘main_nn‘, se utiliza para calcular los valores objetivo durante el proceso de entrenamiento.

La variable ‘optimizer‘ se inicializa con el optimizador Adam y una tasa de aprendizaje de 0.01, este se utilizará para actualizar los pesos del modelo durante el entrenamiento.

Como se quiere registrar el error, ‘loss_fn‘ se establece como la función de pérdida de error cuadrático medio (‘mean_squared_error‘). Esta función se utilizará para calcular la pérdida entre las predicciones del modelo y los valores objetivo durante el entrenamiento.

Finalmente, ‘replay_buffer‘ es un deque (cola doble) que se utiliza para almacenar y muestrear la experiencia pasada del agente de aprendizaje reforzado, tiene una longitud máxima de 10000 y se utilizará para implementar la técnica de replay buffer en el algoritmo de aprendizaje reforzado.

```
41  def epsilon_greedy_policy(state, epsilon=0):
42      if np.random.rand() < epsilon:
43          return np.random.randint(n_outputs)
44      else:
45
46          if isinstance(state, tuple) and len(state) == 2 and isinstance(state[0], np.ndarray) and isinstance(state[1], dict):
47              Q_values = main_nn.predict(state[0][np.newaxis])
48          else:
49              Q_values = main_nn.predict(state[np.newaxis])
50
51          return np.argmax(Q_values[0])
```

Figura 4: Función de política de exploración/explotación para Space Invaders

En la Figura 4, la función ‘epsilon_greedy_policy‘ implementa una política de exploración/explotación epsilon-greedy. Toma como entrada el estado actual del entorno de juego y un parámetro de epsilon que determina el grado de exploración. Se debe tomar en consideración que si un número aleatorio generado por ‘np.random.rand()‘ es menor que epsilon, se elige una acción aleatoria entre las acciones posibles (‘n_outputs‘) y se devuelve; de lo contrario, se calculan los valores Q para el estado actual utilizando el modelo ‘main_nn‘ y se elige la acción con el valor Q máximo utilizando ‘np.argmax()‘, esto implica una estrategia de explotación, ya que se selecciona la acción con mayor valor Q estimado.

Es importante destacar que la función tiene una condición que verifica el tipo y la forma del estado, esto se debe el modelo puede requerir diferentes formatos de entrada dependiendo del entorno de juego. Mediante prueba y error se observaba que en state habían datos tipo np.array y de ves en cuando entraba datos tipo tupla la cual contenia el dato np.array y un diccionario, solo debian de haber datos tipo np.array para poder convertirlo en tensor por lo que se hizo un for con la siguiente evaluacion, si el estado es una tupla de longitud 2, donde el primer elemento es una matriz numpy y el segundo es un diccionario, se aplica el formato adecuado antes de realizar la predicción con ‘main_nn.predict()‘, de lo contrario, se asume que el estado es una matriz numpy y se realiza la predicción directamente.

```
55  def sample_experiences(batch_size):
56      indices = np.random.randint(len(replay_buffer), size=batch_size)
57      batch = [replay_buffer[index] for index in indices]
58      states, actions, rewards, next_states, dones = [
59          np.array([experience[field_index] for experience in batch], dtype=object)
60          for field_index in range(5)]
61      return states, actions, rewards, next_states, dones
```

Figura 5: Función sample_experiences para Space Invaders

Ahora para la Figura 5, la función ‘sample_experiences‘ se utiliza para muestrear experiencias del replay buffer para el entrenamiento del modelo de aprendizaje reforzado, toma como argumento el tamaño de lote (‘batch_size‘) que determina cuántas experiencias se deben muestrear.

Con esto, se generan índices aleatorios (‘indices‘) utilizando ‘np.random.randint()‘ para seleccionar al azar muestras del replay buffer, el tamaño de la muestra se determina mediante el argumento ‘batch_size‘.

Luego se crea un lote (‘batch‘) seleccionando las experiencias correspondientes a los índices generados y se divide el lote en diferentes arreglos (‘states‘, ‘actions‘, ‘rewards‘, ‘next_states‘, ‘dones‘) para cada campo de la experiencia utilizando una comprensión de lista.

Para cada campo de experiencia, se crea un arreglo numpy (‘np.array()‘) tomando los valores correspondientes del lote. El argumento ‘dtype=object‘ se utiliza para permitir que los arreglos contengan objetos de Python en lugar de solo tipos de datos numéricos.

Finalmente, se devuelven los arreglos ‘states‘, ‘actions‘, ‘rewards‘, ‘next_states‘ y ‘dones‘ que contienen las muestras de experiencias seleccionadas aleatoriamente del replay buffer en el tamaño de lote especificado, estos arreglos se utilizarán para el entrenamiento del modelo de aprendizaje reforzado.

```python
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, _,info = env.step(action)
    if next_state.dtype == np.uint8:
        replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward
```

Figura 6: Función play_one_step para Space Invaders

Con respecto a la Figura 6, la función ‘play_one_step‘ realiza una iteración de juego en el entorno (‘env‘) utilizando una política epsilon-greedy, toma como argumentos el entorno, el estado actual (‘state‘) y el valor de epsilon.

En esta primero se selecciona una acción utilizando la función ‘epsilon_greedy_policy‘ pasando el estado actual y el valor de epsilon, la acción se almacena en la variable ‘action‘.

Después se realiza la acción en el entorno llamando al método ‘env.step(action)‘, esto devuelve la siguiente observación del entorno (‘next_state‘), la recompensa obtenida (‘reward‘), una bandera que indica si el episodio ha terminado (‘done‘), y otra información adicional (‘_‘, ‘info‘).

Luego se verifica el tipo de datos de ‘next_state‘ utilizando ‘next_state.dtype‘; si es de tipo ‘np.uint8‘ (entero sin signo de 8 bits), se agrega la experiencia al replay buffer (‘replay_buffer‘) utilizando la función ‘append()‘, la experiencia consiste en una tupla que contiene el estado actual, la acción tomada, la recompensa recibida, el siguiente estado y una bandera que indica si el episodio ha terminado.

Finalmente, se devuelve el siguiente estado (‘next_state‘) y la recompensa (‘reward‘), estos valores se utilizarán en la iteración siguiente del bucle de juego.

```python
discount_rate = 0.99


def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = target_nn.predict(next_states.astype('float32'))
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards + (1 - dones) * discount_rate * max_next_Q_values)
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions.astype('int32'), n_outputs)
    with tf.GradientTape() as tape:
        for i in range(len(states)):
            if isinstance(states[i], tuple) and len(states[i]) == 2 and isinstance(states[i][0], np.ndarray) and isinstance(states[i][1], dict):
                states[i] = states[i][0]
            elif states[i].shape != (210, 160, 3):
                states[i] = states[i-1]
        all_Q_values = main_nn(tf.convert_to_tensor(np.stack([np.array(state, dtype=object) for state in states]).astype('float32')))
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values.astype('float32'), Q_values))
        print(loss)
    grads = tape.gradient(loss, main_nn.trainable_variables)
    optimizer.apply_gradients(zip(grads, main_nn.trainable_variables))
    return loss
```

Figura 7: Función training_step para Space Invaders

La siguiente función en la Figura 7 es ‘training_step‘, realiza un paso de entrenamiento para el modelo de aprendizaje reforzado utilizando la técnica de Double Deep Q-Network (DDQN), toma como argumento el tamaño de lote (‘batch_size‘) que se utilizará para el entrenamiento.

En esta primero se muestrean experiencias del replay buffer llamando a la función ‘sample_experiences‘ y se desempaquetan en las variables ‘states‘, ‘actions‘, ‘rewards‘, ‘next_states‘ y ‘dones‘.

Después se obtienen los valores Q del modelo ‘target_nn‘ para los siguientes estados (‘next_states‘) utilizando ‘target_nn.predict(next_state estos valores se utilizan para calcular el máximo valor Q para cada muestra de lote (‘max_next_Q_values‘) mediante ‘np.max()‘.

Luego se calculan los valores objetivo de Q (‘target_Q_values‘) utilizando la fórmula del algoritmo Q-Learning, de suma la recompensa actual (‘rewards‘) con el descuento (‘discount_rate‘) multiplicado por el máximo valor Q de los siguientes estados, pero solo si el episodio no ha terminado (‘1 - dones‘); los valores objetivo se almacenan en un arreglo de forma (-1, 1).

Se crea una máscara (‘mask‘) utilizando ‘tf.one_hot()‘ para convertir las acciones (‘actions‘) en una representación one-hot, la máscara se utilizará para calcular el Q-value de las acciones seleccionadas.

Posterior a esto se utiliza un bucle ‘for‘ para iterar sobre las muestras de lote (‘states‘) y realizar algunas transformaciones en los estados antes de pasarlos al modelo:

- Si el estado es una tupla de longitud 2 con un arreglo numpy y un diccionario, se asigna solo el arreglo numpy al estado.
- Si el estado no tiene la forma (210, 160, 3), se asigna el estado anterior al estado actual para mantener la coherencia de la forma de los estados.

Se aplica el modelo ‘main_nn‘ a los estados utilizando ‘main_nn(tf.convert_to_tensor())‘ después de convertir los estados en un tensor float32, el resultado son todos los valores Q para las acciones posibles en cada estado.

Se seleccionan los valores Q correspondientes a las acciones tomadas utilizando ‘tf.reduce_sum()‘ junto con la máscara (‘all_Q_values * mask‘), esto devuelve los valores Q para las acciones tomadas en cada estado en una matriz de forma (-1, 1).

Se calcula la pérdida (‘loss‘) utilizando la función de pérdida (‘loss_fn‘) entre los valores objetivo de Q (‘target_Q_values‘) y los valores Q obtenidos (‘Q_values‘); la pérdida se reduce tomando la media de todas las muestras.

Se utiliza un objeto ‘tf.GradientTape()‘ para calcular los gradientes de la pérdida con respecto a las variables entrenables del modelo ‘main_nn‘.

Se aplica la optimización mediante el optimizador (‘optimizer‘) llamando a ‘optimizer.apply_gradients()‘ pasando los gradientes y las variables entrenables del modelo.

Finalmente, se devuelve la pérdida calculada (‘loss‘) durante el paso de entrenamiento.

```python
def play_one_step_train(env, state, model):
    # Verificar si el estado es una tupla
    if isinstance(state, tuple) and len(state) == 2 and isinstance(state[0], np.ndarray):
        state = state[0]

    # Utilizar el modelo para predecir acciones
    Q_values = model.predict(state[np.newaxis])
    action = np.argmax(Q_values[0])

    next_state, reward, done, _,_ = env.step(action)
    if next_state.dtype == np.uint8:
        replay_buffer.append((state, action, reward, next_state, done))
    return next state, reward
```

Figura 8: Función play_one_step_train para Space Invaders

En el caso de la Figura 8, la función ‘play_one_step_train‘ realiza una iteración de juego en el entorno (‘env‘) y actualiza el replay buffer (‘replay_buffer‘) durante el entrenamiento del modelo, toma como argumentos el entorno, el estado actual (‘state‘) y el modelo utilizado para predecir las acciones.

Primero se verifica si el estado es una tupla de longitud 2 y si el primer elemento es un arreglo numpy, si se cumple esta condición, se asigna solo el primer elemento del estado a la variable ‘state‘, esto es para manejar casos especiales donde el estado es una tupla que contiene información adicional.

Después se utiliza el modelo para predecir los valores Q para el estado actual llamando a ‘model.predict(state[np.newaxis])‘; los valores Q se almacenan en la variable ‘Q_values‘.

Se elige la acción con el valor Q máximo utilizando ‘np.argmax(Q_values[0])‘; la acción seleccionada se asigna a la variable ‘action‘.

Se realiza la acción en el entorno llamando al método ‘env.step(action)‘, esto devuelve la siguiente observación del entorno (‘next_state‘), la recompensa obtenida (‘reward‘), una bandera que indica si el episodio ha terminado (‘done‘), y otra información adicional (‘_‘, ‘_‘).

Se verifica el tipo de datos de ‘next_state‘ utilizando ‘next_state.dtype‘, si es de tipo ‘np.uint8‘ (entero sin signo de 8 bits), se agrega la experiencia al replay buffer (‘replay_buffer‘) utilizando la función ‘append()‘. La experiencia consiste en una tupla que contiene el estado actual, la acción tomada, la recompensa recibida, el siguiente estado y una bandera que indica si el episodio ha terminado.

Finalmente, se devuelve el siguiente estado (‘next_state‘) y la recompensa (‘reward‘), estos valores se utilizarán en la iteración siguiente del bucle de juego durante el entrenamiento del modelo.

```python
model_file = 'C:/Users/Pedro/Downloads/Proyecto 2/proyecto2-brayain_proyecto2/proyecto2-brayain_proyecto2/my_dqn_2.h5'

if os.path.isfile(model_file):
    model = keras.models.load_model(model_file)

    env = gym.make('ALE/SpaceInvaders-v5', render_mode='human')
    obs = env.reset()

    while True:
        obs, reward = play_one_step_train(env, obs, model)


else:
    wandb.init(project="SI Project")
    for episode in range(400):

        obs = env.reset()

        for step in range(150):
            epsilon = max(1 - episode / 500, 0.01)
            obs, reward = play_one_step(env, obs, epsilon)

            if episode > 70:
                loss = training_step(70)

                wandb.log({"episode": episode, "total_reward": reward, "loss": loss})
        print(f"Episode: {episode}")
    main_nn.save('my_dqn_f.h5')
```

Figura 9: Loop para Space Invaders

Para el loop del juego en la Figura 9, primero se verifica si existe un archivo de modelo en la ubicación especificada, si el archivo existe, se carga el modelo utilizando ‘keras.models.load_model(model_file)‘.

Si el archivo de modelo no existe, se inicializa el proyecto de WandB llamado ”SI Projectütilizando ‘wandb.init(project=”SI Project”)‘ con el fin de registrar el error y recompensas.

Con esto se inicia un bucle principal que itera a través de los episodios del juego (400 en este caso) y para cada episodio, se resetea el entorno (‘env.reset()‘) y se inicializa el estado observado (‘obs‘).

Luego se inicia otro bucle interno que itera a través de los pasos del episodio (150 en este caso) y se calcula el valor de epsilon para la política epsilon-greedy, el cual disminuye linealmente a lo largo de los episodios, el valor de epsilon se establece en 1 al principio y se reduce a un mínimo de 0.01 utilizando ‘max(1 - episode / 500, 0.01)‘.

Después se realiza una iteración de juego llamando a la función ‘play_one_step‘ con el entorno, el estado observado (‘obs‘) y el valor de epsilon, esto devuelve el siguiente estado observado (‘obs‘) y la recompensa recibida (‘reward‘).

Si el número de episodio es mayor a 70, se realiza un paso de entrenamiento llamando a la función ‘training_step‘ con un tamaño de lote de 70, esto actualiza el modelo de aprendizaje reforzado utilizando las experiencias del replay buffer, además, se registra el episodio actual, la recompensa total y la pérdida en WandB utilizando ‘wandb.log()‘.

Después de completar el bucle interno de pasos, se imprime el número de episodio actual y al finalizar los 400 episodios, se guarda el modelo ‘main_nn‘ en un archivo llamado ‘my_dqn_f.h5‘ utilizando ‘main_nn.save(’my_dqn_f.h5’)‘.

En resumen, el código carga un modelo pre-entrenado si existe, o bien entrena un nuevo modelo a través de múltiples episodios y realiza el registro de las recompensas y pérdidas utilizando WandB, al finalizar, se guarda el modelo entrenado en un archivo.

I-B. Resultados Space Invaders

Mediante Weight & Biases se logró registrar la pérdida y reward del entrenamiento, debido a la limitación tanto del equipo utilizado como de Colab, lo más que se logró registrar fueron 360 épocas, después de estas el entrenamiento se quedaba sin memoria.

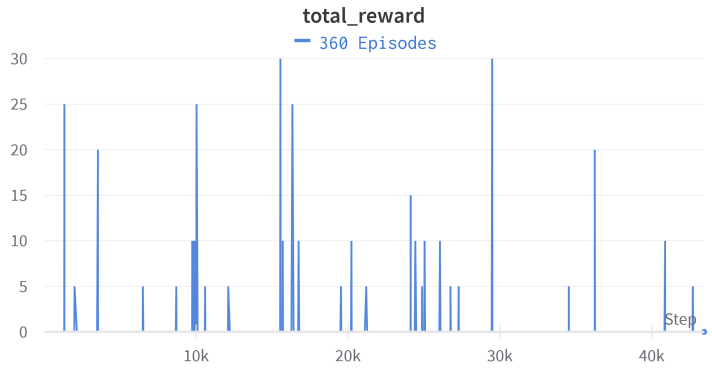

Figura 10: Reward por step para entrenamiento Space Invaders

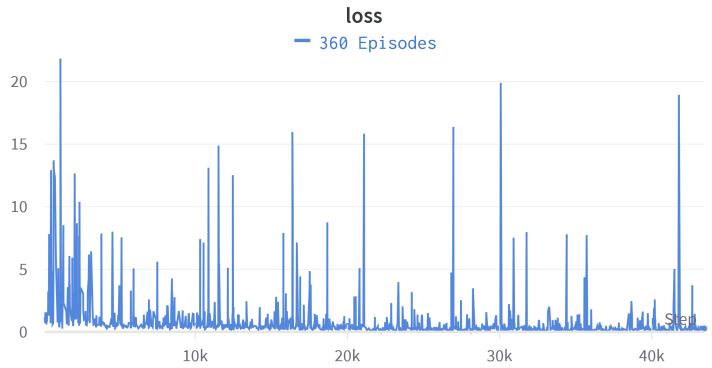

Figura 11: Pérdida por step para entrenamiento Space Invaders

En este caso podemos observar que entre más episodios, la pérdida se reduce gradualmente con ciertos picos que se reducen también al hacer más iteraciones.

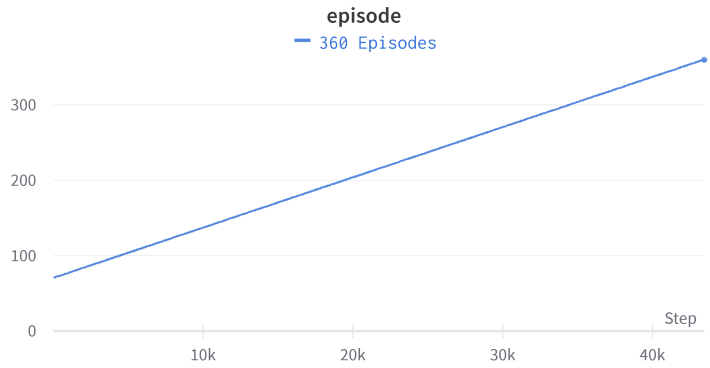

Figura 12: Época por step para entrenamiento Space Invaders

Además se realizó el mismo proceso, pero mediante GPU, el problema de esta es que llegaba a menos épocas y crasheó antes de guardar el modelo.

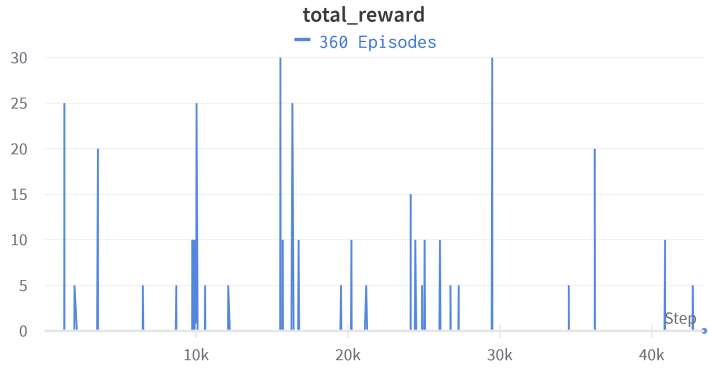

Figura 13: Reward por step para entrenamiento con GPU Space Invaders

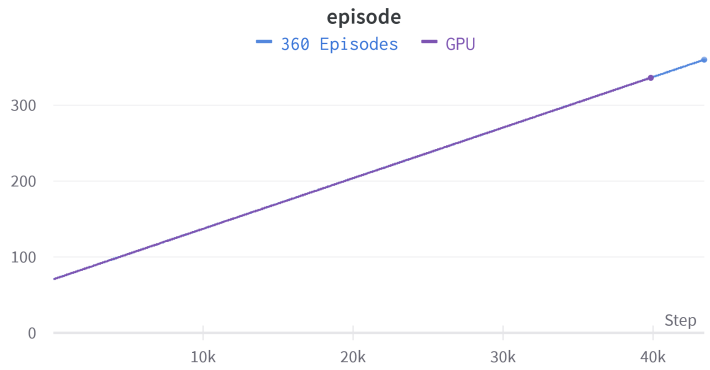

Figura 14: Pérdida por step para entrenamiento con GPU Space Invaders

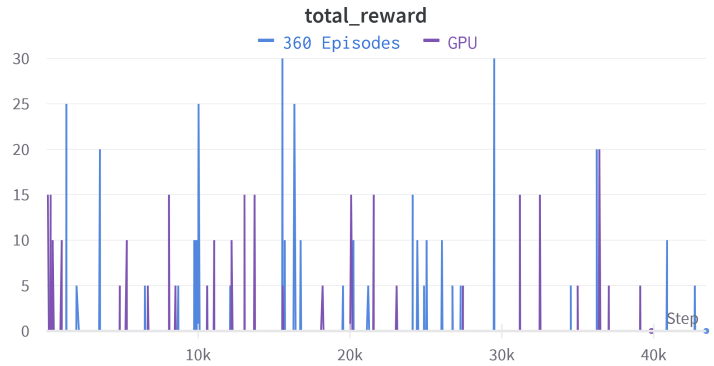

Figura 15: Época por step para entrenamiento con GPU Space Invaders

II. 3.2 Segunda Etapa: Gradiente de política

II-A. Araña MuJoCo

Para este caso se seleccionó la Araña (Ant) de MuJoCo, se pensó en entrenar el modelo para que esta logre caminar en 1000 steps.

```
1  import gymnasium as gym
2  import torch
3  import torch.nn as nn
4  import torch.optim as optim
5  import torch.nn.functional as F
6  import gym
7  import os.path
8  import numpy as np
9  from torch.distributions import Normal
```

Figura 16: Librerías para Araña MuJoCo

Para esto primero se importan varias librerías necesarias para el proyecto de aprendizaje por gradiente de política.

- ‘torch‘: Importa la librería ‘torch‘, que es el principal marco de trabajo utilizado para el aprendizaje automático en PyTorch. Proporciona funciones y estructuras de datos para construir y entrenar modelos de aprendizaje automático.
- ‘torch.nn as nn‘: Importa el módulo ‘nn‘ de PyTorch, que contiene clases y funciones para construir redes neuronales. Se utiliza para definir la arquitectura del modelo de aprendizaje por gradiente de política.
- ‘torch.optim as optim‘: Importa el módulo ‘optim‘ de PyTorch, que proporciona optimizadores para ajustar los pesos de los modelos durante el entrenamiento. Se utiliza para definir el optimizador que actualizará los parámetros del modelo.
- ‘torch.nn.functional as F‘: Importa el módulo ‘functional‘ de PyTorch, que contiene funciones útiles que no tienen parámetros entrenables. Se utiliza para aplicar funciones de activación y otras operaciones a los resultados de las capas de la red neuronal.
- ‘os.path‘: Importa el módulo ‘path‘ del paquete ‘os‘, que proporciona funciones para trabajar con rutas de archivos y directorios. Se utiliza para verificar si existe un archivo de modelo pre-entrenado.
- ‘torch.distributions.Normal‘: Importa la clase ‘Normal‘ del módulo ‘distributions‘ de PyTorch, que representa una distribución normal. Se utiliza para modelar la distribución de las acciones elegidas por el modelo de gradiente de política.

Para la Figura 17 la clase ‘CustomAntEnv‘ es una subclase de ‘gym.Wrapper‘, lo que significa que es un envoltorio (wrapper) personalizado para el entorno ‘env‘ de la clase base, esta realiza lo siguiente:

- ‘__init__(self, env)‘: Este método inicializa la instancia de ‘CustomAntEnv‘ y llama al constructor de la clase base ‘gym.Wrapper‘ utilizando ‘super().__init__(env)‘. Luego, inicializa algunas variables de estado adicionales como ‘prev_x_position‘, ‘prev_y_position‘ y ‘num_steps‘.
- ‘step(self, action)‘: Este método reemplaza el método ‘step()‘ de la clase base ‘gym.Wrapper‘. Toma una acción ‘action‘ y realiza un paso en el entorno llamando al método ‘step()‘ del entorno base. Devuelve el próximo estado, la recompensa, una

```python
class CustomAntEnv(gym.Wrapper):
    def __init__(self, env):
        super().__init__(env)
        self.prev_x_position = None
        self.prev_y_position = None
        self.num_steps = 0

    def step(self, action):
        next_state, reward, done, info,_ = self.env.step(action)
        # Calculate the custom reward based on forward movement in the X-axis
        x_position = next_state[0]
        if self.prev_x_position is not None:
            x_distance = x_position - self.prev_x_position
            if x_distance > 0.01:
                reward += 10  # Give positive reward for moving forward in X-axis
            elif x_distance < -0.01:
                reward -= 10  # Give negative reward for moving backward in X-axis
        self.prev_x_position = x_position


        y_position = next_state[1]
        if self.prev_y_position is not None:
            self.prev_y_position = y_position
        # Ignore Y-axis movement


        return next_state, reward, done, info
```

Figura 17: Clase ambiente personalizado para Araña MuJoCo

bandera que indica si el episodio ha terminado (‘done‘), y una información adicional (‘info‘).
Dentro del método ‘step()‘, se realiza un cálculo personalizado de la recompensa basado en el movimiento hacia adelante en el eje X. El código sigue los siguientes pasos:

1. Obtiene el próximo estado, la recompensa, la bandera ‘done‘, y la información adicional del entorno base utilizando ‘self.env.step(action)‘.
2. Calcula el movimiento en el eje X comparando la posición actual (‘x_position‘) con la posición anterior (‘self.prev_x_position‘). Si ‘x_distance‘ es mayor que 0.01, se agrega una recompensa positiva de 10 para moverse hacia adelante en el eje X. Si ‘x_distance‘ es menor que -0.01, se agrega una recompensa negativa de -10 para moverse hacia atrás en el eje X.
3. Actualiza la posición anterior en el eje X (‘self.prev_x_position = x_position‘).
4. Ignora el movimiento en el eje Y estableciendo ‘self.prev_y_position = y_position‘.
5. Devuelve el próximo estado, la recompensa modificada, la bandera ‘done‘, y la información adicional.

En resumen, la clase ‘CustomAntEnv‘ es un envoltorio personalizado para el entorno del ANT. Realiza un cálculo de recompensa personalizado basado en el movimiento en el eje X, y ajusta la recompensa devuelta por el entorno base antes de devolverla al agente.

```python
env = gym.make('Ant-v4',
               render_mode='human',
               ctrl_cost_weight=0.5,
               use_contact_forces=True,
               healthy_reward=2,
               healthy_z_range=(0.2, 1),
               terminate_when_unhealthy=False)

custom_env = CustomAntEnv(env)
```

Figura 18: Entorno para Araña MuJoCo

Para la Figura 18 se crea una instancia del entorno ‘Ant-v4‘ de MuJoCo utilizando la función ‘gym.make()‘ y se especifican varios argumentos adicionales para configurar el comportamiento del entorno:

- ‘render_mode=’human’‘: Indica que se debe mostrar el entorno con renderizado gráfico mientras se realiza la simulación. Esto permite visualizar el comportamiento del agente mientras aprende.

- ‘ctrl_cost_weight=0.5‘: Especifica el peso de costo de control. Este parámetro controla cuánto se penaliza el uso excesivo de energía por parte del agente para realizar acciones. Un valor más alto penalizará más el esfuerzo del agente.
- ‘use_contact_forces=True‘: Indica que se deben tener en cuenta las fuerzas de contacto en la simulación. Esto permite que el agente interactúe con objetos y superficies en el entorno.
- ‘healthy_reward=2‘: Especifica la recompensa adicional que se otorga al agente cuando se considera saludable. Proporcionar una recompensa adicional por mantener una postura estable o saludable puede incentivar al agente a aprender un comportamiento más estable.
- ‘healthy_z_range=(0.2, 1)‘: Define el rango de valores de altura (‘z‘) considerados como saludables para el agente. Si la posición ‘z‘ del agente se encuentra fuera de este rango, se considera que el agente está en una postura no saludable.
- ‘terminate_when_unhealthy=False‘: Determina si el episodio se debe terminar cuando el agente está en una postura no saludable. En este caso, se establece en ‘False‘, lo que significa que el episodio no se terminará automáticamente cuando el agente esté en una postura no saludable.

Luego se crea una instancia de ‘CustomAntEnv‘ pasando el entorno base ‘env‘ como argumento. Esto aplica el envoltorio personalizado ‘CustomAntEnv‘ al entorno base, lo que permite realizar ajustes adicionales, como el cálculo personalizado de recompensas basado en el movimiento en el eje X.

```
51  class Actor(nn.Module):
52      def __init__(self, state_dim, action_dim):
53          super(Actor, self).__init__()
54          self.fc1 = nn.Linear(state_dim, 64)
55          self.fc2 = nn.Linear(64, 64)
56          self.fc3 = nn.Linear(64, 64)
57          self.mu_head = nn.Linear(64, action_dim)
58          self.sigma_head = nn.Linear(64, action_dim)
59
60      def forward(self, x):
61          x = F.relu(self.fc1(x))
62          x = F.relu(self.fc2(x))
63          x = F.relu(self.fc3(x))
64          mu = torch.tanh(self.mu_head(x))
65          sigma = F.softplus(self.sigma_head(x))
66          return mu, sigma
```

Figura 19: Clase Actor para Araña MuJoCo

La clase ‘Actor‘ es una subclase de ‘nn.Module‘ de PyTorch y representa la arquitectura de la red neuronal del actor en el método de gradiente de política. A continuación se explica la estructura y funcionalidad de esta clase:

- ‘__init__(self, state_dim, action_dim)‘: Este método inicializa la instancia del actor. Toma como argumentos ‘state_dim‘ y ‘action_dim‘, que representan las dimensiones del espacio de estados y acciones respectivamente. Dentro del método, se definen las capas lineales y de salida de la red neuronal del actor.
- ‘forward(self, x)‘: Este método define la propagación hacia adelante (forward pass) de la red neuronal del actor. Toma como entrada ‘x‘, que representa el estado de entrada. El estado se pasa a través de capas lineales y funciones de activación ReLU para extraer características y transformar la entrada. Luego, se calcula la media (‘mu‘) y la desviación estándar (‘sigma‘) de la distribución de probabilidad de las acciones a tomar. La función ‘torch.tanh‘ se utiliza para limitar los valores de ‘mu‘ en el rango (-1, 1), lo que corresponde a las acciones. La función ‘F.softplus‘ se utiliza para asegurar que ‘sigma‘ sea siempre positiva.

En resumen, la clase ‘Actor‘ define la arquitectura de la red neuronal del actor en el método de gradiente de política. Proporciona una forma de obtener la media y la desviación estándar de la distribución de probabilidad de las acciones a partir de un estado de entrada.

```python
class Critic(nn.Module):
    def __init__(self, state_dim):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(state_dim, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, 64)
        self.v_head = nn.Linear(64, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        v = self.v_head(x)
        return v
```

Figura 20: Clase Crítica para Araña MuJoCo

La clase ‘Critic‘ también es una subclase de ‘nn.Module‘ de PyTorch y representa la arquitectura de la red neuronal del crítico en el método de gradiente de política. A continuación se explica la estructura y funcionalidad de esta clase:

- ‘__init__(self, state_dim)‘: Este método inicializa la instancia del crítico. Toma como argumento ‘state_dim‘, que representa la dimensión del espacio de estados. Dentro del método, se definen las capas lineales y de salida de la red neuronal del crítico.

- ‘forward(self, x)‘: Este método define la propagación hacia adelante (forward pass) de la red neuronal del crítico. Toma como entrada ‘x‘, que representa el estado de entrada. El estado se pasa a través de capas lineales y funciones de activación ReLU para extraer características y transformar la entrada. Luego, se calcula el valor (‘v‘) del estado, que es una estimación del valor esperado de las recompensas futuras a partir del estado dado. La capa de salida ‘v_head‘ produce el valor estimado del estado.

En resumen, la clase ‘Critic‘ define la arquitectura de la red neuronal del crítico en el método de gradiente de política. Proporciona una forma de estimar el valor del estado a partir de un estado de entrada. Esta estimación se utiliza en el cálculo del valor de la función de pérdida en el método de actualización del actor mediante el gradiente de política.

```python
class PPO:
    def __init__(self, state_dim, action_dim):
        self.actor = Actor(state_dim, action_dim)
        self.critic = Critic(state_dim)
        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr=1.5e-4) ## 3e-4
        self.critic_optimizer = optim.Adam(self.critic.parameters(), lr=1.5e-4)

    def select_action(self, state):
        with torch.no_grad():
            state = torch.FloatTensor(state)
            mu, sigma = self.actor(state)
            dist = Normal(mu, sigma)
            action = dist.sample()
            action = action.clamp(-1.0, 1.0) # recorta la acción para que esté dentro de los límites
            log_prob = dist.log_prob(action).sum(-1).item() # convierte a escalar
            value = self.critic(state).item() # convierte a escalar
            return action.numpy(), log_prob, value
```

Figura 21: Clase PPO para Araña MuJoCo gbcbcb

La clase ‘PPO‘ implementa el algoritmo Proximal Policy Optimization (PPO) para el aprendizaje de políticas en entornos de RL, consiste en lo siguiente:

- ‘__init__(self, state_dim, action_dim)‘: Este método inicializa la instancia de PPO. Toma como argumentos ‘state_dim‘ y ‘action_dim‘, que representan las dimensiones del espacio de estados y acciones, respectivamente. Dentro del método, se crea una instancia del actor (‘self.actor‘) y del crítico (‘self.critic‘) utilizando las clases ‘Actor‘ y ‘Critic‘, respectivamente. También se definen los optimizadores para el actor (‘self.actor_optimizer‘) y el crítico (‘self.critic_optimizer‘) utilizando el algoritmo Adam con tasas de aprendizaje específicas.

- ‘select_action(self, state)‘: Este método se utiliza para seleccionar una acción en función del estado dado. Toma como argumento ‘state‘, que representa el estado actual. Dentro del método, se realiza una propagación hacia adelante a través del actor para obtener la distribución de probabilidad de las acciones. Se muestrea una acción de esta distribución y se aplica un recorte para asegurarse de que la acción esté dentro de los límites permitidos. También se calcula el logaritmo de la probabilidad

```python
    def update(self, states, actions, log_probs_old, returns):
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        log_probs_old = torch.FloatTensor(log_probs_old).unsqueeze(-1)
        returns = torch.FloatTensor(returns).unsqueeze(-1)

        for _ in range(10): # actualiza varias veces
            mu, sigma = self.actor(states)
            dist = Normal(mu, sigma)
            log_probs_new = dist.log_prob(actions).sum(-1).unsqueeze(-1)
            ratio = (log_probs_new - log_probs_old).exp()
            values_new = self.critic(states)

            advantages_new = returns - values_new
            actor_loss_new = -(ratio * advantages_new).mean()
            critic_loss_new = advantages_new.pow(2).mean()

            # actualiza el actor
            self.actor_optimizer.zero_grad()
            actor_loss_new.backward(retain_graph=True)
            self.actor_optimizer.step()

            # actualiza el crítico
            self.critic_optimizer.zero_grad()
            critic_loss_new.backward(retain_graph=True)
            self.critic_optimizer.step()
        return actor_loss_new.item(), critic_loss_new.item()
```

Figura 22: Continuación de la clase PPO para Araña MuJoCo gbcbcb

de la acción seleccionada y el valor del estado utilizando el crítico. El método devuelve la acción seleccionada como un array NumPy, el logaritmo de la probabilidad de la acción, y el valor del estado.

- ‘update(self, states, actions, log_probs_old, returns)‘: Este método se utiliza para actualizar los parámetros del actor y del crítico mediante el algoritmo PPO. Toma como argumentos ‘states‘, ‘actions‘, ‘log_probs_old‘ y ‘returns‘, que representan los estados, acciones, logaritmo de las probabilidades antiguas y los retornos (recompensas descontadas), respectivamente. Dentro del método, se realiza un bucle de actualización que itera varias veces. En cada iteración, se calculan las nuevas logaritmos de las probabilidades, los valores del estado y las ventajas. Luego se calculan las pérdidas del actor y del crítico utilizando las fórmulas del algoritmo PPO. Finalmente, se realizan las actualizaciones de los parámetros del actor y del crítico mediante los respectivos optimizadores. El método devuelve las pérdidas del actor y del crítico como escalares.

En resumen, la clase ‘PPO‘ encapsula la implementación del algoritmo Proximal Policy Optimization. Proporciona métodos para seleccionar acciones y actualizar los parámetros del actor y del crítico en función de las experiencias de interacción con el entorno.

```python
class RewardNormalizer:
    def __init__(self, gamma=0.99):
        self.gamma = gamma
        self.mean = 0
        self.var = 1

    def normalize(self, rewards):
        # Calcula la media y la varianza con descuento
        discounted_rewards = [self.gamma ** i * r for i, r in enumerate(rewards)]
        mean = np.mean(discounted_rewards)
        var = np.var(discounted_rewards)

        # Actualiza la media y la varianza con un factor de olvido
        alpha = 0.9
        self.mean = alpha * self.mean + (1 - alpha) * mean
        self.var = alpha * self.var + (1 - alpha) * var

        # Normaliza las recompensas
        std = np.sqrt(self.var)
        normalized_rewards = (rewards - self.mean) / (std + 1e-8)
        return normalized_rewards.tolist()

reward_normalizer = RewardNormalizer()

state_dim=env.observation_space.shape[0]
action_dim=env.action_space.shape[0]
```

Figura 23: Normalizador de Rewards para Araña MuJoCo

La clase ‘RewardNormalizer‘ se utiliza para normalizar las recompensas en el algoritmo PPO. A continuación se explica el método y los atributos clave de esta clase:

- ‘__init__(self, gamma=0.99)‘: Este método inicializa la instancia de ‘RewardNormalizer‘. Toma un parámetro opcional ‘gamma‘ que representa el factor de descuento utilizado para calcular las recompensas descontadas. Dentro del método, se inicializan las variables ‘mean‘ y ‘var‘ que se utilizan para realizar el cálculo y la actualización de la media y la varianza de las recompensas.

- ‘normalize(self, rewards)‘: Este método se utiliza para normalizar las recompensas proporcionadas como argumento. Toma como argumento ‘rewards‘, que es una lista de recompensas. Dentro del método, se calcula la media y la varianza de las recompensas descontadas utilizando el factor de descuento ‘gamma‘. Luego, se realiza una actualización de la media y la varianza utilizando un factor de olvido. Finalmente, se normalizan las recompensas utilizando la media y la desviación estándar calculadas y se devuelven como una lista.

En resumen, la clase ‘RewardNormalizer‘ proporciona un mecanismo para calcular y mantener una estimación actualizada de la media y la varianza de las recompensas. Además, permite normalizar las recompensas utilizando esta estimación, lo que puede ser útil para mejorar la estabilidad y la convergencia del algoritmo de RL.

```
157  class ObservationNormalizer:
158      def __init__(self, observation_dim):
159          self.n = 0
160          self.mean = np.zeros(observation_dim)
161          self.var = np.ones(observation_dim)
162
163      def normalize(self, observation):
164          # Actualiza la media y la varianza con el algoritmo Welford
165          self.n += 1
166          delta = observation - self.mean
167          self.mean += delta / self.n
168          delta2 = observation - self.mean
169          self.var += delta * delta2
170
171          # Normaliza la observación
172          std = np.sqrt(self.var / (self.n + 1e-8))
173          normalized_observation = (observation - self.mean) / (std + 1e-8)
174          return normalized_observation
175
176  observation_normalizer = ObservationNormalizer(observation_dim=state_dim)
```

Figura 24: Normalizador de observaciones para Araña MuJoCo

La clase ‘ObservationNormalizer‘ se utiliza para normalizar las observaciones en el algoritmo PPO, esta clase hace lo siguiente:

- ‘__init__(self, observation_dim)‘: Este método inicializa la instancia de ‘ObservationNormalizer‘. Toma como argumento ‘observation_dim‘, que representa la dimensión de las observaciones de entrada. Dentro del método, se inicializan las variables ‘n‘, ‘mean‘ y ‘var‘. ‘n‘ lleva la cuenta del número de observaciones recibidas, ‘mean‘ es un vector de ceros con la misma dimensión que las observaciones y ‘var‘ es un vector de unos con la misma dimensión.

- ‘normalize(self, observation)‘: Este método se utiliza para normalizar una observación proporcionada como argumento. Toma como argumento ‘observation‘, que es una sola observación. Dentro del método, se actualiza la media y la varianza utilizando el algoritmo Welford, que es un método incremental para calcular la media y la varianza. Se actualizan las variables ‘n‘, ‘mean‘ y ‘var‘ en función de la nueva observación recibida. Luego, se calcula la desviación estándar utilizando la varianza actualizada. Finalmente, se normaliza la observación utilizando la media y la desviación estándar calculadas y se devuelve como un vector.

En resumen, la clase ‘ObservationNormalizer‘ proporciona un mecanismo para calcular y mantener una estimación actualizada de la media y la varianza de las observaciones de entrada. Además, permite normalizar las observaciones utilizando esta estimación, lo que puede ser útil para mejorar la estabilidad y la convergencia del algoritmo de RL.

```python
def compute_returns(rewards, values_next, gamma=0.99):
    R = values_next
    returns = []
    for r in reversed(rewards):
        R = r + gamma * R
        returns.insert(0,R)
    return returns

def extract_state(output):
    if isinstance(output, tuple):
        state = output[0]
    else:
        state = output
    return state


ppo=PPO(state_dim=state_dim,
        action_dim=action_dim)

num_episodes=300
num_steps=3000
num_steps_rend=10000
```

Figura 25: Returns, extracción de estados y definición de variables para Araña MuJoCo

Las funciones ‘compute_returns‘ y ‘extract_state‘ son utilidades auxiliares que se utilizan en el entrenamiento del algoritmo PPO.

- ‘compute_returns(rewards, values_next, gamma=0.99)‘: Esta función se utiliza para calcular los retornos descontados a partir de las recompensas y los valores estimados del estado siguiente. Toma como argumentos ‘rewards‘, que es una lista de recompensas, y ‘values_next‘, que es el valor estimado del estado siguiente. El parámetro ‘gamma‘ es el factor de descuento que controla la importancia de las recompensas futuras. La función itera a través de las recompensas en orden inverso y calcula los retornos descontados acumulados utilizando la fórmula R = r + gamma * R, donde R es el retorno descontado acumulado y r es una recompensa. Los retornos descontados se almacenan en una lista en el orden original y se devuelve la lista de retornos.

- ‘extract_state(output)‘: Esta función se utiliza para extraer el estado de salida de un modelo. Toma como argumento ‘output‘, que es la salida del modelo. Si ‘output‘ es una tupla, se asume que el estado está en la primera posición de la tupla, y se devuelve dicho estado. Si ‘output‘ no es una tupla, se asume que el estado es ‘output‘ mismo, y se devuelve sin cambios. Esta función es útil para manejar diferentes formas de salida del modelo y asegurarse de que se obtenga el estado correcto.

En el código proporcionado, se crea una instancia de la clase ‘PPO‘ y se definen variables para el número de episodios (‘num_episodes‘), el número de pasos por episodio (‘num_steps‘), y el número de pasos para mostrar el rendimiento (‘num_steps_rend‘). Estas variables se utilizan posteriormente en el entrenamiento del algoritmo PPO.

```python
if os.path.isfile('C:/Users/pedro/ppo_actor_model.pt')and
os.path.isfile('C:/Users/pedro/ppo_critic_model.pt'):

    # Si los archivos de entrenamiento existen, carga el modelo desde los archivos
    ppo = PPO(state_dim=state_dim, action_dim=action_dim)
    ppo.actor.load_state_dict(torch.load('C:/Users/pedro/ppo_actor_model.pt'))
    ppo.critic.load_state_dict(torch.load('C:/Users/pedro/ppo_critic_model.pt'))

    # Código para mostrar en pantalla como se mueve el objeto con los datos de entrenamiento guardados
    state = extract_state(env.reset())
    for step in range(num_steps_rend):
        action, _, _ = ppo.select_action(state)
        next_state, reward, _, _= custom_env.step(action)
        env.render()
        state = extract_state(next_state)
    env.close()
```

Figura 26: Loop del entrenamiento para Araña MuJoCo

Finalmente se realiza el loop,o sea se realiza el entrenamiento y evaluación del algoritmo PPO (Proximal Policy Optimization) en un entorno de Gym utilizando un modelo de MuJoCo ANT de la siguiente forma:

1. Se verifica si los archivos de entrenamiento existen (‘ppo_actor_model.pt‘ y ‘ppo_critic_model.pt‘) en la ubicación especificada. Si los archivos existen, se carga el modelo desde los archivos y se muestra el movimiento del objeto en el

```
else:
    # Si los archivos de entrenamiento no existen, entrena el modelo y guarda los archivo
    for episode in range(num_episodes):
        state=extract_state(env.reset())
        state = observation_normalizer.normalize(state)
        rewards=[]
        states=[]
        actions=[]
        log_probs=[]
        values=[]
        for step in range(num_steps):
            action,log_prob,value=ppo.select_action(state)
            next_state,reward,_ ,_ =custom_env.step(action)
            next_state = extract_state(next_state)
            next_state = observation_normalizer.normalize(next_state)
            rewards.append(reward)
            states.append(state)
            actions.append(action)
            log_probs.append(log_prob)
            values.append(value)
            state=next_state
        # Normaliza las recompensas
        rewards = reward_normalizer.normalize(rewards)
        _,_,value_next=ppo.select_action(state)
        returns=compute_returns(rewards,value_next)
        actor_loss, critic_loss = ppo.update(states,actions,log_probs,returns)
        print(f'Episode: {episode+1}, Reward: {sum(rewards)}, Actor Loss: {actor_loss},
              Critic Loss: {critic_loss}')
    # Guarda el modelo en un archivo después de que haya sido entrenado
    torch.save(ppo.actor.state_dict(), 'ppo_actor_model.pt')
    torch.save(ppo.critic.state_dict(), 'ppo_critic_model.pt')
```

Figura 27: Continuación del loop para Araña MuJoCo

entorno utilizando los datos de entrenamiento guardados.

2. Si los archivos de entrenamiento no existen, se inicia el proceso de entrenamiento. Se realiza un bucle a lo largo del número de episodios definidos. Para cada episodio: a. Se reinicia el entorno y se obtiene el estado inicial. b. Se normaliza el estado utilizando ‘observation_normalizer‘. c. Se inicializan listas para almacenar las recompensas, estados, acciones, logaritmos de probabilidades y valores. d. Se realiza un bucle a lo largo del número de pasos definidos. Para cada paso: - Se selecciona una acción utilizando el modelo del actor y el estado actual. - Se realiza la acción en el entorno y se obtiene el siguiente estado, recompensa y otras variables. - Se normaliza el siguiente estado utilizando ‘observation_normalizer‘. - Se almacenan la recompensa, estado, acción, logaritmo de probabilidad y valor en las listas correspondientes. - Se actualiza el estado actual con el siguiente estado. e. Se normalizan las recompensas utilizando ‘reward_normalizer‘. f. Se calculan los retornos descontados utilizando las recompensas normalizadas y el valor estimado del siguiente estado. g. Se realiza la actualización del modelo utilizando los estados, acciones, logaritmos de probabilidades y retornos descontados. h. Se imprime la información del episodio, incluyendo la suma de las recompensas, la pérdida del actor y la pérdida del crítico.

3. Después de que el entrenamiento haya finalizado, se guardan los parámetros del modelo en archivos (‘ppo_actor_model.pt‘ y ‘ppo_critic_model.pt‘) para su uso posterior.

En resumen, el código implementa el algoritmo PPO para entrenar y evaluar un modelo en un entorno de Gym, guardando los parámetros del modelo para su reutilización.

II-B. Resultados Araña MuJoCo

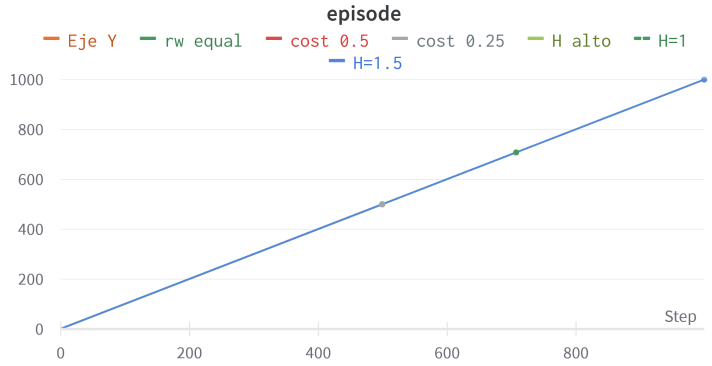

Figura 28: Época por step para entrenamiento Araña MuJoCo

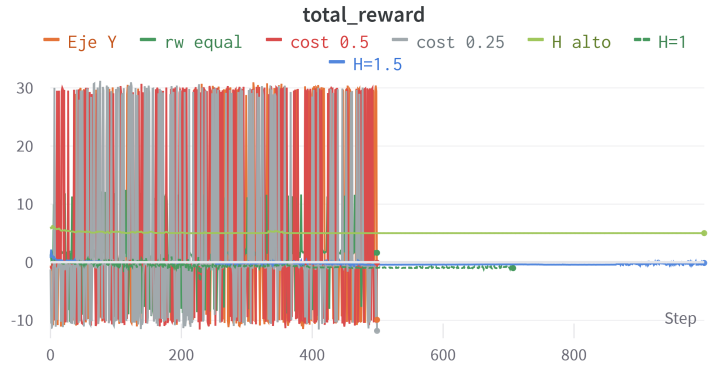

Figura 29: Reward por step para entrenamiento Araña MuJoCo

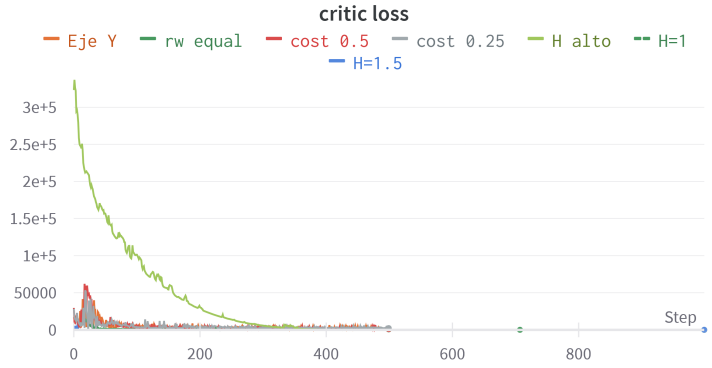

Figura 30: Pérdida crítica por step para entrenamiento Araña MuJoCo

Figura 31: Pérdida de actor por step para entrenamiento Araña MuJoCo

III. Anexos

Para poder todos los códigos base es necesario instalar las siguientes librerías:

- pip install wandb
- pip install tensorflow==2.10.0
- pip install ale_py
- pip install gymnasium[atari]
- pip install gymnasium[accept-rom-license]
- pip install atari-py
- pip install gymnasium[mujoco]