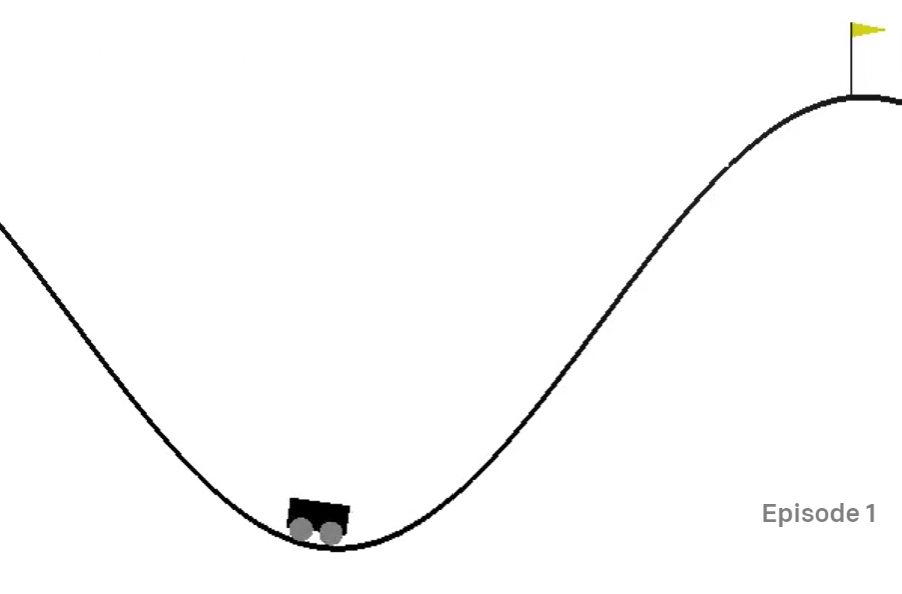
DQN MOUNTAIN CAR CONTINUOUS



Adrián Yared Armas de la Nuez

**Contenido**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

[**1. Objetivo 1**](#_2c35568x6wg8)

[**2. Descripción del problema 2**](#_3q8wnaxm0sv5)

[**3. Código 2**](#_t15scr51mqnp)

[**3.1 Imports y future saving folders 2**](#_t8cwbo6n4466)

[**3.2 Preconfiguración 3**](#_ao835xw092q)

[**3.2.1 Pesos y ajustes del modelo 3**](#_aig92oe7k2m0)

[**3.2.2 Discretización 3**](#_e9uqlcl56mu)

[**3.2.3 Variables globales 3**](#_mglg8vdo1y0q)

[**3.2.4 Creación del entorno del video 4**](#_got4w3vlg98t)

[**3.2.5 Definición de la entrada de la red neuronal 4**](#_mbor3w7wm4w3)

[**3.2.6 Construcción del modelo de red neuronal 4**](#_anvpg7o767zf)

[**3.2.6 Discretización para la entrada de DQN 5**](#_66ntdmoprr5h)

[**3.3 Agente DQN 5**](#_3fwt4vowy5c8)

[**3.3.1 Declaración del Agente 5**](#_c0qqg7i3feyz)

[**3.3.2 Grabación del vídeo 7**](#_pkjt9jtfv0aw)

[**3.3.3 Entrenamiento y selección del mejor modelo 8**](#_fgr30h431o4l)

[**3.3.3.1 Inicio y preparación 8**](#_omg32xcyu3en)

[**3.3.3.2 Barra de progreso 9**](#_19sw4frvrxiv)

[**3.3.3.3 Inicio y configuración del entrenamiento 9**](#_5venld3z8fkt)

[**3.3.3.4 Selección de estado 9**](#_cokq5uapd1zf)

[**3.3.3.5 COntrol de atascos 10**](#_l0jxkfue2mem)

[**3.3.3.6 Cálculo de la recompensa 10**](#_edtue27vv95v)

[**3.3.3.7 Guardado de la experiencia 10**](#_t8kqlch8vz4b)

[**3.3.3.7 Actualización del estado 11**](#_d1kjkv4jx0zk)

[**3.3.3.8 Actualización del progreso 11**](#_ci8955e102v8)

[**3.3.3.9 Actualización de la red objetivo 11**](#_6t0hqmt4dsz4)

[**3.3.3.10 Guardado del mejor modelo y episodio 12**](#_rxndbeq8mqv7)

[**3.3.3.11 Grabación final 13**](#_jzqypk9qvjn0)

[**3.3.3.12 Gráfico de la evolución 13**](#_t299osahqdq7)

[**3.4 Main 13**](#_jv91na9keu3e)

[**4. Ejecución 14**](#_hoduqpq8l0tu)

## **1. Objetivo**

El objetivo de esta tarea es modificar un cuaderno de Google Colab que contiene un agente de aprendizaje por refuerzo entrenado en el entorno CartPole utilizando Keras. Se deberá adaptar el código para que el agente aprenda a resolver el entorno mountainCarContinuous-v0.

🔗 Documentación del entorno: [MountainCarContinuous](https://www.gymlibrary.dev/environments/classic_control/mountain_car_continuous/)

## **2. Descripción del problema**

El entorno MountainCarContinuous consiste en un automóvil ubicado en un valle, que debe alcanzar la cima de una colina. Sin embargo, el motor del automóvil no es lo suficientemente potente como para llegar directamente a la cima, por lo que debe aprender a balancearse hacia adelante y hacia atrás para ganar impulso.

● Estado: Un vector de 2 valores (posición y velocidad del automóvil).

● Acción: Un valor continuo entre -1 y 1 que representa la fuerza aplicada al automóvil.

● Recompensa: Se otorgan recompensas cuando el automóvil alcanza la cima y se penaliza el movimiento ineficaz.

● Objetivo: El agente debe aprender una política óptima para alcanzar la cima en el menor tiempo posible.

## **3. Código**

### **3.1 Imports y future saving folders**

# Import required libraries

import gym

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers

import random

from collections import deque

import matplotlib.pyplot as plt

import os

import threading

import time

from tqdm import tqdm

# Set paths for saving videos and models

video\_folder = r"C:\\Users\\isard\\Desktop\\DQNMCC\\video"

model\_folder = r"C:\\Users\\isard\\Desktop\\DQNMCC\\models"

os.makedirs(video\_folder, exist\_ok=True)

os.makedirs(model\_folder, exist\_ok=True)

### **3.2 Preconfiguración**

#### **3.2.1 Pesos y ajustes del modelo**

# Configuration settings for training and model

config = {

'learning\_rate': 0.0005,

'gamma': 0.99,

'epsilon\_start': 1.0,

'epsilon\_end': 0.01,

'epsilon\_decay': 0.999,

'buffer\_size': 200000,

'batch\_size': 64,

'target\_update': 10,

'hidden\_size': 256,

'video\_interval': 10,

'max\_steps': 10000,

'steps\_per\_video': 2000,

'train\_interval': 4,

'position\_bins': 50,

'velocity\_bins': 50

}

#### **3.2.2 Discretización**

Esto permite usar un agente DQN, que solo funciona con acciones discretas, en un entorno continuo.

# Discretize the continuous action space into fixed values

DISCRETE\_ACTIONS = np.round(np.arange(-1.0, 1.1, 0.1), 2).reshape(-1, 1)

n\_actions = len(DISCRETE\_ACTIONS)

La forma .reshape(-1, 1) asegura que cada acción tenga el formato esperado por el entorno.

#### **3.2.3 Variables globales**

# Global variables

current\_model = None

is\_training = True

episode = 0

#### **3.2.4 Creación del entorno del video**

Creación del entorno para la visualización y guardado del vídeo.

**# Function to create the environment, optionally with video recording**

**def create\_env(record\_video=False, episode\_id=0):**

**env = gym.make("MountainCarContinuous-v0", render\_mode="rgb\_array")**

**if record\_video:**

**env = gym.wrappers.RecordVideo(**

**env,**

**video\_folder=video\_folder,**

**episode\_trigger=lambda ep: True,**

**name\_prefix=f"progress\_{episode\_id}"**

**)**

**return env**

#### **3.2.5 Definición de la entrada de la red neuronal**

Esta parte obtiene la dimensión del espacio de estados (state\_dim) del entorno MountainCarContinuous-v0.

Se crea y cierra el entorno temporalmente solo para leer la información.

# Determine the size of the state space

base\_env = create\_env()

state\_dim = base\_env.observation\_space.shape[0]

base\_env.close()

### 

#### **3.2.6 Construcción del modelo de red neuronal**

Esta función construye una red neuronal que estima los valores Q para cada acción discreta. Se entrena con el optimizador Adam y pérdida MSE.

# Build the neural network model used for Q-value approximation

def build\_model():

model = tf.keras.Sequential([

layers.Input(shape=(state\_dim,)),

layers.BatchNormalization(),

layers.Dense(config['hidden\_size'], activation="relu"),

layers.Dense(config['hidden\_size'], activation="relu"),

layers.Dense(config['hidden\_size'], activation="relu"),

layers.Dense(n\_actions) # Output one Q-value for each action

])

model.compile(

optimizer=tf.keras.optimizers.Adam(learning\_rate=config['learning\_rate']),

loss="mse"

)

return model

#### **3.2.6 Discretización para la entrada de DQN**

Esta función convierte el estado continuo (posición y velocidad) en índices discretos.

Usa bins para dividir ambos valores en rangos y asignar cada uno a una categoría. Esto facilita el uso del estado como entrada para el modelo DQN.

# Convert continuous state to discrete indices for easier processing

def discretize\_state(state):

position, velocity = state

position\_bins = np.linspace(-1.2, 0.6, config['position\_bins'])

velocity\_bins = np.linspace(-0.07, 0.07, config['velocity\_bins'])

position\_idx = np.digitize(position, position\_bins) - 1

velocity\_idx = np.digitize(velocity, velocity\_bins) - 1

return np.array([position\_idx, velocity\_idx])

### **3.3 Agente DQN**

#### **3.3.1 Declaración del Agente**

Esta clase define al agente DQN, que entrena una red para estimar valores Q y elegir acciones.

Y el método train actualiza la red con muestras aleatorias del buffer.

# Deep Q-Network (DQN) agent class

class DQNAgent:

def \_\_init\_\_(self):

self.model = build\_model()

self.target\_model = build\_model()

self.update\_target()

self.gamma = config['gamma']

self.epsilon = config['epsilon\_start']

self.epsilon\_decay = config['epsilon\_decay']

self.epsilon\_min = config['epsilon\_end']

self.replay\_buffer = deque(maxlen=config['buffer\_size'])

self.batch\_size = config['batch\_size']

self.target\_update\_counter = 0

self.min\_buffer\_size = config['buffer\_size'] // 2

self.train\_counter = 0

# Synchronize weights of target model with the main model

def update\_target(self):

self.target\_model.set\_weights(self.model.get\_weights())

# Epsilon-greedy action selection

def get\_action(self, state):

if np.random.rand() < self.epsilon:

return random.randint(0, n\_actions - 1)

discrete\_state = discretize\_state(state)

q\_values = self.model.predict(np.array([discrete\_state]), verbose=0)[0]

return np.argmax(q\_values)

# Store experience in replay buffer

def remember(self, state, action, reward, next\_state, done):

discrete\_state = discretize\_state(state)

discrete\_next\_state = discretize\_state(next\_state)

self.replay\_buffer.append((discrete\_state, action, reward, discrete\_next\_state, done))

# Sample a batch and train the model

def train(self):

if len(self.replay\_buffer) < self.batch\_size:

return

if self.train\_counter % config['train\_interval'] != 0:

return

batch = random.sample(self.replay\_buffer, self.batch\_size)

states = np.array([x[0] for x in batch])

actions = np.array([x[1] for x in batch])

rewards = np.array([x[2] for x in batch])

next\_states = np.array([x[3] for x in batch])

dones = np.array([x[4] for x in batch])

current\_q\_values = self.model.predict(states, verbose=0)

next\_q\_values = self.target\_model.predict(next\_states, verbose=0)

max\_next\_q\_values = np.max(next\_q\_values, axis=1)

targets = current\_q\_values.copy()

for i in range(self.batch\_size):

if dones[i]:

targets[i][actions[i]] = rewards[i]

else:

targets[i][actions[i]] = rewards[i] + self.gamma \* max\_next\_q\_values[i]

self.model.fit(states, targets, epochs=1, verbose=0)

if self.epsilon > self.epsilon\_min:

self.epsilon \*= self.epsilon\_decay

self.train\_counter += 1

#### **3.3.2 Grabación del vídeo**

Esta función corre en un hilo separado y graba videos del agente cada ciertos episodios. Simula un episodio usando el modelo actual, guarda el video y el modelo. Así se puede ver visualmente el progreso del entrenamiento.

# Thread function to record agent progress periodically

def record\_progress\_video():

video\_counter = 0

last\_episode = 0

global episode

while is\_training:

try:

if current\_model is not None and (episode - last\_episode >= config['video\_interval']):

test\_env = create\_env(record\_video=True, episode\_id=f"progress\_{video\_counter}")

obs = test\_env.reset()[0]

done = False

if not hasattr(current\_model, 'predict\_function'):

current\_model.make\_predict\_function()

while not done:

discrete\_obs = discretize\_state(obs)

q\_values = current\_model.predict(np.array([discrete\_obs]), verbose=0)[0]

action\_idx = np.argmax(q\_values)

action = DISCRETE\_ACTIONS[action\_idx]

obs, \_, done, \_, \_ = test\_env.step(action)

test\_env.close()

current\_model.save(f"{model\_folder}/model\_video\_{video\_counter}.h5")

video\_counter += 1

last\_episode = episode

time.sleep(1)

except Exception:

time.sleep(5)

continue

#### **3.3.3 Entrenamiento y selección del mejor modelo**

Entrenamiento del agente DQN en un bucle principal. Interactúa con el entorno, almacena experiencias, entrena el modelo con ellas y guarda el mejor modelo. También graba videos periódicamente y grafica la evolución de recompensas.

##### **3.3.3.1 Inicio y preparación**

Inicialización del agente y carga un modelo si ya existe. También lanza un hilo para grabar videos del progreso. Prepara variables para registrar recompensas y controlar cuándo guardar modelos.

# Main training loop for the agent

def train\_agent():

global episode, current\_model, is\_training

print(" DQN training Start.")

# Load model if exists

if os.path.exists(f"{model\_folder}/best\_dqn\_model.h5"):

agent = DQNAgent()

agent.model = tf.keras.models.load\_model(f"{model\_folder}/best\_dqn\_model.h5")

agent.target\_model = tf.keras.models.load\_model(f"{model\_folder}/best\_dqn\_model.h5")

else:

agent = DQNAgent()

current\_model = agent.model

video\_thread = threading.Thread(target=record\_progress\_video, daemon=True)

video\_thread.start()

rewards\_history = []

best\_reward = float('-inf')

reward\_threshold = 90.0

window\_size = 100

save\_interval = 10

##### **3.3.3.2 Barra de progreso**

Crea una barra de progreso con tqdm para visualizar el avance del entrenamiento del agente.

# Progress bar for training

pbar = tqdm(

total=config['max\_steps'] \* 1000,

desc="Total Training",

position=0,

leave=True,

ncols=100,

bar\_format='{l\_bar}{bar}| {n\_fmt}/{total\_fmt} [{elapsed}<{remaining}] {postfix}'

)

##### **3.3.3.3 Inicio y configuración del entrenamiento**

Inicia el entrenamiento, configurando variables como el número de episodios y pasos totales. Luego, entra en un ciclo de entrenamiento.

episode = 1

total\_steps = 0

while is\_training:

train\_env = create\_env()

state = train\_env.reset()[0]

total\_reward = 0

done = False

steps = 0

last\_position = state[0]

best\_position\_ep = state[0]

stuck\_counter = 0

##### **3.3.3.4 Selección de estado**

En cada paso, el agente selecciona una acción usando epsilon-greedy. Luego, extrae la posición y velocidad del nuevo estado.

# Per-step interaction

while not done:

action\_idx = agent.get\_action(state)

action = DISCRETE\_ACTIONS[action\_idx]

next\_state, reward, done, \_, \_ = train\_env.step(action)

position = next\_state[0]

velocity = next\_state[1]

##### **3.3.3.5 COntrol de atascos**

Si la posición no cambia significativamente durante varios pasos (menos de 0.001), se incrementa un contador. Si este contador supera 100, se termina el episodio.

# Early stop if stuck

if abs(position - last\_position) < 0.001:

stuck\_counter += 1

if stuck\_counter > 100:

done = True

break

else:

stuck\_counter = 0

##### **3.3.3.6 Cálculo de la recompensa**

Se calcula una recompensa modificada que combina diferentes factores: la altura (distancia desde el objetivo), la velocidad, un bono por alcanzar el objetivo (si la posición es mayor a 0.5), una penalización por tiempo y un bono por superar la mejor posición lograda previamente. Todo esto se suma a la recompensa original.

# Reward shaping

height\_reward = abs(position)

velocity\_reward = abs(velocity)

goal\_reward = 100 if position >= 0.5 else 0

time\_penalty = -0.05

progress\_reward = 20 if position > best\_position\_ep else 0

best\_position\_ep = max(best\_position\_ep, position)

modified\_reward = reward + height\_reward + 0.5 \* velocity\_reward + goal\_reward + time\_penalty + progress\_reward

##### 

##### **3.3.3.7 Guardado de la experiencia**

Se guarda la experiencia (estado, acción, recompensa, siguiente estado y si terminó) en el buffer de memoria del agente. Si el tamaño del buffer alcanza el mínimo requerido, se entrena al agente con una muestra aleatoria del buffer

# Store experience and train

agent.remember(state, action\_idx, modified\_reward, next\_state, done)

if len(agent.replay\_buffer) >= agent.min\_buffer\_size:

agent.train()

##### **3.3.3.7 Actualización del estado**

Se actualiza el estado del agente con el siguiente estado, acumulando la recompensa total obtenida hasta el momento.

También se incrementan los contadores de pasos y el contador total de pasos.

# Update state

state = next\_state

total\_reward += modified\_reward

steps += 1

total\_steps += 1

last\_position = position

##### **3.3.3.8 Actualización del progreso**

Se actualiza la barra de progreso, mostrando el número de episodio, la posición actual, la recompensa total acumulada y la mejor posición alcanzada en el episodio.

# Update progress bar

pbar.update(1)

pbar.set\_postfix({

'Ep': f'{episode}',

'Pos': f'{position:.3f}',

'Rew': f'{total\_reward:.2f}',

'BestPos': f'{best\_position\_ep:.3f}'

})

if steps >= config['max\_steps']:

done = True

##### **3.3.3.9 Actualización de la red objetivo**

Se actualiza periódicamente la red objetivo del agente, sincronizándola con la red principal después de un número específico de pasos, y se restablece el contador de actualización de la red objetivo.

# Update target network periodically

agent.target\_update\_counter += 1

if agent.target\_update\_counter >= config['target\_update']:

agent.update\_target()

agent.target\_update\_counter = 0

current\_model = agent.model

##### **3.3.3.10 Guardado del mejor modelo y episodio**

Se guarda el historial de recompensas y el mejor modelo y si el promedio de recompensas supera un umbral, se guarda el modelo final.

También se guarda el modelo con la mejor recompensa en cada episodio y se realiza un guardado periódico cada ciertos episodios.

# Save reward history and best model

rewards\_history.append(total\_reward)

if len(rewards\_history) >= window\_size:

avg\_reward = np.mean(rewards\_history[-window\_size:])

if avg\_reward >= reward\_threshold:

agent.model.save(f"{model\_folder}/final\_model\_threshold\_reached.h5")

break

if total\_reward > best\_reward:

best\_reward = total\_reward

agent.model.save(f"{model\_folder}/best\_dqn\_model.h5")

current\_model = agent.model

if episode % save\_interval == 0:

try:

agent.model.save(f"{model\_folder}/model\_episode\_{episode}.h5")

except:

pass

train\_env.close()

episode += 1

except KeyboardInterrupt:

# Handle user interruption gracefully

is\_training = False

agent.model.save(f"{model\_folder}/final\_model\_interrupt.h5")

pbar.close()

##### **3.3.3.11 Grabación final**

Después de finalizar el entrenamiento, se graba un video del rendimiento final del agente en el entorno. El agente toma decisiones basadas en el modelo entrenado, y se guarda el video del episodio final.

# Final video recording after training ends

final\_env = create\_env(record\_video=True, episode\_id="final")

obs\_final = final\_env.reset()[0]

final\_done = False

while not final\_done:

discrete\_obs = discretize\_state(obs\_final)

q\_values = agent.model.predict(np.array([discrete\_obs]), verbose=0)[0]

act\_idx = np.argmax(q\_values)

act = DISCRETE\_ACTIONS[act\_idx]

obs\_final, \_, final\_done, \_, \_ = final\_env.step(act)

final\_env.close()

##### **3.3.3.12 Gráfico de la evolución**

Se genera y guarda un gráfico que muestra la evolución de las recompensas obtenidas por el agente a lo largo de los episodios.

# Plot and save the reward history

plt.figure(figsize=(10, 5))

plt.plot(rewards\_history)

plt.xlabel("Episode")

plt.ylabel("Reward")

plt.title("C.M.C. With Discretized Actions")

plt.grid(True)

plt.savefig(f"{model\_folder}/training\_rewards.png")

plt.show()

### **3.4 Main**

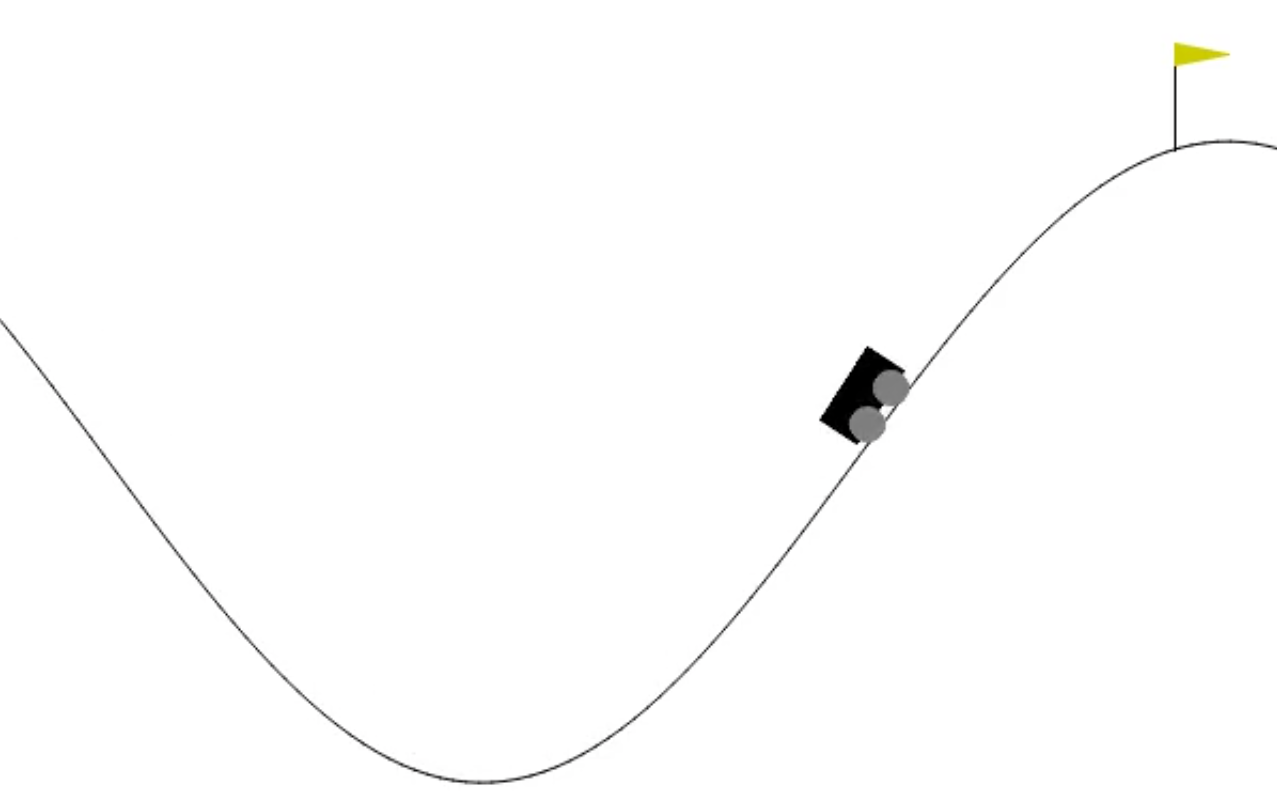
Este bloque asegura que la función train\_agent() se ejecute solo cuando el script sea ejecutado directamente.

# Run training when script is executed

if \_\_name\_\_ == "\_\_main\_\_":

train\_agent()

## **4. Ejecución**

Prueba de ejecución:  


Vídeo de esa ejecución:  
<https://drive.google.com/file/d/1HXcXpC1qA8v3m5J1WXvgP3WzZiShUGO2/view?usp=sharing>

Pese a no llegar al final en esa prueba es debido a la falta de entrenamiento, ya que isard se ha caído multiples veces, por lo que decidí reducir el tiempo de entrenamiento. Pero se ve como aprende y evoluciona con respecto al estado previo.