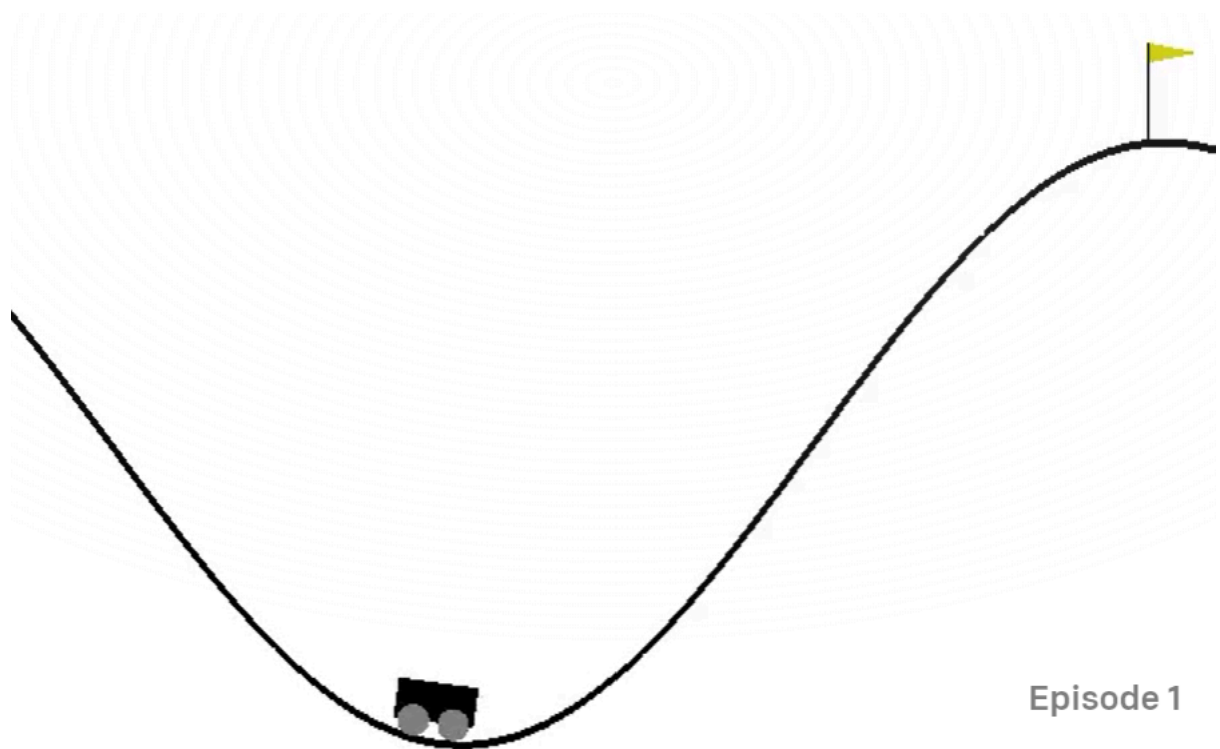


DQN MOUNTAIN CAR CONTINUOUS




Adrián Yared Armas de la Nuez

Contenido

1. Objetivo.....	1
2. Descripción del problema.....	2
3. Código.....	2
3.1 Imports y future saving folders.....	2
3.2 Preconfiguración.....	3
3.2.1 Pesos y ajustes del modelo.....	3
3.2.2 Discretización.....	3
3.2.3 Variables globales.....	3
3.2.4 Creación del entorno del video.....	4
3.2.5 Definición de la entrada de la red neuronal.....	4
3.2.6 Construcción del modelo de red neuronal.....	4
3.2.6 Discretización para la entrada de DQN.....	5
3.3 Agente DQN.....	5
3.3.1 Declaración del Agente.....	5
3.3.2 Grabación del vídeo.....	7
3.3.3 Entrenamiento y selección del mejor modelo.....	8
3.3.3.1 Inicio y preparación.....	8
3.3.3.2 Barra de progreso.....	9
3.3.3.3 Inicio y configuración del entrenamiento.....	9
3.3.3.4 Selección de estado.....	9
3.3.3.5 COntrol de atascos.....	10
3.3.3.6 Cálculo de la recompensa.....	10
3.3.3.7 Guardado de la experiencia.....	10
3.3.3.7 Actualización del estado.....	11
3.3.3.8 Actualización del progreso.....	11
3.3.3.9 Actualización de la red objetivo.....	11
3.3.3.10 Guardado del mejor modelo y episodio.....	12
3.3.3.11 Grabación final.....	13
3.3.3.12 Gráfico de la evolución.....	13
3.4 Main.....	13
4. Ejecución.....	14

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](#)

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()
```

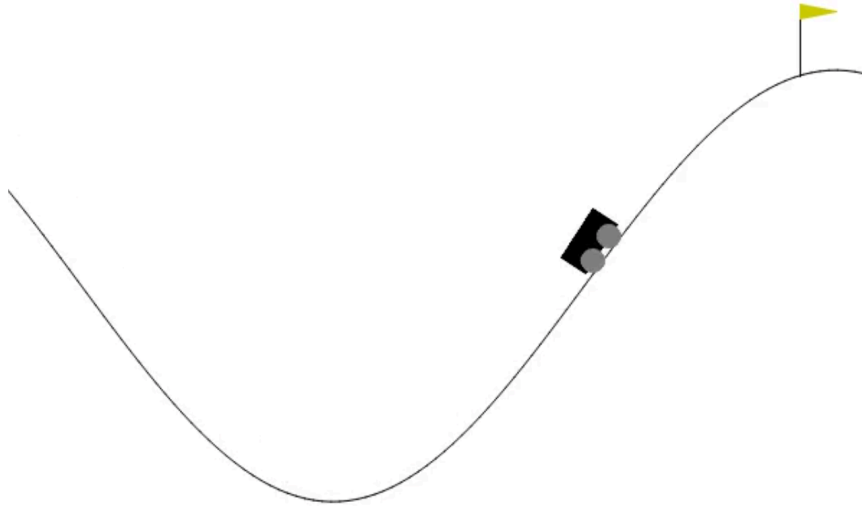
3.4 Main

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.