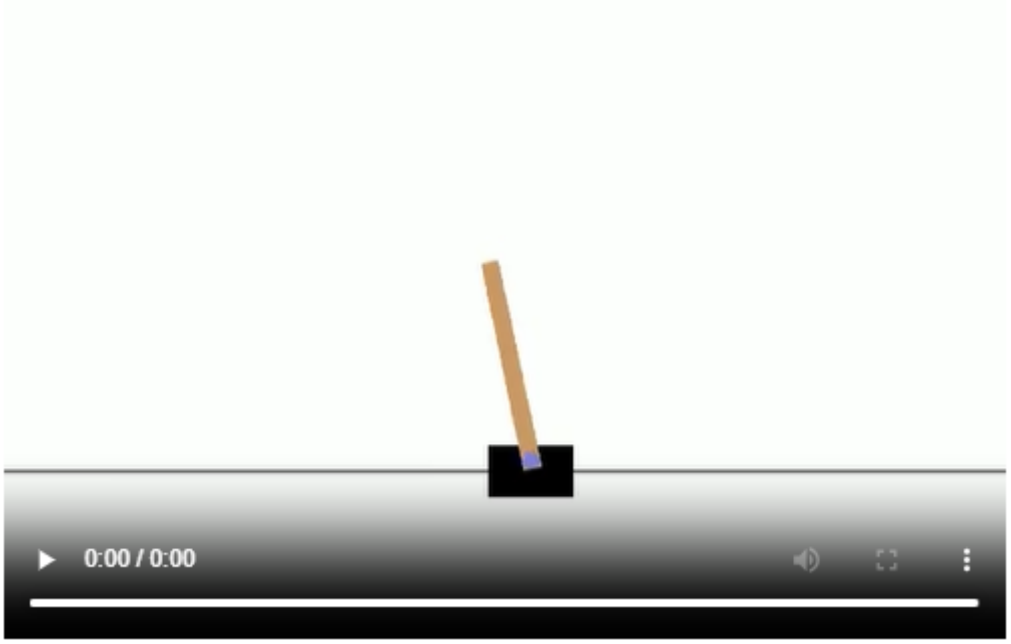
CARTPOLE en Google Colab con Keras



🔎 Explicación detallada del código

1️⃣ Instalación de dependencias en Google Colab

python

```
!sudo apt-get update --fix-missing
!sudo apt-get install -y xvfb ffmpeg
!pip install -U gym
!pip install pygame
!pip install keras
!pip install tensorflow
!pip install pyvirtualdisplay
```

📌 ¿Qué hace esto?

- `apt-get update --fix-missing`: Actualiza la lista de paquetes disponibles en Ubuntu y corrige errores en la descarga.

- **`apt-get install -y xvfb ffmpeg`**:
  - `xvfb`: Permite renderizar gráficos sin una pantalla física (necesario para Gym en Colab).
  - `ffmpeg`: Convierte y maneja archivos de video.
- **`pip install -U gym pygame keras tensorflow pyvirtualdisplay`**:
  - `gym`: Librería para crear y entrenar agentes de IA en entornos simulados.
  - `pygame`: Necesario para algunos entornos de Gym.
  - `keras` y `tensorflow`: Se utilizan para construir y entrenar la red neuronal.
  - `pyvirtualdisplay`: Permite mostrar entornos gráficos en Google Colab.

2️⃣ Habilitar el renderizado en Google Colab

python

```
from pyvirtualdisplay import Display
display = Display(visible=0, size=(400, 300))
display.start()

print("¡Virtual Display iniciado correctamente!")
```

📌 **¿Qué hace esto?**

- `pyvirtualdisplay` crea una pantalla virtual en segundo plano.
- Esto es necesario en Google Colab porque **no tiene una pantalla física para renderizar gráficos**.

3️⃣ Importación de librerías necesarias

python

```
import numpy as np
import pandas as pd
import gym
import random
import cv2
import base64
from collections import deque
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

```
from keras.callbacks import TensorBoard
from IPython.display import HTML
```

📌 **¿Para qué sirven estas librerías?**

- **`numpy`** y **`pandas`**: Para manipulación de datos y cálculos matemáticos.
- **`gym`**: Para crear y manejar el entorno de `CartPole-v1`.
- **`random`**: Se usa para decisiones aleatorias del agente.
- **`cv2`**: Para manipular imágenes y videos.
- **`base64`**: Para mostrar el video en Google Colab.
- **`keras`** (red neuronal):
  - **`Sequential`**: Para definir la arquitectura de la red.
  - **`Dense`**: Capas de la red neuronal.
  - **`Adam`**: Optimizador para mejorar el aprendizaje del agente.
- **`TensorBoard`**: Para visualizar métricas del entrenamiento.
- **`HTML`**: Para incrustar y mostrar videos en Google Colab.

4️⃣ Creación del agente de aprendizaje profundo (DQLAgent)

python

```
class DQLAgent():
    def __init__(self, env):
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n
        self.gamma = 0.95
        self.learning_rate = 0.001
        self.epsilon = 1.0
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01
        self.memory = deque(maxlen=1000)
        self.model = self.build_model()
```

📌 **¿Qué hace esto?**

- **`state_size`**: Número de valores en el estado actual del entorno (posición, velocidad, etc.).
- **`action_size`**: Número de acciones posibles (mover la barra a la izquierda o derecha).

- **`gamma`**: Factor de descuento, cuánto importa la recompensa futura.
- **`learning_rate`**: Velocidad de ajuste de la red neuronal.
- **`epsilon`**: Probabilidad de que el agente **explore** en lugar de **explotar**.
- **`epsilon_decay`**: Reduce `epsilon` gradualmente para que el agente explore menos con el tiempo.
- **`epsilon_min`**: Límite mínimo de `epsilon`.
- **`memory`**: Almacena experiencias pasadas para el entrenamiento.

4.1 Construcción de la red neuronal

python

```
def build_model(self):
    model = Sequential()
    model.add(Dense(48, input_dim=self.state_size, activation='tanh'))
    model.add(Dense(self.action_size, activation='linear'))
    model.compile(loss='mse', optimizer=Adam(learning_rate=self.learning_rate))
    return model
```

📌 ¿Qué hace esto?

- **Crea una red neuronal con dos capas**:
  - **Capa oculta (`Dense(48, activation='tanh')`)**:
    - `48` neuronas en la capa oculta.
    - `tanh`: Función de activación que ayuda a normalizar los valores de entrada.
  - **Capa de salida (`Dense(self.action_size, activation='linear')`)**:
    - Predice la mejor acción a tomar.
  - **Compilación**:
    - `loss='mse'`: Función de error cuadrático medio.
    - `optimizer=Adam(...)`: Optimizador para mejorar la precisión del modelo.

5️⃣ Entrenamiento del agente

python

```
episodes = 10
```

📌 ¿Qué hace esto?

- Define cuántos episodios (partidas) se jugarán.

python

```
for e in range(episodes):
    state = env.reset()
    if isinstance(state, tuple):
        state = state[0]
    state = np.reshape(state, [1, 4])
    time = 0
```

📌 **¿Qué hace esto?**

- **Inicializa el entorno** en cada episodio.
- **Convierte el estado en un formato adecuado** para la red neuronal.

python

```
while True:
    action = agent.act(state)
    next_state, reward, done, _, _ = env.step(action)
    next_state = np.reshape(next_state, [1, 4])
    agent.remember(state, action, reward, next_state, done)
    agent.replay(batch_size)
    agent.adaptiveEGreedy()
    state = next_state

    if done:
        print(f'Episode: {e}, Time: {time}')
        break
    time += 1
```

📌 **¿Qué hace esto?**

- El agente **toma una acción**, observa la recompensa y almacena la experiencia.
- **Ejecuta `replay()`** para entrenar la red neuronal con experiencias pasadas.
- **Reduce `epsilon`** para que el agente explore menos con el tiempo.

6️⃣ Grabación y visualización del video

python

```
!ffmpeg -i cartpole_video.mp4 -vcodec libx264
cartpole_video_fixed.mp4
```

📌 ¿Qué hace esto?

- Convierte el video a un formato compatible con Google Colab.

python

```
def display_video(video_path):
    video_file = open(video_path, "rb").read()
    video_url =
f"data:video/mp4;base64,{base64.b64encode(video_file).decode()}"
    return HTML(f'<video width="600" height="400" controls><source
src="{video_url}" type="video/mp4"></video>')

display_video("cartpole_video_fixed.mp4")
```

📌 ¿Qué hace esto?

- **Lee el archivo de video** y lo convierte a un formato que se puede incrustar en Colab.
- **Lo muestra dentro del notebook** en un reproductor HTML.

📌 Resumen Final

✅ Se instala el entorno necesario en Colab.
✅ Se entrena un agente usando Deep Q-Learning.
✅ Se graba el desempeño del agente.
✅ Se convierte y visualiza el video en Colab.

🚀 ¡Así has implementado un sistema de aprendizaje reforzado desde cero en Google Colab! 🎉

🔎 Explicación de `replay(self, batch_size)`

python
CopiarEditar

```
def replay(self, batch_size):
    if len(self.memory) < batch_size:
        return
    minibatch = random.sample(self.memory, batch_size)
    for state, action, reward, next_state, done in minibatch:
        target = reward if done else reward + self.gamma *
np.amax(self.model.predict(next_state, verbose=0)[0])
        train_target = self.model.predict(state, verbose=0)
        train_target[0][action] = target
        self.model.fit(state, train_target, verbose=0,
callbacks=[tensorboard_callback])
```

📌 Paso a paso del funcionamiento

1️⃣ Comprobar si hay suficientes experiencias en la memoria

python
CopiarEditar

```
if len(self.memory) < batch_size:
    return
```

✅ ¿Qué hace esto?

- Antes de entrenar, verifica si hay suficientes experiencias almacenadas.
- Si la memoria aún no tiene `batch_size` muestras, se **detiene la función** para evitar errores.

2️⃣ Seleccionar un minibatch aleatorio de experiencias

python
CopiarEditar

```
minibatch = random.sample(self.memory, batch_size)
```

✅ ¿Qué hace esto?

- Extrae `batch_size` experiencias **aleatorias** de la memoria (`self.memory`).
- Este **"aleatorización"** ayuda a que el aprendizaje sea más **generalizado** y no dependa solo de las últimas experiencias.

3️⃣ Procesar cada experiencia en el minibatch

python
CopiarEditar
```
for state, action, reward, next_state, done in minibatch:
```

✅ ¿Qué hace esto?

- **Recorre** cada experiencia almacenada en `minibatch`.
- Cada experiencia contiene:
  - `state` → Estado actual.
  - `action` → Acción tomada en ese estado.
  - `reward` → Recompensa obtenida tras tomar la acción.
  - `next_state` → Estado alcanzado después de la acción.
  - `done` → Si el episodio terminó (`True`) o no (`False`).

4️⃣ Calcular el valor objetivo (`target`)

python
CopiarEditar
```
target = reward if done else reward + self.gamma *
np.amax(self.model.predict(next_state, verbose=0)[0])
```

✅ ¿Qué hace esto?

Si el **episodio ha terminado** (`done=True`), el **objetivo** (`target`) es solo la recompensa obtenida (`reward`), porque ya no hay más estados a considerar.
python
CopiarEditar
```
target = reward
```

-

Si el **episodio sigue**, usa la **ecuación de Bellman**:
python
CopiarEditar
```
target = reward + self.gamma * max(Q'(next_state, a'))
```

-
  - **`np.amax(self.model.predict(next_state, verbose=0)[0])`**
    → Predice la mejor acción futura en el estado `next_state`.
  - **`self.gamma`**
    → Factor de descuento (cuánto influye la recompensa futura).

🔍 ¿Por qué usamos `max(Q'(next_state, a'))`?

- Porque **queremos que el agente aprenda a maximizar la recompensa futura**.
- Usamos la **mejor acción posible en `next_state`** para actualizar la estimación.

5️⃣ Ajustar el objetivo en la red neuronal

python
CopiarEditar

```
train_target = self.model.predict(state, verbose=0)
train_target[0][action] = target
```

✅ ¿Qué hace esto?

1. **Obtiene la predicción del modelo actual** para el estado `state`.
2. **Reemplaza el valor de la acción tomada** (`action`) con `target`.

🔍 ¿Por qué no cambiamos todo el `train_target`?

- Porque el modelo debe **seguir aprendiendo sobre otras acciones**.
- Solo ajustamos **la acción que realmente tomó el agente** en ese momento.

6️⃣ Entrenar la red neuronal con la nueva información

python
CopiarEditar

```
self.model.fit(state, train_target, verbose=0,
callbacks=[tensorboard_callback])
```

✅ ¿Qué hace esto?

- Ajusta los **pesos de la red neuronal** para mejorar las predicciones futuras.
- Usa `state` como entrada y `train_target` como salida esperada.
- El `callback tensorboard_callback` registra los datos de entrenamiento en **TensorBoard** para su visualización.

📌 Resumen

1. **Verifica que haya suficientes experiencias en la memoria.**

2. **Selecciona un minibatch aleatorio para entrenar.**
3. **Recorre cada experiencia en el minibatch.**
4. **Calcula el valor objetivo (`target`) usando la ecuación de Bellman.**
5. **Modifica la salida esperada de la red neuronal (`train_target`).**
6. **Ajusta los pesos de la red neuronal para mejorar el aprendizaje.**

🔬 ¿Por qué `replay()` es la función más importante?

✅ **Aprendizaje a partir de experiencias pasadas:**

- Guarda las experiencias en `self.memory` y **aprende de ellas más adelante**.
  ✅ **Rompe la correlación entre experiencias:**
- Selecciona experiencias **aleatorias** para evitar que el agente dependa demasiado de los eventos recientes.
  ✅ **Optimización de la ecuación de Bellman:**
- Usa `Q-learning` para mejorar la estrategia con cada iteración.
  ✅ **Reduce el `epsilon` gradualmente:**
- Al principio **explora** más, pero con el tiempo **explota** lo aprendido.

🚀 ¡Conclusión!

🎯 `replay(self, batch_size)` es el **corazón** del aprendizaje del agente.
🎯 Permite que el agente mejore su toma de decisiones **con cada episodio jugado**.
🎯 Usa la ecuación de Bellman para **predecir las mejores acciones futuras**.

💡 **¡Así funciona Deep Q-Learning en tu código! 🔥🚀**