Entrenamiento de un agente que juega Super Mario mediante RL y CNN

Adrián González Ortega

El siguiente trabajo trata sobre el desarrollo de un agente inteligente para el juego producido por Nintendo Super Mario Bros. En concreto, el objetivo era diseñar, implementar y entrenar un agente con el algoritmo de aprendizaje por refuerzo Q-learning, más concretamente Q-Network Profundo (DQN) y Doble Q-Network Profundo (DDQN).

Keywords— Super Mario Bros, DQN, DDQN, Q-learning, Aprendizaje por Refuerzo

1 INTRODUCCIÓN

En el aprendizaje por refuerzo, el agente interactúa con su entorno, realizando acciones y recibiendo recompensas por estas acciones. A través de este proceso de prueba y error, el agente adquiere información valiosa sobre su entorno y aprende a navegarlo de manera efectiva. Aquí están los componentes clave de este proceso:

- **El Agente**: Este es el personaje que estamos entrenando para resolver un problema específico. Aprende de sus experiencias, tanto de sus errores como de sus éxitos. En este caso, nuestro agente es Mario.

- **El Entorno**: Este es el espacio en el que el agente interactuará y aprenderá. En nuestro caso, el entorno es el nivel del juego en el que Mario se mueve, el primer nivel del primer mundo (1-1). El entorno lo proporciona OpenAI Gym, un conjunto de herramientas para desarrollar y comparar algoritmos de aprendizaje por refuerzo. Permite enseñar a los agentes desde caminar hasta jugar a juegos como el pong o el pinball. Gym es una interfaz de código abierto para tareas de aprendizaje por refuerzo.

- **Las Acciones**: Estas son todas las posibles acciones que el agente puede realizar. Son los medios a través de los cuales el agente interactúa con su entorno. Para Mario, estas acciones podrían incluir moverse hacia adelante o hacia atrás, saltar, agacharse, etc. En el caso particular de mi proyecto, sólo usaré RIGHT_ONLY, es decir Mario solo podrá ir a la derecha y saltar. Esto se hace para no saturar el espacio de acción con un gran número de acciones posibles, lo que dificultaría el aprendizaje del Agente.

- **Las Recompensas**: Este es el núcleo del aprendizaje por refuerzo. Cada vez que el agente realiza una acción, recibe una recompensa que puede ser positiva, negativa o cero. Las recompensas indican al agente si ha elegido una buena o mala acción, permitiéndole aprender de sus acciones y trabajar hacia su objetivo. Las recompensas vienen definidas en el método step de Gymnasium de OpenAI. Esta es su definición:

  La función de recompensa asume que el objetivo del juego es moverse lo más a la derecha posible (aumentar el valor x del agente) lo más rápido posible, sin morir.
  La recompensa r se calcula como r = v + c + d.

- v: la diferencia en los valores x del agente entre estados. En este caso, esto es la velocidad instantánea para el paso dado. Se calcula como v = x1 - x0, donde x0 es la posición x antes del paso y x1 es la posición x después del paso. Si el agente se mueve a la derecha, entonces v > 0. Si el agente no se mueve, entonces v = 0.

- c: la diferencia en el reloj del juego entre frames. La penalización evita que el agente se quede quieto. Se calcula como c = c0 - c1, donde c0 es la lectura del reloj antes del paso y c1 es la lectura del reloj después del paso. Si no avanza, entonces c = 0. Si avanza, entonces c < 0.

- d: una penalización por muerte que penaliza al agente por morir en un estado. Esta penalización anima al agente a evitar la muerte. Si el agente está vivo, entonces d = 0. Si el agente está muerto, entonces d = -15.

La recompensa se define en el rango (-15, 15).

No quería dejar este valor tal cual lo devuelve el entorno, así que en lugar de simplemente devolver la recompensa, añadiré o restaré recompensa basándome en ciertos criterios:

1. **Diferencia de puntuación**: Añado la diferencia entre la puntuación actual y la puntuación anterior, dividida por 40. Esto incentiva al agente a aumentar su puntuación.

2. **Final del juego**: Si el agente ha terminado el nivel (indicado por info['flag_get']), añado 350 a la recompensa. Esto incentiva al agente a completar el nivel. Si el agente no ha terminado el nivel (es decir, ha muerto antes de llegar al final), resto 50 de la recompensa. Esto incentiva al agente a evitar la muerte.

3. **Saltos altos**: Se da una recompensa adicional de 10 si la posición y del agente es mayor que 180. Esto lo añado ya que el agente suele quedarse muy atascado cuando la tubería es muy alta o el agujero es muy largo.

- **El Estado**: Esta es una captura en un determinado tiempo de juego. Para Mario, cada nueva posición que ocupa en el nivel se convierte en un nuevo estado. El estado inicial es donde Mario comienza el nivel, y el estado final es cuando completa el nivel. 1 estado = 4 frames consecutivos (para captar bien el movimiento).

Antes de ponerme a explicar el desarrollo es necesario sentar unas bases teóricas sobre las que se construirá este proyecto.

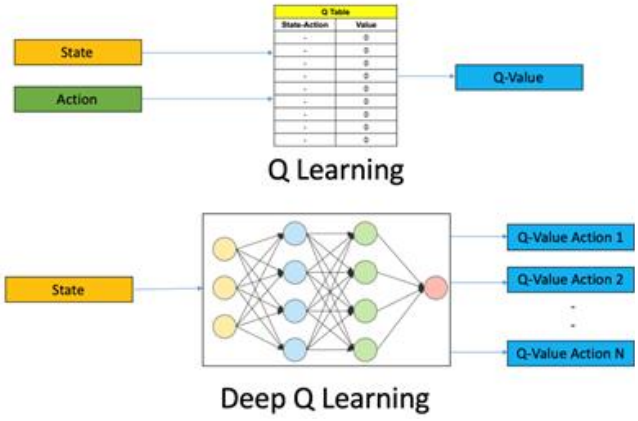

Fig. 1. Q Learning vs Deep Q Learning

El Q-Learning es un método ampliamente utilizado en el aprendizaje por refuerzo. Durante el proceso de entrenamiento, este algoritmo memoriza el estado inicial, la acción ejecutada y la recompensa obtenida, almacenando estos datos en una tabla. Sin embargo, este enfoque puede presentar desafíos cuando se aplica a videojuegos o entornos de gran escala. En estos casos, los datos de entrada son multidimensionales, dependiendo de los píxeles presentes en la imagen, lo que hace que el almacenamiento de datos en una tabla no sea sostenible. Para superar estas limitaciones, Deep Mind introdujo en 2015 el uso de Deep Learning, específicamente una red neuronal convolucional, para complementar el Q-Learning. Esta innovación permitió manejar eficientemente los datos multidimensionales y mejorar la escalabilidad del algoritmo.

DQN representa un avance significativo con respecto al enfoque tradicional de aprendizaje Q, especialmente en contextos complejos como Super Mario Bros. Uno de los retos a los que se enfrenta DQN es la gestión de los espacios de acción extremadamente grandes, típicos de los juegos más complejos. Para hacer frente a esta complejidad, DQN emplea una red neuronal profunda para aproximar la función de valor Q. La arquitectura de DQN está diseñada para recibir como entrada la representación del estado y devolver un vector de valores Q asociados a cada acción posible. El entrenamiento se realiza optimizando la diferencia entre los valores predicción y los valores reales, ayudando a mejorar la capacidad del agente para tomar decisiones inteligentes en el juego.

DDQN representa un paso más en el perfeccionamiento del rendimiento de DQN, centrándose en la mitigación del problema de la sobreestimación del valor Q. En el contexto de Super Mario Bros, la sobreestimación puede llevar a decisiones subóptimas, afectando negativamente al aprendizaje del agente. Para resolver este problema, DDQN introduce la distinción entre selección de acciones y evaluación, utilizando dos redes neuronales separadas, respectivamente, para estimar el valor objetivo y seleccionar la acción. Este enfoque reduce el error de sobreestimación, mejorando la estabilidad y eficacia del aprendizaje.

2 METODOLOGIA

Utilizaremos el primer nivel del juego Super Mario Bros. como entorno. Por defecto, la única observación es una imagen RGB de 240 x 256 píxeles, por lo que tenemos que escribir algunos wrappers para transformarla en una imagen en escala de grises con una resolución de 84 x 84 píxeles. Además, no todas las observaciones son útiles, por lo que utilizaremos sólo una de cada cuatro observaciones y las apilaremos.

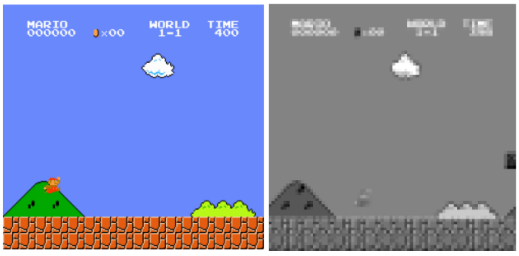

Fig. 2. Vista humano vs modelo

GrayScaleObservation: Wrapper para transformar una imagen RGB a escala de grises. Reduce el tamaño de la representación del estado sin perder información útil. Las acciones de Mario no dependen de que un Goomba sea marrón o de que una tubería sea verde.
Tamaño de estado: [1, 240, 256]

ResizeObservation: reduce las muestras de cada observación en una imagen cuadrada.
Nuevo tamaño: [1, 84, 84]

SkipFrame: Wrapper personalizado que skipea o junta 4 frames en uno e implementa la función step para agregar las recompensas acumuladas en cada fotograma skipeado. No se pierde información ya que no se varia mucho entre frames consecutivos.

FrameStack es un Wrapper que nos permite aplastar fotogramas consecutivos en un único punto de observación para alimentar nuestro modelo de aprendizaje. Así podemos identificar si Mario estaba aterrizando o saltando, basándonos en la dirección de su movimiento en los fotogramas anteriores.

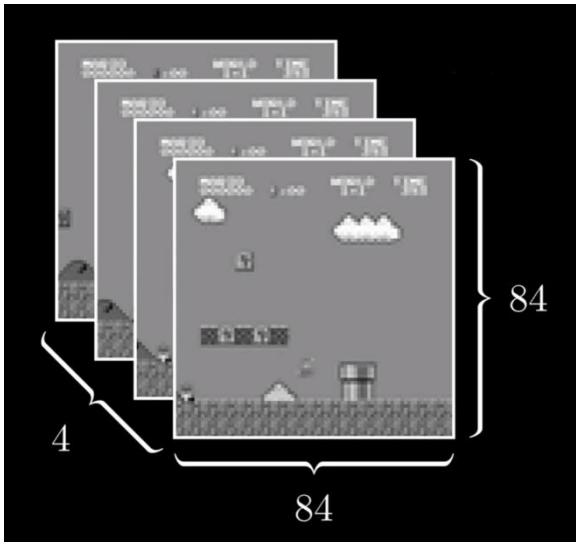

Fig. 3. Preprocesado de entorno

Nuevo tamaño: [4, 84, 84]

Después de preprocesar el entorno, creamos el Agente (Mario). Nuestro agente es capaz de:

1. **Inicializar**: Configura los parámetros iniciales, crea dos redes neuronales (una online y otra objetivo), define el optimizador y la función de pérdida, y establece un buffer de repetición. La red online se actualiza en cada paso de aprendizaje, mientras que la red objetivo se actualiza menos frecuentemente para proporcionar objetivos de aprendizaje más estables.
2. **Elegir una acción**: Basándose en la observación actual, elige una acción. Utilizo una política ε-greedy para seleccionar acciones. Con probabilidad ε, el agente selecciona una acción aleatoria para explorar el entorno. Con probabilidad 1-ε, el agente selecciona la acción que maximiza el valor Q predicho por la red online.
3. **Decaer epsilon**: Reduce el valor de epsilon multiplicándolo por un factor de decaimiento, hasta un mínimo especificado. Esto se utiliza para la exploración durante el aprendizaje.
4. **Almacenar en memoria**: Almacena la transición actual (estado, acción, recompensa, siguiente estado, hecho) en el buffer de repetición.
5. **Sincronizar redes**: Cada cierto número de pasos, copia los pesos de la red en línea a la red objetivo.
6. **Guardar y cargar el modelo**: Puede guardar los pesos actuales de la red en línea en un archivo y cargarlos desde un archivo.
7. **Aprender**: Si el buffer de repetición tiene suficientes muestras, realiza un paso de aprendizaje. Esto implica muestrear un lote del buffer de repetición, calcular los valores Q objetivo y predichos, calcular la pérdida, retropropagar el error, y actualizar los pesos de la red online. También sincronizo las redes si es necesario y se decae el epsilon.

Por último, hablemos de lo que hace que DQN O DDQN sea "profundo". Un estado puede considerarse como un grupo de 4 fotogramas consecutivos de 84×84 píxeles, y hay 5 acciones diferentes. Dado que hay 84×84×4×5 píxeles en un estado, cada píxel tiene una intensidad entre 0 y 255, y hay 5 acciones posibles para cada estado, la tabla Q que crearía tendría 5×256^84×84×4 valores. Es difícil almacenar una tabla Q tan grande. Para ello, utilizaremos una red neuronal convolucional para asignar un estado a sus valores de estado-acción para aproximar la tabla Q.

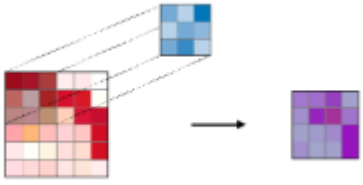

Fig. 4 Capa convolucional (CNN)

La red neuronal tiene la siguiente estructura:
entrada -> (conv2d + ReLU) x 3 -> Flatten -> (capa densa totalmente conectada + ReLU) -> capa densa totalmente conectada -> salida

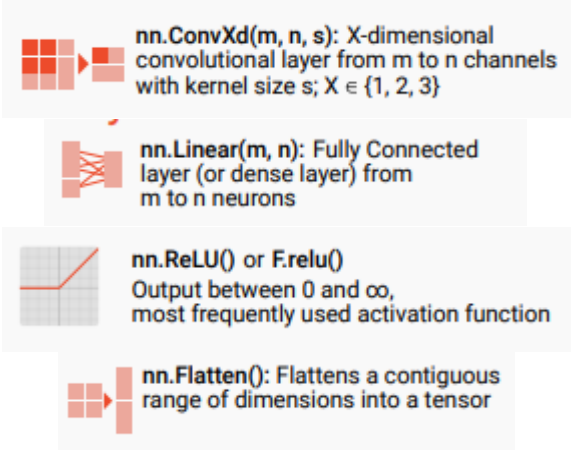

Fig. 5. PyTorch Cheatsheet [3]

3 EXPERIMENTS, RESULTATS I ANÀLISI

3.1 Análisis externo (Comparación DQN vs DDQN)

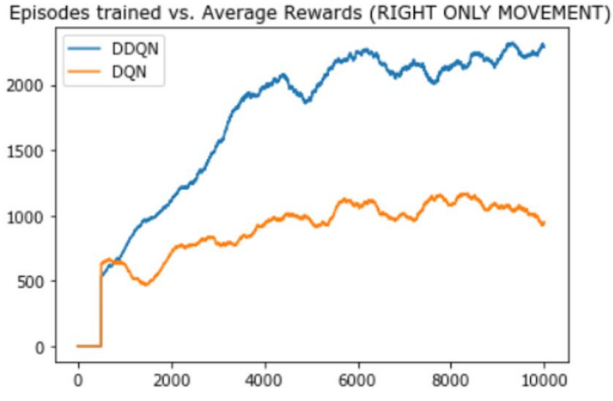

Fig. 6. Comparativa DQN vs DDQN [4]

En un primer momento, al iniciar esta pràctica, he querido comparar el rendimiento de los dos algoritmos DQN y DDQN, pero al hacer unas cuantas búsquedas online me ha quedado claro que el DDQN funciona mucho mejor. Por tanto, usaré este algoritmo para mis experimentos.

3.2 Análisis interno (Step rewards vs custom rewards)

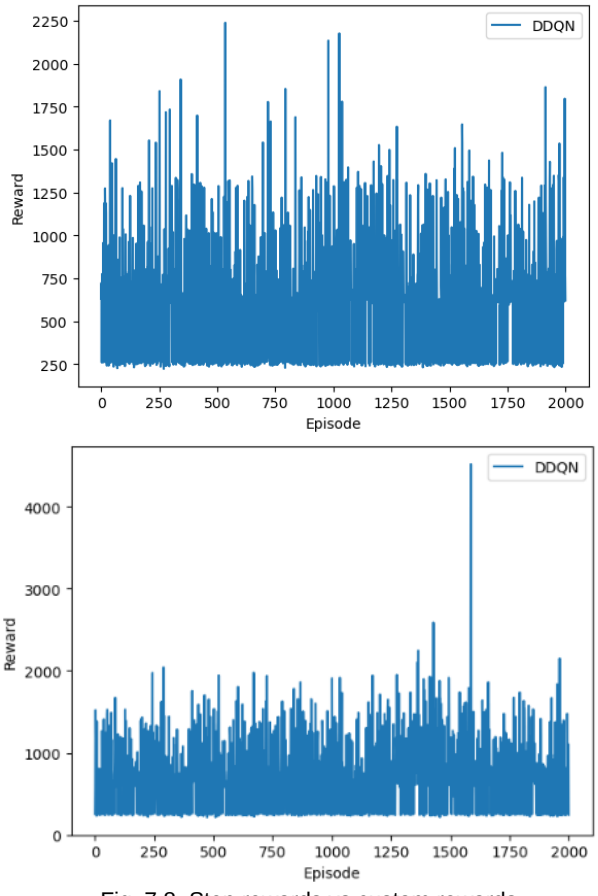

Fig. 7,8. Step rewards vs custom rewards

Estos dos entrenamientos han durado aproximadamente 12h cada uno y me ha servido para sacar unas ciertas conclusiones acerca de mi agente:

- 2000 episodios de entrenamiento no son suficientes para entrenar un buen agente. Según las referencias, son necesarios entre 10.000 y 30.000 episodios para que el agente empiece a ganar partidas con frecuencia.

- Las recompensas custom han tenido un efecto positivo en el desempeño del agente. Se puede observar de forma clara que el DDQN con recompensa custom ha logrado pasarse el nivel 1 vez en el episodio 1600 aprox. Además, se registra una ligera evolución positva en las recompensas del agente, cosa que en la primera gráfica no se percibe.

- El valor de épsilon decay escogido durante el entrenamiento es erróneo. Me he dado cuenta tarde de que al final de los episodios el epsilon seguía siendo 0.89, cuando debería bajar hasta aproximadamente 0.1. Esto lo que hace es que el agente está tomando demasiadas acciones random incluso cuando se acerca el final del entrenamiento, y esto no debería ser así.

Esstos son los hiperparámetros del entrenamiento:

- input_dims: *Tamaño de la entrada de la red neuronal.*
  - Valor: (4, 84, 84)
- num_actions: *Número de acciones que puede realizar el agente.*
  - Valor: 5
- lr: *Tasa de aprendizaje.*
  - Valor: 0.00025
- gamma: *Factor de descuento.*
  - Valor: 0.9
- epsilon: *Probabilidad de realizar una acción aleatoria.*
  - Valor: 1.0
- eps_decay: *Factor de decaimiento de epsilon.*
  - Valor: 0.9999998
- eps_min: *Valor mínimo de epsilon.*
  - Valor: 0.1
- replay_buffer_capacity: *Tamaño del buffer de memoria.*
  - Valor: 100000
- batch_size: *Tamaño del batch.*
  - Valor: 32
- sync_network_rate: *Número de pasos que deben transcurrir para sincronizar las redes.*
  - Valor: 1000
- use_DDQN: *Selección del algoritmo DDQN.*
  - Valor: True

He podido realizar un último entrenamiento (finalizado el 21/12/23 a las 14:38) con un valor de decay epsilon más correcto, haciendo una estimación teniendo en cuenta que con el anterior valor de 0.9999998 en 2000 ep me bajaba hasta 0.89, ahora baja hasta 0.28 al final del entrenamiento.

Los resultados son muchísimo mejores:

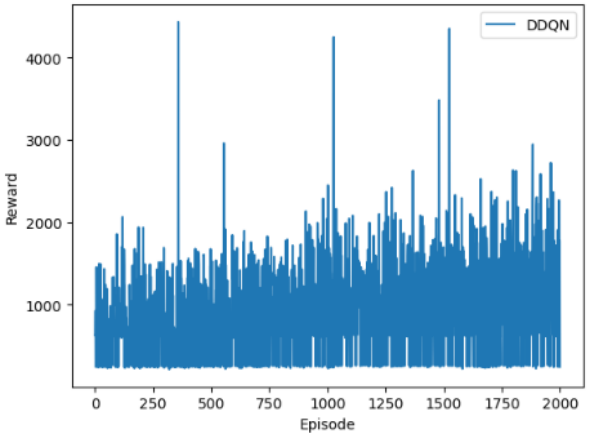

Fig. 9. Step rewards vs custom rewards

Ahora si se ve una clara mejoría en las recompensas de forma ascendente según el agente va sumando episodios. Además, se ha logrado ganar el nivel 3 veces.

input_dims	(4, 84, 84)
num_actions	5
lr	0.00025
gamma	0.9
epsilon	1.0
eps_decay	0.99999604821871
eps_min	0.1
replay_buffer_capacity	100000
batch_size	32
sync_network_rate	500
use_DDQN	True

5 CONCLUSIONES

Finalmente, puedo decir que me lo he pasado muy bien entrenando e implementando el agente de Mario. Siempre me ha gustado la aplicación del Machine Learning y de la Inteligencia Artificial en aspectos que van más allá de la clasificación. Sobretodo, me ha interesado mucho su aplicación en el ámbito social, y una de mis principales motivacions para esta cas Kaggle era mejorar mi conocimiento acerca del aprendizaje por refuerzo para en un futuro poder aplicarlo en un proyecto personal, Memor.IA, que utiliza un sistema de recomendación basado en el aprendizaje por refuerzo para personalizar la terapia de reminiscencia, proporcionando estímulos basados en los recuerdos del usuario con el objetivo de ayudar a prevenir y tratar el Alzheimer. Para saber más del proyecto, ver el vídeo [1] del apartado de Bibliografía.

Fig. 10. Aplicación Memor.IA.

Ahora hablando más acerca de la práctica, he de decir que me ha ayudado mucho a entender el concepto de aprendizaje profundo utilizando DQN Y DDQN, así como de las redes neuronales y la utilización de pytorch. Mi idea inicial era entrenar un agente en Zelda, mi saga favorita de videojuegos, pero no existe un entorno propio de OpenAI y además no está en Kaggle. Además, sería mucho más difícil ya que Mario es un juego simple en comparación. Pero con mis conocimientos actuales podría entrenar una IA para cualquiera de los juegos que proporciona OpenAI.

BIBLIOGRAFIA

[1] Adrián González, "Memor.IA", LinkedIn, 2023 https://www.linkedin.com/posts/adriangonzai_alzheimer-emprendimiento-uab-activity-7136444759140614144-vNjk?utm_source=share&utm_medium=member_desktop

[2] Stanford University, "CNN Cheatsheet", 2019

[3] Matthias Zürl, "PyTorch Cheatsheet", 2021

[4] De-Yu Chao, "Playing Super Mario Bros with Deep Reinforcement Learning", 2021 https://www.analyticsvidhya.com/blog/2021/06/playing-super-mario-bros-with-deep-reinforcement-learning/

[5] Dapeng Hong, "Super Mario Bros Game Playing with Deep Reinforcement Learning", 2018 http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/xwan/poster.pdf

[6] Sebastian Heinz, Using Reinforcement Learning to play Super Mario Bros on NES using TensorFlow, 2019 https://www.statworx.com/en/content-hub/blog/using-reinforcement-learning-to-play-super-mario-bros-on-nes-using-tensorflow/