UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

2021

Trabajo fin de grado

Aprendizaje por refuerzo profundo con OpenAI Gym

Mario García Pascual

Campus Internacional excelencia UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aprendizaje por refuerzo profundo con OpenAI Gym

Autor: Mario García Pascual
Tutor: Luis Fernando Lago Fernandez

junio 2021

Mario García Pascual
Aprendizaje por refuerzo profundo con OpenAI Gym

Mario García Pascual

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Me dijeron:
—O te subes al carro
o tendrás que empujarlo.

Ni me subí ni lo empujé.
Me senté en la cuneta
y alrededor de mí,
a su debido tiempo,
brotaron las amapolas.

Gloria Fuertes

There's an old joke.
Two elderly women are at a Catskills mountain resort,
and one of 'em says,
"Boy, the food at this place is really terrible."
The other one says,
"Yeah, I know, and such small portions."

Woody Allen

Agradecimientos

No podría haber dado ni un paso en esta carrera sin el apoyo incondicional de mis amigos Miguel, David, Mané, Sergio y Alejandro. A ellos van mis agradecimientos. Gracias también a mi tía Julia, por estar por ahí mirando, y a mis padres, mi hermana y mis primos, por el simple hecho de existir.

Resumen

El Aprendizaje por refuerzo profundo (DRL) surge de la inserción de métodos de Aprendizaje profundo (DL) en los algoritmos de Aprendizaje por refuerzo (RL). A pesar de los hitos logrados en este campo durante los últimos años, sigue ocupando un estatus de nicho en el panorama del Aprendizaje automático (ML), y apenas se ha nombrado durante el grado. El objetivo de este trabajo es partir de un estudio del RL clásico para terminar haciendo un estudio detallado de los principales algoritmos de DRL. Luego, hacemos una comparativa del rendimiento de los algoritmos en entornos de OpenAI Gym.

El primer algoritmo de DRL que estudiamos es Deep Q-Network (DQN), que logra fusionar por primera vez RL y DL con éxito. Luego, investigamos sus tres extensiones más conocidas: Double Deep Q-Network (DDQN), Dueling Network y Prioritized Experience Replay (PER). Finalmente, introducimos una familia distinta de algoritmos con el estudio de Advantage Actor-Critic (A2C), que trata de resolver el mismo problema con un enfoque diferente.

La comparativa la hacemos en cuatro entornos clásicos de OpenAI Gym y usando la librería Stable Baselines. Concluimos que, en los entornos sencillos que probamos, no se percibe la diferencia entre DQN y sus extensiones. Por último, comprobamos que las mejoras que introdujo DQN son relevantes, desactivándolas y viendo que no logra aprender.

Palabras clave

Aprendizaje por refuerzo profundo, Aprendizaje por refuerzo, Aprendizaje automático, Deep Q-Network, Advantage Actor-Critic, OpenAI Gym, Stable Baselines

Abstract

Deep Reinforcement Learning (DRL) arises from the insertion of Deep Learning (DL) methods into Reinforcement Learning (RL) algorithms. Despite the milestones achieved in this field during the last few years, it still occupies a niche status in the Machine Learning (ML) landscape, and has hardly been named during the degree. The aim of this work is to start from a survey of classical RL to finish with a detailed study of the main DRL algorithms. Then, we make a performance comparison of the algorithms in OpenAI Gym environments.

The first DRL algorithm we study is Deep Q-Network (DQN), which successfully merges RL and DL for the first time. Then, we investigate its three most popular extensions: Double Deep Q-Network (DDQN), Dueling Network and Prioritized Experience Replay (PER). Finally, we introduce a different family of algorithms with the study of Advantage Actor-Critic (A2C), which tries to solve the same problem with a different approach.

The comparison is done on four classical OpenAI Gym environments and using the Stable Baselines library. We conclude that, in the simple environments we tested, the difference between DQN and its extensions is not noticeable. Finally, we verify that the improvements introduced by DQN are relevant by disabling them and observing that it fails to learn.

Keywords

Deep Reinforcement Learning, Reinforcement Learning, Machine Learning, Deep Q-Network, Advantage Actor-Critic, OpenAI Gym, Stable Baselines

Índice

Listas

Lista de algoritmos

Lista de ecuaciones

Lista de figuras

1

Introducción

1.1. Objetivos

El objetivo de este *Trabajo de Fin de Grado* es hacer una introducción a *Reinforcement Learning* y *Deep Reinforcement Learning*, realizar un estudio teórico de los principales algoritmos de este último, y comparar su rendimiento usando las herramientas *OpenAI Gym* y *Stable Baselines*.

1.2. Motivación

En los últimos años hemos sido testigos del auge del *Deep Reinforcement Learning*. Periódicamente se han colado en nuestra vida noticias sobres *inteligencias artificiales* que dominan los juegos de Atari 2600 [1,2], que vencen a los mejores jugadores de Ajedrez y Go del mundo tan solo partiendo del conjunto de reglas del juego y aprendiendo a través del *self-play* [3,4], o que refutan complicadísimas conjeturas matemáticas sin la intervención de un humano [5].

El *Reinforcement Learning*, pese a considerarse uno de los principales subcampos de *Machine Learning*, casi nunca se estudia en las asignaturas correspondientes junto al *Supervised* y *Unsupervised Learning*, relegándolo a una posición de nicho. La curiosidad nos lleva a interesarnos por este campo. Partimos del *Reinforcement Learning* para llegar al campo que surje de su colaboración exitosa con el *Deep Learning*: *Deep Reinforcement Learning*. Estudiamos con detalle algunos de los algoritmos que hay detrás de los éxitos nombrados antes, como las *Deep Q-Networks* y sus extensiones, e introducimos otra familia de algoritmos conocida como *Policy Gradient*. Este estudio, recogido en el capítulo de Estado del Arte, ocupa la mayor parte del trabajo.

Una vez hecho el estudio teórico, nos interesa comprobar cómo funcionan estos algoritmos en la práctica, para añadirlos a nuestra caja de herramientas de resolución de problemas de Ingenieros Informáticos. Para ello, usamos los entornos de *OpenAI Gym* y las implementaciones de *Stable Baselines* para llevar a cabo experimentos que den cuenta de su rendimiento comparado.

1.3. Toma de decisiones bajo incertidumbre

El tema central que vamos a estudiar en este trabajo es **el problema de la toma de decisiones secuenciales bajo incertidumbre**. Martin L. Puterman empieza su libro *"Markov Decision Procesess: Discrete Stochastic Dynamic Programming"* [6] dando cuenta de este problema:

> *Each day people make many decisions; decisions which have both immediate and long-term consequences. Decisions must not be made in isolation; today's decision impacts on tomorrow's and tomorrow's on the next day's. By not accounting for the relationship between present and future decisions, and present and future outcomes, we may not achieve good overall performance.*

Nuestras acciones tienen consecuencias imprevisibles en el mundo, a corto y a largo plazo, que afectan a las decisiones del futuro, ¿cómo planificamos teniendo en cuenta esto? ¿cuál es la forma óptima de tomar decisiones?

1.3.1. De Programación dinámica a Aprendizaje por refuerzo

El célebre matemático Richard Bellman caracterizó en su libro *"Dynamic Programming"* [7] las estrategias de toma de decisiones (*policies* en inglés) **óptimas** con la siguiente frase:

> *An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

Esta observación es la clave de las ecuaciones de optimalidad de Bellman, que veremos en este trabajo. La Programación dinámica, Dynamic Programming (DP) en inglés, trata de resolver el problema de la toma de decisiones bajo incertidumbre usando estas ecuaciones y un **conocimiento perfecto de las dinámicas del entorno** en el que toma las decisiones el agente. Pero cuando el entorno es demasiado grande o complejo, o directamente no lo conocemos por completo, esta solución no nos vale.

El **Aprendizaje por refuerzo**, Reinforcement Learning (RL) en inglés, busca dar una solución aproximada al problema de la toma de decisiones bajo incertidumbre cuando no conocemos las dinámicas del entorno, pero sí podemos **interactuar** con él directamente, por ejemplo, mediante simulaciones. Llamamos agente al sujeto que toma las decisiones, o acciones, y recompensas al *feedback* de sus decisiones que le da el entorno, ya sea positivo o negativo.

1.3.2. Aprendizaje por refuerzo y Aprendizaje automático

Se suele explicar que el Aprendizaje automático, o Machine Learning (ML) , se divide en tres subcampos distintos, como aparece en la Figura 1.1: el Aprendizaje supervisado (SL), el Aprendizaje no-supervisado (UL), y el Aprendizaje por refuerzo (RL). Los dos primeros se suelen explicar en paralelo, mientras que el último se ve por separado.

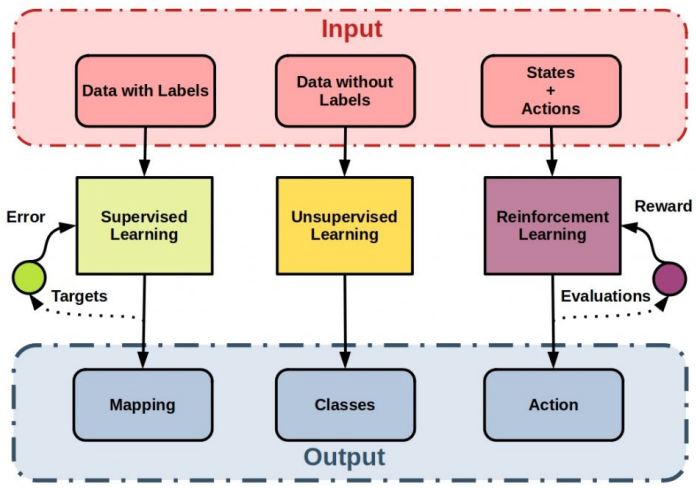

Figura 1.1: División del Aprendizaje automático en sus tres subcampos: Aprendizaje supervisado, Aprendizaje no-supervisado y Aprendizaje por refuerzo [1]

Esa separación tiene sentido, pues el Aprendizaje por refuerzo es ciertamente el *perro verde*. En RL desaparecen los conjuntos de datos: las interacciones con el entorno son lo más parecido a los datos que hay en este campo, y el agente puede generar tantos como quiera. Además, estos no vienen de una única distribución de probabilidad, sino que esa distribución depende de la manera en la que el agente tome las decisiones en el entorno. Los conceptos de *test set* y *validation set* no tienen sentido en RL, pues sólo hay un entorno, y el mismo entorno nos da una medida del rendimiento de las decisiones que toma el agente. La idea de *overfitting* sigue existiendo, pero pierde su posición central.

1.3.3. De Aprendizaje por refuerzo a Aprendizaje por refuerzo profundo

El primer gran éxito de RL fue **TD-Gammon**, el jugador artificial de backgammon desarrollado por Gerald Tesauro en 1992 [8], que logró un nivel de juego casi a la altura de los grandes maestros de la época. Al aprender a jugar mediante *self-play*, el programa usaba estrategias poco ortodoxas, que los humanos no habían considerado y no entendían, pero que funcionaban. En la Figura 1.2 vemos una imagen del programa.

TD-Gammon ya contenía algunos de los elementos que compartirían los futuros éxitos del campo,

[1] Fuente: `https://starship-knowledge.com/supervised-vs-unsupervised-vs-reinforcement`

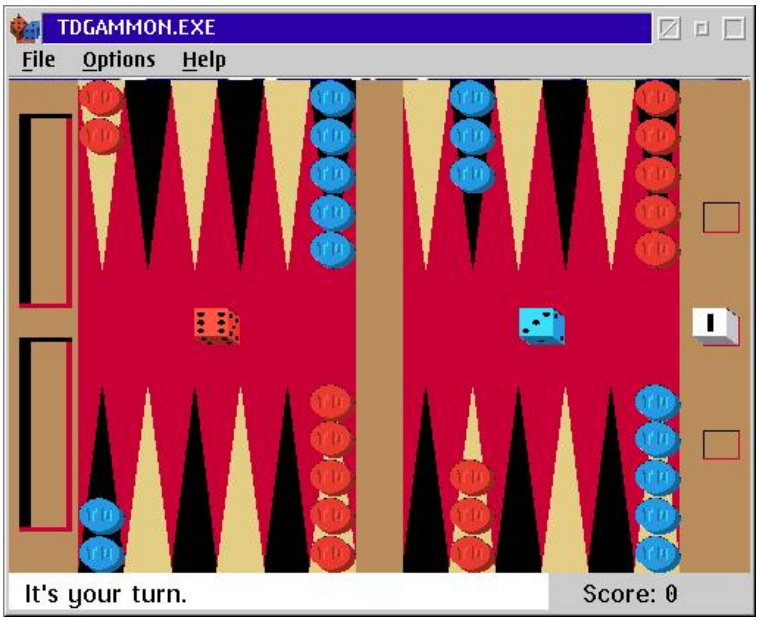

Figura 1.2: Imagen del programa TD-Gammon [2]

como el uso de redes neuronales (de una capa en el caso de TD-Gammon) y el *self-play*. Por desgracia, se intentó aplicar las técnicas de TD-Gammon a otros juegos pero fallaron, lo que llevó a pensar que su éxito se debía a las particularidades del juego de backgammon [1].

Más de 20 años después se publica el artículo *"Playing Atari with Deep Reinforcement Learning"* [1], del que hablaremos en este trabajo, donde se logra insertar con éxito técnicas de Aprendizaje profundo, Deep Learning (DL) en inglés, a los algoritmos de RL, dando lugar al campo del Aprendizaje por refuerzo profundo, Deep Reinforcement Learning (DRL) en inglés. El resultado es un jugador artificial de Atari 2600 que aprende a jugar usando como entrada los píxeles de la pantalla. Entre 2016 y 2017 se presentan los jugadores artificiales de Ajedrez y Go, AlphaZero y AlphaGo, que partiendo sólo de las reglas del juego y en 24 horas de *self-play* lograron alcanzar un nivel de juego superhumano [3, 4]. Grandes maestros de ambos juegos de todo el mundo elogian la creatividad y el ingenio de estas *inteligencias artificiales*.

En 2021 el campo sigue vibrante. En el mes de mayo, cuatro eminencias de RL publicaron *Reward is Enough"* [9], donde se establece la hipótesis de que la maximización de la recompensa acumulada, tal y como se entiende en RL, es suficiente para alcanzar la Inteligencia Artificial General (IAG) . Es uno de esos artículos que sin duda pasará a la historia del campo.

[2] Fuente: `https://www.os2world.com/games/index.php/native-games/board/114-os-2-td-gammon`

2

Estado del arte

2.1. Conceptos básicos de Aprendizaje por refuerzo

En la introducción hemos visto a grandes rasgos, de manera informal, en qué consiste y qué estudia el subcampo de Machine Learning conocido como Reinforcement Learning. En esta sección profundizaremos un poco más en el tema, y veremos:

- los procesos de decisión de Markov, Markov Decision Processes (MDP) en inglés, que componen el formalismo teórico de RL. Definiremos la estrategia de toma de decisiones del agente, o *policy* en inglés, y los importantes conceptos de:
  - función de valor para una política $V^{\pi}(s)$ y su versión óptima $V(s)$
  - función de acción-valor para una política $Q^{\pi}(s, a)$ y su versión óptima $Q(s, a)$
  - las relaciones que cumplen $V(s)$ y $Q(s, a)$, conocidas como *ecuaciones de optimalidad de Bellman*, que caracterizan las estrategias de toma decisiones óptimas
- el dilema Exploración *versus* Explotación
- la clasificación de las distintas clases de algoritmos en:
  - *Model-based* y *Model-free*
  - *On-policy* y *Off-policy*
  - *Value-based* y *Policy-based*
- tres algoritmos clásicos: Sarsa, Q-learning y Reinforce

2.1.1. Procesos de decisión de Markov

El modelo del entorno y del agente

El problema de la toma decisiones secuenciales bajo incertidumbre se puede modelizar con la estructura matemática conocida como *proceso de decisión de Markov* (MDP). La definición que vamos

a utilizar es la siguiente [10, 11]:

Definición 2.1.1. *Un proceso de decisión de Markov es una 4-tupla (S, A, R, P) formada por los siguientes elementos:*

1. *un conjunto no-vacío S llamado el* ***espacio de estados***
2. *un conjunto no-vacío A llamado el* ***espacio de acciones***
3. *una función real acotada $R\colon S \times A \to \mathbb{R}$, conocida como* ***función de recompensa esperada****. $R(s, a)$ es la recompensa que recibe el agente por tomar la acción a en el estado s*
4. *una función $P\colon S \times A \to \mathcal{P}(S)$, donde $\mathcal{P}(S)$ es el espacio de distribuciones de probabilidad sobre S. La llamamos* ***función de probabilidad de transición****. Para cada par (s, a) nos devuelve una distribución de probabilidad sobre los estados. Utilizamos*

$$P(j|s, a) \coloneqq P(s, a)(j) \in [0, 1], \quad s \in S, \quad a \in A,$$

para denotar el uso de la distribución de probabilidad $P(s, a) \in \mathcal{P}(S)$. $P(j|s, a)$ es la probabilidad de transitar al estado j habiendo tomado la acción a en el estado s.

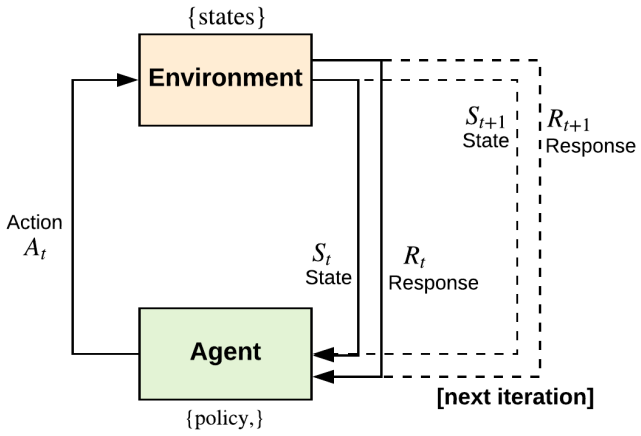

Figura 2.1: Esquema del ciclo básico de RL [1]

La forma de interpretar el modelo es la siguiente (Figura 2.1): el entorno se encuentra en el estado $s \in S$; el agente observa este estado y usa su estrategia de toma de decisiones para seleccionar una acción $a \in A$; el agente recibe una recompensa $r_1 = R(s, a) \in \mathbb{R}$; el siguiente estado del entorno se toma de la distribución de probabilidad $P(s, a)$, digamos que es $s' \in S$; y volvemos a empezar, esta vez partiendo desde el estado $s' \in S$. Si repetimos este paso indefinidamente, el agente obtiene una secuencia de recompensas $(r_1, r_2, r_3, \ldots)$. El objetivo del agente es asumir un estrategia de toma de

[1] Fuente: `https://towardsdatascience.com/the-exploration-exploitation-trade-off-7bc369027ba1`

decisiones que optimice, en algún sentido, esta secuencia de recompensas [2].

¿Cómo toma el agente la decisión sobre qué acción escoger? A continuación describimos el modelo del agente, contenido en el concepto de *política*.

Definición 2.1.2. *Una **política** es una función $\pi\colon S \to \mathcal{P}(A)$, que para cada estado $s \in S$ devuelve una distribución de probabilidad sobre las acciones. Escribimos*

$$\pi(s,a) \coloneqq \pi(s)(a) \in [0,1], \quad s \in S, \quad a \in A,$$

para denotar el uso de la distribución de probabilidad $\pi(s) \in \mathcal{P}(A)$. $\pi(s,a)$ es la probabilidad que tiene el agente de seleccionar la acción a en el estado s, y Π es el espacio de todas las políticas.

*Si la distribución de probabilidad es degenerada y se acumula en una sola acción $a \in A$ para cada estado $s \in S$, decimos que la política es **determinista** y la identificamos con la función $\mu\colon S \to A$ que asigna a cada estado $s \in S$ la acción escogida $a \in A$, tal que*

$$\pi(s, \mu(s)) = 1,$$

para todo $s \in S$.

La interpretación de la política de un agente con política $\pi \in \Pi$ es clara. El entorno se encuentra en el estado $s \in S$, entonces el agente elige una acción de acuerdo a la distribución de probabilidad $\pi(s)$, digamos $a \in A$. O, en el caso de que el agente siga una política determinista μ, toma directamente la acción $a = \mu(s)$.

A veces llamamos entorno $\mathcal{E}$ a un sistema que se puede modelizar con un proceso de decisión de Markov, y decimos que el agente interacciona con el entorno $\mathcal{E}$ siguiendo una política $\pi \in \Pi$. En RL, la unidad atómica de **interacción** es la transición (s, a, r, s') (o **experiencia**), formada por el estado $s \in S$ del que se parte, la acción $a \in A$ tomada por el agente, la recompensa $r = R(s,a)$ obtenida y el siguiente estado $s' \in S$ al que transitó el entorno.

Funciones de valor y Ecuaciones de optimalidad

Un proceso de decisión de Markov (S, A, R, P) junto con una política $\pi \in \Pi$ induce un espacio de probabilidad bien definido sobre las trayectorias $\tau = (s_1, a_1, s_2, \ldots)$, por tanto estamos legitimados a usar la medida de probabilidad $\mathbb{P}^\pi$ y la esperanza $\mathbb{E}^\pi$.

Sean X_t e Y_t las variables aleatorias que observan el estado y la acción a tiempo $t \in \mathbb{N}$, tal que

$$X_t(\tau) = s_t, \quad Y_t(\tau) = a_t,$$

[2] Existe una pluralidad de *criterios de optimalidad*. Nosotros usaremos el llamado *expected total discounted reward*. Para ver otros criterios, consúltese [6].

definimos la variable aleatoria de **ganancia** γ-descontada G, o recompensa total descontada, como

$$G(\tau) = \sum_{t=1}^{\infty} \gamma^{t-1} R(X_t(\tau), Y_t(\tau)),$$

con $\gamma \in [0, 1)$, conocido como el **factor de descuento** [3]. Es decir se suman las recompensas que ha obtenido el agente en cada tiempo, descontadas con el correspondiendo factor γ^{t-1}. Sobre esta variable aleatoria se define la **función de valor** para una política $\pi \in \Pi$, tal que

$$V^{\pi}(s) = \mathbb{E}^{\pi}[G|X_1 = s] \tag{2.1}$$

y, de un modo muy similar, la **función de acción-valor** para una política $\pi \in \Pi$,

$$Q^{\pi}(s, a) = \mathbb{E}^{\pi}[G|X_1 = s, Y_1 = a]. \tag{2.2}$$

Estas funciones representan el valor que da el agente a estar en un estado s usando la política π, y el valor de haber tomado la acción a en el estado s con la política π, respectivamente. Sobre estas funciones, se definen sus versiones **óptimas** tomando el supremo [4] sobre el espacio de políticas, tal que

$$V(s) = \sup_{\pi \in \Pi} V^{\pi}(s), \qquad Q(s, a) = \sup_{\pi \in \Pi} Q^{\pi}(s, a). \tag{2.3}$$

Estas funciones representan el valor más alto que puede obtener el agente usando cualquier política. Decimos que una política π es **óptima** si $V(s) = V^{\pi}(s)$ para todo $s \in S$. Estas funciones óptimas cumplen unas relaciones que reciben el nombre de **ecuaciones de optimalidad de Bellman**, o simplemente **ecuaciones de Bellman**, que son las siguientes

$$V(s) = \sup_{a \in A} Q(s, a), \qquad Q(s, a) = R(s, a) + \gamma \sum_{j \in S} P(j|s, a) V(j). \tag{2.4}$$

Si asumimos que el espacio de acciones A es **finito**, entonces estamos legitimados a usar cambiar el supremo de la primera ecuación por un máximo. A veces escribimos las dos ecuaciones en forma de esperanza y mostrando la recursividad de $V(s)$ y $Q(s, a)$ como

$$V(s) = \sup_{a \in A} \mathbb{E}_{s' \sim P(s,a)} \big[R(s, a) + \gamma V(s') \big], \tag{2.5}$$

y

$$Q(s, a) = \mathbb{E}_{s' \sim P(s,a)} \left[R(s, a) + \gamma \max_{a' \in A} Q(s', a') \right], \tag{2.6}$$

[3] El descuento γ nos dice cuánto valen para el agente las recompensas del futuro. En concreto, si el agente se encuentra en tiempo $t = 1$, una unidad (1) de recompensa en tiempo $t = 2$ vista desde tiempo $t = 1$ vale exactamente $\gamma < 1$, es decir, menos que una unidad en $t = 1$. Algo similar ocurre con el dinero en tiempos de inflación, que un euro hoy de vale más que uno de mañana.

[4] Ponemos supremo, y no máximo, porque el espacio de políticas Π no tiene suficiente estructura para asegurar que el supremo se alcanza para una política π^*.

donde el subíndice de la esperanza indica que el estado siguiente s' se disitribuye de acuerdo a $P(s, a)$.

2.1.2. Exploración versus Explotación

A diferencia del Supervised Learning, en RL el agente debe **explorar** el entorno explícitamente. Lo ejemplarizamos con uno de los problemas más sencillos en RL: los k-*armed bandits*. El agente está en una sala con k máquinas tragaperras y se le permite un número determinado de partidas. Cada tragaperras da una recompensa de 1 o 0 con una probabilidad distinta desconocida. El objetivo del agente es maximizar la ganancia. Imaginemos que empieza a jugar con la primera tragaperras y, tras unas partidas, observa que podría tener una probabilidad relativamente alta, ¿debe intentar explotar esto, o explorar otras tragaperras con la posibilidad de encontrar una mejor?

La respuesta depende de cuántas partidas pueda jugar el agente. En la práctica, la solución **heurística** más común para asegurar que el agente mantiene un cierto grado de exploración se conoce como ε***-greedy***. Sea $\varepsilon \in (0, 1)$, la estrategia ε-greedy consiste en tomar una acción aleatoria cualquiera con probabilidad ε, y la acción a la que en ese momento se asigna el mayor rendimiento el resto del tiempo [5]. Es decir, esta fuertemente inclinada hacia la explotación. Pero, pese a su simpleza, es el remedio más común para olvidarnos del complicado dilema de la Exploración *versus* Explotación.

2.1.3. Clasificación de algoritmos

Vamos a describir tres criterios distintos para la clasificación de algoritmos en RL: *Model-based* y *Model-free*, *On-policy* y *Off-policy*, y *Value-based* y *Policy-Based* [10, 11].

Model-based y *Model-free*

Cuando hablamos del problema de la toma de decisiones bajo incertidumbre, la incertidumbre viene de que no sabemos con certeza el siguiente estado $s' \in S$ al que transitará el sistema cuando tomamos la acción $a \in A$ desde el estado $s \in S$, y este estado afecta a la estrategia del agente.

El método *model-based* por antonomasia es la Programación dinámica (DP), donde se asume un conocimiento total de las dinámicas del entorno $\mathcal{E}$ por parte del agente, esto es, se conoce la función de probabilidad de transición $P(s'|s, a)$ y la función de recompensa esperada $R(s, a)$. En general, los métodos *model-based* son aquellos en los que, o bien se **conoce** el entorno, o bien se **aproxima** un modelo de este, lo que permite hacer **planificaciones** anticipando los futuros estados y recompensas. Si se tienen suficientes transiciones (s, a, r, s'), los valores de $P(s'|s, a)$ y $R(s, a)$ se pueden predecir

[5] Puesto en términos del refranero español, ε*-greedy* sólo escoge el *bueno por conocer* con probabilidad ε, optando el resto de veces por la seguridad del *malo conocido*.

mediante Supervised Learning.

Por el contario, los métodos *model-free* sólo conocen el entorno $\mathcal{E}$ **interactuando** con él a través de **simulaciones**. Todos los modelos que estudiamos en este trabajo pertenecen a esta clase.

Off-policy y *On-policy*

En el libro *Reinforcement Learning: An Introduction"* [10] de Richard S. Sutton y Andrew Barto, se define la distinción entre estas dos clases de algoritmos como

> *On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data.*

En los métodos *on-policy* la política que interactúa con el entorno y la política a mejorar son la misma. Los métodos *off-policy* no se tienen que ajustar a esto, y la experiencia de otros agentes interactuando con el entorno también se puede utilizar para mejorar la política. Esta diferencia un tanto críptica se ejemplificará en la siguiente sección con los algoritmos Sarsa y Q-learning, que son los métodos *on-policy* y *off-policy* por antonomasia.

Value-based y *Policy-based*

En los métodos *policy-based*, se aproxima la **política** $\pi(s, a)$ directamente, a través de una parametrización. Esta parametrización puede ser desde una aproximación lineal simple hasta una red neuronal. Estos métodos actualizan la política de modo iterativo hasta maximizar su rendimiento.

En cambio, en los métodos *value-based* se aproxima la **función de acción-valor** $Q(s, a)$, y se actualizan de modo iterativo usando reglas que se asemejan a las ecuaciones de optimalidad de Bellman 2.6. Una función de acción-valor $Q(s, a)$ representa implícitamente una política determinista μ, aquella que para cada estado toma la acción que la maximiza, es decir

$$\mu(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a). \tag{2.7}$$

Decimos que esta política es **codiciosa** porque toma siempre la acción que alcanza el valor máximo. En última instancia, ambos métodos necesitan una política para interactuar con el entorno. La diferencia es que en los métodos basados en políticas se aproxima directamente y en los métodos basados en valores se deriva de la función $Q(s, a)$.

2.1.4. Q-learning y Sarsa

Q-learning y Sarsa son dos de los algoritmos mas conocidos de RL. En la clasificación de algoritmos descrita en la sección anterior, ambos son métodos *model-free* y *value-based*, es decir, que no estiman un modelo del entorno $\mathcal{E}$ y operan aproximando la función de acción-valor $Q(s, a)$. Suponemos que los espacios de acciones y estados son **finitos**, de modo que $Q(s, a)$ se puede ver como una tabla donde las filas recorren los estados y las columnas las acciones. A esta situación se la conoce como RL tabular.

TD-error

Los dos algoritmos se dice que son TD-control (Temporal Difference), ya que ambos utilizan el concepto de TD-error en sus diseños. El TD-error se propuso originalmente para agentes que estimaban $V(s)$. Digamos que tenemos una transición (s, a, r, s') del estado s a s', entonces definimos el TD-error δ como

$$\delta = r + \gamma V(s') - V(s), \tag{2.8}$$

donde $\gamma \in [0, 1)$ es el factor de descuento. Si ahora consideramos transiciones entre dos pares estado-acción (s, a) y (s', a'), obtenemos una expresión similar para el TD-error pero para funciones de acción-valor $Q(s, a)$, tal que

$$\delta = r + \gamma Q(s', a') - Q(s, a). \tag{2.9}$$

Sarsa

El nombre de **Sarsa** viene de que la unidad atómica de experiencia que utiliza el algoritmo es la transición entre dos pares estado-acción (s, a, r, s', a'). Una vez conocidos estos 5 elementos, la regla que utiliza para actualizar la tabla $Q(s, a)$ es

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a)), \tag{2.10}$$

donde $\alpha \in (0, 1)$ es el factor de aprendizaje, y lo que va multiplicando es el TD-error 2.9. En un paso genérico del algoritmo, partimos de una transición (s, a, r, s') que viene del paso anterior; escogemos, pero no ejecutamos, a' siguiendo una estrategia ε-greedy; actualizamos la tabla con la regla 2.10, y, ahora sí, ejecutamos la acción en el entorno. Si reescribimos la ecuación 2.10 de esta manera

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(r + \gamma Q(s', a')),$$

vemos que la regla de actualización es un *soft-update* del antiguo valor $Q(s,a)$ con el *target* Sarsa $r+\gamma Q(s',a')$. El pseudocódigo del algoritmo se puede ver en 2.1.

Q-learning

Sarsa es un algoritmo *on-policy* ya que la acción a' que usa la regla de actualización es la que después se ejecuta en el entorno. **Q-learning** opera de forma distinta. Si partimos de una transición (s,a,r,s') en un paso genérico del algoritmo, Q-learning escoge, pero no ejecuta, la acción **codiciosa** $a^* = \operatorname{argmax}_{a'\in A} Q(s',a')$, y utiliza la misma regla de Sarsa. Como la acción escogida es de esta forma particular, la regla queda

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a'\in A} Q(s',a') - Q(s,a)), \tag{2.11}$$

que llamamos la regla de actualización de Q-learning. Una vez actualizada la tabla, se escoge y ejecuta una acción siguiendo la estrategia ε-greedy. Como la acción que usa la regla de actualización es **distinta** a la que después se ejecuta en el entorno, Q-learning es un algoritmo *off-policy*. El pseudocódigo del algoritmo se puede ver en 2.2.

input : Descuento γ, parámetro de aprendizaje α, parámetro exploración ε-greedy ε

Inicializar tabla $Q(s,a)$ para todos los pares estado-acción
for *episodio* $1,\dots,M$ **do**
  Inicializa el entorno y obtén el estado inicial s_0
  Escoge acción a_0 desde s_0 usando estrategia ε-greedy derivada de Q
  for $t = 0\dots,T$ **do**
    Ejecuta acción a_t y observa recompensa r_t y estado s_{t+1}
    Escoge acción a_{t+1} desde s_{t+1} usando estrategia ε-greedy derivada de Q
    $\delta \leftarrow r_t + \gamma Q(s'_{t+1}, a'_{t+1}) - Q(s_t, a_t)$
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \cdot \delta$
  end
end

Algoritmo 2.1: Pseudocódigo del algoritmo de on-policy TD-control Sarsa.

2.1.5. Reinforce

Motivación

Este es el primer algoritmo de RL *policy-based* que nos encontramos. La idea de algunos algoritmos de esta familia es **parametrizar** la política $\pi_\theta(s,a)$ con parámetros θ, de modo que la función que manda $\theta \mapsto \pi_\theta(s,a)$ sea diferenciable para cada $s\in S$ y $a\in A$. Si tenemos una noción $J(\pi_\theta)$ del rendimiento de una política con parámetros θ, podemos intentar optimizar este rendimiento directamente

input : Descuento γ, parámetro de aprendizaje α, parámetro exploración ε-greedy ε

Inicializar tabla $Q(s, a)$ para todos los pares estado-acción
for *episodio* $1, \ldots, M$ **do**
  Inicializa el entorno y obtén el estado inicial s_0
  for $t = 0 \ldots, T$ **do**
    Escoge acción a_t desde s_t usando estrategia ε-greedy derivada de Q
    Ejecuta acción a_t y observa recompensa r_t y estado s_{t+1}
    $\delta \leftarrow r_t + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t)$
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \delta$
  end
end

Algoritmo 2.2: Pseudocódigo del algoritmo de off-policy TD-control Q-learning.

desde el espacio de parámetros mediante un algoritmo de *gradient ascent* [6], tal que

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\pi_\theta), \tag{2.12}$$

donde η es un hiperparámetro de aprendizaje que indica cuántos nos movemos en la dirección del gradiente que acabamos de estimar [12].

Policy Gradient Theorem

En esta sección vamos a considerar la versión sin descuento de la ganancia, o recompensa acumulada, para simplificar la exposición, de modo que $G(\tau) = \sum_{t=1}^{\infty} r(X_t(\tau), Y_t(\tau))$. La única diferencia que provoca esto es que ahora desaparece el factor de descuento γ de todas las definiciones. Definimos el rendimiento de la política π_θ como la **recompensa total acumulada esperada**

$$J(\pi_\theta) = \mathbb{E}^{\pi_\theta}[G] = \mathbb{E}^{\pi_\theta}\left[\sum_{t=1}^{\infty} R(X_t, Y_t)\right]. \tag{2.13}$$

Para el rendimiento $J(\pi_\theta)$ definido de esta manera, tenemos a nuestra disposición el Policy Gradient Theorem (PGT) , que nos permite expresar el gradiente del rendimiento $\nabla_\theta J(\pi_\theta)$ de modo que lo podemos usar en nuestros algoritmos [13]. El teorema dice lo siguiente

Teorema 2.1.3. *El gradiente del rendimiento se puede expresar de forma exacta como*

$$g = \nabla_\theta J(\pi_\theta) = \mathbb{E}^{\pi_\theta}\left[\sum_{t=1}^{\infty} \nabla_\theta \log \pi_\theta(X_t, Y_t) Q^{\pi_\theta}(X_t, Y_t)\right]. \tag{2.14}$$

La prueba de este teorema se puede encontrar en [10]. Con el gradiente expresado de esta forma ya podemos describir el algoritmo Reinforce.

[6] Se trata de ascenso, y no descenso, porque queremos *maximizar* la recompensa acumulada, y por ello hemos de movernos en la dirección a la que apunta el gradiente.

Reinforce

El algoritmo Reinforce consiste en aplicar directamente la idea del ascenso por gradiente para maximizar la recompensa total acumulada, aplicando el PGT para estimar el gradiente y usando un estimador directo de la función de acción-valor $Q^{\pi_\theta}(s,a)$. Tenemos una política parametrizada $\pi_\theta(s,a)$, e interactuando con el entorno generamos un episodio de longitud T, $\tau = (s_1, a_1, r_1, s_2, \ldots, s_T)$ [7]. Entonces, estimamos el gradiente mediante

$$\hat{g} = \sum_{t=1}^{T-1} \nabla_\theta \log \pi_\theta(s_t, a_t) \sum_{t'=t}^{T-1} r_{t'}, \tag{2.15}$$

donde $\sum_{t'=t}^{T-1} r_{t'} = \hat{Q}_t$, a veces llamado *reward to go*, es un estimador insesgado de $Q^{\pi_\theta}(s_t, a_t)$. Una vez tenemos la estimación del gradiente $\hat{g}$, realizamos el ascenso por gradiente $\theta \leftarrow \theta + \eta\hat{g}$, y volvemos a empezar. El pseudocódigo del algoritmo se puede ver en 2.3.

input : Parámetro de aprendizaje η

1 Inicializar parámetro θ de la política
2 **for** *episodio* $1, \ldots...$ **do**
3 Genera episodio $\tau = (s_1, a_1, r_1, s_2, \ldots, s_T)$ usando la política π_θ
4 $\hat{g} \leftarrow \sum_{t=1}^{T-1} \nabla_\theta \log \pi_\theta(s_t, a_t) \sum_{t'=t}^{T-1} r_{t'}$
5 $\theta \leftarrow \theta + \eta\hat{g}$
6 **end**

Algoritmo 2.3: Pseudocódigo del algoritmo de Monte Carlo Policy Gradient Reinforce.

2.2. Métodos basados en Deep Q-Networks

En esta sección exponemos el algoritmo Deep Q-Network (DQN) [1, 2] y sus tres extensiones más famosas: Double Deep Q-Network (DDQN) [14], Dueling Network [15] y Prioritized Experience Replay (PER) [16]. Cada una afecta a una parte distinta del algoritmo original, y por tanto todas son compatibles. Como resumen de la sección:

- DQN aproxima la función acción-valor $Q(s,a)$ con una red neuronal. Utiliza una memoria de repetición, sobre la que se toman *mini-batches*, para entrenar la red con una función de pérdida inspirada en Q-learning
- DDQN cambia la función de pérdida que se utiliza para el algoritmo de optimización
- Dueling Network cambia la arquitectura de la red neuronal que aproxima la función acción-valor $Q(s,a)$

[7] Un episodio es una trayectoria finita, que termina porque el entorno ha llegado a un estado terminal.

- PER cambia la forma en la que se toman *mini-batches* de experiencias almacenadas en la memoria de repetición

Por último veremos Rainbow DQN [17], un modelo que consiste en unir todas estas extensiones (y alguna más), logrando el actual estado del arte entre los métodos basados en DQN. Se dará sentido a todos estos conceptos en las próximas páginas.

2.2.1. Deep Q-Network

Motivación

> *We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning.*

Así comienza el resumen del célebre artículo llamado *"Playing Atari with Deep Reinforcement Learning"* [1], publicado en 2013 por un equipo de investigadores de DeepMind (la filial de inteligencia artificial de Google) liderado por Volodymyr Mnih. En él se expone por primera vez el algoritmo conocido como DQN [8], responsable del hito de aprender políticas para siete juegos de Atari 2600 (Fig. 2.2) usando tan solo píxeles con un mínimo procesamiento como entrada, superando todos los modelos anteriores en seis de ellos. Dos años más tarde, el mismo equipo de investigadores extendido publica en *Nature "Human-level control through deep reinforcement learning"* [2], donde repiten este logro para 49 juegos de Atari 2600. Estos dos artículos dan el pistoletazo de salida al campo científico-tecnológico del Deep Reinforcement Learning, que no ha parado de cosechar éxito tras éxito desde entonces.

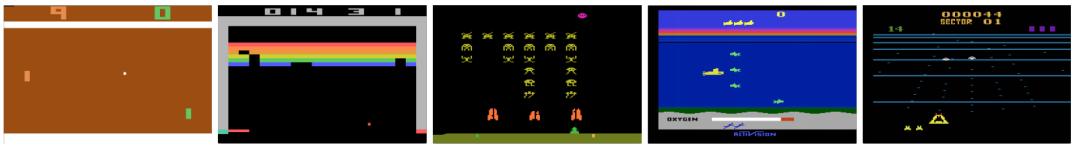

Figura 2.2: Capturas de pantalla de 5 juegos de la Atari 2600: (de izquierda a derecha) Pong, Breakout, Space Invaders, Seaquest, Beam Rider. Extraído del artículo *"Playing Atari with Deep Reinforcement Learning"* [1].

Las DQN son una extensión de Q-Learning que utilizan una red neuronal para aproximar la función acción-valor $Q(s, a)$, para luego usar un algoritmo de optimización como SGD para su entrenamiento. Curiosamente, esta no sería la principal aportación del artículo. A la fecha de su publicación ya existían métodos para el entrenamiento de redes neuronales en aprendizaje por refuerzo, pero estos eran, o bien demasiado inestables, o bien demasiado ineficientes para ser usados con redes neuronales

[8] La versión de DQN que aparece en el artículo [1] no es la misma que la del artículo [2],donde se introduce la idea de sincronizar los parámetros de la red objetivo separada cada C pasos del algoritmo, en lugar de cada paso.

grandes. Por ejemplo, antes hemos hablado de TD-gammon, el programa que logró un nivel de juego superhumano en backgammon en 1992 [18]. Pues este algoritmo ya utilizaba un Multilayer Perceptron (MLP) de una capa para aproximar la función de valor $V(s)$. La clave del artículo no es la idea de insertar métodos de DL en los algoritmos de RL, sino la combinación de técnicas que se utilizan para que esta inserción tenga éxito, para estabilizar el método y para hacerlo más eficiente.

La causa de la inestabilidad era de sobra conocida [1, 2]. En RL, si se modifica la política $\pi(s, a)$ de toma de decisiones del agente también se modifica la frecuencia con la que se visitan los estados y con ello la distribución los datos. Por esto, decimos que en RL la distribución de los datos es **no-estacionaria**. Por la naturaleza de Q-Learning, pequeños cambios en los valores $Q(s, a)$ pueden conllevar grandes cambios en esta distribución. Además, por la secuencialidad del proceso de aprendizaje, las transiciones no se dan de forma independiente sino que están **correlacionadas** unas con otras.

Esto implica que los datos sobre los que aplicamos el algoritmo de optimización no son ni **independientes** ni **idénticamente distribuidos**, que son las dos hipótesis estadísticas básicas que requieren la mayoría de estos algoritmos para su funcionamiento.

Primer intento

El siguiente algoritmo *ingenuo* que no funciona [2]. Vamos a aproximar la función de valor estado-acción con una red neuronal $Q(s, a|\theta)$ con parámetros θ. Entonces, interactuamos con el entorno $\mathcal{E}$ usando una estrategia ε-greedy, y, para cada transición (s, a, r, s') que observemos, usamos el algoritmo de optimización sobre los parámetros de la red con la función de coste:

$$L(\theta) = (r + \gamma \max_{a' \in A} Q(s', a'|\theta) - Q(s, a|\theta))^2.$$

Obsérvese el parecido de esta función de pérdida con la función de actualización de los Q-valores en Q-Learning. Como en el algoritmo clásico, se intenta que los Q-valores actuales se aproximen a los valores mejorados que predicen las **ecuaciones de optimalidad de Bellman**. En la función de pérdida se mira el error cuadrático entre dos valores: el valor predicho por la red neuronal $Q(s, a|\theta)$; y el valor objetivo $y = r + \gamma \max_a Q(s', a|\theta)$. El valor predicho depende, naturalmente, de los valores de la red, pero es extraño que el valor objetivo dependa también de ellos. En este algoritmo, cada transición (s, a, r, s') se utiliza una sola vez para actualizar los parámetros e inmediatamente se descarta.

Esta forma de insertar redes neuronales en un algoritmo de RL, quizás la más directa, no funciona. Es víctima de los dos problemas mencionados, la distribución no-estacionaria y los datos correlacionados.

input : Capacidad de memoria N, descuento γ, pasos para sincronizar red objetivo C, parámetro exploración ε-greedy ε

Inicializar replay memory D con capacidad N
Inicializar red Q con pesos aleatorios θ
Inicializar red objetivo $\hat{Q}$ con pesos $\theta^- = \theta$
for *episodio* $1, \ldots, M$ **do**
  Inicializa el entorno y obtén el estado inicial s_0
  for $t = 0 \ldots, T$ **do**
    Con probabilidad ε escoge acción aleatoria a_t;
    de lo contrario $a_t = \text{argmax}_a \, Q(s_t, a)$
    Ejecuta acción a_t y observa recompensa r_t y estado s_{t+1}
    Almacena transición (s_t, a_t, r_t, s_{t+1}) en D
    Toma un minibatch aleatorio (s_j, a_j, r_j, s_{j+1}) de D
    $y_j = \begin{cases} r_j & \text{si el episodio termina al paso } j+1 \\ r_j + \gamma \, \text{máx}_a \, \hat{Q}(s_{j+1}, a|\theta^-) & \text{en caso contrario} \end{cases}$
    Descenso gradiente sobre $(y_j - Q(s_j, a_j|\theta))^2$ con respecto a θ
    Sincroniza red objetivo $\hat{Q}$ cada C pasos
  end
end

Algoritmo 2.4: Pseudocódigo de Deep Q-Network estándar. Sobre este se construyen el resto de extensiones de Deep Q-Network.

Modelo

El algoritmo de DQN estándar es el algoritmo *ingenuo* que se acaba de explicar con dos mejoras clave:

- el uso de una memoria de repetición (*replay memory* en inglés) D, en la se almacenan las últimas N transiciones (s, a, r, s'), y que se utiliza para tomar *mini-batches* aleatorios sobre los que se utiliza el algoritmo de optimización.
- el uso de una red objetivo (*target network* en inglés) $Q(s, a|\theta^-)$ distinta a la red predictora (a veces llamada *online network*) $Q(s, a|\theta)$. Los pesos θ^- de esta red objetivo separada se sincronizan cada C pasos con la red principal, $\theta^- \leftarrow \theta$.

Con estas dos mejoras, la función de pérdida final [9] queda

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(D)} \left[(y^{DQN} - Q(s, a|\theta))^2 \right], \tag{2.16}$$

donde $\mathcal{U}(D)$ es la distribución uniforme sobre la memoria de repetición D, y

$$y^{DQN} = r + \gamma \max_{a' \in A} Q(s', a'|\theta^-) \tag{2.17}$$

[9] Siendo rigurosos, deberíamos indicar que hay una dependencia de la función de pérdida del paso del algoritmo, y que lo que estamos minimizando es una secuencia de funciones de pérdida L_i. Cada vez que sincronizamos los parámetros de la red objetivo θ^- estamos cambiando la función de pérdida.

es el *target* de DQN. Para realizar el máximo, el **espacio de acciones** A tiene que ser **finito**. Esto es una condición necesaria para aplicar DQN y el resto de algoritmos de la familia, que le impide ser utilizado en entornos con acciones continuas. El pseudocódigo de DQN se puede ver en el algoritmo 2.4.

Si atendemos a la clasificación de algoritmos que se ha dado antes, DQN es un algoritmo *model-free* y *off-policy*. Es *model-free* porque no construye un modelo del entorno, sino que aprende directamente interactuando con el simulador; y *off-policy* porque, al igual que Q-Learning, la política que se utiliza para generar los datos es distinta a la política que se está aprendido. Es más, el uso de la memoria de repetición hace que el algoritmo sea *off-policy* directamente.

El objetivo de estas dos mejoras es **aliviar** los problemas de la distribución no-estacionaria de los datos y su correlación. La red objetivo separada reduce la divergencia y las oscilaciones, pero el elemento más importante es la memoria de repetición.

Experience Replay

Llamamos Experience Replay (ER) [19] al uso de la memoria de repetición, que permite al agente de RL recordar y reusar experiencias pasadas. El ER permite al agente no procesar las experiencias en el mismo orden en el que las vivió mientras interaccionaba con el entorno. Este es el elemento más importante para aplicar DL con éxito a los algoritmos de RL, ya que ataca directamente a los dos problemas que hemos mencionado:

- al almacenar la experiencia en una memoria, se rompen las correlaciones temporales de los datos, ya que mezclamos experiencias antiguas con otras más nuevas y tomamos muestras aleatorias dentro de ella
- el problema de la distribucion no-estacionaria se ve atenuado ya que estas distribuciones distintas se promedian al unirse en la memoria, suavizando las diferencias y evitando grandes oscilaciones
- además, aumenta la eficiencia con la que se aprovechan los datos, puesto que una misma experiencia puede usarse, potencialmente, para muchas actualizaciones de los parámetros

El ER aumenta considerablemente el coste computacional del algoritmo y su uso de memoria. Pero el precio a pagar sale rentable, pues a cambio tenemos un algoritmo estable y que requiere menos experiencia para aprender, puesto que aprovecha más cada experiencia vivida.

Implementación

La memoria de repetición D se suele implementar como cola First In First Out (FIFO) , donde, una vez ha llenado su capacidad con N experiencias, las experiencias más antiguas son sustituidas por las más recientes.

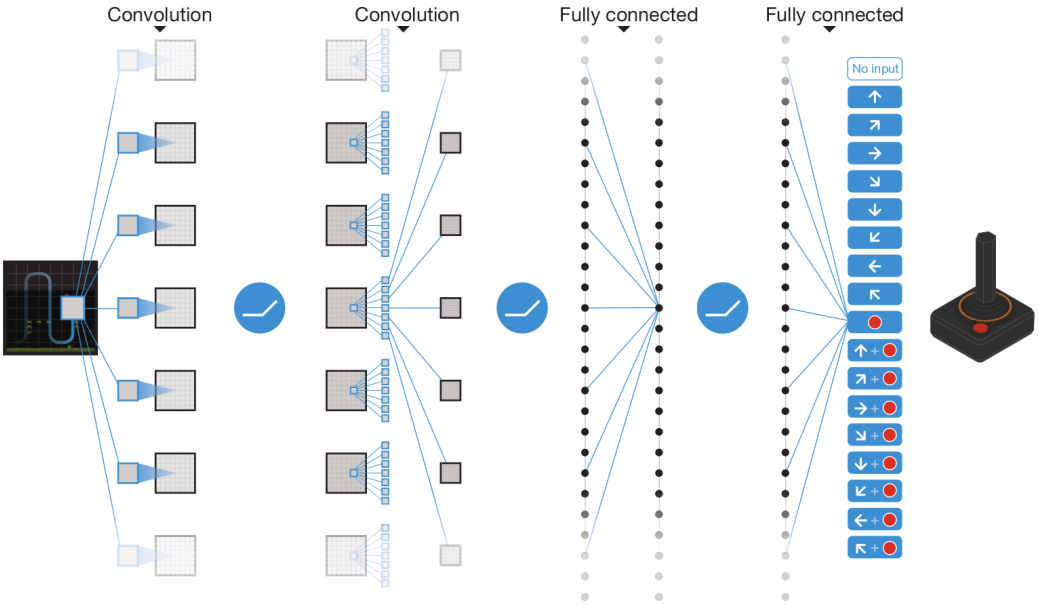

Figura 2.3: Muestra la arquitectura de la red neuronal utilizada en el artículo *"Human-level control through deep reinforcement learning"* [2], del que se ha extraído. Obsérvese como recibe como entrada el estado, y la salida es un vector con las acciones.

Para implementar la red neuronal que aproxima la función $Q(s, a|\theta)$, lo primero que se nos ocurre es crear un modelo que reciba como entrada un par estado-acción $(s, a) \in S \times A$ y devuelva un escalar $Q(s, a|\theta) \in \mathbb{R}$, tal que

$$(s, a) \longmapsto Q(s, a|\theta) \in \mathbb{R}.$$

Esta forma es ineficiente, ya que, por la naturaleza del algoritmo, tenemos que calcular el valor de todas las acciones de un mismo estado para hallar el máximo, y eso requiere tantas operaciones de *feed-forward* en la red como acciones tenga el entorno.

En su lugar, la arquitectura de red neuronal que utilizaremos recibe como entrada un estado $s \in S$ y devuelve un vector de valores acción-estado $Q(s, \cdot|\theta) \in \mathbb{R}^m$. Es decir, parametrizamos la red neuronal de modo que sea una función que manda

$$s \longmapsto \begin{bmatrix} Q(s, a_1|\theta) \\ \vdots \\ Q(s, a_m|\theta) \end{bmatrix} \in \mathbb{R}^m. \tag{2.18}$$

donde m es el número de acciones. De este modo, si el estado del entorno se puede representar como un vector n-dimensional y tenemos m acciones, la red neuronal sería una función de $\mathbb{R}^n$ a $\mathbb{R}^m$. Esto nos permite obtener el valor de todas las acciones asociadas a un mismo estado con una sola operación de *feed-forward*. Se puede ver esta arquitectura en la Figura 2.3.

2.2.2. Double Deep Q-Network

Motivación

En 2015, un equipo de investigadores de DeepMind liderado por Hado van Hasselt publica el artículo *"Deep Reinforement Learning with Double Q-Learning"* [14]. En él se estudia con mucho detalle el problema de la **sobreestimación** y el **sobreoptimismo** en DQN, como se puede ver en la Figura 2.4. Muestran que la sobreestimación de los valores acción-estado es un hecho muy común en los algoritmos basados en Q-learning, y que esto puede suponer un obstáculo en el proceso de aprendizaje de buenas políticas. Visto esto, proponen una extensión de DQN que previene, en cierta medida, el problema de la sobreestimación, y lo aplican a los juegos de Atari 2600, superando [10] el rendimiento de DQN estándar publicado en [2] y convirtiéndose en el nuevo estado del arte incontestable de DRL. Ese método es DDQN , que exponemos a continuación.

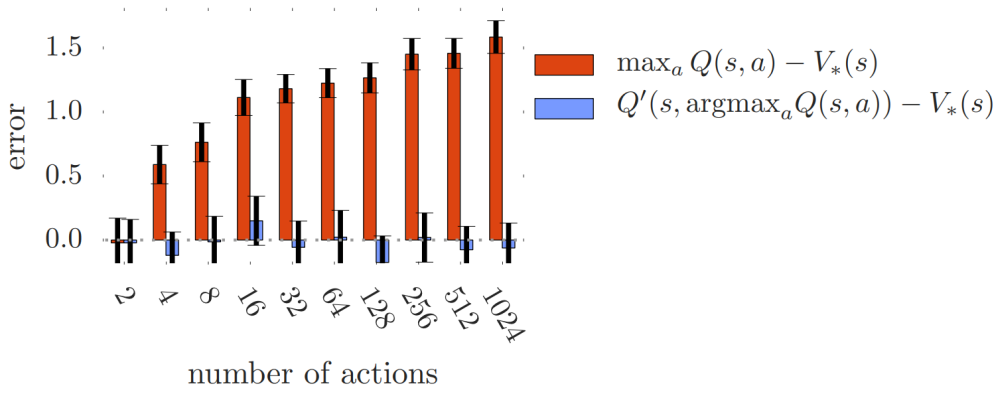

Figura 2.4: Histograma que muestra la cantidad de sobreestimación en Q-learning y Double Q-learning en función del número de acciones. Imagen extraída del artículo *"Deep Reinforement Learning with Double Q-Learning"* [14].

La observación clave es que, a través del operador $\max$ sobre las acciones, DQN utiliza la misma red tanto para seleccionar la acción como para evaluarla, lo que lo hace más susceptible a la sobreestimación de los valores. Esto se ve mejor si desarrollamos de la siguiente manera el *target* de DQN

$$y^{DQN} = r + \gamma \max_{a' \in A} Q(s', a'|\theta^-) = r + \gamma Q(s', \operatorname*{argmax}_{a' \in A} Q(s', a'|\theta^-)|\theta^-).$$

Esta ecuación (un tanto tautológica, como un juego de palabras) dice que, fijado un estado $s \in S$, es lo mismo:

[10] De hecho, superaron al DQN estándar en la gran mayoría de juegos utilizando los mismos hiperparámetros. Por si fuera poco, optimizan los hiperparámetros para la nueva extensión y obtienen unos resultados todavía más impresionantes.

- hallar el valor máximo que alcanza $Q(s, \cdot)$ en el conjunto de acciones

- evaluar $Q(s, a^*)$ en la acción $a^* = \text{argmax}_{a \in A} Q(s, a)$ que alcanza el máximo de $Q(s, \cdot)$

En el segundo punto se ven las dos redes en acción: la seleccionadora, que escoge la acción óptima $a^* \in A$ con respecto a sus valores acción-estado; y la evaluadora, que se aplica sobre la acción seleccionada. En DQN la misma red ejerce las dos funciones.

Modelo

Pues bien, el modelo que propone el artículo consiste en **desacoplar la red en dos**, una que haga la función de **seleccionadora** y otra **evaluadora**, con el objetivo de prevenir la sobreestimación de los valores. Esta idea no es la principal aportación del artículo. Double Q-learning [20] fue propuesto varios años antes como algoritmo de RL clásico por Hado van Hasselt, el mismo autor de este artículo. Lo principal es cómo aplicar la idea del algoritmo en el contexto tabular a la arquitectura de DQN. La respuesta es particularmente sencilla, puesto que en DQN ya tenemos dos redes (relativamente) desacopladas: la red predictora $Q(s, a|\theta)$, y la red objetivo $Q(s, a|\theta^-)$, cuyos parámetros θ^- se mantienen fijos y se sincronizan con los de la red predictora cada cierto número de pasos del algoritmo.

Por esta razón, la red objetivo parece la candidata perfecta para ser la segunda red. En concreto, la propuesta del artículo es utilizar la red predictora (θ) para seleccionar la acción y la red objetivo (θ^-) para estimar su valor. Dicho todo esto, la única diferencia con el algoritmo de DQN estándar es la función de coste que se utiliza, por tanto basta cambiar la línea 12 del pseudocódigo de DQN 2.4 para obtener el pseudocódigo de DDQN. La función de coste final queda

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{U}(D)} \left[(y^{DDQN} - Q(s, a|\theta))^2 \right], \tag{2.19}$$

donde $\mathcal{U}(D)$ es la distribución uniforme sobre la memoria de repetición D, y

$$y^{DDQN} = r + \gamma Q(s, \underset{a' \in A}{\text{argmax}}\, Q(s, a'|\theta)|\theta^-). \tag{2.20}$$

es el *target* de DDQN, que ahora depende de los dos parámetros. El cambio a nivel del algoritmo es mínimo, y el aumento computacional también, puesto que sólo tenemos que realizar la operación *feed-forward* una vez más por cada dato en esta extensión del algoritmo.

2.2.3. Dueling Network

Motivación

El artículo *"Dueling Network Architectures for Deep Reinforcement Learning"* [15], publicado en 2015 por (una vez más) un equipo de investigadores de DeepMind [11], asume un enfoque distinto a la hora de buscar mejoras para el algoritmo DQN estándar. Los autores se dan cuenta de que, si bien en los últimos años se han insertado con éxito arquitecturas de DL en los algoritmos de RL, estas eran siempre arquitecturas estándar en el campo, como MLP, CNN, LSTM, Autoencoders, etc. Es decir, ninguna incorporaba elementos específicos de RL ni se aprovechaba de la estructura de la función que se estaba aproximando, la función acción-valor $Q(s, a)$.

El objetivo del artículo no es proponer un algoritmo nuevo de DRL ni aplicar DL a uno clásico, sino innovar una **arquitectura de red neuronal** diseñada específicamente para problemas de *model-free* RL, que se pueda combinar con gran parte de los los algoritmos existentes en el campo.

Generalización

En Machine Learning y Deep Learning, llamamos Generalización a la capacidad de un modelo de funcionar sobre datos que no ha visto durante el proceso de aprendizaje. Podemos hablar de algo similar en el contexto del Reinforcement Learning. Por ejemplo, el algoritmo de RL clásico Q-learning, en su versión tabular, no generaliza nada en absoluto el conocimiento que ha adquirido de cada par estado-acción $(s, a) \in S \times A$. Cada paso del algoritmo modifica una casilla de la tabla y deja el resto exactamente como estaban. Para Q-learning, los pares (s, a), (s, b), (t, a) y (t, b) son todos igual de distintos, y no utiliza las relaciones que tienen entre ellos (que compartan el estado o la acción) para transferir información entre ellos: decimos que Q-learning no generaliza entre los pares estado-acción.

La mayor parte de problemas tienen demasiadas acciones y estados como para aprenderlos por separado: necesitamos la generalización. Desde el inicio del campo se utilizaron funciones de aproximación lineales y no lineales [12] para la función de valor. De esta manera, al hacer depender la función entera de unos parámetros, los estados dejan de estar aislados entre sí. Con la llegada de DRL, esta aproximación se hace con una red neuronal, lo cual permite extraer características relevantes y obtener representaciones alternativas de los estados que faciliten el aprendizaje.

Aquí entra en juego Dueling Network. Las redes neuronales estándar en DL ya tienen cierta capacidad de generalizar por lo que hemos dicho antes. Dueling Network pretende fomentar más la generalización en el contexto específico RL. En concreto, se da una estructura especial a la red neuronal, de tal forma que estime:

- por un lado, el valor $V(s)$ de estar en un estado $s \in S$

[11] Excepto Nando de Freitas, que, aunque ahora trabaja en la filial de Google, en el momento de la publicación del artículo todavía no pertenecía al grupo.

[12] Se conoce como *Deadly Triad Issue* [21] a la inestabilidad y la dificultad para converger de los métodos que combinan tres elementos: *bootstrapping*, algoritmo *off-policy* y función de aproximación no-lineal. Pero, como hemos visto, DQN usa los tres y es estable.

- y por otro, la ventaja $A(s, a)$ (*advantage* en inglés) de tomar una acción $a \in A$ desde el estado $s \in S$. La función de ventaja de una política π se define como

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s). \tag{2.21}$$

Finalmente, estos dos valores son combinados de un modo especial para obtener la función de valor estado-acción $Q(s, a)$. Al estar estimando explícitamente $V(s)$, este conocimiento se generalizará para todas las acciones desde un mismo estado, permitiendo a la red aprender qué estados son valiosos sin necesidad de conocer el efecto de cada acción.

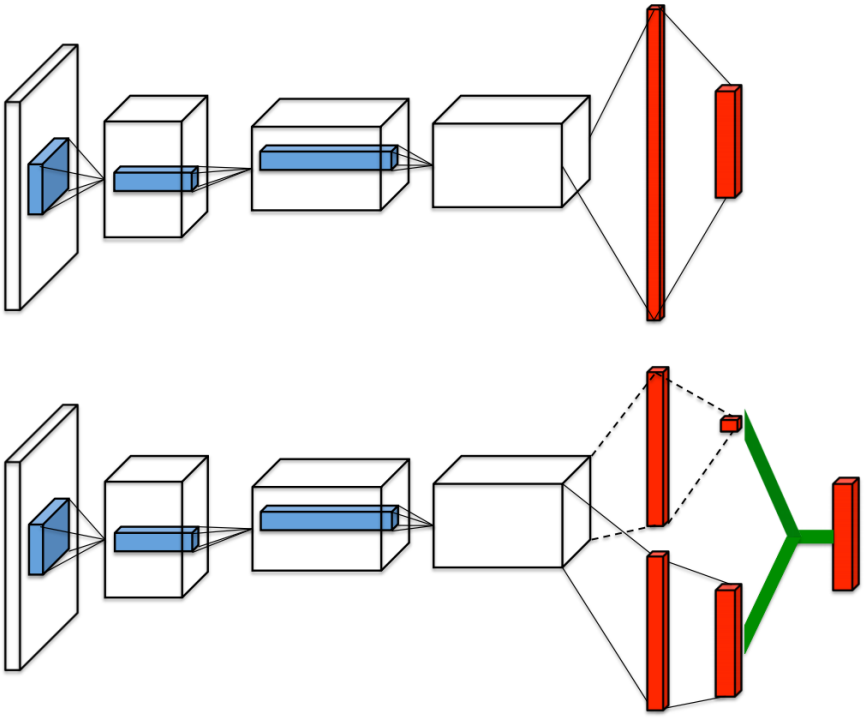

Figura 2.5: Diagrama que muestra la diferencia entre (de arriba a abajo) la red neuronal CNN + MLP estándar y una Dueling Network. Imagen extraída del artículo *"Dueling Network Architectures for Deep Reinforcement Learning"* [15].

Modelo

Vamos a fijarnos en la Figura 2.5. La arquitectura de la Dueling Network empieza con un único flujo que procesa el estado, dándole una representación alternativa; luego, se bifurca en dos flujos computacionales, uno para la función de valor y otro para la función de ventaja (dependiente del estado); finalmente, los flujos se combinan usando un capa agregadora especial. Si ξ son los parámetros del flujo inicial común, η los de la función de valor y ψ los de la función de ventaja, la capa agregadora queda

$$Q(s,a|\theta) = V(s|\xi,\eta) + A(s,a|\xi,\psi) - \frac{1}{m}\sum_{a'\in A} A(s,a'|\xi,\psi), \tag{2.22}$$

donde $\theta = \{\xi,\eta,\psi\}$ y m es el número de acciones. Nótese que los parámetros ξ del flujo inicial sólo se utilizan para codificar el estado, tal que $s \mapsto F(s|\xi)$. El flujo que codifica la función de valor tiene que terminar en un escalar, mientras que el de la de ventaja termina en un vector de valores para cada acción $A(s,\cdot|\xi,\psi) \in \mathbb{R}^m$, como sucedía en la parametrización de la red de DQN estándar.

Dueling Network sólo cambia este aspecto de la arquitectura de la red neuronal, y el sobrecoste computacional es mínimo. Sólo afecta a la arquitectura de la red, sin ningún esfuerzo adicional ni ninguna modificación del algoritmo.

Es curioso que tan sólo bifurcando el flujo para luego unirlo con la capa agregadora especial se obtenga una red con la capacidad de proveer de estimadores separados de $V(s)$ y $A(s,a)$ si miramos cada uno de los flujos. Esto se debe a que la misteriosa capa agregadora se ha diseñado con detenimiento.

Justificación

En la ecuación 2.21 definimos cuál es la función de ventaja para una política determinada. Podemos intentar utilizar una capa agregadora del tipo

$$Q(s,a|\theta) = V(s|\xi,\eta) + A(s,a|\xi,\eta),$$

pero esto claramente no funciona, ya que no identifica unívocamente a las funciones V y A. Basta sumar una constante k a una función y restársela a la otra para confirmar esto. Por tanto, si construyésemos la red neuronal con esta capa agregadora no obtendríamos estimadores de las funciones de valor y ventaja en sendos flujos. Necesitamos una capa que indentifique unívocamente estas funciones. Para esto, observamos que si $a^* = \operatorname{argmax}_{a'\in A} Q(s,a')$, entonces $Q(s,a^*) = V(s)$ [13], y por tanto $A(s,a^*) = 0$. Basándonos en esto, construimos la capa para que la función de ventaja valga 0 cuando la política escoge la acción con mayor valor. La aproximación directa es hacer

$$Q(s,a|\theta) = V(s|\xi,\eta) + A(s,a|\xi,\psi) - \max_{a'\in A} A(s,a'|\xi,\psi),$$

que produce una estimación correcta en el flujo del valor ya que

$$a^* = \operatorname*{argmax}_{a'\in A} Q(s,a'|\theta) = \operatorname*{argmax}_{a'\in A} A(s,a'|\xi,\psi), \tag{2.23}$$

y por tanto $Q(s,a^*|\theta) = V(s|\xi,\eta)$. La capa original de Dueling Network es suficiente para identificar las dos funciones, pero se pierde la semántica, ya que ahora V y A no son verdaderos estimadores de la función de valor y de ventaja porque están desplazados una constante, pero la política que se

[13] Debido a las ecuaciones de optimalidad de Bellman.

representa sigue siendo la misma porque A preserva el orden relativo del valor de sus acciones. A cambio, dice el artículo que se obtiene estabilidad durante la optimización. En última instancia, ambas capas logran estimar (o excepto por una constante) V y A en sus flujos y podemos decidir utilizar cualquiera de las dos, pero se suele utilizar la versión de la media, que se puede ver en 2.22.

2.2.4. Prioritized Experience Replay

Motivación

Ya en el artículo fundacional de DRL [1] se menciona que la memoria de repetición D se podría mejorar de modo que diferenciase experiencias según su importancia para el proceso de aprendizaje, en lugar de dar a todas la misma importancia usando un muestreo uniforme.

Un equipo de investigadores de (nuevamente) DeepMind liderado por Tom Schaul publica en el año 2016 el artículo *"Prioritized Experience Replay"* [16], donde recogen el guante del artículo de 2013. El concepto clave que desarrollan es que los agentes de RL pueden aprender más de unas experiencias que de otras, y por lo tanto se puede optar por una estrategia mejor que simplemente repetir las experiencias con la misma frecuencia con la que fueron vividas. El resultado de aplicar este método unido a DDQN a los juegos de Atari 2600 fue sorprendente: se aceleró el aprendizaje por un factor de 2 y se logró un nuevo estado del arte.

Si queremos priorizar unas experiencias frente a otras en base a su importancia, es crucial la forma en la que medimos esta importancia. En una situación idealizada, la medida que nos gustaría utilizar es *cuánto puede aprender nuestro agente de una transición en el estado actual.* Por desgracia esta medida no es accesible, así que tendremos que usar una alternativa.

Priorización

La idea central de PER es utilizar el TD-error δ de una experiencia (s, a, r, s'), definido

$$\delta = r + \gamma \max_{a' \in A} Q(s', a'|\theta) - Q(s, a|\theta), \tag{2.24}$$

como métrica de su importancia. Ya hemos visto este valor en la sección de RL clásico, pues se utiliza en el algoritmo de Sarsa y Q-learning. La priorización codiciosa (*greedy* en inglés), donde se escoge la experiencia de la memoria de repetición con el TD-error más alto, tiene varios problemas. Entre ellos, la rigidez del método codicioso hace perder mucha diversidad a las experiencias que repite el agente, y lo hace propenso al sobreajuste.

Entre el método puramente codicioso determinista y el muestreo uniforme de la memoria de datos, se encuentra el método de la *priorización estocástica*. Esta forma de tomar muestras aleatorias de la memoria de repetición consiste en asignar a cada experiencia una probabilidad tal que

- todas las experiencias tengan probabilidad positiva
- si una experiencia tiene mayor TD-error que otra, entonces tienen mayor probabilidad

Definimos la probabilidad de muestreo de la transición i-ésima como

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \tag{2.25}$$

donde $p_i > 0$ es la prioridad que asignamos a la transición i-ésima, y α es un parámetro de temperatura que determina el grado de priorización. Consideramos dos variantes dependiendo de la elección de la prioridad p_i.

- $p_i = |\delta_i| + \varepsilon$, donde δ_i es el TD-error de la transición i-ésima y $\varepsilon > 0$ una constante que asegura que todas las transiciones tiene probabilidad positiva
- $p_i = (\text{Rank}(i))^{-1}$, donde $\text{Rank}(i)$ es la posición de la transición i-ésima al ser ordenadas con respecto al valor absoluto del TD-error $|\delta_i|$.

Ambas se pueden utilizar, pero la basada en $\text{Rank}(i)$ es más robusta, puesto que un TD-error excepcionalmente alto no puede cambiar por completo la distribución. Para terminar, utilizar este tipo de priorización cambia la distribución sobre la que aprende nuestro algoritmo, y debemos corregir ese sesgo añadiendo unos pesos w_i sobre los δ_i, tal que

$$w_i = \left(\frac{1}{NP(i)}\right)^\beta. \tag{2.26}$$

Para más información sobre cómo se usan estos pesos, consultar el artículo [16].

Implementación

La implementación directa de este algoritmo es tan ineficiente que convierte el algoritmo en inviable. Los autores del artículo original se esfuerzan mucho en explicar cómo se puede implementar de manera eficiente haciendo un uso inteligente de las estructuras de datos y algoritmos. La primera variante admite una implementación en una estructura conocida como *sum-tree*, mientras que para la segunda se puede aproximar la función de distribución con una función lineal a trozos con k segmentos equiprobables. Para más información, acudir al artículo original [16].

2.2.5. Rainbow DQN

En el artículo *Rainbow: Combining Improvements in Deep Reinforcement Learning"* [17], se combinan de una modo especial las tres extensiones descritas en este trabajo y tres mas: Multi-step learning, Distributional RL y Noisy Nets. Ejemplarizando el lema que dice que *la unión hace la fuerza*, Rainbow DQN obtuvo unos resultados espectaculares, como se puede observar en la Figura 2.6.

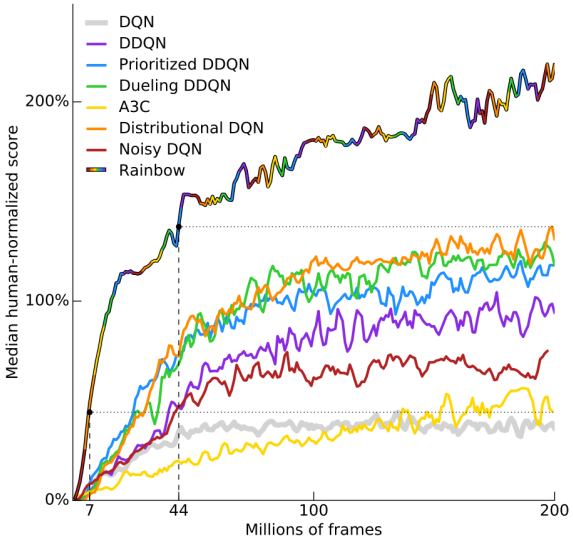

Figura 2.6: Gráfica comparativa entre seis extensiones de DQN y Rainbow DQN. A los 7 millones de *frames* alcanza el rendimiento de DQN, y a los 44 millones supera el mejor rendimiento obtenido con cualquier método. Fuente: *Rainbow: Combining Improvements in Deep Reinforcement Learning"* [17].

2.3. Métodos basados en Policy Gradient

Los métodos basados en Policy Gradient difieren en los vistos en la sección anterior en que, en lugar de centrarse en aprender una función acción-valor $Q(s, a)$, se lleva a cabo el aprendizaje directamente sobre una política parametrizada π_θ con una red neuronal. Son un enfoque atractivo en RL porque optimizan directamente la recompensa acumulada en el espacio de parámetros de la política. Los principales beneficios de estos métodos son:

- se pueden utilizar en espacios de acciones continuos y de dimensión alta, ya que no requieren tomar un máximo sobre las acciones para su funcionamiento, como ocurría en los métodos basados en DQN
- modelizan políticas estocásticas directamente, pueden controlar el nivel de exploración de forma natural sin necesidad de estrategias ε-*greedy*
- al utilizar información del gradiente para la optimización, suelen tener mejores propiedades de convergencia
- por último, a veces la política óptima simplemente es más fácil de aprender que la función acción-valor óptima

En este trabajo vamos a estudiar el algoritmo Advantage Actor-Critic [22], que parametriza tanto la política π_θ como la función de valor $V_\psi^{\pi_\theta}$, así como su versión paralelizada síncrona.

2.3.1. Advantage Actor-Critic

Motivación

El algoritmo Reinforce es conceptualmente muy simple, pero se ha comprobado el estimador que usa del gradiente tiene una gran varianza, haciéndolo inviable para entornos complejos y con longitud de episodio mayores [13, 22]. Para aliviar este problema, se introduce la idea de función *baseline* $b(s)$, donde $b(s)$ es una función que depende solamente del estado, y no de la acción. Se puede comprobar que el PGT se sigue cumpliendo si sustituimos $Q^{\pi_\theta}(s,a)$ por $(Q^{\pi_\theta}(s,a) - b(s))$ en el enunciado del teorema. Si recordamos la definición de la función de ventaja

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s),$$

lo primero que se nos ocurre es tomar como función *baseline* la función de valor $V^{\pi_\theta}(s)$, de modo que la formulación del nuevo PGT queda así [14]:

Teorema 2.3.1. *El gradiente del rendimiento se puede expresar como*

$$g = \nabla_\theta J(\pi_\theta) = \mathbb{E}^{\pi_\theta}\left[\sum_{t=1}^{\infty} \nabla_\theta \log \pi_\theta(X_t, Y_t) A^{\pi_\theta}(X_t, Y_t)\right]. \tag{2.27}$$

La idea clave del algoritmo Advantage Actor-Critic es aproximar la función de valor $V_\psi^{\pi_\theta}(s)$ con parámetros ψ, a la vez que se parametriza la política. El paso del ascenso por gradiente en el espacio de parámetros será exactamente igual que en Reinforce, a través de la estimación del gradiente provista por el PGT. Lo que nos queda por explicar es:

- cómo estimamos $A^{\pi_\theta}(s,a)$ para poder obtener una estimación del gradiente $\hat{g}$
- y cómo entrenamos la nueva aproximación de $V_\psi^{\pi_\theta}(s)$

Advantage Actor Critic

Llamamos **actor** a la política parametrizada $\pi_\theta(s,a)$ y **crítico** a la función de valor aproximada $V_\psi^{\pi_\theta}(s)$. Supongamos que hemos generado una episodio de longitud T, $\tau = (s_1, a_1, r_1, s_2, \ldots, s_T)$ interactuando con el entorno usando la política $\pi_\theta(s,a)$. Para estimar la función de ventaja, se nos puede ocurrir utilizar el mismo estimador de $Q^{\pi_\theta}(s,a)$ que usamos en Reinforce y simplemente restarle nuestra aproximación de la función de valor, de modo que

$$\hat{A}_t = \sum_{t'=t}^{T-1} r_{t'} - V_\psi^{\pi_\theta}(s_t).$$

[14] Para ver otras formulaciones equivalentes del PGT, ver [13].

Esto se puede utilizar, pero en su lugar usamos un estimador que aprovecha mejor el hecho de que estemos aproximando la función de valor $V_\psi^{\pi_\theta}(s)$, utilizándola a modo de *bootstrap*, de modo que

$$\hat{A}_t = r_t + V_\psi^{\pi_\theta}(s_{t+1}) - V_\psi^{\pi_\theta}(s_t). \tag{2.28}$$

Para entrenar la función $V_\psi^{\pi_\theta}(s)$, usamos un algoritmo de optimización para minimizar la función de pérdida

$$L(\psi) = \sum_{t=1}^{T-1} \hat{A}_t^2. \tag{2.29}$$

Por aclarar más este paso, pensemos que la trayectoria τ nos induce un conjunto de datos $\mathcal{D} = \{(s_t, r_t + V_\psi^{\pi_\theta}(s_{t+1}))\}_{t=1}^{T-1}$, y buscamos minimizar el error cuadrático medio $\sum_{t=1}^{T-1}(y_t - V_\psi^{\pi_\theta}(s_t))^2$, donde el target es $y_t = r_t + V_\psi^{\pi_\theta}(s_{t+1})$. Entonces, mediante un algoritmo de optimización obtenemos una estimación del gradiente $\nabla_\psi L(\psi)$, que podemos utilizar para hacer un descenso gradiente en el espacio de parámetros, tal que

$$\psi \leftarrow \psi - \eta_\psi \nabla_\psi L(\psi). \tag{2.30}$$

Esto en cuanto al uso del crítico $V_\psi^{\pi_\theta}(s)$ para estimar las ventajas y su entrenamiento usando un algoritmo de optimización. En cuanto al actor $\pi_\theta(s, a)$, operamos del mismo modo que en Reinforce, obteniendo un estimador del gradiente mediante el PGT, tal que

$$\hat{g} = \sum_{t=1}^{T-1} \nabla_\theta \log \pi_\theta(s, a) \hat{A}_t, \tag{2.31}$$

para luego usarlo haciendo un ascenso por gradiente en el espacio de parámetros vía

$$\theta \leftarrow \theta + \eta_\theta \hat{g}. \tag{2.32}$$

El pseudocódigo se puede ver en 2.5.

2.3.2. Synchronous Advantage Actor-Critic

El algoritmo que vamos a usar en este trabajo es la versión paralelizada síncrona de Advantage Actor-Critic, conocida como A2C [22] [15]. La idea es tener un conjunto de agentes, todos con los mismos parámetros θ y ψ, que generen en paralelo los gradientes del actor y del crítico. Para mantenerlos sincronizados se introduce un coordinador, que les informa de los parámetros globales θ y ψ, y luego espera a recibir los gradientes de todos los agentes para hacer una única actualización con todos los

[15] Aunque A2C es el acrónimo de Advantage Actor-Critic, se usa para referirse a la versión paralelizada síncrona de este algoritmo.

gradientes acumulados. Entonces, el coordinador les informa de los nuevos parámetros globales, y vuelta a empezar.

input : Parámetros de aprendizaje η_θ y η_ψ

Inicializar parámetros θ y ψ

for *episodio* $1, \ldots ...$ **do**

  Genera episodio $\tau = (s_1, a_1, r_1, s_2, \ldots, s_T)$ usando la política π_θ

  Estimas ventajas $\hat{A}_t \leftarrow r_t + V_\psi^{\pi_\theta}(s_{t+1}) - V_\psi^{\pi_\theta}(s_t)$

  $\hat{g} \leftarrow \sum_{t=1}^{T-1} \nabla_\theta \log \pi_\theta(s_t, a_t)\hat{A}_t$

  $L(\psi) = \sum_{t=1}^{T-1} \hat{A}_t^2$

  $\theta \leftarrow \theta + \eta_\theta \hat{g}$

  $\psi \leftarrow \psi - \eta_\psi \nabla_\psi L(\psi)$

end

Algoritmo 2.5: Pseudocódigo del algoritmo de Advantage Actor-Critic.

3

Desarollo

3.1. OpenAI Gym

OpenAI Gym [23] es una herramienta *open source* que sirve para implementar y comparar algoritmos de aprendizaje por refuerzo. En concreto, Gym provee de una colección de entornos, o problemas de test, a los que se puede acceder mediante una interfaz simple y unificada. Los entornos son de lo más diversos, desde entornos de control clásico hasta juegos de Atari 2600, pasando por humanoides que aprenden a caminar. A todos se accede desde la misma interfaz.

3.1.1. Entornos

Vamos a seleccionar unos entornos sobre los que realizaremos los experimentos. Dado que la mayoría de modelos de DRL que hemos estudiado son del tipo DQN, estamos restringidos a modelos cuyo espacio de acciones sea **finito**. Además, nuestra limitada capacidad computacional nos impide enfrentarnos a los juegos de Atari 2600. Por ello, hemos decidido escoger los siguientes cuatro modelos: tres de control clásico, *CartPole-v1*, *MountainCar-v0* y *Acrobot-v1*; y *LunarLander-v2*, del simulador *Box2D*. De ahora en adelante nos referiremos a ellos sin el sufijo con la versión y sin tipografía cursiva.

A continuación describimos el espacio de estados, el espacio de acciones y las recompensas en cada uno de los entornos. Para datos más técnicos como cuáles son las condiciones de finalización de los episodios, cómo se escoge el estado inicial, o cuáles son las condiciones de resolución que define OpenAI, véase la wiki de OpenAI Gym en Github [1], aunque a veces no hay mucha información, como en el caso de LunarLander.

- **CartPole**: Tenemos un poste unido a un carro mediante una articulación, que se mueve por una pista sin fricción. El objetivo del agente es mantener el poste en posición vertical. El espacio de estados son las 4-tuplas $(x, \dot{x}, \theta, \dot{\theta})$, donde x es la posición del carro, $\dot{x}$ su velocidad, θ el ángulo de poste y $\dot{\theta}$ su velocidad angular; las acciones son empujar a la izquierda o a la derecha; la

[1] Enlace: `https://github.com/openai/gym/wiki/Table-of-environments`.
[2] Enlace: `https://gym.openai.com/envs/#classic_control`.

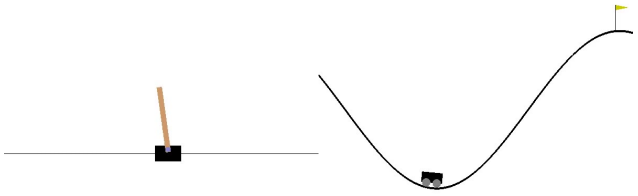

Figura 3.1: Imagen del entorno *CartPole-v1* y *MountainCar-v0*, obtenidas de OpenAI Gym [2].

recompensa es $+1$ por cada paso que se mantiene el poste lo suficientemente vertical. Se puede ver una imagen de este entorno en 3.1 (izquierda).

- **MountainCar**: Un coche con poca potencia se encuentra entre dos colinas. El objetivo del agente es alcanzar la cima de una de ellas. El espacio de estados es $(x, \dot{x})$, donde x es la posición del coche y $\dot{x}$ su velocidad; las acciones son tres, empujar a la izquierda, no empujar, y empujar a la derecha; la recompensa es -1 cada paso hasta que se alcanza la cima. Se puede ver una imagen de este entorno en 3.1 (derecha).

- **Acrobot**: Dos enlaces unidos mediante una articulación. El objetivo es alcanzar la altura de la línea negra con la punta del enlace inferior. Los estados son $(\sin\theta_1, \cos\theta_1, \sin\theta_2, \cos\theta_2, v_1, v_2)$, donde θ_1, θ_2 son los ángulos de las dos articulaciones, y v_1, v_2 las velocidades angulares; las acciones son aplicar $-1, 0$ o $+1$ de torque a la articulación entre los dos enlaces; y la recompensa es -1 cada paso hasta que se alcanza la línea negra. Se puede ver una imagen de este entorno en 3.2 (izquierda).

- **LunarLander**: Una nave espacial quiere alunizar entre dos banderas. Su objetivo es hacerlo lo mejor posible. El estado es una 8-tupla, cuyas dos primeras componentes son las coordenadas de la nave; hay cuatro acciones: no hacer nada, activar motor principal, motor izquierdo y motor derecho; la recompensa depende de un conjunto de reglas que evalúa las maniobras del agente. Se puede ver una imagen de este entorno en 3.2 (derecha).

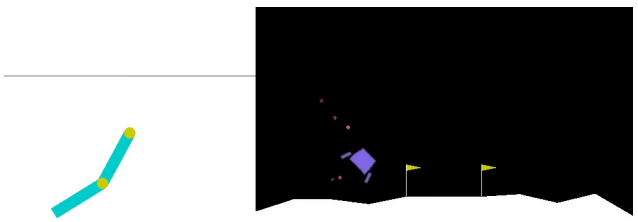

Figura 3.2: Imágenes del entorno *Acrobot-v1* y *LunarLander-v2*, obtenidas de OpenAI Gym [3]

[2] Enlace: `https://gym.openai.com/envs/#classic_control`.
[3] Enlace: `https://gym.openai.com/envs/#box2d`.

3.2. Stable Baselines

Stable Baselines [24] es un conjunto de implementaciones en *Tensorflow* de algoritmos de Deep Reinforcement Learning. Contiene gran parte de los algoritmos del estado del arte en DRL. En particular, los cinco que hemos estudiado en este trabajo: DQN, sus extensiones DDQN, Dueling Network y PER, y el algoritmo basado en gradientes A2C.

Vamos a utilizar esta librería para realizar los experimentos del siguiente capítulo. Como hemos dicho, el objetivo de este trabajo es hacer un estudio teórico de los principales algoritmos de DRL, en especial aquellos basados en DQN, para después hacer una comparativa de su rendimiento probándolos en entornos de OpenAI Gym. La implementación eficiente de estos algoritmos no es baladí, y sería perseguir un objetivo muy distinto al nuestro. Sería como intentar hacer una comparativa de modelos de DL y no usar Tensorflow o Pytorch. Queremos que nuestra comparativa sea fiable, y por ello nos apoyamos en las implementaciones estables y comprobadas de Stable Baselines.

3.2.1. Funcionalidad

A continuación explicamos la funcionalidad básica de la librería Stable Baselines.

Crear, entrenar y evaluar un modelo

Para crear un modelo basta seleccionar uno de los disponibles en la librería e instanciarlo con los hiperparámetros deseados. Los dos primeros parámetros son siempre la arquitectura de red neuronal que se usará para aproximar la función de acción-valor o la política, dependiendo del algoritmo; y el entorno de entrenamiento.

Una vez tenemos el modelo, para entrenarlo basta llamar al método `learn`, indicándole el número de pasos que queremos que dure el entrenamiento. Terminado el aprendizaje, podemos evaluar al agente usando la función `evaluate_policy`, que recibe el modelo, el entorno y el número de episodios para evaluar, y devuelve la media y la desviación típica de la recompensa episódica. A continuación, un ejemplo de los tres pasos:

```
model = DQN('MlpPolicy', 'CartPole-v1', prioritized_replay=False)
model.learn(total_timesteps=100000)
mean_reward, std_reward = evaluate_policy(model, model.get_env(), n_eval_episodes=5)
```

Callbacks

Los *Callbacks* son funciones que se ejecutan internamente en el modelo durante el entrenamiento. Nos permiten acceder a variables internas, monitorizar el rendimiento, guardar versiones del modelo, etc. A nosotros nos interesan dos: `CheckpointCallback` y `EvalCallback`. El primero guarda una

copia del modelo cada cierto número de pasos, y lo utilizamos para guardar copias del modelo durante el entrenamiento y hacer grabaciones del proceso. Mientras que el segundo evalúa al agente en un entorno separado cada cierto número de pasos, y guarda el mejor modelo que se vaya encontrando. A continuación, un ejemplo de los dos:

```
checkpoint_callback = CheckpointCallback(save_freq=checkpoint_save_freq)
eval_callback = EvalCallback(eval_env, eval_freq=eval_freq)
model.learn(total_timesteps=100000, callback=[checkpoint_callback,eval_callback])
```

Modelos DQN y A2C

Los modelos de la librería que utilizaremos serán `DQN` y `A2C`. Todas las versiones de DQN que hemos estudiado se pueden activar y desactivar a través de los hiperparámetros de `DQN`. Ambos modelos comparten los hiperparámetros `gamma` (el factor de descuento γ), y `learning_rate`, que indica el tamaño del paso en la dirección del gradiente. Por ejemplo, si queremos instanciar una DQN con Dueling Network, tamaño de memoria de repetición 1000 y factor de descuento 0,95.

```
model1 = DQN('MlpPolicy',
    'CartPole-v1',
    gamma=0.95,
    buffer_size=1000,
    policy_kwargs={'dueling': True})
```

Y si queremos instanciar un A2C con *learning rate* 0,001 y `n_steps` 5.

```
model2 = A2C('MlpPolicy',
    'CartPole-v1',
    learning_rate=0.001,
    n_steps=5)
```

4

Experimentos y Resultados

En este capítulo describiremos los experimentos que hemos realizado, así como el análisis de los resultados. Como se ha dicho antes, los entornos de experimentación son los cuatro entornos de Open AI Gym: CartPole, MountainCar, Acrobot y LunarLander.

4.1. Optimización de hiperparámetros

La optimización de hiperparámetros es un elemento muy importante de ML, y también lo es de DRL. En DRL se suelen mantener los mismos hiperparámetros para los problemas que son del mismo tipo, como ocurre en los artículos que usan los juegos de Atari 2600. En nuestro caso, para cada modelo tendremos dos conjuntos de hiperparámetros: uno para los entornos de control clásico, y otro para LunarLander.

Para hacer la optimización, usaremos búsqueda aleatoria en el espacio de hiperparámetros. En cuanto a la evaluación de cada conjunto de hiperparámetros, se hará lo siguiente:

- se entrenaran 5 modelos con los mismos hiperparámetros el mismo número de pasos
- al terminar, cada modelo se evaluará generando 5 episodios, y tomando la media de las recompensas de cada uno
- la evaluación final de los hiperparámetros será la media de la media de los episodios de cada modelo.

De esta forma aliviamos la variación tanto en el entrenamiento del modelo como en su evaluación. En ambos se ha optimizado el factor de descuento γ y el *learning rate*. Además, para cada modelo se han optimizado: en DQN el tamaño de la memoria de repetición, el número de pasos para sincronizar las redes, y el *batch size*; en A2C, el número de pasos que se toma para hacer las estimaciones. La red neuronal que se usa en ambos casos es una MLP de dos capas ocultas de 64 neuronas.

4.2. Comparativa de algoritmos

El objetivo de este trabajo era hacer un estudio teórico de algunos de los principales algoritmos de DRL para luego hacer un análisis de su rendimiento en entornos de OpenAI Gym. Por tanto, el experimento principal será probar todos estos modelos en cada uno de los entornos que hemos seleccionado y analizar los resultados. Hemos estudiado 5 modelos: DQN, sus tres extensiones, y DQN. Además de estos, también probaremos con DQN con las tres extensiones simultáneamente, ya que estas son compatibles entre sí.

CartPole

Comentamos la Figura 4.1. A2C no logra aprender CartPole, al menos con un entrenamiento de 200000 pasos. El modelo combinado Dueling DDQN con PER alcanza el mejor rendimiento, pero en general todos los modelos son bastante inestables. Este es un hecho conocido sobre el entorno CartPole: pese a ser uno de los más sencillos, muchos modelos tienen problemas de estabilidad. Logran alcanzar la recompensa máxima por episodio, 500 en el caso de CartPole, y a los pocos episodios colapsan, fenómeno conocido como *catastrophic forgetting* [25].

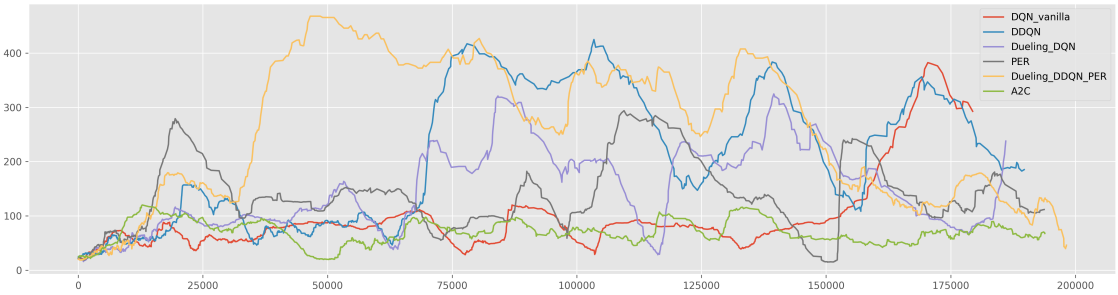

Figura 4.1: Gráfica de la recompensa total por episodio durante el entrenamiento en el entorno CartPole, suavizada con una media móvil de 30 episodios.

Acrobot

Comentamos la Figura 4.2. De nuevo, A2C no logra aprender Acrobot. El resto se estabilizan en torno a -100, que es una buena recompensa por episodio para resolver Acrobot. No hay diferencias apreciables entre los modelos basados en DQN.

MountainCar

Comentamos la Figura 4.3. La recompensa de MountainCar es muy poco informativa, pues solo da *feedback* al agente cuando logra subir a la colina. Observamos que hasta el paso 50000 ningún algoritmo empieza a aprender. A2C tampoco aprende el MountainCar, y el resto de modelos, excepto

quizás PER, obtienen un rendimiento similar.

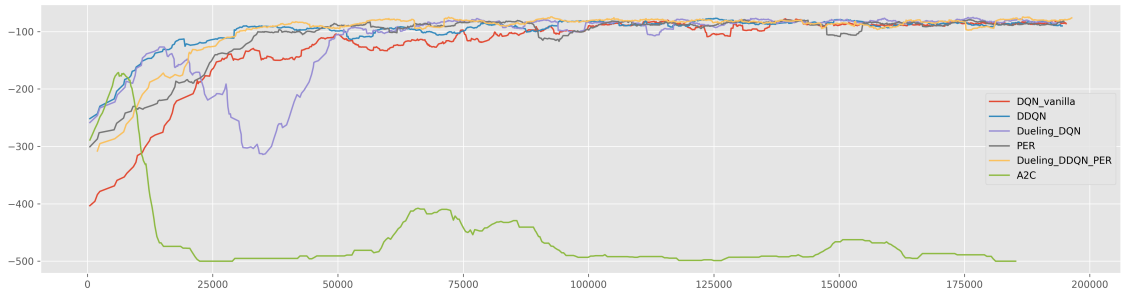

Figura 4.2: Gráfica de la recompensa total por episodio durante el entrenamiento en el entorno Acrobot, suavizada con una media móvil de 30 episodios.

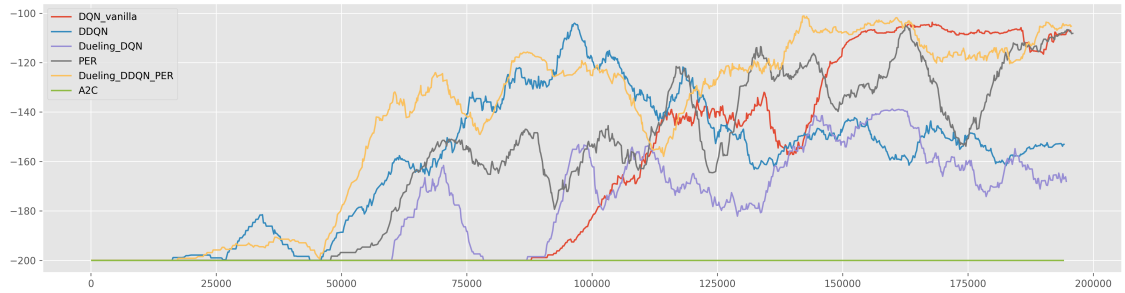

Figura 4.3: Gráfica de la recompensa total por episodio durante el entrenamiento en el entorno MountainCar, suavizada con una media móvil de 30 episodios.

LunarLander

Comentamos la Figura 4.4. En este entorno entrenamos durante 400000 pasos por su mayor complejidad. Esta vez A2C sí que consigue aprender el entorno, obteniendo un rendimiento similar al resto de modelos. Casi todos ellos alcanzan en algún momento la recompensa de 200, que se define como el umbral de superación del entorno.

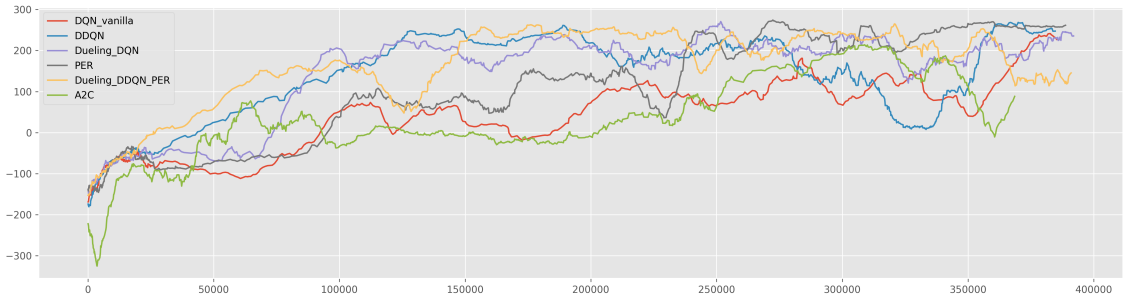

Figura 4.4: Gráfica de la recompensa total por episodio durante el entrenamiento en el entorno LunarLander, suavizada con una media móvil de 30 episodios.

4.2.1. Conclusión

No se observa que ningún modelo basado en DQN obtenga sistemáticamente mejores resultados que el resto. Su rendimiento tiene las fluctuaciones típicas del entrenamiento. Esto se debe probablemente a que los entornos que hemos probado no son lo suficientemente complejos como para captar estas diferencias, como sí ocurre en los juegos de Atari 2600. En cuanto a A2C, una hipótesis podría ser que necesitase más pasos de entrenamiento para lograr aprender los entornos. Gracias a la memoria de repetición, los algoritmos basados en DQN utilizan los datos de manera más eficiente, pero también entrenan más lento. Quizá no es justo comparar A2C con DQN entrenando el mismo número de pasos. Otra posibilidad es que los hiperparámetros usados no sean los óptimos y haya que hacer una búsqueda sistemática.

4.2.2. Vídeos de los agentes

En el canal de Youtube `https://www.youtube.com/channel/UCks3rkK4LGZh5E2S3X5_DFw` se pueden ver vídeos de los agentes durante el entrenamiento y en el momento en el que alcanzaron su mayor rendimiento. El modelo de los vídeos es DQN con las tres extensiones activadas, y el vídeo durante el entrenamiento se toma cuando se alcanza un octavo de los pasos totales.

4.3. Segundo experimento

El artículo de DQN [2] dice que se ha comprobado que las mejoras introducidas son importantes desactivándolas y viendo como empeora el rendimiento. Replicamos este experimento entrenando una DQN con memoria de repetición de tamaño y *batch size* 1, y que sincroniza el la red *target* con la *online* cada paso. Se puede ver en la Figura 4.5 como este modelo degradado no logra aprender en absoluto.

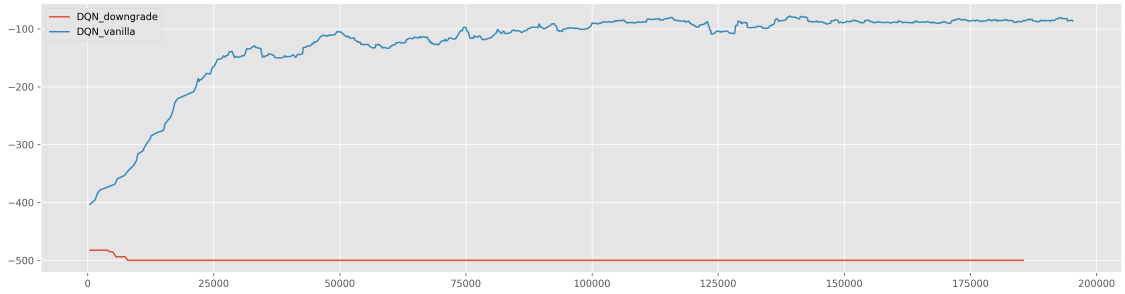

Figura 4.5: Gráfica de la recompensa total por episodio durante el entrenamiento en el entorno LunarLander, suavizada con una media móvil de 30 episodios.

5

Conclusiones y Trabajo futuro

5.1. Conclusiones

En este trabajo hemos recorrido el camino que une el campo del *Reinforcement Learning* con el reciente *Deep Reinforcement Learning*. Los procesos de decisión de Markov formalizan el problema de la toma de decisiones bajo incertidumbre, y nos aportan herramientas para diseñar algoritmos que lo resuelven aproximadamente, como las funciones de valor y acción-valor, y las ecuaciones de optimalidad de Bellman. Q-learning se basa en estos principios para encontrar estrategias óptimas, pero no funciona cuando el entorno es complejo y multidimensional.

En este contexto surgen las Deep Q-Networks, que fusionan con éxito las ideas Q-learning y el *Deep Learning* para resolver problemas antes impensables en RL. La mejora principal es el uso de una memoria de repetición, que estabiliza el proceso y mejora el aprovechamiento de los datos. Estudiamos tres extensiones: DDQN alivia el problema de la sobreestimación; Dueling Network es una arquitectura de red neuronal específica para problemas de RL; y PER da más importancia a unas experiencias que a otras de la memoria de repetición.

Basados en una idea distinta, los algoritmos de *Policy Gradient* tratan de maximizar el rendimiento directamente desde el espacio de políticas. El algoritmo clásico de esta familia, Reinforce, tiene su reflejo en DRL en el algoritmo A2C, que combina este enfoque con el que vimos antes.

Después del extenso estudio teórico, queríamos comparar el rendimiento de los algoritmos de DRL en varios entornos de entrenamiento sencillos de Open AI Gym usando las implementaciones de los algoritmos que ofrece Stable Baselines. Los resultados de los experimentos indicaban que no había una diferencia notable en el rendimiento de DQN y sus extensiones, probablemente debido a que los entornos escogidos no suponían un reto para algoritmos tan potentes. Además, el algoritmo A2C queda por debajo del resto, sobre lo que se proponen varias hipótesis. Para finalizar, se comprueba en otro experimento que las técnicas que utiliza DQN para estabilizar el aprendizaje efectivamente funcionan, desactivándolas y observando que baja su rendimiento.

5.2. Trabajo futuro

Al ser un trabajo de introducción a un tema reciente hay varios caminos distintos para seguir. Las tres vías que deja abiertas este trabajo son: seguir estudiando más algoritmos *state-of-the-art*, hacer implementaciones propias de los algoritmos vistos, o intentar aplicarlos a entornos de entrenamiento más desafiantes.

- Para seguir estudiando, el camino más directo es explorar más algoritmos de la familia *Policy Gradient*, que apenas llegamos a tocar en este trabajo. Algoritmos como *Trust Region Policy Optimization*, y su mejora *Proximal Policy Optimization*, pueden ser los siguientes pasos a dar.
- Implementar un algoritmo es la mejor forma de asegurarnos de que lo comprendemos totalmente. Para seguir esta vía habría que: estudiar implementaciones anteriores, entenderlas, realizar un diseño modular propio, y llevarlo a cabo. Una vez hecho esto, habría que comprobar que es funcional y eficiente.
- Los entornos que hemos visto en el trabajo son divertidos pero no suponen un reto para nuestros algoritmos; no los llevan al límite. Por ello merece la pena enfrentarse a entornos más complejos como los juegos de Atari o los humanoides del simulador MuJoCo, también accesibles desde Open AI Gym.

Bibliografía

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[5] A. Z. Wagner, "Constructions in combinatorics via neural networks," *arXiv preprint arXiv:2104.14516*, 2021.

[6] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[7] R. Bellman, *Dynamic programming*, vol. 153. American Association for the Advancement of Science, 1966.

[8] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[9] D. Silver, S. Singh, D. Precup, and R. S. Sutton, "Reward is enough," *Artificial Intelligence*, p. 103535, 2021.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[12] H. Dong, Z. Ding, S. Zhang, H. Yuan, H. Zhang, J. Zhang, Y. Huang, T. Yu, H. Zhang, and R. Huang, *Deep Reinforcement Learning: Fundamentals, Research, and Applications*. Springer Nature, 2020. `http://www.deepreinforcementlearningbook.org`.

[13] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[14] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, 2016.

[15] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," pp. 1995–2003, 2016.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015.

[17] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[18] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[19] L.-J. Lin, "Reinforcement learning for robots using neural networks," Carnegie Mellon University, 1992.

[20] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, pp. 2613–2621, 2010.

[21] H. Van Hasselt, Y. Doron, F. Strub, M. Hessel, N. Sonnerat, and J. Modayil, "Deep reinforcement learning and the deadly triad," *arXiv preprint arXiv:1812.02648*, 2018.

[22] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, pp. 1928–1937, PMLR, 2016.

[23] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[24] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines." `https://github.com/hill-a/stable-baselines`, 2018.

[25] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," in *Psychology of learning and motivation*, vol. 24, pp. 109–165, Elsevier, 1989.

Acrónimos

A2C Synchronous Advantage Actor-Critic.

CNN Convolutional Neural Network.

DDQN Double Deep Q-Network.

DL Deep Learning.

DP Dynamic Programming.

DQN Deep Q-Network.

DRL Deep Reinforcement Learning.

ER Experience Replay.

FIFO First In First Out.

IAG Inteligencia Artificial General.

LSTM Long Short-Term Memory.

MDP Markov Decision Processes.

ML Machine Learning.

MLP Multilayer Perceptron.

PER Prioritized Experience Replay.

PGT Policy Gradient Theorem.

RL Reinforcement Learning.

SGD Stochastic Gradient Descent.

SL Supervised Learning.

TD Temporal Difference.

UL Unsupervised Learning.


UAM
UNIVERSIDAD AUTONOMA
DE MADRID