Deep Q-networks

PID_00275573

Laura Ruiz Dern

Tiempo mínimo de dedicación recomendado: 3 horas

Universitat Oberta de Catalunya

Laura Ruiz Dern

Doctora en Astronomía y Astrofísica por el Observatoire de Paris. Licenciada en Física y MSc en Astrofísica, Cosmología y Física de Partículas por la Universidad de Barcelona, y doble MSc en Gestión de Proyectos y Cooperación Internacional por la ESNECA Business School. Investigadora y científica de datos con experiencia en *data mining* y *machine learning*, y actualmente especializada en *deep learning* y *deep reinforcement learning*. Ha trabajado en varios proyectos de investigación en astrofísica y ciencia de datos del Centre National d'Études Spatiales (CNES); del Centre National de la Recherche Scientifique (CNRS), en colaboración con la European Space Agency (ESA), y del Centre Tecnològic de Catalunya (Eurecat). Desde 2014 compagina sus investigaciones con la divulgación científica y con la docencia universitaria (Observatoire de Paris, Universitat Oberta de Catalunya).

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Jordi Casas Roma

Primera edición: septiembre 2021

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Laura Ruiz Dern
Producción: FUOC

Índice

Introducción

Tras introducir las soluciones aproximadas y repasar los conceptos principales de las redes neuronales y su aplicación e influencia en el aprendizaje por refuerzo, así como las diferencias con el aprendizaje supervisado, en este módulo nos centraremos en el uso de las redes neuronales como función de aproximación de una función de valor, para el caso concreto del Q-*learning*. Son las llamadas *deep Q-networks*.

En los primeros módulos introduciremos estas *deep Q-networks*, las diferencias con el *Q-learning* y su arquitectura básica. Estudiaremos también las problemáticas que surgen y los métodos que permiten corregirlas y mejorar la arquitectura con tal de asegurar su eficiencia y máximo rendimiento.

Finalmente dedicaremos un último módulo a las mejoras y variaciones que se han hecho a lo largo de los últimos años en las *deep Q-networks*, con el objetivo de mejorar su eficiencia y rendimiento. Veremos cómo el hecho de incorporar redes neuronales profundas en el aprendizaje por refuerzo ha permitido llegar a superar a humanos expertos y campeones mundiales en muchos videojuegos Atari, así como también en los juegos Starcraft II y Go. Las aplicaciones de las *deep Q-networks* van más allá de los videojuegos, aunque su estado del arte se sigue desarrollando en este ámbito.

Objetivos

Los principales objetivos que estudiaremos en este módulo son:

1. Estudiar las *deep Q-networks* como solución aproximada.
2. Comprender las técnicas que mejoran el rendimiento y estabilidad de las *deep Q-networks*.
3. Descubrir las variaciones más importantes y el estado del arte de las *deep Q-networks*.

1. Introducción a las DQN

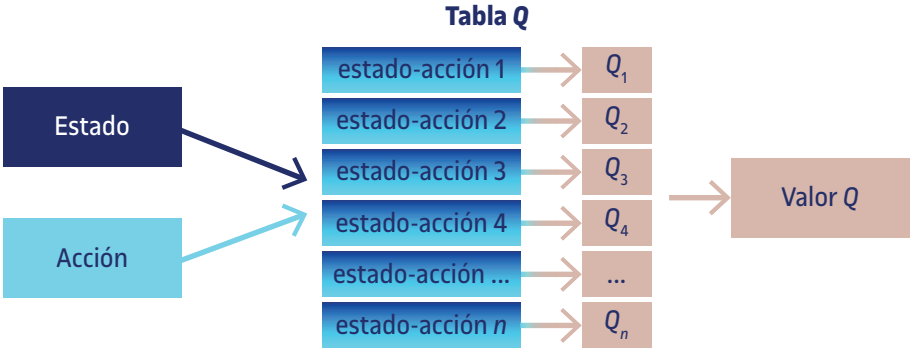
Anteriormente, vimos el método del *Q-learning*, en el cual usábamos matrices para representar la tabla con los valores de Q. Esta tabla permitía guiarnos en la elección de la mejor acción para cada estado, proporcionando la recompensa futura que esa elección nos dará. Por ejemplo, dada una tabla Q:

$$\mathbf{Q} = \begin{array}{c} \begin{matrix} a_0 & a_1 & a_2 & a_3 & a_4 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 25 & 0 \\ 0 & 0 & 17 & 0 & 74 \\ 12 & 0 & 80 & 100 & 0 \\ 21 & 0 & 0 & 74 & 62 \\ 5 & 0 & 70 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \end{matrix}$$

las acciones (a) se definían en las columnas y los estados en las filas (s). Así, por ejemplo, si en el estado s_2 tomamos la acción a_0 la recompensa será de 12, mientras que si realizamos la acción a_3 obtendremos una recompensa de 100. El objetivo siempre es elegir la acción que nos da una recompensa máxima.

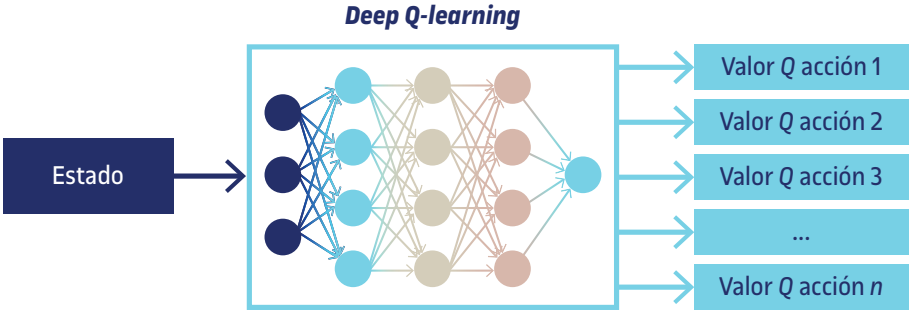
Para obtener cada valor de la tabla Q, usábamos el algoritmo de *Q-learning* en el que inicializábamos toda la tabla con ceros, $Q(s_t,a_t) = 0$, se elegía una acción en ese estado inicial s_t por la que obteníamos un nuevo estado s_{t+1} y una recompensa $Q(s_{t+1},a_{t+1})$ (siguiendo la estrategia exploración-explotación ϵ – *greedy*), y tras esa elección se actualizaban todos los valores de la tabla mediante la ecuación de Bellman. Conforme se iba actualizando la tabla, el agente podía ir tomando cada vez mejores acciones. Y así, iteravitamente, hasta terminar el aprendizaje, como se puede ver en la figura 1.

Figura 1. Esquema de *Q-learning*



Como hemos visto anteriormente, cuando tenemos un número muy grande de estados-acción el método *Q-learning* no nos sirve porque es técnicamente imposible almacenar todos los posibles valores de Q en una tabla. Pero sí que podemos mapear estado y acción en un valor por medio de funciones no lineales. La opción más común es usando una red neuronal, es decir, usar técnicas de aprendizaje profundo o DL para representar la tabla Q (figura 2). De este modo generalizamos la aproximación del valor de Q en lugar de registrar todas las posibles soluciones. Así, cada vez que se requiere el valor de Q para una acción determinada, la red lo proporciona. Y con este nuevo valor de Q se actualiza también la propia red neuronal.

Figura 2. Esquema del *deep Q-learning*



Esta combinación del *Q-learning* con el aprendizaje profundo es lo que se conoce como *deep Q-network* (DQN). Y el algoritmo de aprendizaje para aproximar la función $Q(s,a)$ con una DQN se llama, análogamente, *deep Q-learning* (DQL). Este es uno de los métodos basados en la función de valor (*value-based*) más potentes y usados dentro del aprendizaje por refuerzo profundo.

El equipo de DeepMind fue el primero que propuso combinar redes neuronales convolucionales con el aprendizaje por refuerzo (Mnih *et al.*, 2013), presentando por primera vez las DQN. Esta unión del aprendizaje profundo con el aprendizaje por refuerzo abría las puertas a tratar problemas con muchos estados, dando al agente la capacidad de observar un entorno de altas dimensiones y aprender a interactuar con él.

Lectura complementaria

V. Mnih; K. Kavukcuoglu; D. Silver; *et al.* (2013). «Playing Atari with Deep Reinforcement Learning». *Nature Journal*.

En el siguiente apartado veremos el algoritmo *Q-learning* cuando la función de aproximación es una red neuronal, y estudiaremos cuáles son los problemas que surgen de la diferencia con el aprendizaje supervisado y los distintos métodos para solventarlos.

2. Arquitectura de las DQN

El Algoritmo 1 muestra el algoritmo base de una DQN, tal y como se describe conceptualmente.

Algorithm 1 DQN base

Inicializar $Q(s,a)$ con una aproximación inicial aleatoria
Obtener el estado inicial s
for k = 1,... el proceso converge **do**
  Obtener el vector (s,a,r,s') (estado, acción, recompensa, nuevo estado)
    para la acción $a = \max_{a' \in \mathcal{A}} Q(s,a')$ (acción con mayor valor de Q)
  Fijar el valor objetivo de Q para esta acción como $r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$
  if el episodio ha finalizado **then**
    Calcular la función de pérdida según: $\mathcal{L} = [Q(s,a) - r]^2$
  else
    Calcular la función de pérdida según:
      $\mathcal{L} = \left[Q(s,a) - (r + \gamma \max_{a' \in \mathcal{A}} Q(s',a'))\right]^2$
  end if
  Actualizar $Q(s,a)$ con *backpropagation* y descenso del gradiente
end for

Pero a pesar de su sencillez, este algoritmo sufre de algunos problemas que impiden que funcione correctamente. El valor objetivo cambia con cada iteración, es decir, en DRL no hay la estabilidad que encontrábamos en el aprendizaje profundo. En DRL el agente continuamente aprende de lo que explora y cada vez sabe más sobre los distintos estados y acciones, por lo que la variable objetivo no es única, también cambia.

Con esta idea vemos que el algoritmo anterior no está teniendo en cuenta algunos factores importantes. Por un lado, no incluye la exploración-explotación y, por el otro, no tiene presente que en DRL los datos no están independiente e idénticamente distribuidos como requiere el algoritmo SGD, lo que implica a su vez una alta correlación entre estados.

En el siguiente apartado exploraremos algunos métodos que nos permiten corregir estos factores.

2.1 Técnicas de mejora del algoritmo DQN

Para resolver los mencionados problemas con el algoritmo inicial de las DQN, se proponen tres soluciones: método ϵ-*greedy*, *experience replay buffer* y red objetivo. A continuación, explicamos cada uno de ellos.

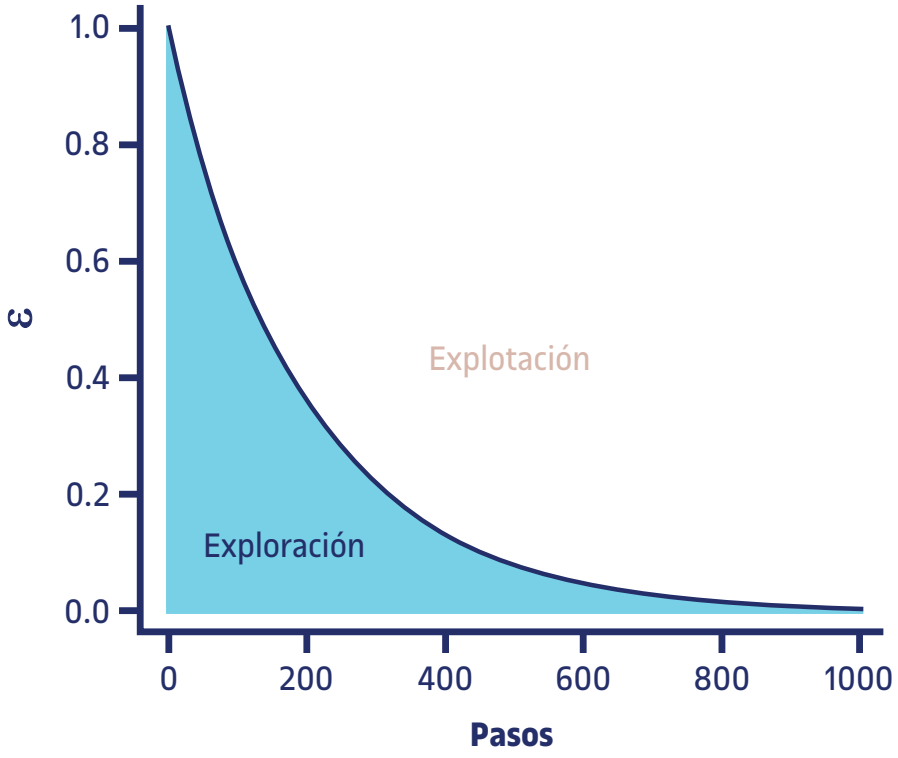
2.1.1 Método ϵ-*greedy*

Es evidente que el agente necesita interaccionar con el entorno para poder generar datos que le permitan ir aprendiendo para luego seguir interaccionando cada vez con mejores decisiones. Sin embargo, para que el aprendizaje prospere hace falta que las aproximaciones de Q sean suficientemente buenas para que esas experiencias aporten información relevante al agente. Si no se consiguen buenos valores, el agente corre el riesgo de estancarse entre decisiones malas sin mostrar ninguna mejora. Ese es el principal peligro, sobre todo al principio del entrenamiento, cuando el agente se enfrenta por primera vez al entorno sin ninguna experiencia previa. Y aun cuando lleve unas pocas experiencias todavía será insuficiente. Nos encontramos de nuevo con el problema de exploración y explotación que ya hemos visto anteriormente.

El agente necesita explorar el entorno para hacerse una idea de los estados vistos y del resultado de las acciones tomadas en ellos, y eso implica empezar tomando acciones sin experiencia previa, un poco al azar. A la vez también necesita seguir interaccionando con el entorno para recibir más datos, pero con una cierta certeza de que sus acciones aportan algo, pues no tendrá sentido elegir solo acciones al azar cuando muchas de ellas puede ya haberlas intentado antes y, por lo tanto, conocer su resultado. Por ejemplo, si el agente tiene como misión recomendar canciones al usuario, al inicio no sabrá qué le gusta y le propondrá canciones de estilos y artistas distintos al azar (exploración); pero una vez reciba el retorno del usuario tras varias selecciones al azar, podrá afinar más en sus gustos (explotación). No tendría sentido que le vuelva a proponer canciones que ya sabe que no le gustan al usuario; si solo se queda en la exploración, su rendimiento sería muy pobre.

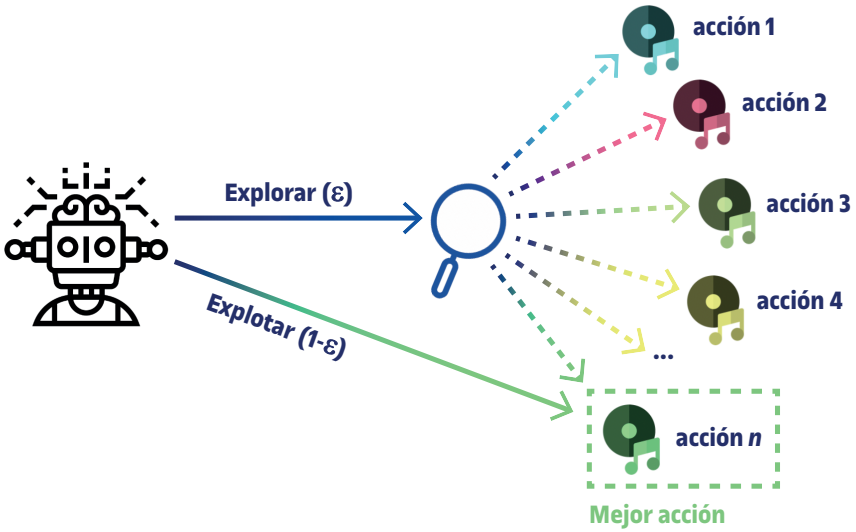
El **método ϵ-*greedy*** permite considerar estas dos necesidades del agente: explorar aleatoriamente al principio, cuando aún no se tiene suficiente información (la aproximación de Q es mala); y utilizar la aproximación de Q, dejando la aleatoriedad, para decidir las acciones cuando el aprendizaje se encuentra en un estado más avanzado. El método introduce un parámetro de probabilidad ϵ con el que indica cuándo pasar de una política aleatoria a una política Q. Cuando el valor es 1, todas las acciones que se toman son aleatorias. Conforme se vaya reduciendo esta probabilidad, el agente pasará a tomar más acciones acorde con la política Q (figura 3).

Figura 3. Transición de las elecciones del agente de exploración a explotación a lo largo del proceso de aprendizaje, en función de ϵ



La figura 4 muestra este proceso de forma sencilla con el ejemplo de las canciones. Cada vez que el agente tenga que elegir una acción (qué canción ofrezco al usuario), considerará dos opciones: explorar o explotar, es decir, elegir al azar una de las múltiples que hay (con igual probabilidad todas de ser elegidas), o elegir la mejor (la más afín al usuario, según lo aprendido hasta el momento). Su decisión entre explorar o explotar dependerá del valor de ϵ.

Figura 4. Representación del método ϵ-*greedy* con el ejemplo de las canciones



2.1.2 *Experience replay buffer*

El problema principal del algoritmo DQN inicial es que los datos que se introducen a la red neuronal son secuenciales. El agente interacciona con el entorno y obtiene una recompensa; pero la siguiente acción que se toma será consecuencia de la primera. Según lo que ocurra fruto de la primera acción elegirá una nueva acción. Y así sucesivamente. Los estados están fuertemente correlacionados.

En una DQN, cada vez que hay una interacción con el entorno se actualiza el valor de Q, igual que hacíamos en *Q-learning* con el método tabular. Dicho de otro modo, el nuevo valor de Q sobreescribe el anterior; las experiencias pasadas quedan sobreescritas con las nuevas experiencias. En la práctica eso significa que el agente no recuerda nada de lo que pasó en los estados anteriores. Imaginemos, por ejemplo, un juego con distintos niveles: el agente pasaría al segundo nivel (nuevo estado) pero olvidaría lo que hizo o pasó en el primer nivel. Pero lo que nos interesa es que el agente aprenda y que sea mejor en cada nivel o cada partida.

Nuestro objetivo es aproximar una función más compleja y no lineal $Q(s,a)$ con una red neuronal. Para ello seguimos los mismos pasos que en *Q-learning* actualizando los valores con la ecuación de Bellman igual que si fuera un problema de aprendizaje supervisado. Pero en el momento en que queremos calcular la función de pérdida y aplicar SGD para mejorar los parámetros, olvidamos que el método del descenso del gradiente requiere que los datos sean separables. Es decir, tienen que estar independiente e idénticamente distribuidos (*i.i.d.*). Sin embargo, en DRL los datos no tienen ninguna de estas propiedades:

- no son independientes porque, aunque almacenáramos una cantidad de datos previos al estado actual, dado que los estamos introduciendo de forma secuencial, están muy relacionados entre ellos

- tampoco tendrán una distribución idéntica a los ejemplos proporcionados por la política que esperamos aprender. En DRL recolectamos los datos ya sea por la política actual en un estado dado, o de forma aleatoria, o ambas a la vez (ϵ-*greedy*), por lo que no tendrán nada que ver con su distribución según la política final.

Pero lo cierto es que tampoco nos interesa recolectar datos de forma aleatoria. De ser así, las probabilidades de acertar en cada estado con la mejor acción (la que nos proporciona mayor recompensa) seguramente sean muy bajas. Sería como jugar un partido de tenis y moverse hacia un lado (acción) sin tener ni idea de por dónde viene o puede venir la pelota. Cada punto de la trayectoria de la pelota sería un nuevo estado; si solo conocemos estados aleatorios del partido es prácticamente imposible saber hacia dónde va, necesitamos una secuencia de su movimiento. Vemos claro que las probabilidades de coincidir

la posición del agente y la de la pelota se reducen al mínimo, ¡y las de ganar el partido son todavía peor! ¡El agente dará más «raquetazos» al aire que a la pelota!

La solución al problema pasa por combinar ambas necesidades:

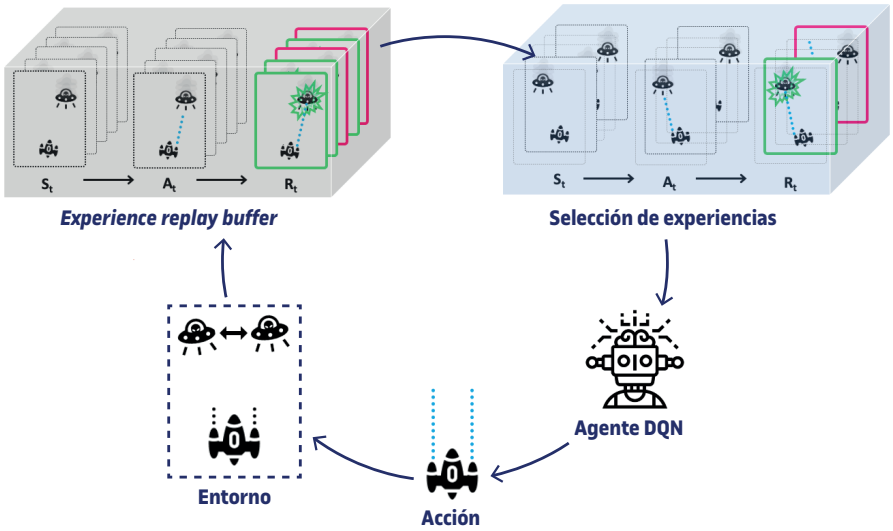
1) almacenar una cierta cantidad de experiencias mientras el agente va experimentando, de modo que este pueda aprender de experiencias recientes.

2) seleccionar de forma aleatoria un subconjunto de estos datos almacenados para la red neuronal con tal de reducir la correlación entre ellos.

Esta técnica se conoce como *buffer* de repetición de experiencias, en inglés ***experience replay buffer*** (llamada también *replay buffer* o *experience replay*). Se guardan experiencias pasadas en un *buffer* de tamaño fijo del cual extraemos un subconjunto aleatorio para pasarlo a la red neuronal. Al tener un tamaño fijo, las experiencias se irán acumulando en el *buffer*, eliminándose las más antiguas conforme entren nuevas experiencias.

La figura 5 muestra esquemáticamente el funcionamiento de este método. Imaginemos un simple juego de eliminar extraterrestres. El agente interaccionará con el entorno con una acción (disparar donde esté el extraterrestre), y las experiencias resultantes (conjunto de estado, acción y recompensa) se almacenarán en el *replay buffer*. La red neuronal seleccionará, posteriormente, un subconjunto aleatorio de estas experiencias para entrenar y mejorar el aprendizaje.

Figura 5. Esquema del funcionamiento del *experience replay buffer*



Aplicando el *replay buffer* conseguiremos tener unos datos de entrenamiento un poco más independientes entre ellos (se hace una selección aleatoria de los datos almacenados en el *buffer*, rompiendo así la correlación temporal),

y a la vez serán suficientemente recientes para que estén casi idénticamente distribuidos (la política asociada en el estado actual será más parecida a la política final conforme se llega al final del proceso).

2.1.3 Red objetivo

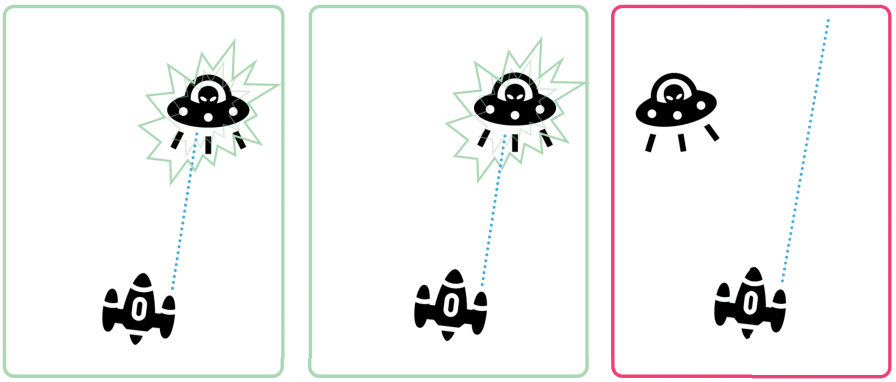
En el subapartado anterior hablábamos de la correlación entre estados debido a que son secuenciales y necesitábamos pasar a la red neuronal unos datos «más independientes». Pero esta correlación tiene otro efecto (también relacionado con que los datos no son *i.i.d.*), y es que cada acción tomada afecta directamente al siguiente estado. Los vectores (s,a,r,s') de un estado y el siguiente serán muy similares, casi indistinguibles para la red neuronal.

i.i.d.

En inglés, *independent and identically distributed variables*

Esto puede causar que cada vez que actualicemos el valor de Q, su valor vaya empeorando en los siguientes estados en lugar de mejorar, provocando un entrenamiento muy inestable. Estaremos viciando nuestro agente a tomar acciones similares a las que hizo en el estado anterior, independientemente de la nueva situación. Retomemos el juego del extraterrestre del subapartado anterior. El juego consiste, como se ilustra en la figura 6, en eliminar extraterrestres desde una nave espacial. Pero los extraterrestres a veces aparecen a la izquierda y a veces a la derecha, por lo que según donde esté el extraterrestre el agente deberá disparar con el arma derecha o la izquierda. Cuando empieza el entrenamiento (panel izquierdo de la figura 6) el extraterrestre aparece a la derecha y el agente dispara con el arma derecha, acertando. Después otro extraterrestre aparecerá a la derecha con un 70 % de probabilidades, por lo que el agente vuelve a disparar a la derecha (panel central de la figura 6). Y así sucesivamente. El problema es que si en sucesivos estados ocurre esto, el agente no aprende cuándo disparar a la izquierda (solo un 30 % de los extraterrestres aparecerán a la izquierda). El valor de Q por disparar a la derecha siempre será mucho mayor. Si el agente no ve muchos casos de disparar a la izquierda, terminará disparando siempre a la derecha independientemente de dónde esté el extraterrestre (panel de la derecha en la figura 6), lo cual no tiene ningún sentido.

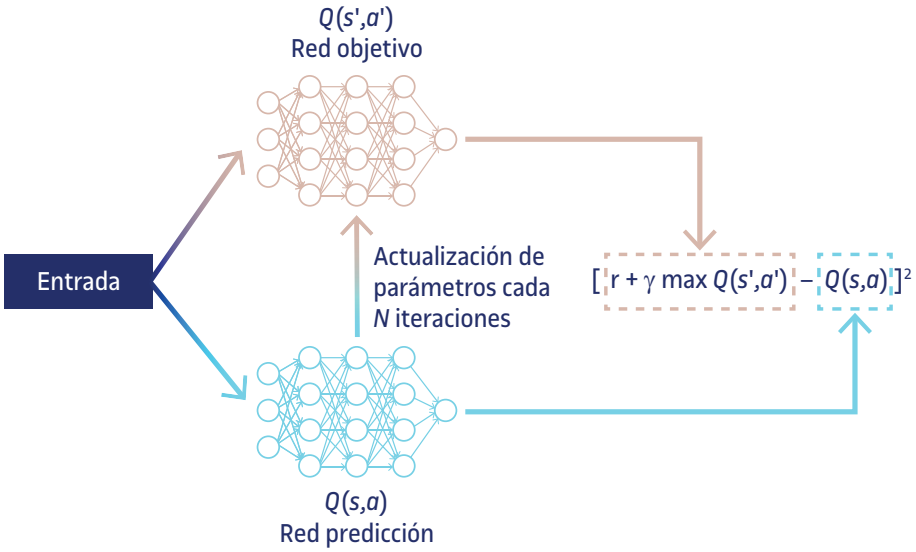
Figura 6. Ejemplo del problema de la correlación entre estados



Para solucionar este problema podemos introducir una segunda red neuronal $\hat{Q}$ para mejorar el aprendizaje. En lugar de obtener el valor objetivo $Q(s',a')$ y el valor predicho $Q(s,a)$ con la misma red neuronal, calcularemos el valor objetivo con esta segunda red neuronal a la que llamaremos **red objetivo** o *target network*.

Esta segunda red $\hat{Q}$ será una copia de la principal pero con los pesos fijos. Cada un cierto número de iteraciones N (entre 1000 y 10000, generalmente) los coeficientes/parámetros de la red principal (la red de predicción) se introducen en la red objetivo y esta explora durante un tiempo el espacio de estados con esos valores (figura 7). Dicho de un modo más visual, lo que estaremos haciendo es parar el juego en esta segunda red neuronal para extraer más información del espacio de estados. Las experiencias aprendidas en esta red objetivo también se pueden ir almacenando en el *replay buffer* para poder emplearlas en el proceso de aprendizaje.

Figura 7. Esquema de la red objetivo



Así, el valor objetivo $Q(s',a')$ (fijo durante un cierto tiempo, N iteraciones) necesario en la ecuación de Bellman será proporcionado ahora por la red objetivo $\hat{Q}$, y calcularemos la pérdida o error por cada acción con este valor. Pasadas las N iteraciones, se sincronizará con la red principal y volverá a recibir nuevos coeficientes con los que explorar el espacio de estados de nuevo.

Con este mecanismo se consigue estabilizar el entrenamiento y generalizar mejor los patrones en los datos, evitando centrarse mucho en una región del espacio de estados, lo que causa, como hemos visto, un estancamiento del agente que termina por repetir siempre la misma acción.

2.2 Algoritmo DQN final

El algoritmo DQN propuesto en el artículo original de DeepMind publicado en 2013 donde testeaban el rendimiento y potencial de este tipo de modelo en siete juegos Atari distintos, no incluía ninguna red objetivo. Fue posteriormente, en 2015, que Mnih *et al*. introdujeron esta mejora en el algoritmo, implementándolo en 49 juegos de Atari.

Lectura complementaria

V. Mnih; K. Kavukcuoglu; D. Silver; *et al.* (2015). «Human-level control through deep reinforcement learning». *Nature Journal*.

El Algoritmo 2 muestra el algoritmo de una DQN teniendo en cuenta las mejoras hechas en el artículo de 2015.

Algorithm 2 DQN con ϵ-*greedy*, *experience replay* y *target network*

Inicializar el *replay buffer D*
Inicializar Q_θ con pesos aleatorios θ
Inicializar $\hat{Q}_{\theta^-}$, idéntica a Q, $\theta^- = \theta$
Inicializar ϵ
for episodio = 1 a M **do**
  for t = 1 a T **do**
    Según la probabilidad ϵ seleccionar una acción aleatoria a dado un estado s_t (*epsilon-greedy*)
    Ejecutar la acción a y observar la recompensa r y el siguiente estado s'
    Almacenar el vector (s,a,r,s') en el *replay buffer D*
    Extraer subconjunto aleatorio (s_j,a_j,r_j,s'_j) de J transiciones del *replay buffer D*
    for cada transición en el buffer **do**
      if el episodio ha terminado **then**
        Calcular variable objetivo $y_j = r_j$
      else
        Calcular variable objetivo $y_j = r_j + \gamma \max_{a^- \in \mathcal{A}} \hat{Q}(s',a^-;\theta^-)$
      end if
    end for
    Calcular la pérdida $\mathcal{L} = \frac{1}{J}\sum_{j=0}^{J-1}\left[Q(s_j,a_j;\theta) - y_j\right]^2$
    Actualizar Q con SGD minimizando la pérdida
    Cada N iteraciones clonar los pesos de Q a $\hat{Q}$ (i.e. $\theta^- = \theta$)
  end for
end for

Las DQN fueron el primer método de DRL propuesto por DeepMind en 2013 y 2015. Los prometedores resultados obtenidos con las técnicas usadas en DQN dieron pie a un sinfín de opciones en el campo del aprendizaje por refuerzo profundo. Las DQN usan, por lo general, redes neuronales convolucionales, especialmente cuando los estados son imágenes. Pero estas se pueden combinar de múltiples formas dando lugar a una gran variedad de modelos y mejoras. En el siguiente apartado veremos algunos de los modelos más conocidos y usados.

3. Variaciones de las DQN

En este apartado estudiaremos las mejoras más importantes que se han hecho a las DQN desde su nacimiento, siempre con el objetivo de mejorar su estabilidad e incrementar su rendimiento. Sin entrar en detalles matemáticos, introduciremos el concepto fundamental de cada variación en orden cronológico, y veremos cómo la combinación de estas permite crear modelos más robustos y consistentes, con resultados sustancialmente mejores.

3.1 Aprendizaje multipaso

Una primera variación que podemos incluir a las DQN es el **método multipaso** (o *n-step* en inglés) ya conocido de las aproximaciones tabulares (Sutton, 1988), con tal de mejorar y reducir el tiempo de aprendizaje. La idea es hacer la actualización de los parámetros, de la DQN, con secuencias de transición más largas.

Empecemos recordando la ecuación de Bellman:

$$Q(S_t,A_t) = R_t + \gamma \max_a Q(S_{t+1},A_{t+1}) \quad (1)$$

Si ahora escribimos cómo sería la siguiente actualización, tendríamos:

$$Q(S_{t+1},A_{t+1}) = R_{a,t+1} + \gamma \max_{a'} Q(S_{t+2},A_{t+2}) \quad (2)$$

donde $R_{a,t+1}$ es la recompensa obtenida en $t+1$ tras realizar la acción a.

Si sustituimos esta segunda ecuación en la primera, $Q(S_t,A_t)$ se puede describir como:

$$Q(S_t,A_t) = R_t + \gamma \max_a [R_{a,t+1} + \gamma \max_{a'} Q(S_{t+2},A_{t+2})] \quad (3)$$

Consideremos ahora que la acción a que hemos tomado en $t+1$ era ya la mejor opción, la más óptima. En ese caso, el primer $\max_a$ ya no nos hace falta, y $R_{a,t+1} = R_{t+1}$. Así, podemos simplificar la ecuación anterior un poco más:

$$Q(S_t,A_t) = R_t + \gamma[R_{t+1} + \gamma \max_{a'} Q(S_{t+2},A_{t+2})] \quad (4)$$

$$= R_t + \gamma R_{t+1} + \gamma^2 \max_{a'} Q(S_{t+2},A_{t+2}) \quad (5)$$

De esta expresión se observa claramente la propiedad recursiva de la ecuación de Bellman. Podríamos ir añadiendo componentes por cada transición, hasta tener una secuencia de n transiciones, y que la actualización de la función de valor no fuera sobre un único paso sino sobre múltiples, de modo que con una n grande aceleraríamos mucho el proceso de aprendizaje.

Lectura complementaria

F. J. Hernández-García; R. S. Sutton (2019). *Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target*. University of Alberta.

Pero la realidad es que con una n grande el modelo no convergería. Fijémonos que al omitir el término $\max_a$ estamos asumiendo que la acción que cogemos siempre es la óptima, sin validarlo. Pero hay muchas ocasiones, como al inicio del proceso cuando el agente elige acciones más aleatoriamente, que podría perfectamente estar obteniendo valores de Q inferiores al valor óptimo. Si esto lo hacemos muchas veces, n, en una misma transición, la probabilidad de empeorar la actualización en lugar de mejorarla aumenta considerablemente.

En consecuencia, tenemos que por un lado nos interesa un valor de n mayor que 1 para poder acelerar el proceso, pero por el otro lado tampoco queremos que sea muy alto para asegurar la convergencia. En la práctica, n suele oscilar entre valores de 2-3, aunque siempre dependerá de cada problema y hará falta ajustarlo como un hiperparámetro más.

3.2 *Double Q-network*

Una de las mejoras propuestas en el artículo de 2015 de DeepMind fue la introducción de una segunda red neuronal al proceso, la red objetivo (*target network*). Esta red nos proporcionaba la mejor acción a tomar y su valor Q, con el que calculábamos la pérdida con la ayuda de la ecuación de Bellman. El valor objetivo y lo obteníamos según:

$$y = r + \gamma \max_{a^-} \hat{Q}(s',a^-;\theta^-) \quad (6)$$

Pero esta técnica no nos asegura que la acción que ha elegido la red objetivo es realmente aquella que en nuestro problema, en la red principal, nos da el valor más alto de Q. Este hecho provoca que la DQN siempre tenga cierta tendencia a sobreestimar los valores de Q, lo cual perjudica el rendimiento y los resultados obtenidos, llevándonos a políticas subóptimas.

La solución es un pequeño cambio en quién proporciona la acción a tomar. En lugar de dejar a la red objetivo que elija la acción con valor más alto de Q (que, repetimos, no tiene por qué ser la acción con valor más alto de Q para la red principal), lo que haremos ahora es que será la misma red principal la que elegirá la acción con mayor valor de Q, y la red objetivo la que proporcionará

el valor objetivo de Q para esa acción, la que dirá qué recompensa tiene esa acción elegida por la DQN.

De este modo, la nueva fórmula para obtener el valor objetivo de Q será:

$$y = r + \gamma \hat{Q}(s', \text{argmax}_a Q(s', a; \theta); \theta^-) \tag{7}$$

Este nuevo algoritmo, cuyo pseudocódigo se muestra en el Algoritmo 3, es el que conocemos como las **redes DQN dobles o *double Q-networks*** (DDQN), y fue presentado por primera vez por el equipo de DeepMind en el artículo de Hasselt *et al.* (2015), probando que así se eliminaba completamente el problema de la sobreestimación de Q.

Lectura complementaria

H. van Hasselt; A. Guez; D. Silver (2015). *Deep Reinforcement Learning with Double Q-learning*. Google DeepMind.

Algorithm 3 Double DQN

Inicializar red principal Q_θ con pesos aleatorios θ
Inicializar $\hat{Q}_{\theta^-}$, idéntica a Q, $\theta^- = \theta$
Inicializar el *replay buffer D* vacío
for episodio = 1 a M **do**
  for t = 1 a T **do**
    Observar estado s y seleccionar $a \approx \pi(a,s)$ (*epsilon-greedy*)
    Ejecutar la acción a y observar la recompensa r y el siguiente estado s'
    Almacenar el vector (s,a,r,s') en el *replay buffer D*
    Extraer subconjunto aleatorio (s_j,a_j,r_j,s'_j) de J transiciones del *replay buffer D*
    for cada transición en el buffer **do**
      if el episodio ha terminado **then**
        Calcular variable objetivo $y_j = r_j$
      else
        Calcular variable objetivo $y_j = r_j + \gamma \hat{Q}(s', \text{argmax}_a Q(s', a; \theta); \theta^-)$
      end if
    end for
    Calcular la pérdida $\mathcal{L} = \frac{1}{J} \sum_{j=0}^{J-1} \left[Q(s_j,a_j;\theta) - y_j \right]^2$
    Actualizar Q con SGD minimizando la pérdida
    Cada N iteraciones clonar los pesos de Q a $\hat{Q}$ (i.e. $\theta^- = \theta$)
  end for
end for

En la tabla 1 se resumen las principales diferencias entre el método red objetivo y la DDQN.

Tabla 1. Diferencia entre el método red objetivo y la DDQN

Modelo	Red de predicción	Red objetivo
Método red objetivo	Proporciona los hiperparámetros a la red objetivo cada N iteraciones	Determina la mejor acción a tomar y proporciona su valor $Q(s',a')$
DDQN	Proporciona los hiperparámetros a la red objetivo cada N iteraciones y la acción con mayor Q según esta red	Calcula el valor de $Q(s',a')$ para la acción elegida por la red de predicción

3.3 *Prioritized experience replay*

En el apartado anterior veíamos que uno de los problemas en DRL era que los datos son secuenciales y que no están idénticamente distribuidos. Para resolverlo usábamos la técnica de *experience replay buffer* con la que guardábamos un conjunto de experiencias pasadas del agente, pero suficientemente recientes, en un *buffer* de tamaño fijo. Posteriormente se seleccionaban aleatoriamente unas cuantas experiencias del *buffer* con las que se entrenaba la DQN, rompiendo así un poco la correlación temporal entre estados seguidos.

Pero con este procedimiento podemos darnos cuenta de que una selección aleatoria del conjunto de entrenamiento nos proporciona una distribución uniforme que nunca favorecerá experiencias que puedan ser más relevantes. Todas son tratadas por igual.

La idea que subyace en esta técnica es intentar que aquellas experiencias que puedan ser más importantes en el proceso de aprendizaje tengan mayor representación y prioridad en el conjunto de entrenamiento, mejorando así la calidad de la política de la DQN y su convergencia. Todas las nuevas experiencias que entren en el *buffer* también tendrán que tener prioridad alta, que luego se irá ajustando según la pérdida obtenida para ese ejemplo al actualizar con la ecuación de Bellman.

Esta mejora de la técnica *experience replay buffer* se conoce como **repetición de la experiencia prioritaria**, más conocida en inglés por ***prioritized experience replay*** (PER), y fue expuesta por primera vez por Schaul *et al.* (2015) del equipo de DeepMind.

Lectura complementaria

T. Schaul; J. Quan; I. Antonoglou; *et al.* (2016). *Prioritized Experience Replay*. Google DeepMind.

Necesitamos un criterio que nos permita detectar y seleccionar esas experiencias más relevantes, es decir, esas experiencias que presentan mayor diferencia entre nuestra predicción y el valor objetivo (es decir, de las que todavía se puede aprender mucho más), pero a la vez evitando que siempre se elijan las mismas experiencias (causaríamos sobreajuste).

Para ello utilizamos una priorización estocástica que nos dé la probabilidad de cada ejemplo en el *buffer* de ser elegido:

$$P(i) = \frac{{p^{\alpha}}_i}{\sum_k {p^{\alpha}}_k} \tag{8}$$

donde p_i es la prioridad del i-ésimo ejemplo en el *buffer* que puede calcularse a partir del error TD $p_i = |\delta_i| + \epsilon$ (con ϵ una constante), y α un valor que nos dice cuánta importancia le damos a esa prioridad y que formará parte de los hiperparámetros a ajustar. Cuando $\alpha = 0$ volvemos al caso clásico del *experience replay* con una distribución de datos uniforme, y cuando $\alpha = 1$ estaremos seleccionando únicamente las experiencias con máxima prioridad. Se recomienda un valor alrededor de 0.6 para empezar.

En cada iteración obtendremos esta distribución de probabilidad con la que seleccionaremos el conjunto de datos para entrenar.

Es natural pensar que procediendo así estamos introduciendo un sesgo clarísimo en la selección de datos hacia las experiencias con prioridad más alta. Este efecto necesita ser compensado para que el SGD funcione bien y no causemos sobreajuste. Para corregir este sesgo utilizaremos lo que se llama un muestreo de importancia (en inglés *importance sampling*, IS) de los pesos, con el fin de ajustar las actualizaciones reduciendo los pesos de los ejemplos vistos más veces. El valor del peso para cada ejemplo viene dado por:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta} \tag{9}$$

donde N es el tamaño del *buffer*, P(i) la distribución de probabilidad, y β un hiperparámetro que permite establecer en qué proporción este muestreo de importancia de los pesos afecta al aprendizaje. Conviene que β sea prácticamente 0 al inicio del aprendizaje y que vaya creciendo hasta 1 a lo largo del proceso de entrenamiento, ya que estos pesos serán cada vez más importantes hacia el final del aprendizaje cuando los valores de Q empiezan a converger.

Así, los pesos pertenecientes a experiencias de alta prioridad necesitarán ajustarse muy poco en cada iteración porque serán experiencias que la red verá muchas veces, todo lo contrario de los pesos de las experiencias menos prioritarias.

Gracias al método de PER, el algoritmo de la red neuronal (una DDQN, por ejemplo) será mucho más robusto permitiendo una convergencia más rápida hacia resultados más óptimos. En el siguiente subapartado veremos otra arquitectura de DQN que combinándola con PER también suele mejorar su rendimiento.

3.4 *Dueling Q-network*

Como hemos ido diciendo en repetidas ocasiones, el objetivo de la DQN clásica es calcular el valor Q para cada acción posible y elegir aquella acción con el valor de Q mayor. Pero si nos paramos a pensar un momento, ¿realmente en todos los estados es necesario calcular el valor de Q de todas las acciones? Puede haber estados en los que no haya ninguna acción que sea relevante, por lo que no importa el valor de Q que tengan, nos podemos ahorrar ese cálculo y ganar en eficiencia.

Para ello, la clave está en descomponer la función Q en dos componentes:

$$Q(s,a) = A(s,a) + V(s) \tag{10}$$

donde,

- $V(s)$: el valor de estar en el estado s, equivalente a la recompensa esperada con descuento que es posible conseguir en el estado s

- $A(s,a)$: ventaja de tomar la acción a en el estado s, cuánto mejor es esa acción a (qué cantidad extra de recompensa nos da) respecto a las otras posibles acciones

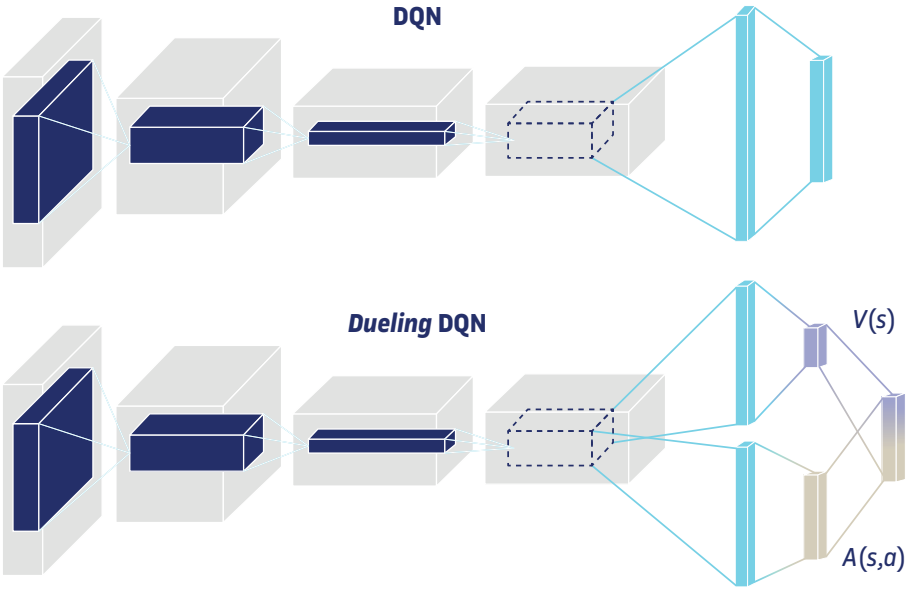
Este tipo de arquitectura que permite obtener separadamente la estimación de la función de valor de estado y la función de ventaja es la que se conoce como *Dueling Q-Network* o *dueling* DQN, y fue propuesta de nuevo por DeepMind en el artículo de Wang *et al.* (2016).

Lectura complementaria

Z. Wang; T. Schaul; M. Hessel; *et al.* (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. Google DeepMind.

En la figura 8 se muestra la arquitectura de esta red. El primer bloque es similar al de la DQN clásica, pero al final, la red toma las características de todo ese primer bloque y las procesa por dos caminos independientes, en dos subredes paralelas. Una predice el valor de $V(s)$ (un número), y la otra los valores individuales de la ventaja de cada acción. Después, ambos resultados se combinan en una capa específica que agrega $V(s)$ a cada valor de $A(s,a)$ obteniendo así finalmente el valor de $Q(s,a)$. Finalmente la *dueling* DQN usa la técnica de actualización de la *double*-DQN, y se entrena la red como ya conocemos.

Figura 8. Comparativa de la arquitectura de una DQN clásica con la *dueling* DQN



Es importante tener presente que esta agregación de $V(s)$ y $A(s)$ no se puede hacer sumando ambos como en la ecuación anterior. Esa ecuación no es identificable, es decir, que no se puede recuperar V y A de forma única partiendo de Q. Es lo que se conoce como **problema de identificabilidad**. Para solucionarlo, substraemos la ventaja media de todas las acciones posibles en ese estado.

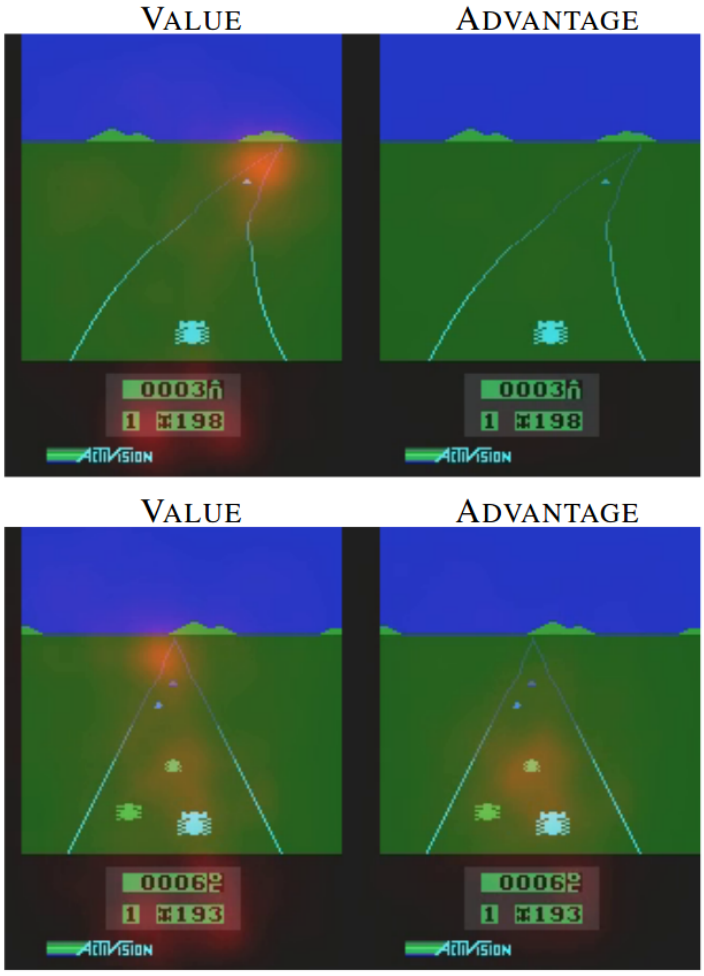
$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{N_{acciones}}\sum_{a'} A(s,a';\theta,\alpha)\right) \qquad (11)$$

donde θ son los parámetros típicos de la red, α los parámetros de la subred de la ventaja A, y β los parámetros de la subred de V.

Con este desacoplamiento de $Q(s,a)$ la *dueling* DQN es capaz de aprender qué estados son o no relevantes sin tener que aprender el efecto de cada acción para cada estado, lo cual, como decíamos al principio de este apartado, es práctico para aquellos estados en los que cualquier acción es equivalente y no causa ningún cambio en el entorno. De esta manera, el entrenamiento será mucho más rápido que en una DQN clásica o una DDQN.

Para entender mejor el funcionamiento de este tipo de red neuronal, podemos coger como ejemplo directamente el que presentan Wang *et al.* (2016) en su artículo, basado en el juego de Enduro (figura 9).

Figura 9. Ejemplo de la función de $V(s)$ y de $A(s,a)$ en el juego de Enduro



Fuente: Wang *et al.* (2016)

En esta figura tenemos dos escenarios, uno en el que el coche solo tiene a otro coche por delante pero a lo lejos (arriba), y otro en la que tiene un coche que le pasa por la izquierda y otro que se le acerca por delante invadiendo su carril (abajo). En el lado izquierdo de ambos escenarios se muestra en qué se fija (punto anaranjado) la función $V(s)$, mientras que en el lado derecho vemos en qué se fija la función $A(s,a)$.

Podemos observar que la función $V(s)$ se fija más en los coches que se encuentran lejos que en los cercanos, donde empiezan a verse, en cualquiera de los escenarios. También se fija en la puntuación. En cambio, la función $A(s,a)$ en el primer escenario no señala nada porque no hay ningún coche delante y cercano que sea motivo para forzar una acción; cualquier acción en ese estado será totalmente irrelevante. Pero en el segundo escenario sí que señala el coche que está invadiendo el carril, indicando que hay una acción a tomar para evitar la colisión. En este ejemplo, los valores Q del estado del primer escenario no serían calculados y la *dueling* DQN pasaría directamente a evaluar el siguiente estado.

Finalmente cabe mencionar que las *dueling* DQN se pueden combinar con el *prioritized experience replay* que introducíamos en el subapartado anterior. Como hemos visto, ambas aproximaciones tratan aspectos totalmente diferentes del proceso de aprendizaje, por lo que su combinación es una opción con mucho potencial. Así lo muestran Wang *et al*. en su artículo con varios experimentos en 57 juegos de Atari, comparando los resultados obtenidos por un humano, con una DQN, una DDQN, una *dueling* DQN, y una *dueling* DQN + PER. Una gran mayoría de juegos presentan un rendimiento significativamente mejor cuando se combinan *dueling* DQN + PER.

3.5 DQN categórica

Otra alternativa de mejora de las DQN es pensar en distribuciones de probabilidad del valor de Q en lugar de los valores de Q en sí mismos. En entornos más complejos nos podemos encontrar con situaciones estocásticas que devuelvan diferentes valores de Q con distintas probabilidades, y que pueden justificar trabajar con distribuciones en lugar de valores individuales, proporcionando así un aprendizaje más estable que permita un mejor rendimiento final.

Lectura complementaria

M. G. Bellemare; W. Dabney; R. Munos (2017). *A Distributional Perspective on Reinforcement Learning*. Google DeepMind.

Imaginemos que el jefe de una empresa tiene que viajar regularmente a París desde Barcelona. Tiene tres opciones de transporte: ir en avión, en tren o en coche. Cada una de estas opciones es una acción que el agente puede tomar. Pero ¿cuál de ellas es mejor? ¿Cuál le proporciona una mayor recompensa? El tiempo de viaje es un factor importante: con el avión tarda unas 4 horas (entre llegar al aeropuerto, pasar el control, esperar a subir, el viaje de 1 hora y 25 minutos, y el viaje del aeropuerto de destino al lugar de la reunión), en tren

unas 7 horas y con el coche unas 10 horas. En primera instancia uno diría que el avión es la mejor opción porque tarda menos. Pero imaginemos que una de las veces que el agente coge el avión, se encuentra con una huelga de personal de tierra y se produce un retraso de 3 horas del avión. Y en otra ocasión hay un buen temporal y el avión no puede salir a la hora. Y otro día se encuentra con un atasco al llegar entre el aeropuerto de destino y el lugar de la reunión y pierde dos horas. Podríamos imaginar situaciones parecidas también con el tren y con el coche. Analizándolo así observamos que en una situación real la recompensa de coger un transporte u otro cada vez (es decir, en cada estado) no siempre será la misma. Nos damos cuenta de que describiremos mejor la recompensa si hablamos de una distribución de probabilidad que dependa del estado y de la acción a tomar. Si hay un buen temporal, la recompensa por coger el tren puede ser mayor; o si hay una manifestación en la frontera, la recompensa por coger el avión será más grande que no el tren o el coche, y si hay huelga de transportes de tren y avión, la recompensa será mayor por coger el coche. Si solo conocemos el tiempo de desplazamiento, el avión siempre será la acción elegida. ¿Pero es la mejor opción? Si en cambio conocemos el número de veces que hay problemas, cancelaciones o huelgas con el avión, comparado con el tren, y vemos la distribución entera, en media quizás el tren es mejor si el jefe de la empresa quiere asegurarse que llegará a la reunión a la hora establecida. Estadísticamente hablando, nos interesa la varianza de cada acción.

Así, ¿no sería mejor en estos casos considerar la distribución de probabilidades del valor para una acción en lugar de predecir siempre el valor medio?

Consideremos el retorno aleatorio Z cuyo valor esperado es Q. Por analogía con la ecuación de Bellman podemos llamar Z al **valor de la distribución** y generalizar la ecuación según:

$$Z(S_t,A) \overset{\mathrm{D}}{=} R(S_t,a) + \gamma Z(S_{t+1},a') \tag{12}$$

donde tanto el valor como la recompensa ahora son distribuciones de probabilidad en lugar de valores concretos como teníamos antes. Y si antes hablábamos de la función de valor de la política V_π ahora su análoga es la distribución de valor Z_π. Esta distribución Z_π es la que nos permite describir la aleatoriedad intrínseca de las interacciones del agente con su entorno. La evaluación de la política π se realiza por medio del operador de evaluación de política T_π que, junto con Z, define la distribución compuesta $T_\pi Z$. Esta distribución compuesta viene definida por tres fuentes de aleatoriedad independientes: la recompensa R, la transición entre estados P_π y el siguiente valor de la distribución $Z(S_{t+1},A_{t+1})$; y es la que luego utilizaremos para entrenar la red y obtener mejores predicciones de $Z\pi$ para cada acción del estado en cuestión. En realidad el proceso termina siendo el mismo que en *Q-learning*, pero adaptando la función de pérdida a una capaz de comparar distribuciones (por ejemplo, la métrica Wasserstein, o la divergencia de Kullback-Leibler o entropía cruzada).

A esta aproximación se la conoce como **DQN categórica** (C51) o RL distribucional, (en inglés, *categorical 51-atom* DQN).

El Algoritmo 4 muestra el proceso de una DQN categórica. Llegar a este algoritmo implica un desarrollo y unas consideraciones matemáticas mucho más extensas y complejas comparado con los algoritmos estudiados en los subapartados anteriores. En este curso no entraremos en tal detalle, pero se invita al lector curioso a leer el artículo recomendado de Bellemare *et al.* (2017) de DeepMind en el cual se introdujo esta distinta manera de abordar el aprendizaje por refuerzo.

Algorithm 4 DQN categórica

Entrada: una tupla (S_t, A_t, r_t, S_{t+1}), $\gamma_t \in [0,1]$
$Q(S_{t+1}, a) := \sum_i z_i p_i(S_{t+1}, a)$
$a^* \leftarrow \arg\max_a Q(S_{t+1}, a)$
$m_i = 0$, $i \in 0,\ldots,N-1$
for $j \in 0,\ldots,N-1$ **do**
  Calcula la proyección de la muestra de la actualización de Bellman $\hat{T}z_j$ en el soporte de z_j:
  $\hat{T}z_j \leftarrow [r_t + \gamma_t z_j]_{V\text{MIN}}^{V\text{MAX}}$
  $b_j \leftarrow (\hat{T}z_j - V_{\text{MIN}})/\Delta z$, con $b_j \in [0, N-1]$
  $l \leftarrow \lfloor b_j \rfloor$ (función suelo: valor entero inferior más cercano a b_j)
  $u \rightarrow \lceil b_j \rceil$ (función techo: valor entero superior más cercano a b_j)
  Obtener la distribución de probabilidad de $\hat{T}z_j$
  $m_l \leftarrow m_l + p_j(S_{t+1}, a)(u - b_j)$
  $m_u \leftarrow m_u + p_j(S_{t+1}, a)(b_j - l)$
end for
Salida: $-\sum_i m_i \log p_i(S_t, a_t)$ (entropía cruzada)

3.6 *NoisyNet*: red con ruido

Anteriormente vimos el problema de exploración-explotación, por el que al inicio el agente está obligado a seleccionar acciones aleatorias porque desconoce qué acciones son mejores o peores, pero conforme interacciona con el entorno necesita ir tomando decisiones más certeras según lo aprendido con las acciones precedentes. Solucionábamos este problema introduciendo un hiperparámetro de probabilidad ϵ (método ϵ-*greedy*) con el que de alguna manera regulábamos esta transición de tomar acciones aleatorias ($\epsilon = 1.0$) a tomar a acciones únicamente sobre la base del valor de Q ($\epsilon \leftarrow 0$).

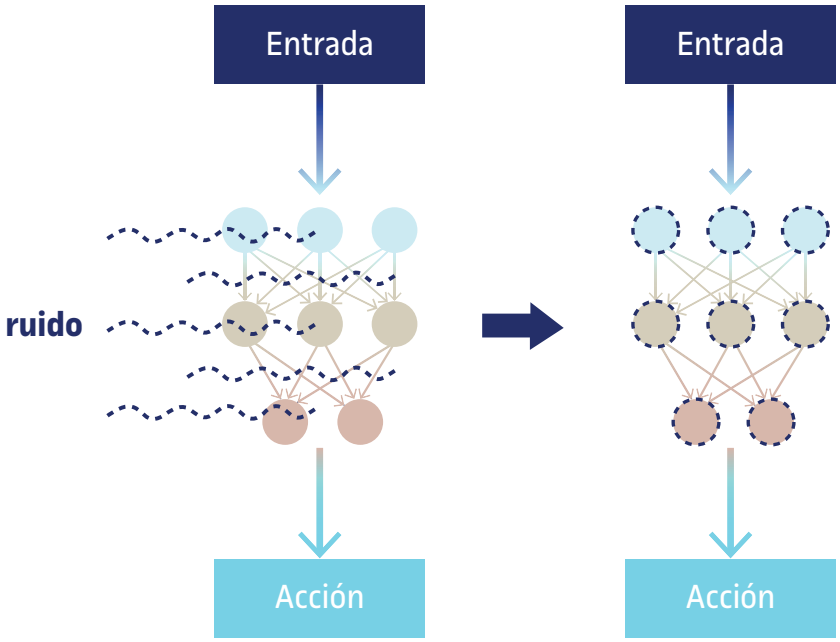
Esta evolución de la exploración a la explotación requiere alterar (*tuning*) el valor de ϵ hasta encontrar un valor inicial adecuado al problema, con tal de asegurar un proceso de entrenamiento eficiente. Esto implica que si el entorno no es muy complejo, o la no estacionaridad es pequeña durante el juego, o el número de episodios es reducido, la modificación de ϵ en búsqueda de su mejor valor es relativamente factible. Pero si no es el caso, puede resultar muy tedioso y dificultar un buen rendimiento del modelo.

La alternativa propuesta por Fortunato *et al.* (2017) del equipo de DeepMind es en realidad muy sencilla, y se basa simplemente en añadir ruido a los pesos y sesgos de las capas completamente conectadas de la red neuronal. Los autores demuestran que inducir un cierto comportamiento estocástico en la política del agente conlleva un proceso de exploración más eficiente. La idea principal es que un pequeño cambio en el vector de pesos puede resultar en un cambio de la política –ahora dependiente del estado, contrario al método ϵ-*greedy* que era independiente– mucho más complejo y más consistente. A esta variación se le llamó ***NoisyNet***, o **red con ruido**.

Lectura complementaria

M. Fortunato; M. Gheshlaghi Azar; B. Pio; *et al.* (2017). *Noisy Networks for Exploration.* Google DeepMind.

Figura 10. Idea conceptual de las *NoisyNets*



Inicialmente se introduce una distribución de ruido parametrizada principalmente por su varianza (que equivaldría a la cuantificación de la fuerza del ruido inducido).

$$y = f_\theta(x) \tag{13}$$

donde θ son los parámetros de la función de ruido.

A lo largo del proceso de aprendizaje estos parámetros se actualizan y adaptan gracias al descenso del gradiente, de la misma forma que hacíamos con los pesos y otros hiperparámetros, lo que permite al agente decidir cuándo y en qué proporción introducir una perturbación en los pesos.

Fortunato *et al.* (2017) proponen dos tipos de distribución para modelizar el ruido:

- **Ruido blanco gaussiano** (*white gaussian noise* o *independent gaussian noise*): es un ruido de distribución gaussiana en la que sus valores son *i.i.d.* (están

idéntica e independientemente distribuidos, es decir, no correlacionados). La idea es que a cada peso de la red se le añade un valor aleatorio de esta distribución, de modo que, a efectos prácticos, cada peso será totalmente independiente y tendrá sus propios parámetros μ y σ (los cuales se irán actualizando a lo largo del proceso con la propagación hacia atrás).

- **Ruido gaussiano factorizado** (*factorized gaussian noise*): en lugar de extraer un valor aleatorio por cada peso, otra alternativa es hacer el producto diádico o tensorial (conocido en inglés como *outer product*, $v \otimes w$) de un vector con el tamaño de la entrada y otro vector con el tamaño de la salida. La matriz aleatoria resultante es el ruido que se añade a los pesos.

Aunque resulte ser una variación muy sencilla, sin ninguna complejidad matemática ni conceptual, el método de *NoisyNet* condujo a una mejora muy considerable en las puntuaciones obtenidas en un gran número de juegos de Atari, al combinarlas no solo con las DQN clásicas sino con las *dueling* DQN, o con los métodos *actor-critic*, superando incluso algunos récords humanos (se invita al lector a leer el artículo de Fortunato *et al.* (2017) para más detalles de las curvas de aprendizaje y los resultados obtenidos en 57 juegos Atari).

3.7 *Rainbow* DQN: la DQN híbrida

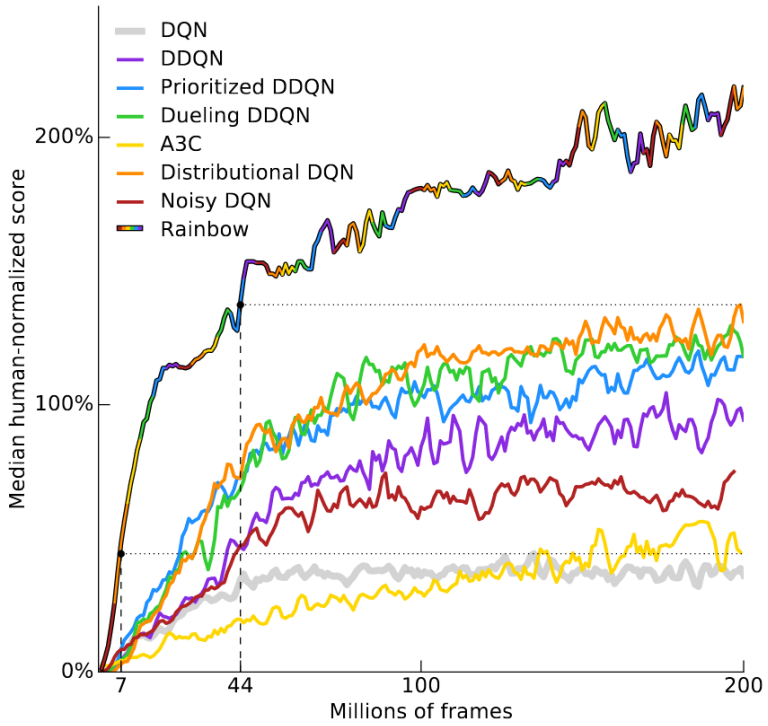
Tras todas las mejoras introducidas en los subapartados anteriores, es lícito preguntarse si no podrían combinarse todas ellas para obtener así un modelo aún mejor. Las DDQN nos permiten lidiar con el sesgo del valor de Q debido a la correlación entre estados, el PER solventa el problema de no dar más importancia a las experiencias relevantes, las *dueling* DQN ayudan a mejorar la eficiencia calculando los valores de Q únicamente en aquellos estados en los que una acción puede ser significativa, las DQN distribucionales resuelven el problema en el que obtener una distribución de Q es más preciso que un único valor de Q, y las *NoisyNet* aportan una solución más eficiente al problema de la exploración. En efecto vemos que cada mejora se ciñe a un aspecto distinto del proceso de entrenamiento y aprendizaje, por lo que uno puede pensar que todas se podrían integrar y encajar como piezas de un puzle, obteniendo así un modelo mucho más robusto. Y así es, todas son altamente complementarias.

En cierto modo, ya hemos visto que algunas de estas variaciones ya se combinan entre sí, como el método PER con las DDQN y las *dueling* DQN, o las *dueling* DQN que se complementan a las DDQN, o el método *NoisyNet* con las DQN o las *dueling* DQN, etc. En este apartado trataremos brevemente de combinarlas todas en un único **modelo híbrido**, al que Hessel *et al.* (2017) pusieron el nombre de ***rainbow DQN*** (traducido como DQN arcoíris), dado que visualmente se puede decir que une en una única curva de aprendizaje todos los colores de las curvas de aprendizaje de cada mejora individualmente (figura 11).

Lectura complementaria

M. Hessel; J. Modayil; H. van Hasselt; *et al.* (2017). *Rainbow: Combining Improvements in Deep Reinforcement Learning*. Google DeepMind.

Figura 11. Comparación del rendimiento medio de 57 juegos Atari entre el modelo híbrido *rainbow* y los distintos métodos de mejora de las DQN individualmente, respecto del modelo clásico DQN (en gris)



Fuente: Hessel *et al.* (2017)

La integración de todos sigue, *grosso modo*, el siguiente esquema:

1) La arquitectura de la red será en su esencia una *dueling* DQN, separando la función de valor y la ventaja en dos subredes lineales.

2) Estas dos subredes lineales serán ahora redes con ruido (*NoisyNets*) con una distribución de ruido gaussiano factorizada, para minimizar la cantidad de variables de ruido independientes.

3) Gracias a las subredes se predice una distribución de acciones, siguiendo las propiedades de la DQN distribucional.

4) Las dos subredes se agregan en una misma capa con función de activación *softmax* para obtener las distribuciones parametrizadas.

5) Se estima la distribución del retorno a partir de estas distribuciones parametrizadas.

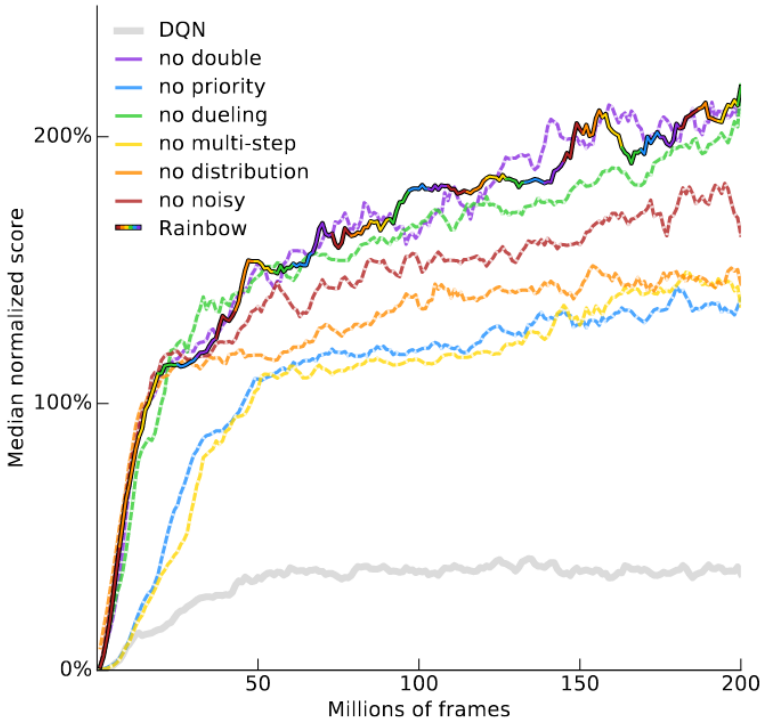
6) La actualización se hará ahora con la ecuación de Bellman con n-pasos.

7) Se aplica el proceso de selección del valor de Q de las DDQN con la introducción de una red objetivo (*target network*) para prevenir la sobreestimación de los valores de los estados.

8) En cada iteración se priorizan las experiencias para el conjunto de entrenamiento con el método PER siguiendo una distribución de pérdida de KL, asegurando una convergencia más rápida hacia resultados más óptimos.

A este esquema le sigue un proceso de modificación de hiperparámetros ahora mucho más complejo debido a la gran cantidad de variables. En el caso del artículo de Hessel *et al.* (2017) se partió de los mejores modelos de cada variación individual, introducidos en los respectivos artículos, y se modificaron ligeramente aquellos más significativos.

Figura 12. Comparación del rendimiento mediano de los mismos 57 juegos Atari entre el modelo híbrido y el mismo modelo al que se le elimina una mejora u otra, respecto del modelo clásico DQN (en gris)



Fuente: Hessel *et al.* (2017)

El gráfico de la figura 11 compara el rendimiento de los distintos modelos vistos en este módulo con el modelo híbrido *rainbow* para los 57 juegos Atari de los que hemos hablado anteriormente. También se incluye la comparación con otro tipo de modelo, el *actor-critic* A3C, que veremos más adelante. El gráfico de la figura 12 compara el rendimiento de estos mismos 57 juegos entre el modelo híbrido completo y el modelo híbrido eliminándole alguna de las mejoras explicadas, respecto de la DQN clásica (en gris), para ver en qué proporción afecta cada mejora. Podemos observar muy claramente cómo eliminar PER o el método multipaso tiene una afectación importante en el rendimiento medio. Para más detalles de este análisis se invita al lector a leer el mencionado artículo.

Podemos concluir que la integración de todas las mejoras de las DQN resultaron en general en un incremento muy positivo de los resultados en los 57 juegos Atari elegidos, convirtiendo esta DQN híbrida en el estado del arte de las DQN en aprendizaje por refuerzo profundo, tanto en eficiencia como en rendimiento.

Resumen

Partiendo del *Q-learning* y usando las redes neuronales como funciones de aproximación de la función de valor, surgen las *deep Q-networks*. Estas constituyen el primer método dentro del DRL que tiene como objetivo solucionar problemas en los que el número de estados no es posible gestionarlo con una tabla.

Hemos visto que el algoritmo inicial que las define es en realidad relativamente sencillo, pero cuando se analiza con detalle hay varios aspectos que es importante tener en cuenta para evitar la inestabilidad del aprendizaje. En primer lugar, a diferencia del lenguaje supervisado, en DRL necesitamos estrictamente que haya un ejercicio de exploración-explotación del agente puesto que pasa de un estado inicial en el que desconoce prácticamente todo, a estados más avanzados en los que ya ha adquirido ciertas experiencias. Y, por otro lado, los estados suelen suceder tan continuos que la correlación es muy alta entre ellos. Es decir, no están idéntica e independientemente distribuidos. La introducción del método ϵ-*greedy*, el *buffer* de repetición de experiencias y la red objetivo, nos permiten obtener un algoritmo base de las DQN mucho más robusto y que tiene presentes todos estos factores.

A partir de este algoritmo, se han desencadenado una serie de nuevos algoritmos más complejos con tal de mejorar aún más la eficiencia de las DQN. Así, tanto el aprendizaje multipaso, como las DQN dobles, el *buffer* de repetición de experiencias priorizadas, las *dueling*-DQN, las distribucionales o las *NoisyNets*, tienen como último fin esta competición por la búsqueda del rendimiento óptimo de las DQN. Como hemos visto, el estado del arte en este método lo concluye la DQN híbrida, en la que se intenta integrar, no sin complejidades matemáticas, todos los algoritmos mencionados en uno solo.

Glosario

aprendizaje profundo *m* Conjunto de algoritmos de aprendizaje automático que intenta modelar abstracciones de alto nivel utilizando redes neuronales de múltiples capas o niveles.
sigla **DL**
en deep learning

deep learning *m* Véase **aprendizaje profundo**.

DL *m* Véase **aprendizaje profundo**.

i.i.d. *m* Véase **independientes e idénticamente distribuidos**.

independent and identically distributed *m* Véase **independientes e idénticamente distribuidos**.

independientes e idénticamente distribuidos *m* Conjunto de datos o variables aleatorias que son mutuamente independientes entre sí y además cada variable tiene la misma distribución de probabilidad.
sigla **i.i.d.**
en independent and identically distributed

Bibliografía

Bellemare, M. G.; Dabney, W.; Munos, R. (2017). *A Distributional Perspective on Reinforcement Learning*. Google DeepMind.

Fortunato, M.; Azar, M. G.; Pio, B.; *et al.* (2017). *Noisy Networks for Exploration*. Google DeepMind.

Hasselt, H. van; Guez, A.; Silver, D. (2015). *Deep Reinforcement Learning with Double Q-learning*. Google DeepMind.

Hernández-García, J. F.; Sutton, R. S. (2019). *Understanding Multi-Step Deep Reinforcement Learning: A Systematic Study of the DQN Target*. University of Alberta.

Hessel, H.; Modayil, J.; van Hasselt, H.; *et al.* (2017). *Rainbow: Combining Improvements in Deep Reinforcement Learning*. Google DeepMind.

Lapan, M. (2018). «Deep Reinforcement Learning Hands-On». *Expert Insight*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; *et al.* (2013). «Playing Atari with Deep Reinforcement Learning». *Nature Journal*.

Mnih, V.; Kavukcuoglu, K.; Silver, D; *et al.* (2015). «Human-level control through deep reinforcement learning». *Nature Journal*.

Schaul, T.; Quan, J.; Antonoglou, I.; *et al.* (2016). *Prioritized Experience Replay*. Google DeepMind.

Silver, D.. *Univerity College London Course on RL*. [Fecha de consulta: 14 de mayo de 2020]. <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>.

Sutton, R. S. (1988). «Learning to Predict by the Methods of Temporal Differences». *Machine Learning*, vol. 3, n.º 1, págs. 9-44.

Sutton; R. S.; Barto, A. G. (2019). *Reinforcement Learning. An introduction*. Cambridge, MA: MIT Press.

Tamar, A.; Wu, Y.; Thomas, G.; *et al.* (2016). *Value iteration networks*. Advances in Neural Information Processing Systems.

Thomas, S.. *Deep Reinforcement Learning Course*. [Fecha de consulta: 25 de mayo de 2020]. <https://simoninithomas.github.io/Deep_reinforcement_learning_Course/>.

Wang, Z.; Schaul, T.; Hessel, M.; *et al.* (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. Google DeepMind.