

APRENDIZAJE POR REFUERZO

10 1 011 1000

10 1 011 1000

10 1 011 1000

10 1 011 1000

**Adrián Colomer Granero
Gabriel Enrique Muñoz Ríos**

**MÁSTER UNIVERSITARIO EN INTELIGENCIA
ARTIFICIAL**



Universidad
Internacional
de Valencia



Universidad
Internacional
de Valencia

Este material es de uso exclusivo para los alumnos de la Universidad Internacional de Valencia. No está permitida la reproducción total o parcial de su contenido ni su tratamiento por cualquier método por aquellas personas que no acrediten su relación con la Universidad Internacional de Valencia, sin autorización expresa de la misma.

Edita

Universidad Internacional de Valencia

Máster Universitario en
Inteligencia Artificial

Aprendizaje por refuerzo
6 ECTS

Adrián Colomer Granero
Gabriel Enrique Muñoz Ríos

Leyendas



Enlace de interés



Ejemplo



Importante



abc Los términos resaltados a lo largo del contenido en color **naranja** se recogen en el apartado **GLOSARIO**.

Índice

| | |
|--|----|
| CAPÍTULO 1. INTRODUCCIÓN | 7 |
| 1.1. Clasificación de algoritmos de aprendizaje basados en datos | 8 |
| 1.1.1 Aprendizaje no supervisado | 9 |
| 1.1.2 Aprendizaje supervisado | 9 |
| 1.1.3 Aprendizaje por refuerzo | 9 |
| 1.2. Clasificación de proyectos basados en aprendizaje por refuerzo | 10 |
| 1.2.1 Proyectos basados en entornos simulados | 10 |
| 1.2.2 Proyectos basados en entornos simulados y productivos | 10 |
| 1.3. Características del aprendizaje por refuerzo | 11 |
| 1.4. Resumen | 12 |
| | |
| CAPÍTULO 2. CONCEPTOS Y DEFINICIONES | 13 |
| 2.1. Conceptos básicos | 14 |
| 2.1.1 Entorno | 14 |
| 2.1.2 Agente | 15 |
| 2.1.3 Acción | 15 |
| 2.1.4 Observación | 16 |
| 2.1.5 Estado | 17 |
| 2.1.6 Recompensa | 17 |
| 2.1.7 Episodio e iteración | 18 |
| 2.2. Conceptos avanzados | 19 |
| 2.2.1 Estrategia | 19 |
| 2.2.2 Exploración | 20 |
| 2.2.3 Explotación | 20 |
| 2.2.4 Experiencia | 21 |
| 2.3. Resumen | 22 |
| | |
| CAPÍTULO 3. CLASIFICACIÓN DE ALGORITMOS DE APRENDIZAJE POR REFUERZO | 23 |
| 3.1. Basados en estrategia | 24 |
| 3.2. Basados en modelo | 25 |
| 3.3. Esquema de algoritmos de aprendizaje por refuerzo | 26 |
| 3.4. Resumen | 28 |

| | |
|---|----|
| CAPÍTULO 4. ALGORITMOS BÁSICOS DE APRENDIZAJE POR REFUERZO | 30 |
| 4.1. Deep Q-Networks | 30 |
| 4.1.1 Pseudocódigo del algoritmo Deep Q-Networks | 33 |
| 4.1.2 Aclaraciones del pseudocódigo del algoritmo Deep Q-Networks | 35 |
| 4.2. Policy Gradients | 36 |
| 4.2.1 Pseudocódigo del algoritmo Policy Gradients..... | 37 |
| 4.2.2 Aclaraciones del pseudocódigo del algoritmo Policy Gradients..... | 38 |
| 4.3. Resumen | 39 |
| | |
| CAPÍTULO 5. ALGORITMOS AVANZADOS DE APRENDIZAJE POR REFUERZO | 40 |
| 5.1. Actor Critic..... | 40 |
| 5.1.1 Pseudocódigo del algoritmo Actor-Critic..... | 43 |
| 5.2. A2C y A3C | 44 |
| 5.2.1 Pseudocódigo del algoritmo A2C..... | 45 |
| 5.2.2 Pseudocódigo del algoritmo A3C..... | 47 |
| 5.3. Proximal Policy Optimization | 48 |
| 5.3.1 Pseudocódigo del algoritmo PPO | 49 |
| 5.4. Deep Deterministic Policy Gradient..... | 51 |
| 5.4.1 Pseudocódigo del algoritmo Deep Deterministic Policy Gradient..... | 52 |
| 5.5. Caso de uso model-based: AlphaGo | 54 |
| 5.5.1 Análisis de la solución AlphaGo | 54 |
| 5.5.2 Comentarios sobre AlphaGo | 57 |
| 5.6. Resumen | 58 |
| | |
| GLOSARIO | 59 |
| | |
| BIBLIOGRAFÍA | 67 |



Capítulo 1

Introducción

En los últimos años, la **inteligencia artificial** ha cambiado el modo en el que vivimos. El uso de este tipo de tecnologías en las aplicaciones y plataformas de nuestro día a día ha proporcionado un cambio de paradigma en la experiencia de los usuarios, enriqueciendo a unos niveles desconocidos hasta el momento a nivel de personalización, calidad y velocidad.

Aunque este boom se haya desarrollado a una velocidad vertiginosa en la última década, la inteligencia artificial no es una “promesa” nueva. Tenemos que remontarnos a los años 50 del s. XX para situar el punto de partida de esta rama de las ciencias de la computación. La mayoría de algoritmos y enfoques implementados en las soluciones actuales datan de estos primeros años de estudio e investigación. El obstáculo que encontraron fue que la teoría se desarrolló mucho más rápido que la ingeniería necesaria, de ahí que la explosión total no se haya producido hasta estos últimos años.

Dentro de las distintas líneas de trabajo y desarrollo actuales, **la aplicación principal se ha centrado alrededor de lo que se conoce como aprendizaje automático o machine learning**. El aprendizaje automático cubre los algoritmos y técnicas que se desarrollan desde el punto de vista del aprendizaje a partir de conjuntos de datos. Mediante la explotación de la información disponible en estos conjuntos de datos, los algoritmos de *machine learning* son capaces de extraer nuevos patrones ocultos o resolver tareas complejas de carácter predictivo o analítico. Al hablar de datos, no debemos olvidar la importancia que ha tenido el **big data** a nivel tecnológico, ofreciendo la posibilidad de usar grandes volúmenes de información que hace unos años eran impensables.

Además de los datos, el hecho de que el aprendizaje automático sea la referencia en términos de inteligencia artificial en la actualidad se debe, principalmente, a dos ingredientes básicos. En primer lugar, el servicio y mejora constante en cuanto a los proveedores *cloud* o recursos *computacionales* disponibles. Como hemos comentado, este fue uno de los primeros obstáculos durante los años 50-60, siendo ahora posible la implementación y desarrollo de soluciones de aprendizaje automático que hace solo unos años eran inabarcables computacionalmente.

El segundo elemento que ha facilitado el desarrollo de soluciones basadas en aprendizaje automático es la madurez de los *frameworks* y librerías disponibles. A nivel tecnológico, este ha sido un paso fundamental para implementar este tipo de soluciones, siguiendo metodologías y aproximaciones del mundo del *software* en soluciones y proyectos de inteligencia artificial.

1.1. Clasificación de algoritmos de aprendizaje basados en datos

Cuando hablamos de aprendizaje automático, normalmente encontramos dos enfoques principales: métodos supervisados y métodos no supervisados. Por métodos supervisados nos referimos a aquellos algoritmos que relacionan los datos disponibles con una etiqueta o valor objetivo. Esta etiqueta o valor objetivo puede estar contenida en los datos (como una variable más) o se puede definir a partir de reglas de negocio que se tienen que aplicar sobre el mismo conjunto de datos.

Por otro lado, los métodos no supervisados forman la familia de algoritmos que se aplica sobre conjuntos de datos en los que no se conoce un objetivo a priori. Un ejemplo de este segundo tipo son los algoritmos de clusterización.

Además de estos dos grupos principales, podemos encontrar otro grupo de técnicas y problemas conocido como aprendizaje por refuerzo. A partir de la definición que encontramos en Barto, A. y Sutton, R. (1992), podemos definir el aprendizaje por refuerzo como un tipo de aprendizaje basado en prueba y error, llevado a cabo mediante la interacción de un agente y un entorno. Teniendo como punto de partida la comparativa entre aprendizaje por refuerzo y los aprendizajes supervisados y no supervisados que también encontramos en Barto y Sutton (1992), podemos analizar en qué nivel se sitúa el aprendizaje por refuerzo.

Para muchos expertos, y al ser una familia de algoritmos que aprende a partir de la experiencia y datos pasados, el aprendizaje por refuerzo se sitúa al mismo nivel que el aprendizaje supervisado y el aprendizaje no supervisado. Por otro lado, y debido a la relación del aprendizaje por refuerzo con otras ramas de la inteligencia artificial como **visión por computador**, **robótica** o sistemas expertos, existe otra gran parte de expertos que define el aprendizaje por refuerzo como una rama en sí.



En otras ramas de la inteligencia artificial los grandes volúmenes de datos sirven como un repositorio de experiencia; en el aprendizaje por refuerzo el dato se genera en tiempo real, a partir de las simulaciones y los entornos donde estos algoritmos se desarrollan.

Más allá de entender en qué parte podemos incluir al aprendizaje por refuerzo, es importante entender qué puede aportar a partir del aprendizaje basado en datos. A partir del repaso analizado en Lepenioti, K. et al. (2020), podemos visualizar una relación entre el tipo de aprendizaje y el resultado esperado, como por ejemplo:

- a. Aprendizaje no supervisado → análisis descriptivo.
- b. Aprendizaje supervisado → análisis predictivo.
- c. Aprendizaje por refuerzo → análisis prescriptivo.

1.1.1 Aprendizaje no supervisado

Cuando hablamos de aprendizaje no supervisado nos referimos al tipo de extracción de conocimiento sin un objetivo definido a priori. Trabajamos directamente sobre el conjunto de variables disponibles y el objetivo es encontrar correlaciones y patrones a partir de la similitud de los elementos que componen el conjunto de datos.



El resultado de este aprendizaje es un análisis descriptivo, ya que el conocimiento que se extrae es pasado; es decir, el conocimiento necesita de una interpretación para verdaderamente descubrir nuevas relaciones o nuevos comportamientos que antes no se conocían. Esta interpretación está muy ligada a un conocimiento de dominio específico.

1.1.2 Aprendizaje supervisado

En este caso sí que disponemos de un objetivo a alcanzar mediante el algoritmo de aprendizaje. A este objetivo también se le conoce como “variable objetivo”, ya que es una variable que podemos encontrar en nuestro conjunto de datos o que podemos definir a partir de reglas de negocio conocidas.



Al relacionar un conjunto de variables de nuestros datos con otra variable como objetivo para conocer sus valores futuros, el tipo de análisis en el que se basa el aprendizaje supervisado se conoce como “análisis predictivo”.

1.1.3 Aprendizaje por refuerzo

La última rama que encontramos es el aprendizaje por refuerzo. En los dos casos anteriores, los conjuntos de datos son datos recopilados a lo largo del tiempo, normalmente por grupos de personas o por procesos automáticos. Es por ello que estos conjuntos de datos pueden pecar de estar muy sesgados hacia el objetivo que se quiere alcanzar con ellos. **En el caso del aprendizaje por refuerzo, en el que el aprendizaje se produce from scratch, a partir de la interacción de un agente con su entorno,** las reglas de conocimiento que se infieren son originales y no siguen ningún tipo de sesgo, ya que se crean en tiempo real durante la ejecución.



El análisis que se puede aplicar en este sentido se conoce como “análisis prescriptivo”, ya que el conocimiento que surge aporta enfoques desconocidos hasta el momento, proporcionando un nuevo paradigma en la toma de decisiones, a la hora de solucionar el problema propuesto.

1.2. Clasificación de proyectos basados en aprendizaje por refuerzo

Como hemos visto, el aprendizaje por refuerzo se basa en la obtención de datos a partir de la interacción de un agente con su entorno. Dependiendo del tipo de entorno y de la naturaleza del proyecto, este tipo de recolección conlleva ciertas singularidades a la hora de afrontar proyectos basados en aprendizaje por refuerzo.

En esta asignatura veremos dos grupos principales:

- a. Proyectos basados en entornos simulados.
- b. Proyectos basados en entornos simulados y productivos.

1.2.1 Proyectos basados en entornos simulados

Este es el caso más común dentro de proyectos de aprendizaje por refuerzo. Nos referimos a que, en todo momento, el entorno con el que el agente interactúa **es una simulación, como, por ejemplo, un videojuego o un software diseñado a partir del reto a resolver**. El hecho de que este sea el tipo de proyecto más común es, precisamente, por la facilidad de comunicación con el entorno y por tener siempre la lógica del mismo bajo control.

Este tipo de enfoque es muy útil para testear nuevos algoritmos y para llevar a cabo el desarrollo de comparativas o **benchmarks**.

1.2.2 Proyectos basados en entornos simulados y productivos

Dependiendo del dominio de aplicación, **hay escenarios donde es necesario probar y testear la solución de aprendizaje por refuerzo en su entorno real**. Este es el caso de soluciones donde hay una parte de implementación *hardware*, como pueden ser entornos de robótica (OpenAI, Andrychowicz, M. et al. (2018)) o conducción autónoma (Kiran et al. (2021)). Los proyectos de este tipo se implementan en un híbrido entre la parte de **laboratorio** y la parte de **producción**.

El reto surge en el sentido de que el desarrollo del entorno para el laboratorio debe ser lo más preciso posible con relación al mundo real o producción. Por ejemplo, en el caso de un brazo robótico se deberían tener en cuenta multitud de aspectos, como los sensores disponibles desde los que se puede recolectar información o los grados de libertad sobre los que se puede aplicar un movimiento. Cualquier variación mínima entre la simulación y el mundo real podría conllevar errores en la solución.

La mayoría de estos proyectos se desarrollan bajo pruebas de concepto o con un peso mayor del lado del laboratorio, ya que el despliegue en entornos productivos también conlleva un análisis exhaustivo y una validación del lado de la seguridad o situaciones extremas, lo que en muchas ocasiones deriva en una inversión en tiempo y esfuerzo demasiado costosa.

1.3. Características del aprendizaje por refuerzo

Hasta ahora nos hemos centrado en una presentación del aprendizaje por refuerzo en relación con otros enfoques de aprendizaje basado en datos y con los distintos escenarios de ejecución en la interacción agente-entorno que podemos encontrar. Vamos a continuar con otras características que tienen mucho peso a la hora de entender la complejidad y el potencial que tiene esta rama de la inteligencia artificial.

Uno de los principales retos del aprendizaje por refuerzo se produce a nivel de hiperparámetros.

En aprendizaje automático, y más concretamente en **deep learning**, una buena configuración de hiperparámetros como el número de epochs, tamaño del batch de datos o el valor del *learning rate* es crítica para poder asegurar que la solución alcanzada es óptima.

En el caso de aprendizaje por refuerzo, la hiperparámetrización se vuelve más compleja, ya que las soluciones son una combinación de algoritmos de aprendizaje automático junto con los propios algoritmos de aprendizaje por refuerzo, lo que implica una mayor configuración y decisión a la hora de entrenar nuestros modelos y diseñar la solución.



Aunque hablamos de aprendizaje automático, los algoritmos y modelos que desarrollaremos en la asignatura pertenecen todos al conjunto de soluciones de deep learning. Por ello, es recomendable tener un conocimiento previo en esta rama. Un recurso de referencia es Goodfellow et al., (2016).

Otro aspecto importante a tener en cuenta es que las soluciones basadas en aprendizaje por refuerzo son muy demandantes en cuanto a recursos computacionales se refiere, sobre todo en cuanto a entrenamiento e interacción entre el agente y el entorno. Esto se debe, principalmente, a tener una mayor hiperparametrización y a que **los entrenamientos, al estar basados en prueba y error, necesitan de muchas iteraciones**. En relación con esto último, la resolución de los problemas sigue un proceso totalmente empírico. Esta es una situación común en aprendizaje automático, pero en el caso del aprendizaje por refuerzo resalta la complejidad a la hora de evaluar cómo de bien o mal lo está haciendo el agente.

Más allá de los obstáculos que podemos encontrarnos con este tipo de desarrollos, el aprendizaje por refuerzo proporciona una forma muy original a la hora de diseñar soluciones de inteligencia artificial. Como veíamos al principio, conceptualmente es un enfoque de inteligencia artificial muy cercano a la idea preconcebida que desde el comienzo se prometió. El hecho de que un agente sea capaz de aprender y adaptarse para maximizar un objetivo sin supervisión abre un abanico de posibilidades infinitas en los desarrollos basados en inteligencia artificial del futuro.

1.4. Resumen

Este primer capítulo se ha centrado en una introducción al aprendizaje por refuerzo desde un punto de vista de dónde se sitúa dentro de la inteligencia artificial y del aprendizaje automático. El momento actual de la inteligencia artificial se centra en el aprendizaje a partir de grandes volúmenes de datos y ahí es donde el aprendizaje por refuerzo introduce un nuevo prisma a la hora de abordar estos retos.

La interacción entre el agente y el entorno a base de prueba y error es una aproximación muy cercana a la idea original de inteligencia artificial con la que se creó esta rama de las ciencias de la computación. Las posibilidades que este tipo de soluciones ofrece van más allá de la resolución de problemas complejos y llega hasta un tipo de análisis prescriptivo en el que el propio resultado de la ejecución nos comunica cómo llevar a cabo la solución.

Es interesante destacar también el aprendizaje por refuerzo desde el punto de vista del entorno. El entorno es un elemento crucial dentro de las soluciones de aprendizaje por refuerzo, por lo que el tipo de acceso y de comunicación que se establezcan es crítico para un buen desarrollo. Aquí encontramos principalmente entornos basados completamente en simulaciones y también entornos en los que el desarrollo de la solución se produce en simulaciones, para luego usarse en el mundo real. Un ejemplo de este caso son entornos basados en problemas de robótica.

Para finalizar, es importante tener en cuenta que las propias características de este tipo de soluciones hacen que aparezcan otros obstáculos que no podemos perder de vista, como son una mayor hiperparametrización para configurar los experimentos y la necesidad de mayores recursos computacionales para entrenar y ejecutar soluciones tan demandantes. Aun así, el potencial que tienen desde el punto de vista conceptual es tan amplio que no hay dudas de su uso para abordar algunos de los nuevos retos que presenta la inteligencia artificial.

En el siguiente capítulo entraremos en la terminología y los conceptos necesarios para poder sacarle todo el partido a las soluciones de aprendizaje por refuerzo. Será un capítulo muy importante para ir ganando confianza y aprender la nomenclatura necesaria para el resto de la asignatura.



Capítulo 2

Conceptos y definiciones

Como comentamos en el capítulo anterior, el aprendizaje por refuerzo (Barto, A. y Sutton, R. (1992)) es un tipo de aprendizaje basado en prueba y error donde **sus dos componentes principales son el entorno y el agente**. Mediante la interacción entre el agente y el entorno, el agente va adaptándose y aprendiendo para alcanzar un objetivo predefinido. Este objetivo estará definido en términos de recompensa, que es el valor que el entorno va devolviendo al agente en cada paso de la interacción. **El objetivo final del agente es encontrar una estrategia óptima que maximice la recompensa obtenida.**

Si se analiza la interacción entre el agente y el entorno durante las distintas iteraciones de la ejecución, encontraremos la base sobre la que se sustenta conceptualmente el aprendizaje por refuerzo, un modelado siguiendo un enfoque de **cadenas de Markov**. En cada instante de tiempo, el agente dispone de la información proporcionada por el entorno para así tomar una decisión al respecto. Tanto la información del entorno como la decisión del agente provocan una transición entre estados, afectando a todos los elementos de la ejecución:

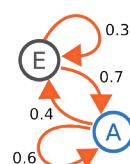


Figura 1. Ejemplo de una cadena de Markov. Por Joxemai4. Dominio público.

Esta sucesión de transiciones se traduce directamente en el objetivo del agente de maximizar una estrategia. **La estrategia será, justamente, la evaluación y adopción de acciones que el agente lleva a cabo en cada una de las situaciones a las que se enfrenta para maximizar el objetivo del problema propuesto.**

Cuando hablamos de aprendizaje por refuerzo, ocurre que muchos de los términos que se utilizan son ya conocidos en el mundo de la tecnología o el software, pero que tienen una definición concreta en este contexto. Es importante tener este punto en cuenta cuando nos referimos a agentes, entorno o estados, por ejemplo.

Como curiosidad, en Barto y Sutton (1992) se analiza también la relación de esta rama con otras como la psicología o la neurociencia; de ahí que muchos de los conceptos que se revisarán a lo largo del capítulo sean representativos de estos campos.

2.1. Conceptos básicos

Comenzaremos con una revisión de los conceptos básicos. Algunos de los términos que abordaremos en esta sección ya nos suenan porque se han ido utilizando desde el primer capítulo. Estos términos son los ingredientes básicos de cualquier solución de aprendizaje por refuerzo. Además de su definición teórica, añadiremos en cada caso algunos aspectos a tener en cuenta desde el punto de vista de cómo sería la implementación.

Es muy importante entender claramente lo que significa cada elemento ya que, conforme avance la asignatura, algunos de estos términos irán evolucionando y complementándose desde un punto de vista más complejo.

2.1.1 Entorno

El entorno es el medio donde nuestro agente se va a desarrollar. Está formado por un conjunto de elementos con los que el agente podrá interactuar para alcanzar su objetivo. Además de elementos interactivos, el entorno también se define en términos de la lógica de funcionamiento que afectan directamente a la forma en la que se desarrollará el proceso de aprendizaje.

Normalmente, el entorno es accesible por medio de funciones o llamadas a una **API** de comunicación. Dependiendo del entorno con el que se trabaje, esta API puede existir o será necesario desarrollar una a medida. **En cada una de las iteraciones, el agente le envía al entorno la acción que quiere llevar a cabo y el entorno le responde con la información disponible y accesible para el agente.**



El hecho de tener que desarrollar la API de comunicación con un entorno desde cero es una de las razones principales de la costosa adopción del aprendizaje por refuerzo. Son muchas las situaciones en las que el desarrollo de esta API de comunicación es un gran proyecto en sí mismo.

2.1.2 Agente

Junto con el entorno, el agente es el elemento fundamental en cualquier solución de aprendizaje por refuerzo. **Es una entidad capaz de aprender y adaptarse en el entorno en el que se desarrolla y que, por medio de su interacción con él, va obteniendo experiencia y va mejorándose con el fin de encontrar una estrategia óptima para alcanzar el objetivo definido.**

En cualquier solución de aprendizaje por refuerzo que desarrollemos, el agente es la pieza o artefacto software que contendrá la lógica y la mayor parte de la implementación. Normalmente, se compone de una serie de funciones enfocadas en la interacción con el entorno y en la monitorización de lo que está ocurriendo durante la ejecución. Es por ello que podemos encontrar funciones para hacer un tratamiento de la recompensa, para seleccionar las acciones, para traducir la información disponible, etc.

Otro componente que tenemos que tener en cuenta en la implementación del agente es la parte correspondiente al modelo. Dependiendo de la complejidad del problema a resolver, el modelo que utiliza el agente puede ser simple o una arquitectura avanzada de *deep learning*. Pero, en cualquier caso, la definición e implementación se producen también del lado del agente.



Las implementaciones de aprendizaje por refuerzo que estudiaremos en la asignatura se conocen como “aprendizaje por refuerzo profundo”, ya que combinan, en una única solución, el uso de algoritmos de aprendizaje por refuerzo y arquitecturas de *deep learning*.

Como apunte, hoy en día podemos encontrar multitud de proyectos en los que trabajaremos con más de un agente, conocidos como “sistemas multiagente” (Nguyen et al. (2019)). En estos escenarios, es común encontrar una implementación por cada uno de los agentes necesarios en la ejecución y un agente global encargado de orquestar la comunicación entre ellos. Otra opción es que los agentes se comuniquen entre ellos de manera independiente para desarrollar un aprendizaje individual a partir de las interacciones entre ellos y el entorno. Con este enfoque visualizamos cómo el agente no necesariamente tiene que representarse con un elemento físico en el entorno, sino que puede ser también un elemento abstracto encargado de tomar decisiones durante la ejecución.

2.1.3 Acción

Desde el primer capítulo hemos comentado que el agente es capaz de interactuar con el entorno para aprender y adaptarse mediante la toma de decisiones. Esta toma de decisiones tiene una relación directa con el concepto de acción.

Las acciones forman el conjunto de posibilidades que un agente puede ejecutar en un momento determinado. Dependiendo de la acción que se ejecute, el entorno proporcionará un **feedback** específico al agente. Normalmente, el conjunto de acciones disponible para realizar un experimento de aprendizaje por refuerzo es fijo, para así poder modelar las acciones a partir del feedback obtenido durante las interacciones pasadas.

Es importante tener en cuenta que el conjunto de acciones disponibles puede ser intratable dependiendo del problema, lo que impacta en el tipo de técnicas que se pueden usar para llegar a una solución. En estas situaciones podemos encontrar combinaciones con otras técnicas de inteligencia artificial como **metaheurísticas, planificación o búsquedas en árbol**, que ayuden en la selección de la mejor acción en este tipo de escenarios.



Ejemplo

A nivel de implementación, las acciones serán las salidas del modelo del agente. Dependiendo de la complejidad del modelo, las acciones pueden corresponderse con un tipo de datos u otro.

Por ejemplo, si el agente dispone de cuatro posibles acciones y su modelo es una red neuronal, nuestro grupo de acciones se corresponde con las cuatro neuronas que son las salidas de la red, con valores que oscilan dependiendo de la función de activación en la capa de salida. Esto obliga a que las acciones tengan que traducirse entre el entorno y la lógica del agente. El entorno entiende las acciones de una forma específica, con un número real, un número entero o incluso una etiqueta, mientras que el agente va a procesar las acciones del lado del modelo, con un dominio diferente al del entorno.

Es importante tener en cuenta siempre esta situación en la comunicación entre el agente y el entorno, ya que en entornos reales, como en el ejemplo de la robótica, la correspondencia entre los valores de los sensores y los valores del modelado necesita ser lo más precisa posible para una correcta ejecución y un aprendizaje completo.

2.1.4 Observación

Como parte del *feedback* que el entorno devuelve al agente cuando este toma una acción en un momento determinado, se encuentra la observación. **La observación es una fotografía/resumen del punto en el que se encuentra el entorno en ese momento.** Es común que el acceso al estado del entorno no sea completo: es decir, el agente solo puede acceder a los elementos e indicadores que el entorno permite.



Ejemplo

Si nuestro entorno fuera un videojuego, la observación en un momento determinado sería una imagen de la pantalla del videojuego. Además de la imagen, en la observación podemos encontrar otros elementos disponibles en el entorno como pueden ser contadores de tiempo, medidas de puntuación, barras de vida, etc. Toda esta información forma parte de un momento específico de la ejecución, de ahí que pueda ser observado.

En función de la información que contenga, las observaciones se presentan en forma de matrices o vectores de datos. Si volvemos al ejemplo de un videojuego y la observación se corresponde con la pantalla del mismo, en cada instante de tiempo donde hacemos una ingesta de la observación encontraremos que se corresponde con una matriz de píxeles con tres **canales RGB**. Esta sería la información mínima que el agente tendría disponible para decidir qué acción tomar.

2.1.5 Estado

En aprendizaje por refuerzo, el estado es un concepto que se puede interpretar desde dos puntos de vista. En primer lugar, se puede corresponder con el **estado en el que se encuentra la ejecución**, haciendo una conexión directa con la idea de **cadenas de Markov** que veíamos al comienzo del capítulo. Dependiendo de las acciones tomadas por el agente y cómo vayan evolucionando las iteraciones, el estado se correspondería con cada uno de los momentos en los que el agente se encuentra para tomar una decisión.

La otra interpretación del concepto de estado está relacionada con **el procesamiento de la observación**. La información que encontramos en las observaciones es lo que se conoce como información en crudo o **raw**. En la mayoría de ocasiones, el agente no es capaz de interpretar satisfactoriamente esta información ya que necesita algún tipo de preprocesamiento y preparación antes de poder utilizarla como entrada al modelo. Por ello, se define el estado como la información lista para ser consumida por el agente para poder tomar una acción.

A nivel de implementación, el estado es el resultado de aplicar todas las tareas necesarias de preprocesamiento (normalización, estandarización, transformación de píxeles, etc.) sobre la observación y así tener los datos listos para poder usarlos como datos de entrada en el modelo.



De hecho, dependiendo del autor y/o recurso, los conceptos de observación y estado se intercambian arbitrariamente. Lo importante es tener en cuenta que siempre es necesario valorar si la información necesita ser procesada antes de usarla como datos de entrada del modelo.

2.1.6 Recompensa

Dentro del *feedback* que el entorno comparte con el agente encontramos un elemento crucial en cualquier solución de aprendizaje por refuerzo: la recompensa. **La recompensa es el resultado que el entorno devuelve al agente a partir de la acción que se ha ejecutado**. Desde el punto de vista del aprendizaje, **la recompensa es la medida que nos indica si se está aprendiendo o no**. El objetivo del agente es encontrar la estrategia óptima para maximizar la recompensa que se obtiene.

Generalmente, la recompensa se corresponde con un valor numérico devuelto directamente por el entorno. Hay casos en los que se diseñan funciones de recompensa para modificar o adaptar el valor proporcionado por el entorno, ya sea para suavizar el problema o para testear cambios en los comportamientos del agente en relación con un cambio en la recompensa.

Al ser un valor numérico proporcionado por el entorno, la recompensa se utiliza durante el entrenamiento del agente como indicador de cómo de buena o mala ha sido la acción seleccionada en un momento determinado. En los ejemplos que veremos durante la asignatura, este número lo encontraremos normalizado o reducido a valores de señal binarios, ya que son ejemplos en los que la recompensa se devuelve como un valor de puntuación acumulada, perdiendo la relación directa sobre la acción tomada.

2.1.7 Episodio e iteración

Las ejecuciones en aprendizaje por refuerzo se miden y controlan en función de dos indicadores, episodios e iteraciones. **Un episodio se corresponde con una ejecución completa, desde el comienzo o estado inicial hasta un estado final o terminal.**



Ejemplo

Por hacer una analogía con el ejemplo de los videojuegos, el episodio sería desde que comienza una partida hasta que llegamos al game over. Dependiendo de la configuración del entorno y del experimento, la ejecución del episodio se podría corresponder con solo una oportunidad para jugar la partida o con todas las oportunidades (o vidas) que estén predefinidas.

A efectos prácticos, los episodios nos permiten medir el número de ejecuciones que lleva nuestro agente, el tiempo medio que está tardando y, teniendo en cuenta también la recompensa, cómo está aprendiendo respecto al tiempo de entrenamiento.

En un episodio, cada instante de tiempo durante la ejecución es lo que se conoce como iteración o step. Intuitivamente, podemos entender un step como **un momento de tiempo congelado durante la ejecución de la solución.** Al igual que el episodio, las iteraciones también se utilizan como medida de tiempos de entrenamiento y convergencia de la solución en benchmarks y comparaciones entre algoritmos.

La implementación de los episodios e iteraciones en una solución de aprendizaje por refuerzo se corresponde con dos bucles anidados con los que se va controlando la ejecución. Encontraremos un bucle global para los episodios (o partidas) que queremos llevar a cabo y un bucle interno para monitorizar todas las iteraciones que se van sucediendo en cada episodio.



Como hemos visto, en los videojuegos, un episodio sería una partida completa desde su inicio hasta el game over mientras que la iteración se correspondería con cada frame de la partida. En otros campos, como la robótica, la definición del final del episodio depende de la configuración del experimento y del objetivo, teniendo que ser definida por el usuario previamente.

2.2. Conceptos avanzados

A continuación vamos a cubrir una serie de conceptos avanzados que complementarán los conceptos básicos vistos hasta el momento. El hecho de agrupar las siguientes definiciones dentro de conceptos avanzados se debe a que, dependiendo del algoritmo de aprendizaje por refuerzo que se use, pueden variar en su comportamiento e impacto en la solución. Trataremos este hecho en los siguientes capítulos.

Aunque hay muchos más conceptos dentro del aprendizaje por refuerzo, nos quedaremos con cuatro de los conceptos más importantes que nos servirán para guiar el resto de la asignatura: estrategia, exploración, explotación y experiencia.

2.2.1 Estrategia

La estrategia, o policy, define cómo el agente selecciona una acción en un momento determinado de la ejecución. Es el componente que va evolucionando a medida que el agente va aprendiendo mediante la interacción con el entorno. En este sentido, hay que tener en cuenta que el objetivo final del agente es maximizar la recompensa obtenida, por lo que el aprendizaje está sesgado hacia los valores de recompensa que se van obteniendo. Cuando el agente es capaz de alcanzar el valor de recompensa máximo, diremos que el agente sigue una estrategia óptima. En teoría, todo problema con una solución basada en aprendizaje por refuerzo tiene una estrategia óptima alcanzable.

La estrategia se representa mediante las siguientes fórmulas:

$$\pi(s_t) = a_t$$

$$\pi(a_t | s_t) = P(a_t | s_t)$$

donde π representa la propia función de la estrategia. La primera opción se utiliza para ejecuciones deterministas, donde un estado siempre devuelve la misma acción.

El segundo caso es un enfoque estocástico, siendo actualmente el caso más común. El parámetro de entrada es el estado en el que se encuentra el agente, teniendo en cuenta la acción seleccionada en ese instante de tiempo. El valor de salida varía dependiendo del tipo de algoritmo y de cómo se lleva a cabo la selección de la acción, normalmente asociada a una probabilidad. Respecto a este tipo de policy, entraremos en detalle en los capítulos siguientes. Como apunte, la estrategia óptima se representaría con π^* .



La mayoría de soluciones de aprendizaje por refuerzo que podemos encontrar hoy en día hacen uso de modelos de *deep learning* para seleccionar las acciones a partir de los estados en los que se encuentra el agente. Por ello, hay recursos en los que la estrategia o *policy* se interpreta como el propio modelo de *deep learning*. En este sentido, es importante tener en cuenta que la selección de la acción también depende del algoritmo de refuerzo seleccionado, por lo que es una combinación de ambas partes.

2.2.2 Exploración

Hasta ahora hemos indicado en varias ocasiones que el agente aprende a partir de la interacción con el entorno. Además, acabamos de introducir el concepto de estrategia, que es el método que el agente sigue para seleccionar las acciones en cada momento de la ejecución. Nos falta conocer cómo comienza el proceso de aprendizaje, es decir, cómo el agente empieza a valorar cuáles son las mejores acciones a tomar en cada estado.

Para ello tenemos el concepto de exploración. En aprendizaje por refuerzo, la exploración **es un intervalo de tiempo durante el entrenamiento del agente en el que el agente prueba y valida las acciones que toma en base a la recompensa recibida del entorno. Es un proceso que comienza siendo totalmente aleatorio y que, a partir de esas respuestas, va perdiendo esa aleatoriedad para dar paso al conocimiento adquirido.**

Podemos encontrar diferentes tipos de proceso de exploración, aunque uno de los más conocidos y utilizados para introducir este concepto es el llamado *epsilon-greedy*. **En epsilon-greedy controlaremos el proceso de exploración mediante un hiperparámetro llamado “épsilon”**, cuyo valor oscila entre 1 y 0. Este parámetro nos permitirá ir monitorizando el grado de aleatoriedad durante el proceso de exploración.



Ejemplo

Un ejemplo de este proceso sería:

- a. Comenzamos con un valor alto de épsilon, normalmente 0,99.
- b. En cada step, calculamos un valor aleatorio entre 0 y 1 y lo comparamos con épsilon. Si el valor aleatorio es menor, ejecutamos una acción seleccionada por el modelo. Si no, ejecutamos una acción aleatoria.
- c. Al finalizar el step, reducimos el valor de épsilon para, conforme el entrenamiento se va desarrollando, ir dejando cada vez más espacio a acciones seleccionadas con el modelo en vez de acciones aleatorias.

Este proceso de exploración se realiza durante la fase de entrenamiento, ya que está ligado al descubrimiento de acciones y a la prueba y error en la ejecución. Cabe destacar que el **número de iteraciones sobre las que vamos a ejecutar el proceso de exploración es un nuevo hiperparámetro que, dependiendo del entorno y del reto a resolver, puede tomar un valor del orden de millones de steps**.

2.2.3 Explotación

Una vez el agente ha llevado a cabo el proceso de exploración, entramos en una nueva fase del entrenamiento conocida como explotación. En realidad, el modo de explotación será también el que utilice el agente durante su fase de **inferencia**, una vez esté totalmente entrenado.

En explotación, las posibilidades de ejecutar una acción aleatoria son muy reducidas, dándole todo el peso de la decisión al modelo del agente. El objetivo de esta fase es asegurar todo lo posible el aprendizaje desarrollado hasta el momento, a partir de las decisiones que el modelo toma en los estados que visita el agente.



Una decisión importante en la relación entre exploración y explotación es la definición de los límites en cada caso durante el proceso de entrenamiento. Hay una rama de investigación en este sentido conocida como *Exploration-Exploitation trade-off*, sobre la que también podemos encontrar una sección en Barto y Sutton (1992).



Dependiendo del algoritmo que se use, los conceptos de exploración y explotación pueden variar. Intuitivamente, la idea es saber que al comienzo del aprendizaje el agente ejecuta acciones aleatoriamente para ir adquiriendo experiencia y, seguidamente, emplear ese conocimiento acumulado e ir perfeccionando la toma de decisiones. Al final, su conocimiento será suficiente para tomar la mejor acción disponible.

2.2.4 Experiencia

El último de los conceptos avanzados que veremos es el concepto de experiencia. En aprendizaje por refuerzo, **la experiencia se refiere a la estructura de datos donde se irá almacenando la información generada durante la ejecución**. Hay que tener en cuenta que en la mayoría de ejecuciones los datos generados se almacenan en buffers de memoria, de tal forma que tenemos que definir protocolos o estrategias para hacer una gestión óptima. Dentro de posibles estructuras de datos que ayuden a esta gestión óptima podemos encontrar colas de mensajería, listas o arrays multidimensionales.

Junto con la experiencia encontramos el concepto de transición. **La transición es el elemento mínimo que se almacena en la experiencia e incluye cuatro componentes fundamentales: el estado actual, la acción tomada, la recompensa recibida y el siguiente estado.**

Las transiciones se obtienen en cada iteración de la ejecución a partir de la información generada por el agente y por el entorno en un momento determinado. **El conjunto de transiciones recolectadas en una secuencia de iteraciones se conoce como “trayectoria”**. Con los cuatro elementos que forman una transición seremos capaces de entrenar a nuestros agentes en cualquier algoritmo de aprendizaje por refuerzo.

Durante el proceso de entrenamiento y dependiendo del tipo de algoritmo que se use, el uso que se hace de las transiciones es a partir de conjuntos o paquetes de datos, o de trayectorias completas que contienen un grupo de transiciones, similar a los entrenamientos en *deep learning*. Entraremos en detalle en el siguiente capítulo cuando estudiemos las clasificaciones de los algoritmos.

2.3. Resumen

En este capítulo nos hemos centrado en una introducción a los principales conceptos con los que trabajaremos durante la asignatura. Como en cualquier ámbito científico, es importante manejar y controlar la nomenclatura y terminología necesaria para poder llevar a cabo nuestros desarrollos e implementaciones.

Hemos dividido los conceptos en dos grupos. En el primer grupo, conceptos básicos, hemos presentado los términos más importantes dentro del aprendizaje por refuerzo y que son transversales a cualquier algoritmo que usemos. Los elementos vistos en este grupo siempre se comportan de la misma manera desde un punto de vista conceptual, sea cual sea la implementación que se lleve a cabo.

El segundo grupo, conceptos avanzados, cubre conceptos cruciales dentro del aprendizaje por refuerzo pero que varían dependiendo del algoritmo que se use y de la configuración de hiperparámetros que definimos en los experimentos.

El próximo capítulo será un puente entre los términos que hemos presentado y los algoritmos que veremos en la asignatura. Por ello, nos centraremos en definir dos grupos principales de algoritmos en base al uso que hacen de los datos disponibles y a la información del entorno que tienen a su disposición durante la ejecución.



Capítulo 3

Clasificación de algoritmos de aprendizaje por refuerzo

El objetivo de los capítulos anteriores ha sido sentar las bases del aprendizaje por refuerzo, centrándonos en las definiciones y conceptos importantes que vamos a utilizar durante la asignatura. Entre estos conceptos, hemos resaltado en varias ocasiones el hecho de que el agente aprende a partir de la interacción con el entorno, pero sin entrar en detalle sobre el propio proceso en sí. A partir de este capítulo comenzaremos a vislumbrar cómo y de qué manera se lleva a cabo el proceso de aprendizaje.

Tal y como encontramos en Arulkumaran, K. (2017), existen diversas formas a la hora de hacer una clasificación de los distintos enfoques que podemos utilizar en las soluciones de aprendizaje por refuerzo. En nuestro caso, nos centraremos en **dos grupos principales**, un primer grupo centrado en **la gestión que se hace de los datos recolectados para el aprendizaje del agente** y un segundo grupo en base al **tipo de información que hay disponible del entorno**. Estos grupos se conocen como “**algoritmos basados en estrategia**” y “**algoritmos basados en modelo**”, respectivamente.

Hay otras clasificaciones y agrupaciones para las diferentes familias de algoritmos, como, por ejemplo:

- a. Basadas en el método de optimización que el agente usa durante el aprendizaje
- b. Basadas en si tienen la posibilidad de usar multiproceso.
- c. Basadas en si el espacio de acciones es discreto o continuo.

Algunas de estas agrupaciones las veremos en los próximos capítulos, mucho más específicos de las características de los algoritmos.

El hecho de agrupar los diferentes algoritmos por características similares es muy importante para tener una primera idea a la hora de abordar un problema con una solución de aprendizaje por refuerzo. Como vimos en el primer capítulo, son muchas las decisiones y restricciones que vamos a encontrar respecto a hiperparámetros, recursos computacionales o decisiones de implementación. De ahí que decidir cuál es el mejor algoritmo a utilizar en cada caso sea crucial.

A continuación, comenzaremos con la primera de las clasificaciones, algoritmos basados en estrategia. Este grupo tiene un impacto crítico en la forma en la que el agente aprende, principalmente por la gestión de los datos y la *policy*. De ahí que sea muy recomendable haber repasado y estudiado estos conceptos antes de continuar.

3.1. Basados en estrategia

Como veíamos en el capítulo anterior, la estrategia se refiere a la forma en la que el agente selecciona una acción en un estado. Dependiendo del algoritmo que se implemente y del momento en el que se encuentre el proceso de aprendizaje, la selección de acciones es diferente, ya que se tienen en cuenta elementos como el conjunto de experiencias que se está usando, el tipo de evaluación que se hace de cada acción o el grado de aleatoriedad que se tendrá en cuenta. En este sentido, diferenciamos dos grupos dependiendo de la estrategia que se siga: *on-policy* y *off-policy*.



Como indicamos en el capítulo anterior, el concepto de *policy* está muy ligado al modelo que el agente utiliza para, a partir de un estado, seleccionar la acción a ejecutar. Es por ello que en las clasificaciones que veremos a continuación tiene mucho peso la estrategia que se lleve a cabo, ya que afectará al modo en el que el modelo se actualizará.

El objetivo de una estrategia *on-policy* es hacer uso de la experiencia a corto plazo. Por experiencia a corto plazo nos referimos al conjunto de transiciones que se han ido almacenando en un número de *steps* fijo. Hay un componente importante a tener en cuenta y es que **la experiencia recolectada durante este número de *steps* finito se obtiene utilizando la *policy* actual**: por eso, el nombre de *on-policy*.

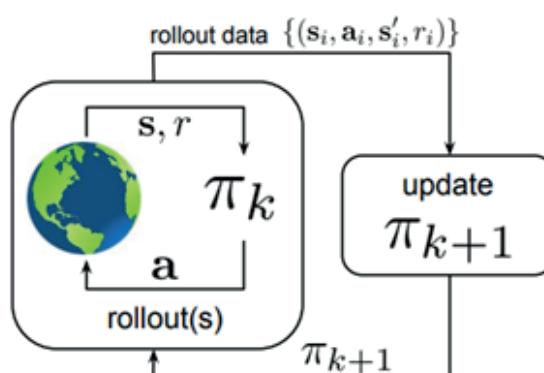


Figura 2. Diagrama general de una estrategia *on-policy*. Adaptado de Levine, S. et al. (2020). Offline Reinforcement Learning: Tutorial, Review and Perspectives on Open Problems. Cornell University-Arxiv, pág. 2.

A efectos prácticos, este tipo de estrategia implica que cada vez que el modelo del agente se entrena lo hará solo teniendo en cuenta la experiencia almacenada con la versión del modelo más reciente. Una vez que el modelo actualice sus parámetros, la experiencia se desechará y se comenzará una trayectoria nueva.

En el otro lado encontramos la estrategia **off-policy**. En esta estrategia, **la actualización del modelo también se realiza cada cierto número de steps, pero teniendo en cuenta toda la experiencia que se ha ido almacenando hasta ese momento.**

La experiencia recolectada está compuesta por un conjunto finito de transiciones que se han ido generando durante toda la interacción entre el agente y el entorno, es decir, que se han ido generando con diferentes versiones del modelo del agente. Por ejemplo, podemos encontrarnos en la situación de tener datos del comienzo de la ejecución, cuando el agente se comportaba de manera totalmente aleatoria, y que esos datos se sigan teniendo en cuenta en las actualizaciones del modelo en el instante de tiempo actual.

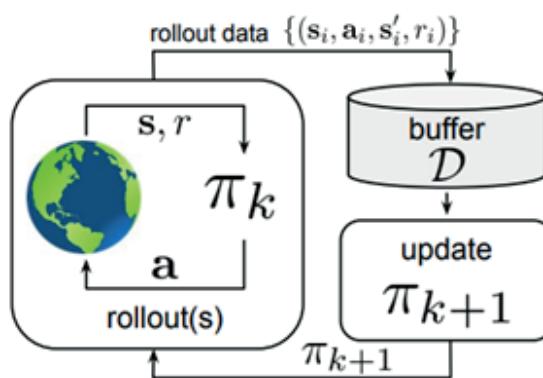


Figura 3. Diagrama general de una estrategia off-policy. Adaptado de Levine, S. et al. (2020). Offline Reinforcement Learning: Tutorial, Review and Perspectives on Open Problems. Cornell University-ArXiv, pág. 2.

Por ello es importante tener en cuenta que la experiencia es un conjunto finito de transiciones que también necesita ir actualizándose. La forma más común de actualización es *new-in old-out*, donde en el momento que se llegue al límite predefinido comenzaremos a eliminar los elementos más antiguos para dar paso a los nuevos, de forma similar a cómo se hace en algunas colas de mensajería en software.

Cada vez que queremos hacer una actualización del modelo en algoritmos basados en off-policy seleccionaremos un subconjunto aleatorio a partir de los datos disponibles, como ocurre en los entrenamientos de soluciones con arquitecturas de *deep learning*. El tamaño de este subconjunto es también un hiperparámetro que tenemos que definir al comienzo de la ejecución.

3.2. Basados en modelo

En esta clasificación, lo primero que es importante aclarar es que **el concepto de modelo que manejamos se refiere al modelo del entorno, no al modelo del agente**. Hasta ahora, todos los ejemplos y comentarios relativos a la idea de modelo eran del lado del agente; de ahí que también hayamos mencionado en varias ocasiones que este modelo se puede implementar mediante una red neuronal o arquitectura basada en *deep learning*. En este caso, el modelo es relativo al entorno y **se centra en si tenemos conocimiento de las dinámicas del mismo o no**.

En base a las características y la información disponible del entorno durante la interacción con el agente, el modo de aprendizaje será diferente, pudiendo aprovechar esta información para adaptar sus procesos y el tratamiento de los datos. Dependiendo de si disponemos de ese conocimiento extra del entorno o no, encontraremos **dos grupos principales de soluciones: *model-free* y *model-based***.

Una solución ***model-based*** se corresponde justamente con lo que indicábamos al comienzo de la sección: **existe información suficiente y completa para estimar las transiciones entre estados y las recompensas esperadas**. Este tipo de soluciones se suelen apoyar en otras técnicas dentro de la inteligencia artificial, como, por ejemplo, algoritmos de planificación o búsquedas basadas en (meta)heurísticas. Al conocer el modelo que rige el entorno, podemos estimar las mejores acciones en el estado actual a partir de simulaciones a futuro. Un ejemplo de este caso es la solución de AlphaGo (Silver, D. et al (2016)), que explicaremos en detalle en el Capítulo 5.



El obstáculo que podemos encontrarnos es que, en problemas complejos, el espacio de acciones y de estados es muy amplio y, por ello, la mayoría de las veces tendremos que combinar nuestra solución de aprendizaje por refuerzo con otros algoritmos que nos permitan aplicar una solución más tratable desde el punto de vista computacional.

En el caso de no conocer cómo se comporta nuestro entorno, trabajaremos en un dominio *model-free*. Este escenario es el que nos encontraremos en la mayoría de retos actuales relacionados con aprendizaje por refuerzo, como es el caso de la aplicación de estas soluciones sobre videojuegos de Atari en Mnih, V., Kavukcuoglu, K. et al. (2015). **Al no conocer un modelo del entorno, nuestra solución se centrará en aproximar lo mejor posible las mejores acciones para distintos estados del entorno a partir del conocimiento que se tiene en el estado actual.**

3.3. Esquema de algoritmos de aprendizaje por refuerzo

Con la idea de conectar estos dos grupos de clasificaciones con los algoritmos que vamos a estudiar en la asignatura, vamos a basarnos en una organización de los algoritmos más conocidos dentro del estado del arte en aprendizaje por refuerzo. El esquema sobre el que trabajaremos es:

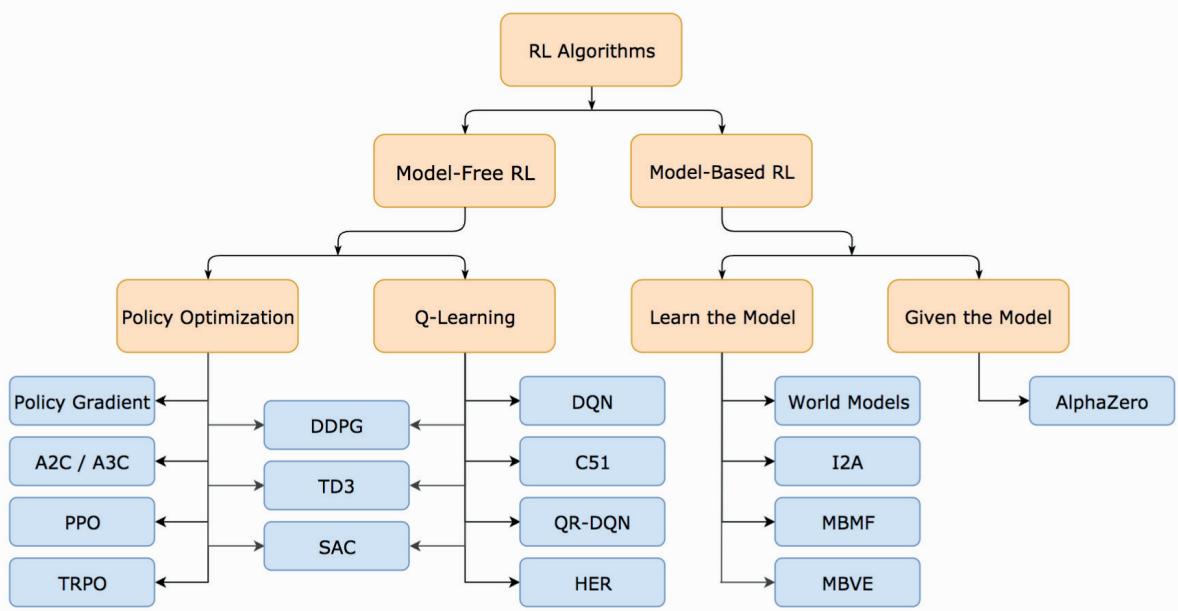


Figura 4. Clasificación de algoritmos de aprendizaje por refuerzo. Recuperado de Achiam, J. y OpenAI (2018). OpenAI Spinning Up RL. MIT License

Como vemos, los algoritmos están agrupados por niveles a partir de las clasificaciones presentadas. La primera diferencia que encontramos es en relación con las implementaciones, si son *model-based* o *model-free*. Hoy en día, la mayoría de soluciones y ejemplos son *model-free*, de ahí que tengamos un abanico de posibilidades más amplio que en *model-based*. Aunque no olvidemos que en las soluciones *model-based* se utiliza una combinación entre diferentes algoritmos (que pueden pertenecer al grupo de *model-free* también) con otras técnicas de inteligencia artificial, como es el caso de AlphaGo.

A efectos prácticos, vamos a darle más peso a los ejemplos del conjunto de *model-free*, ya que son los más representativos actualmente y han alcanzado resultados del estado del arte en multitud de problemas.



Respecto a AlphaGo, aunque entraremos más en detalle en el Capítulo 5, la solución final hace uso de modelos basados en Actor-Critic junto con modelos de búsqueda óptima en árboles, aprovechando la estimación de bondad de las acciones junto con la comprobación del movimiento en futuras posiciones del tablero. Es un claro ejemplo de la combinación de una solución de aprendizaje por refuerzo con otros enfoques basados en inteligencia artificial.

Los algoritmos *model-free* podemos dividirlos en dos grupos principales, dependiendo del modo de aprendizaje que el agente utilice durante la ejecución del experimento. Podemos encontrar también un grupo mixto que aprovecha las características de cada una de las divisiones, aunque en la asignatura nos centraremos en entender las diferencias entre ambas aproximaciones para conocer el límite y las particularidades dependiendo del experimento que queramos desarrollar.

La organización de los algoritmos *model-free* parte de dos algoritmos básicos de aprendizaje por refuerzo, **Deep Q-Networks** (Mnih, V., Kavukcuoglu, K. et al. (2015)) y **Policy Gradients** (Sutton, R. et al. (2000)).

Aunque entraremos en detalle en cada algoritmo en el siguiente capítulo, algunos de los puntos principales que los definen son:

- a. El uso que se hace de la experiencia para el proceso de aprendizaje. **Deep Q-networks usa una estrategia off-policy mientras que Policy Gradients usa una estrategia on-policy.**
- b. Las **funciones de coste** que usan para guiar el proceso de aprendizaje.
- c. **Deep Q-networks** solo funciona con un **espacio de acciones discreto** mientras que **Policy Gradients** puede trabajar en un **espacio de acciones discreto y continuo**.
- d. **El método para escoger la acción en cada algoritmo es diferente.** Deep Q-networks utiliza un **método voraz** a partir de la recompensa estimada, mientras que Policy Gradients utiliza una distribución aleatoria de probabilidades ponderada.

Estas diferencias son la base que define cada familia de los algoritmos que vemos en la clasificación. El resto de algoritmos más avanzados han ido apareciendo estos últimos años, como A3C (Mnih, V. et al. (2016)), PPO (Schulman, J. et al. (2017)) o DDPG (Lillicrap, T. et al. (2016)). Son nuevas versiones que han ido evolucionando, mejorando e incluso combinando algunos de estos aspectos, como, por ejemplo:

- a. Un **uso más óptimo de la experiencia** durante el proceso de aprendizaje.
- b. **Adaptaciones en las funciones de coste** para potenciar situaciones favorables y la toma de decisiones.
- c. Explotar la capacidad computacional y la **ejecución multiproceso**.

3.4. Resumen

En este capítulo hemos presentado una serie de clasificaciones para entender cómo podemos organizar los algoritmos de aprendizaje por refuerzo en relación con sus características. Esto nos ayudará a saber qué algoritmo es el que mejor se adapta al reto que tengamos que resolver.

Las clasificaciones se han basado en dos puntos cruciales en cualquier técnica de aprendizaje: el uso que se hace de los datos recolectados y si tenemos información extra que nos permita mejorar el propio proceso de aprendizaje. En relación con esto, las dos clasificaciones principales que hemos visto son algoritmos basados en estrategia y algoritmos basados en modelo.

En los algoritmos basados en estrategia encontramos dos grupos, *on-policy* y *off-policy*. Los algoritmos *on-policy* hacen uso de los datos recolectados solo con la última versión de la *policy* aprendida hasta el momento, para luego actualizarla con la nueva información. Una vez la *policy* es actualizada, la información se desecha para comenzar una nueva trayectoria. En cambio, los algoritmos *off-policy* mantienen una memoria con experiencia recolectada con distintas versiones de la *policy* del agente durante toda la interacción con el entorno. Durante el proceso de entrenamiento, en el momento de actualizar la *policy*, escogen un conjunto aleatorio de transiciones para llevar a cabo la actualización.

La otra clasificación que hemos visto es la de algoritmos basados en modelo. Es importante recordar que por modelo nos referimos al modelo del entorno, no al del agente. Aquí volvemos a encontrarnos dos posibilidades, *model-free* y *model-based*. En *model-free* no conocemos el modelo del entorno, por lo que el agente utiliza el conocimiento acumulado hasta el estado actual para estimar las mejores acciones a tomar. En el caso de *model-based*, sí que conocemos un modelo de cómo el entorno se comporta en sus transiciones dependiendo de las acciones que el agente tome, por lo que tomamos ventaja de esa información para hacer una estimación más precisa de cómo de buena o mala es la acción seleccionada.

La mayoría de algoritmos que encontramos hoy en día en la literatura se usan para problemas del tipo *model-free*, pudiendo luego pertenecer a un enfoque *on-policy/off-policy*. En este sentido, hemos visto algunos ejemplos de estos algoritmos juntos con sus características principales para conocer qué es lo que hace que pertenezcan a un grupo o al otro.

En el siguiente capítulo comenzaremos con la explicación de los algoritmos básicos para saber cómo el agente es capaz de aprender durante el proceso de entrenamiento, resaltando los puntos principales que hemos visto en la clasificación presentada durante este capítulo.



Capítulo 4

Algoritmos básicos de aprendizaje por refuerzo

En los capítulos anteriores hemos introducido la mayoría de conceptos alrededor de la idea de que el agente aprende a partir de su interacción con el entorno. Sabemos que ese aprendizaje se produce a partir de las acciones que el agente va tomando y de la recompensa que el entorno le devuelve en cada momento.

Para entrar en el detalle de este proceso, comenzaremos con las explicaciones de los algoritmos más representativos en las soluciones de aprendizaje por refuerzo actuales. En concreto, introduciremos dos algoritmos que son la base del resto: Deep Q-networks y Policy Gradients.

4.1. Deep Q-Networks

Deep Q-Networks (Mnih, V., Kavukcuoglu, K. et al. (2015)) **es un algoritmo de aprendizaje por refuerzo basado en el algoritmo de Q-learning, siendo su aportación principal la combinación de este algoritmo con técnicas de deep learning.**



El modelo de *deep learning* más usado en las soluciones Deep Q-Networks son las redes convolucionales. Esto se debe a que, en la mayoría de casos, el entorno de simulación está basado en videojuegos y el tipo de dato disponible es a nivel de píxel.

Q-learning (Watkins, C. J. C .H., Dayan, P. (1992)) **es un algoritmo que se centra en cuantificar la calidad (la Q del nombre viene del inglés quality) de seleccionar una acción en un estado determinado. La calidad, a su vez, será definida como recompensa esperada a futuro.**



El objetivo del algoritmo es encontrar la función Q óptima, esto es, la función que en cada estado ejecuta la mejor acción de entre todas las disponibles. Es importante notar que la función Q se definirá en relación a los pares estado-acción.

Esta función Q también se puede interpretar como la función encargada de definir la estrategia que sigue el agente. En el caso de Q-learning, la estrategia que comúnmente se utiliza es una estrategia *greedy* apoyada en el proceso de exploración-explotación que veíamos en el segundo capítulo. Matemáticamente, la función Q óptima se define tal que:

$$Q^*(s, a) = \max_{\pi} E R_t | S_t = s, a_t = a, \pi]$$

Cada vez que el agente tiene que decidir qué acción tomar, la estrategia estimará las recompensas esperadas de cada una de las acciones y seleccionará la acción que le retorne el valor más alto. Siguiendo esta estrategia durante todas las iteraciones de la ejecución, se llegaría a la secuencia de acciones óptima para alcanzar la mayor recompensa posible.

Cuando hablamos de **recompensa esperada nos referimos a la suma de recompensas futuras que espera el agente en un estado determinado.** Un elemento clave en esta definición es el concepto de futuro. Para hacer una estimación del valor futuro se definirá la recompensa futura en términos de expectativa:

$$Q^\pi(s, a) = E [R_t | S_t = s, a_t = a]'$$

$$R_t = \sum_{k=0}^{\infty} y^k r_{t+k}$$

La fórmula de la izquierda la podemos interpretar como la estimación de la recompensa esperada a partir de la información que se va obteniendo durante la ejecución. En el componente de la derecha, nótese el cálculo de la recompensa para el estado actual basado en la recompensa inmediata y la suma de las recompensas futuras. Es importante tener en cuenta **el factor multiplicativo de cada recompensa de los estados siguientes, conocido como “discount factor”**. Este factor define la importancia que le damos a las recompensas futuras en nuestra estimación, estando su valor por defecto entre 0.95 y 0.99.

Para el cálculo de la recompensa esperada utilizaremos la ecuación de Bellman como en Mnih, V., Kavukcuoglu, K. et al. (2015). La ecuación de Bellman se define tal que:

$$Q^*(s, a) = E [r + \gamma \max_{a'} Q^*(s', a') | s, a]$$



Ejemplo

Vamos a analizar los distintos elementos que componen esta ecuación. Lo primero, está compuesta a partir del conjunto de elementos que hemos ido almacenando en nuestra experiencia: estado, acción, recompensa y siguiente estado. Esta es la razón por la que se almacenan estos campos en cada transición: porque serán los elementos que se necesiten para los cálculos que hay que realizar durante el proceso de aprendizaje. Otro punto a destacar es la recurrencia definida en la fórmula. En cada estado se selecciona una acción, lo que nos devuelve la recompensa actual:

$$Q^*(s, a) = E [r + \max_{a'} Q^*(s', a') | s, a]$$

Ahora tenemos que incluir cómo de bueno es el estado al que nos movemos. De ese objetivo se encarga el segundo operando, analizando todas las acciones disponibles para ver qué recompensa nos devuelve cada una y así saber cuál es la mejor acción en el siguiente estado:

$$Q^*(s, a) = E [r + \max_{a'} Q^*(s', a') | s, a]$$

Uniendo los dos operandos obtenemos la recompensa esperada en el estado actual para la acción seleccionada. Esta fórmula definida en términos de recurrencia temporal nos permite ejecutar la ecuación de Bellman durante un número finito de estados para ir construyendo la función Q de forma cada vez más precisa.

Con las fórmulas definidas previamente somos capaces de obtener una estrategia óptima en espacios de estados y de acciones manejables computacionalmente. Si el problema tiene demasiada complejidad, no seremos capaces de modelar nuestros pares de estado-acción debido a la cantidad de situaciones y estados distintos que nuestro agente encontrará. **Es justo en estas situaciones cuando se usará un modelo de red neuronal como función aproximadora**, como en Mnih, V., Kavukcuoglu, K. et al. (2015).

El hecho de reemplazar nuestra función Q por una red neuronal hace que necesitemos tener en cuenta todos los detalles necesarios para que una solución basada en *deep learning* funcione correctamente. Justo esto es lo que indicábamos en el primer capítulo cuando nos referímos a esa complejidad extra a la hora de definir valores para los hiperparámetros de ambas técnicas, o sea, el modelo del agente basado en una arquitectura de *deep learning* y el algoritmo de aprendizaje por refuerzo seleccionado.

4.1.1 Pseudocódigo del algoritmo Deep Q-Networks

Una vez conocemos la fórmula que usaremos para el proceso de aprendizaje y que vamos a usar una arquitectura de *deep learning* para aproximar la selección de acciones, vamos a analizar el pseudocódigo del algoritmo para desglosar sus partes más importantes:

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

Figura 5. Pseudocódigo del algoritmo de Deep Q-Networks. Recuperado de Mnih, V. et al. (2013) Playing Atari with Deep Reinforcement Learning. Cornell University-Arxiv

En primer lugar, **inicializamos la estructura de datos que vamos a utilizar como memoria para almacenar las transiciones que usaremos para entrenar nuestro agente**. Es importante notar cómo la memoria se inicializa con una capacidad finita, por lo que también tendríamos que definir una política de actualización de datos.

En el caso de Deep Q-Networks, la estrategia que seguiremos es una estrategia *off-policy*, de ahí que necesitemos una estructura de este tipo donde almacenar los datos que se van generando. Hay que recordar que en *off-policy* las transiciones almacenadas se producen con diferentes versiones del modelo del agente en distintos momentos de la ejecución.



Recordemos que la policy del agente puede ser interpretada como el modelo de deep learning que usamos para estimar las acciones en un estado determinado; de ahí que hablemos de diferentes versiones del modelo durante la ejecución.

Además de la memoria, inicializamos la función Q con valores aleatorios. **En el caso de usar arquitecturas de deep learning, esta función Q se corresponde con nuestro modelo de deep learning. En otro caso, la función Q se correspondería con una matriz donde relacionamos los pares estado-acción con los valores de recompensa esperados.** Es por ello que si el espacio de acciones y/o de estados es muy grande, representar las relaciones en una matriz sería inviable, por lo que el uso de redes neuronales para aproximar los valores es más adecuado y útil.

A continuación encontramos dos bucles para controlar los episodios y las iteraciones. Cuando definimos estos conceptos en el segundo capítulo indicamos cómo se encargan de monitorizar la ejecución y llevar un orden en la misma. Por eso, se implementan como bucles anidados para englobar las tareas y ejecuciones específicas

en cada caso. Podemos observar que antes del comienzo del bucle de *steps*, justo después del comienzo del episodio, inicializamos la secuencia s_1 y la preprocesamos.

Justamente esto se corresponde con la primera observación de la ejecución y el preprocesamiento de la misma, que sería lo que en el Capítulo 2 definimos como estado. El hecho de que sea una secuencia lo abordaremos en la siguiente sección de este apartado.

Dentro del bucle de *steps*, es interesante comprobar cómo al comienzo del mismo se produce el proceso de exploración. A partir de la comprobación del valor de ϵ seleccionamos una acción aleatoria o la acción con mayor recompensa esperada a partir del modelo Q. Nótese que, en el caso de usar el modelo Q, el valor de entrada sería el estado en ese instante de tiempo y la versión del modelo dependería de los parámetros en la ejecución actual.

Una vez seleccionada la acción, se ejecuta en el entorno y recolecta la información que este devuelve. Se hacen las actualizaciones necesarias para la próxima iteración de la ejecución y almacenamos los campos necesarios de la transición que se ha generado. Estos campos se corresponden con los que presentamos en el segundo capítulo: estado, acción, recompensa y siguiente estado.

Al utilizar una estrategia *off-policy*, el entrenamiento se realizará usando subconjuntos aleatorios de transiciones a partir de la experiencia global. Por ello necesitamos obtener un **minibatch** aleatorio de transiciones para llevar a cabo una actualización del modelo. Es en estos subconjuntos donde se pueden encontrar transiciones de ejecuciones pasadas que fueron recolectados con otras versiones de la *policy* del agente.

Una vez conocida la estrategia que utilizaremos para usar la experiencia adquirida durante el entrenamiento pasamos a la definición de la función de coste. La función de coste para medir el error que se va cometiendo está basada en la ecuación de Bellman que hemos explicado al comienzo del apartado. De hecho, lo que haremos será diferenciar en la ecuación de Bellman entre **el componente objetivo y el componente predictivo**:

$$L = E((r + \max_{a'} Q^*(s', a')) - Q^*(s, a)) ;$$

$$\text{Objetivo: } (r + \max_{a'} Q^*(s', a')) ;$$

$$\text{Predicción: } Q^*(s, a)$$

Por ello, lo primero que hacemos es calcular el componente objetivo y_j . Si el estado actual es terminal, el valor de y_j será el de la recompensa actual. Si no, utilizamos Bellman para calcular la recompensa actual y la esperada del siguiente estado. Una vez tenemos el valor objetivo, la predicción la realizamos con el modelo y la acción seleccionada, para así medir el error cometido y poder llevar a cabo una actualización del modelo.

Para ir entrenando el modelo se usará esta función de coste cada un cierto número de *steps* fijado previamente, que será el intervalo de tiempo en el que el agente irá recolectando experiencia reciente para luego poder tenerla en cuenta en el momento de actualizar los pesos del modelo.

4.1.2 Aclaraciones del pseudocódigo del algoritmo Deep Q-Networks

Una vez entendido el algoritmo de Deep Q-Networks, todavía hay algunos detalles que necesitan de cierta aclaración. El primero de ellos es que, en la función de coste, ambos componentes dependen de la misma función Q. Si se implementa la función de coste de manera literal a esta definición, tanto la predicción como el valor esperado estarán basados en la misma función Q, el modelo de red neuronal, lo que implicará que van a estar cambiando constantemente durante el proceso de aprendizaje. Esto provocará que el modelo no sea capaz de aprender correctamente, llegando a una situación de divergencia en la mayoría de los casos.

Para solventar esta situación, en Mnih, V., Kavukcuoglu, K. et al. (2015) se propone definir **dos modelos neuronales iguales en su arquitectura**. Uno de los modelos se encargará de predecir las acciones y el otro será el responsable de **devolver el resultado para comparar con la acción elegida**. Este segundo modelo se conoce como **modelo target** y es un punto crítico para un correcto funcionamiento del algoritmo de Deep Q-networks. Este modelo va a tener los mismos pesos que el modelo predictivo, pero, a diferencia de este, **se actualizará en intervalos de tiempos más largos**, manteniéndose congelado el resto del tiempo y, por tanto, manteniendo los mismos valores durante ese intervalo de tiempo para realizar las comparaciones siempre contra el mismo resultado.

Otro detalle importante está relacionado con el almacenamiento de las observaciones. Hemos explicado que las observaciones se corresponden con la inicialización de una secuencia pero, ¿por qué una secuencia si, en principio, en cada iteración obtenemos sólo una observación?

Esta pregunta está muy ligada al caso de uso sobre el que se desarrolló el algoritmo de Deep Q-networks, los videojuegos. **En el caso de los videojuegos hay un componente fundamental que es la evolución de los frames o pantallas durante la ejecución**. Esta evolución nos permite entender movimientos y tendencias para poder tomar una acción acorde.

En Deep Q-networks, durante el preprocesamiento de la observación, se define una **ventana de frames u observaciones pasadas** para tenerlas en cuenta en el estado actual como entrada al modelo. El valor por defecto utilizado en el artículo de referencia es de una ventana de cuatro observaciones. Los resultados empíricos aplicando esta modificación mejoraron notablemente, de ahí que se utilice ampliamente en otros entornos y problemas dentro del aprendizaje por refuerzo.



Una alternativa al uso de secuencias de *frames* es la combinación de una arquitectura de modelo convolucional con una arquitectura recurrente, como una red LSTM, que sea capaz de mantener el contexto de una secuencia de observaciones. Podemos encontrar un ejemplo de este tipo en Lampe, G. et al. (2017).

4.2. Policy Gradients

Como en cualquier solución de aprendizaje por refuerzo, el algoritmo de Policy Gradients (Sutton, R. et al. (2000)) es un algoritmo centrado en obtener la estrategia óptima relacionada con la recompensa que el agente va obteniendo. En su versión base, y a diferencia de Deep-Q networks, en **Policy Gradients** no se hace una estimación de la recompensa esperada relacionando los pares estado y acción, sino que **en cada estado se obtiene la probabilidad de cada acción**:

$$\pi(a_t | s_t) = P(a_t | s_t; \theta)$$

Trabajar sobre el espacio de probabilidades de las acciones implicará que **no se apliquen los procesos de exploración y explotación** tal y como hemos visto en el manual. En vez de ello, cada vez que el agente necesite seleccionar una acción lo hará a partir de **una distribución de probabilidades aleatoria, ponderada por las probabilidades de cada acción**.

Además de la forma en la que se seleccionan las acciones, hay otra diferencia clave para entender Policy Gradients. Aunque el aprendizaje se seguirá desarrollando en términos de recompensa esperada, es necesario definir un intervalo finito de steps, provocando que el **cálculo de la recompensa esperada se pueda realizar para cada conjunto de steps sin necesidad de definir una función de recurrencia que trabaje con un límite a futuro**.

Este comportamiento hace que Policy Gradients sea un algoritmo on-policy. Cada vez que se ejecute el conjunto de steps finito, la experiencia se usará para actualizar el modelo. Después de la actualización de los pesos del modelo, la experiencia se reseteará por completo. De esta forma solo se tendrá en cuenta la experiencia obtenida con la última versión del modelo para actualizar los pesos del mismo.

La estimación de las probabilidades de cada acción se obtiene a partir del estado en el que se encuentra el agente y de la versión del modelo en ese instante de tiempo. A lo largo de la ejecución, se usará la recompensa obtenida con cada acción para ir modificando y precisando las probabilidades de selección dependiendo del estado.

Intuitivamente, en cada estado se seleccionará una acción y se obtendrá una recompensa. Si la recompensa es positiva, esa acción es positiva por lo que, para ese estado, esa acción se querrá repetir en el futuro. Si la recompensa es negativa, la acción se intentará evitar. De ahí que en la función de coste de *Policy Gradients* se utilice la recompensa como factor de la *policy*:

$$L = \nabla_{\theta} \log \pi(a_t | s_t; \theta) R_t$$



Esta forma de actualización de las probabilidades de las acciones se ve impactada por la varianza de los datos respecto a las recompensas obtenidas. Sobre todo al comienzo de las ejecuciones, la misma acción en un estado determinado puede devolver recompensas positivas o negativas, por lo que usar solo la recompensa como factor de importancia de la acción es demasiado básico.

Al igual que ocurría con Deep Q-networks, la estimación de la estrategia se complica cuando los espacios de estados y acciones se vuelven demasiado complejo en cuanto a posibles combinaciones y decisiones. Es por ello que, nuevamente, el agente utilizará un modelo de *deep learning* para estimar las probabilidades del espacio de acciones a partir de un estado dado.

4.2.1 Pseudocódigo del algoritmo Policy Gradients

Analicemos a continuación el pseudocódigo del algoritmo de Policy Gradients:

| REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_* |
|---|
| Input: a differentiable policy parameterization $\pi(a s, \theta)$ Algorithm parameter: step size $\alpha > 0$ Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (e.g., to 0) Loop forever (for each episode): Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot s, \theta)$ Loop for each step of the episode $t = 0, 1, \dots, T - 1$: $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$ $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t S_t, \theta)$ |

Figura 6. Pseudocódigo del algoritmo REINFORCE/PolicyGradients. Recuperado de Sutton, R. y Barto, A. (2018). Reinforcement Learning: An Introduction. MIT License, pp 328.

Comenzamos con la inicialización aleatoria de los pesos del modelo que usará el agente para estimar las probabilidades de las acciones a partir del estado en el que se encuentra. En cuanto a la memoria, aunque no se defina una estructura de datos como en el caso de Deep Q-Networks, a efectos prácticos **sí hay que definir una variable donde almacenar las trayectorias durante la ejecución de los steps**.

Seguidamente comenzamos el bucle de ejecución. En este ejemplo se da un caso interesante y común en otras implementaciones de aprendizaje por refuerzo basadas en Policy gradients. En vez de tener dos bucles controlados para los episodios y los steps, definimos la ejecución solo en función de un solo bucle para almacenar la trayectoria. Este diseño se debe a que en los algoritmos *on-policy* tenemos que asignar un valor finito de steps para recolectar las transiciones en esa trayectoria; de ahí que tenga sentido controlar todo el experimento solo en base a los steps.

La selección de la acción a ejecutar depende de la distribución de probabilidades que se obtiene del modelo a partir del estado actual. **Esta distribución se utiliza, como los pesos de las acciones, en una selección aleatoria, para favorecer la exploración cuando las probabilidades son muy equitativas entre sí.** Una vez se selecciona la acción, se ejecuta en el entorno y se recolecta la información relativa a la observación y la recompensa para preparar la transición actual e incluirla en la trayectoria. Todo esto se generaliza en la quinta línea del pseudocódigo.

Como comentamos al comienzo del apartado, al trabajar con un conjunto finito de elementos en la trayectoria somos capaces de calcular las recompensas esperadas en cada instante de tiempo. Eso es justo lo que se calcula en las líneas seis, para poder utilizar los valores en la función de coste. **Este cálculo se conoce como discounted rewards.**

En Policy Gradients, la función de coste se basará en la optimización sobre la distribución de probabilidades introducida en el punto anterior:

$$L = \nabla_{\theta} \log \pi(a_t | s_t; \theta) R_t$$

Esta función está definida en términos de maximizar el valor resultante, por lo que normalmente se cambia el signo de la misma para transformarla en una función de minimización y así poder aplicar el algoritmo de *backpropagation* para actualizar los pesos del modelo del agente. La única diferencia con el pseudocódigo es que, en vez de la recompensa en un instante de tiempo, se usa la variable R_t a partir de la estimación de las recompensas en la trayectoria.

4.2.2 Aclaraciones del pseudocódigo del algoritmo Policy Gradients

Respecto a la función de coste, el elemento más importante es el **factor multiplicativo de la probabilidad de la acción**. Anteriormente se indicó que usar la recompensa directamente puede provocar una gran varianza en los datos y, por tanto, guiar el proceso de aprendizaje hacia mínimos locales o a la divergencia de la solución. En el siguiente capítulo se analizarán una serie de alternativas para evitar esta situación.

Hay otro componente de la función de coste que todavía no hemos explicado en detalle, **la aplicación de logaritmo a la selección de acciones**. Si recapitulamos, en Policy Gradients estamos actualizando directamente la probabilidad de las acciones a partir de la trayectoria que hemos almacenado. Esto implica que, dependiendo del número de veces que una acción es seleccionada, su impacto en la actualización del modelo va a ser mayor o menor.

Para equilibrar cómo afectan las probabilidades de las acciones en la actualización del modelo, el gradiente de la *policy* se divide por la probabilidad de la acción:

$$\frac{\nabla_{\theta} \pi(a_t | s_t; \theta)}{\pi(a_t | s_t; \theta)}$$

Y a partir de la regla de la cadena y del hecho de que la derivada de $\log x$ es $1/x$, podemos hacer la transformación al *log* de la *policy*:

$$\nabla \ln f(x) = \frac{\nabla f(x)}{f(x)}$$

Más allá de ser una transformación elegante desde un punto de vista matemático, esta adaptación de la función de coste de Policy Gradients facilita los cálculos que se llevan a cabo durante el proceso de aprendizaje. Es por ello que se utiliza en toda su familia de algoritmos.

4.3. Resumen

En este capítulo se han presentado los dos algoritmos básicos en aprendizaje por refuerzo con redes neuronales, Deep Q-networks y Policy Gradients. Ambos algoritmos se incluyen dentro del grupo *model-free*, por usarse comúnmente en entornos donde el objetivo es llevar a cabo la mejor estimación posible a partir de la información que el agente tiene disponible en un momento determinado.

En el caso de la clasificación basada en estrategia sí que encontramos una diferenciación más clara. Deep Q-networks es un claro ejemplo de una estrategia *off-policy*, mientras que Policy gradients forma parte de los algoritmos que siguen una estrategia de tipo *on-policy*.

En cuanto al proceso de aprendizaje, y más concretamente a las funciones de coste utilizadas, hemos estudiado la ecuación de Bellman y la actualización directamente sobre la *policy* que se está optimizando. El primer caso es la función que utilizaremos en algoritmos de la familia de Deep Q-networks, mientras que el segundo caso será el utilizado en toda la familia de Policy Gradients.

El siguiente capítulo es el último del manual y se centrará en una introducción a algunos de los algoritmos más reconocidos actualmente en soluciones de aprendizaje por refuerzo. Estos algoritmos son evoluciones y mejoras a partir de las dos técnicas que acabamos de estudiar. De ahí la importancia y recomendación de entender bien este capítulo como paso previo al estudio de los algoritmos más avanzados que presentaremos.



Capítulo 5

Algoritmos avanzados de aprendizaje por refuerzo

Durante los capítulos del manual hemos ido desarrollando, de una manera escalonada, todos los conceptos y términos que son necesarios para poder sacarle todo el partido a soluciones basadas en aprendizaje por refuerzo y *deep learning*. En el presente capítulo vamos a introducir los principales algoritmos que se usan hoy en día y que son el estado del arte en multitud de dominios.

Estos algoritmos se basan en las dos aproximaciones que estudiamos en el capítulo anterior, *Deep Q-networks* y *Policy Gradients*. De hecho, son evoluciones y mejoras de los aspectos más básicos donde *Deep Q-networks* y *Policy Gradients* pueden verse más limitados.

Es interesante tener en cuenta que en todos los algoritmos que vamos a cubrir se mantendrá la idea de clasificarlos dentro de los grupos estudiados, dependiendo del uso que hagan de los datos recolectados y de cómo aplican las funciones de coste durante el procesos de entrenamiento.

5.1. Actor Critic

Antes de entrar en detalle en el algoritmo de Actor-Critic (Konda, V. R. (2000)), volvamos al reto que teníamos en *Policy Gradients* respecto a la elección del factor multiplicativo a la hora de ponderar las acciones seleccionadas.

Comentamos cómo el uso del valor de la recompensa directamente puede derivar en situaciones no deseadas, debidas, principalmente, a la gran varianza que introduce en la solución. Para solventar esta situación, podemos encontrar en Schulman, J. et al. (2016) las siguientes alternativas:

$$g = \mathbb{E} \left[\sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right], \quad (1)$$

where Ψ_t may be one of the following:

- 1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
- 2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t .
- 3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
- 4. $Q^{\pi}(s_t, a_t)$: state-action value function.
- 5. $A^{\pi}(s_t, a_t)$: advantage function.
- 6. $r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$: TD residual.

The latter formulas use the definitions

$$V^{\pi}(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty}, \\ a_{t+1:\infty}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad Q^{\pi}(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1:\infty}, \\ a_{t+1:\infty}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad (2)$$

$$A^{\pi}(s_t, a_t) := Q^{\pi}(s_t, a_t) - V^{\pi}(s_t), \quad (\text{Advantage function}). \quad (3)$$

Figura 7. Alternativas para los factores utilizados en Policy Gradients. Recuperado de Schulman, J. et al. (2016). High Dimensional Continuous Control Using Generalized Advantage Estimation. Cornell University-ArXiv

Lo que resalta de cada versión de estas funciones son los diferentes factores que multiplican la probabilidad de la acción. **La primera y la segunda fórmula se centran en un refinamiento de la recompensa obtenida**, pero siguen la misma definición presentada en el algoritmo de Policy Gradients.

En la tercera fórmula se utiliza un baseline o umbral de referencia. Este umbral establecerá el límite para considerar una acción positiva o no. El factor de importancia de la acción se define usando la recompensa actual y este umbral, el cual, normalmente, se obtiene a partir del análisis del entorno y de la distribución de recompensas para saber en torno al rango de valores que se mueve.

En las siguientes fórmulas aparece un nuevo concepto, la estimación del *value*. El *value* (Barto, A. y Sutton, R (1992)) es un concepto muy destacado en aprendizaje por refuerzo y, de una manera intuitiva, **significa cómo de bueno es el estado en el que se encuentra el agente**. Por ello se puede usar para medir la bondad del estado y, por tanto, saber si se va por buen camino con la estrategia actual.

En las soluciones actuales, es muy común definir el factor de ponderación de las probabilidades de las acciones en función del *value*, ya que devuelve un umbral realista del valor medio del estado. Si la recompensa obtenida es mayor que el *value*, interpretaremos que la acción realizada es positiva y viceversa. Matemáticamente, el *value* se define:

$$V^{\pi}(s_t) = \mathbb{E}_{\substack{s_{t+1:\infty}, \\ a_{t+1:\infty}}} \left[\sum_{k=0}^{\infty} r_{t+k} \right]$$

Usando como valor de entrada el estado actual, ofrece un valor estimado sobre la recompensa esperada que el agente puede conseguir a partir de ese estado.



Otra interpretación del *value* se corresponde con la media de recompensas de todas las acciones que se pueden tomar en un estado.

El uso del *value* con la recompensa actual como factor de ponderación nos lleva a la definición de ventaja, que es el parámetro que se observa en la ecuación número cinco. Las tres últimas ecuaciones hacen uso de una función Q similar a la que hemos visto en Deep Q-networks y a funciones basadas en diferencias temporales teniendo en cuenta el *value* del siguiente estado también.



De entre todas las fórmulas presentadas, la más usada actualmente es la ecuación número cinco, definiendo el factor de ponderación en términos de una función de ventaja. En ella, tomamos como referencia el valor del estado y lo comparamos con la recompensa obtenida en ese instante de tiempo. La diferencia será el factor que se utilice para potenciar o minimizar la probabilidad de la acción. Esta será la función que usaremos en el resto de algoritmos del capítulo.

Es justo en este momento, al estimar las probabilidades de las acciones y el *value* del estado, donde el algoritmo de Policy Gradients se transforma en el conocido algoritmo de Actor-Critic (Konda, V.R. (2000)). **En este algoritmo, el actor correspondería a la distribución de probabilidades y el critic al value estimado, que es el que se encarga de decir cómo de buena es la decisión que se toma.**



El nombre de Actor-Critic viene justamente de su similitud con una audición donde un actor se expone a un crítico que valora su actuación. En este caso, esa misma situación se daría con las probabilidades que el actor propone y el valor del critic para indicar si se va por el camino acertado o no.

Al trabajar con dos salidas diferentes, distribución de probabilidades y el valor esperado, durante el proceso de entrenamiento estaríamos optimizando el aprendizaje en función de estos dos valores.

5.1.1 Pseudocódigo del algoritmo Actor-Critic

Vamos a echar un vistazo al pseudocódigo de Actor-Critic para poder desglosar las secciones que son diferentes respecto a Policy Gradients, principalmente en la función de coste:

Algorithm 1 Monte Carlo on policy actor-critic.

Require: Initialize policy π with parameters θ_π and value critic v_π with parameters θ_v

```

1: for each episode do
2:   Get initial state  $s$ 
3:   Initialize storage buffer  $S, A, R, S'$ 
4:   for  $i = 1, 2, 3 \dots N$  steps do
5:     Sample action with policy:  $a \sim \pi_\theta(s)$ 
6:     Run action through environment, obtain reward and post state:  $r, s' \leftarrow ENV(s, a)$ 
7:     Collect and store:  $S, A, R, S' \leftarrow s, a, r, s'$ 
8:      $s \leftarrow s'$ 
9:   end for
10:  Compute discount returns:  $\hat{V} = \sum_{t=0}^{N-1} \gamma^t r_{t+1}$ 
11:  Update  $\theta_v$  to minimize  $\sum_{n=1}^N \|v_\pi(s_n) - \hat{V}_n\|^2$ 
12:  With learning rate  $\alpha$ , update policy:  $\theta_\pi \leftarrow \theta_\pi + \alpha \nabla_\theta \log \pi(A|S)v_\pi(S)$ 
13: end for

```

Figura 8. Pseudocódigo del algoritmo de Actor-Critic. Recuperado de *Monte Carlo actor critic algorithm*. Por gopal_chitalia, en Reddit.

Como en los ejemplos del capítulo anterior, comenzamos nuevamente inicializando aleatoriamente los pesos del modelo del agente. En este caso encontramos dos modelos diferentes, uno para el *Actor* y otro para el *Critic*. Aunque en el pseudocódigo se traten como modelos distintos, en la mayoría de implementaciones actuales ambos modelos usan la misma arquitectura y solamente cambian la capa de salida. **Es decir, es el mismo modelo pero con una doble capa de salida, una para la distribución de probabilidades de las acciones y otra para un valor real que se corresponde con el value.**

Este pseudocódigo también difiere del ejemplo de Policy Gradients ya que, aunque siga una estrategia *on-policy*, utiliza un doble bucle de episodios e iteraciones. Podríamos encontrar otros ejemplos solo con el bucle de iteraciones donde lo importante es almacenar la experiencia durante la trayectoria actual antes de realizar una actualización del modelo del agente.

El bucle principal donde ocurre la lógica del algoritmo es similar al caso de Policy Gradients, salvo en el caso de la función de coste. En Actor-Critic la función de coste se divide en dos funciones diferentes, una para el *Critic* y otra para el *Actor*.

Por parte del *Critic*, la función de coste se transforma en una minimización del error que se comete al hacer la estimación del *value* siguiendo la filosofía de una regresión. El elemento con el que se compara el *value* es el valor \hat{v} , calculado en la línea diez a partir de los valores conocidos de recompensa. La función para el *Actor* es similar a Policy Gradients, a diferencia del factor multiplicativo de la probabilidad de la acción. Aunque en este caso se use el propio *value*, se podrían implementar otras alternativas como hacer **una estimación de la acción seleccionada mediante una función Q**, en vez de simplemente usar la recompensa inmediata.

En resumen, en el caso de Actor-Critic hay una evolución en la función de coste para dotar de más robustez el aprendizaje del agente en términos de certidumbre sobre qué acción realizar en cada estado de la ejecución. Esta robustez se consigue por medio de la estimación del *value* para valorar cómo de bueno es el estado en el que se encuentra el agente y, a partir de la comparación con la recompensa obtenida, adaptar la probabilidad de la acción seleccionada de manera oportuna.

5.2. A2C y A3C

A2C (Advantage Actor-Critic) (Clemente, A. et al (2017)) y A3C (Asynchronous Advantage Actor Critic) (Mnih, V. et al. (2016)) surgen de seguir evolucionando el algoritmo de Actor-Critic, nuevamente enfocados en el factor de relevancia de la acción seleccionada. Además, **estos nuevos algoritmos también tienen como objetivo intentar solventar problemas computacionales en sus ejecuciones**, como es la necesidad de usar **GPU** en los experimentos para el entrenamiento de las arquitecturas de *deep learning*.

Respecto al factor de relevancia que usamos en esta ocasión, recordemos las opciones que veíamos en la sección de Policy Gradients:

$$g = \mathbb{E} \left[\sum_{t=0}^{\infty} \Psi_t \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right], \quad (1)$$

where Ψ_t may be one of the following:

- 1. $\sum_{t=0}^{\infty} r_t$: total reward of the trajectory.
- 2. $\sum_{t'=t}^{\infty} r_{t'}$: reward following action a_t .
- 3. $\sum_{t'=t}^{\infty} r_{t'} - b(s_t)$: baselined version of previous formula.
- 4. $Q^{\pi}(s_t, a_t)$: state-action value function.
- 5. $A^{\pi}(s_t, a_t)$: advantage function.
- 6. $r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$: TD residual.

The latter formulas use the definitions

$$V^{\pi}(s_t) := \mathbb{E}_{\substack{s_{t+1:\infty} \\ a_{t+1:\infty}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad Q^{\pi}(s_t, a_t) := \mathbb{E}_{\substack{s_{t+1:\infty} \\ a_{t+1:\infty}}} \left[\sum_{l=0}^{\infty} r_{t+l} \right] \quad (2)$$

$$A^{\pi}(s_t, a_t) := Q^{\pi}(s_t, a_t) - V^{\pi}(s_t), \quad (\text{Advantage function}). \quad (3)$$

Figura 7. Alternativas para los factores utilizados en Policy Gradients. Recuperado de Schulman, J. et al. (2016). High Dimensional Continuous Control Using Generalized Advantage Estimation. Cornell University-ArXiv

En este caso, usaremos la conocida como “función de ventaja”. Como se ha mencionado anteriormente, **la función de ventaja es una función que relaciona la recompensa obtenida (o la recompensa esperada a futuro, según el caso que se quiera aplicar) por el agente a partir de la acción seleccionada en un estado determinado con el value del estado.**

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t)$$

A partir del cálculo de esta relación, somos capaces de establecer una referencia sobre cómo de buena o mala es la acción seleccionada con lo que se espera en ese estado. Esta referencia es mucho más robusta que otras opciones, ya que directamente evalúa la acción seleccionada respecto al valor medio esperado.

Otra gran diferencia entre los algoritmos de A2C y A3C respecto Actor-Critic **es el uso de diferentes workers (agentes en nuestra simulación) que se ejecutan en paralelo**. Intuitivamente, este enfoque permite incrementar la experiencia que se va obteniendo durante el aprendizaje, así como las diferentes situaciones a las que cada agente se enfrenta, lo que implica más diversidad en las observaciones recolectadas y en las acciones seleccionadas. Básicamente, al ejecutar más agentes en paralelo estamos generando más transiciones y más diversas, lo que ayuda a que el entrenamiento sea más generalizable y permita una convergencia más óptima.

Es interesante resaltar que, en este caso, todos los agentes que se ejecutan en paralelo hacen uso del mismo modelo de *deep learning* a modo de inteligencia común. Justo aquí se produce la diferencia entre A2C y A3C, dependiendo de si la actualización de este modelo común se produce de manera síncrona o asíncrona. Veamos el pseudocódigo en cada caso para conocer mejor cómo se produce esta agregación del conocimiento a partir de la ejecución de los agentes en paralelo.

5.2.1 Pseudocódigo del algoritmo A2C

En una situación donde el modelo se va actualizando de manera síncrona estaríamos en el contexto de A2C. Esto conllevaría una comunicación entre los agentes que se ejecutan durante el entrenamiento para, de manera ordenada, ir modificando los parámetros del modelo conforme van terminando sus trayectorias. Esta modificación, normalmente, se realiza mediante una agregación de los gradientes de cada proceso, para así compartir el conocimiento que se va adquiriendo en cada caso.

Un ejemplo de pseudocódigo de A2C sería:

Algorithm 1 Parallel advantage actor-critic

```

1: Initialize timestep counter  $N = 0$  and network weights  $\theta, \theta_v$ 
2: Instantiate set  $e$  of  $n_e$  environments
3: repeat
4:   for  $t = 1$  to  $t_{max}$  do
5:     Sample  $a_t$  from  $\pi(a_t | s_t; \theta)$ 
6:     Calculate  $v_t$  from  $V(s_t; \theta_v)$ 
7:     parallel for  $i = 1$  to  $n_e$  do
8:       Perform action  $a_{t,i}$  in environment  $e_i$ 
9:       Observe new state  $s_{t+1,i}$  and reward  $r_{t+1,i}$ 
10:      end parallel for
11:    end for
12:     $R_{t_{max}+1} = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_{t_{max}+1}; \theta) & \text{for non-terminal } s_t \end{cases}$ 
13:    for  $t = t_{max}$  down to 1 do
14:       $R_t = r_t + \gamma R_{t+1}$ 
15:    end for
16:     $d\theta = \frac{1}{n_e \cdot t_{max}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{max}} (R_{t,i} - v_{t,i}) \nabla_\theta \log \pi(a_{t,i} | s_{t,i}; \theta) + \beta \nabla_\theta H(\pi(s_{e,t}; \theta))$ 
17:     $d\theta_v = \frac{1}{n_e \cdot t_{max}} \sum_{i=1}^{n_e} \sum_{t=1}^{t_{max}} \nabla_{\theta_v} (R_{t,i} - V(s_{t,i}; \theta_v))^2$ 
18:    Update  $\theta$  using  $d\theta$  and  $\theta_v$  using  $d\theta_v$ .
19:     $N \leftarrow N + n_e \cdot t_{max}$ 
20: until  $N \geq N_{max}$ 

```

Figura 9. Pseudocódigo del algoritmo Advantage Actor-Critic (A2C). Recuperado de Clemente, A. et al. (2017). Efficient Parallel Methods for Deep Reinforcement Learning. Cornell University – Arxiv e-Print.

A grandes rasgos, el cuerpo de todo el pseudocódigo es muy similar a las versiones que ya hemos visto de Policy Gradients y Actor-Critic. La experiencia se sigue acumulando en trayectorias siguiendo una estrategia *on-policy*. Hay que tener en cuenta el componente multiproceso, ya que, en realidad, estaríamos recolectando varias trayectorias al mismo tiempo, una por cada proceso que se esté ejecutando.

Nuevamente, encontramos la principal diferencia con respecto a su versión anterior en la función de coste. Concretamente, en el uso de la función de ventaja como factor multiplicativo de la acción seleccionada. Será esta función de ventaja la que nos definirá cómo de buena o mala es la acción seleccionada:

$$\text{función de ventaja en el pseudocódigo: } A^{\pi}(s_t, a_t) \rightarrow (R_{t,i} - V_{t,i})$$

No olvidemos que en este caso seguimos estimando la distribución de probabilidad de las acciones y el *value* del estado. Al igual que en Actor-Critic encontraremos la función de coste dividida en dos bloques, cada bloque enfocado en la minimización del error para cada objetivo.



En la función de coste para el Actor encontramos un nuevo componente, la función λ , introducida en Mnih, V. et al. (2016). Esta función es una estimación de la entropía de la policy que se utiliza para mejorar la exploración.

Volvamos al hecho de que podemos ejecutar varios procesos en paralelo. En este caso, cada proceso va desarrollando su propia interacción con el entorno, con su versión del modelo y recolectando su propia experiencia. Una vez que cada proceso ha actualizado su modelo, todos los procesos se sincronizan para compartir sus versiones de los modelos y cargar esta nueva versión en cada proceso.

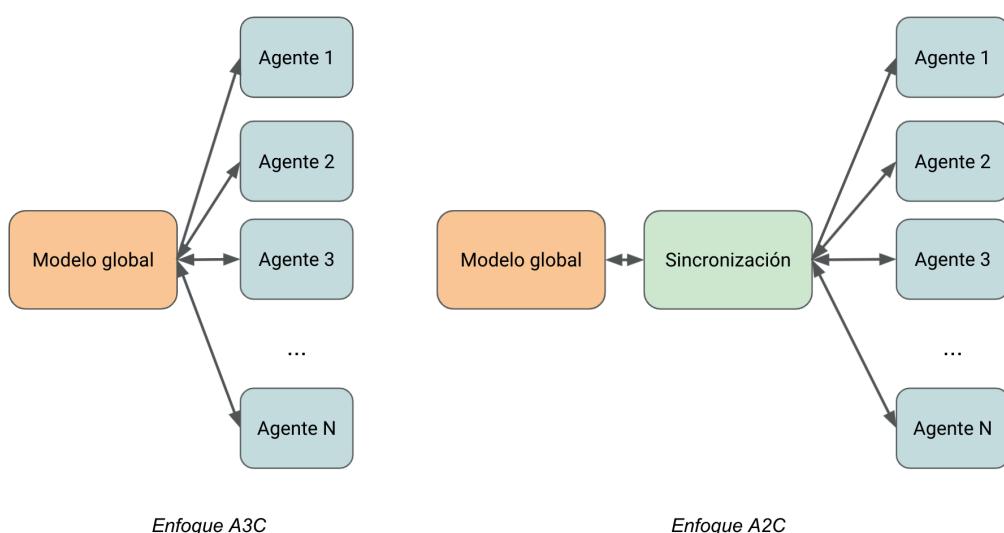


Figura 10. Arquitectura de comunicación A2C vs A3C. Adaptado de Lilian Weng, *Policy Gradient Algorithms* (<https://lilianweng.github.io/>)

Esta forma de comunicación entre procesos es segura y ordenada, pero tiene el inconveniente de poder presentar latencias altas al tener que esperar a que todos los agentes terminen su ejecución. Para intentar solventar este problema, veamos una versión alternativa con A3C.

5.2.2 Pseudocódigo del algoritmo A3C

La principal característica que encontramos en A3C viene de **la asincronía en la ejecución de los procesos. A3C se diseñó con el objetivo de no depender computacionalmente de recursos muy restrictivos**, como el uso de GPU o TPU. Partiendo de este punto, se desarrolla la paralelización de los procesos a nivel CPU para aprovechar todas las ventajas de tener varios agentes ejecutándose en paralelo durante el proceso de entrenamiento.

Al implementar la ejecución multiproceso de manera asíncrona, **cada vez que un agente termine su episodio actualiza el modelo con la versión que tiene actualmente, sin tener en cuenta el estado de otros agentes.**

A priori, esto puede parecer negativo, debido a que agentes con versiones del modelo obsoletas o de menor calidad van a modificar el modelo general en el que todos los agentes se basan. En realidad, al ejecutar un número de agentes considerables durante un tiempo de entrenamiento suficiente, este comportamiento no se reproduce de esta manera, ya que la propia experiencia que se va acumulando y las diferentes versiones de los modelos llegan a un punto de equilibrio para asegurar la convergencia de la solución.

A continuación se muestra un ejemplo de pseudocódigo para A3C:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```

// Assume global shared parameter vectors  $\theta$  and  $\theta_v$  and global shared counter  $T = 0$ 
// Assume thread-specific parameter vectors  $\theta'$  and  $\theta'_v$ 
Initialize thread step counter  $t \leftarrow 1$ 
repeat
  Reset gradients:  $d\theta \leftarrow 0$  and  $d\theta_v \leftarrow 0$ .
  Synchronize thread-specific parameters  $\theta' = \theta$  and  $\theta'_v = \theta_v$ 
   $t_{start} = t$ 
  Get state  $s_t$ 
  repeat
    Perform  $a_t$  according to policy  $\pi(a_t|s_t; \theta')$ 
    Receive reward  $r_t$  and new state  $s_{t+1}$ 
     $t \leftarrow t + 1$ 
     $T \leftarrow T + 1$ 
  until terminal  $s_t$  or  $t - t_{start} == t_{max}$ 
   $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t // \text{Bootstrap from last state} \end{cases}$ 
  for  $i \in \{t - 1, \dots, t_{start}\}$  do
     $R \leftarrow r_i + \gamma R$ 
    Accumulate gradients wrt  $\theta'$ :  $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta') (R - V(s_i; \theta'_v))$ 
    Accumulate gradients wrt  $\theta'_v$ :  $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$ 
  end for
  Perform asynchronous update of  $\theta$  using  $d\theta$  and of  $\theta_v$  using  $d\theta_v$ .
until  $T > T_{max}$ 
  
```

Figura 11. Pseudocódigo del algoritmo Asynchronous Advantage Actor Critic (A3C). Recuperado de Mnih, V. et al. (2016). Asynchronous Methods for Deep Reinforcement Learning. Cornell University – Arxiv e-print.

Como se aprecia, el algoritmo es similar a A2C a excepción de los momentos de actualización del modelo global de la ejecución. **En vez de esperar a que terminen todos los agentes que se ejecutan en paralelo, cada vez que un agente termina actualiza los pesos del modelo.**

Como apunte, es interesante destacar la forma en la que se monitoriza el entrenamiento tanto en A2C como A3C. Normalmente se dispone de todos los agentes que van a ejecutarse en modo entrenamiento, que son los que están preparados para cargar y modificar el modelo global. Junto con estos *workers*, se define un nuevo agente para ejecutarse en modo inferencia, cuya labor es cargar la versión del modelo global en ese instante de tiempo y ejecutar un episodio. Este agente no modifica el modelo global en ningún caso. De esta manera, se puede ir controlando la calidad del modelo global y, por tanto, tener una idea del punto en el que se encuentra el entrenamiento.

5.3. Proximal Policy Optimization

Continuando con la evolución de los algoritmos que pertenecen a la familia de Policy Gradients llegamos a PPO o Proximal Policy Optimization (Schulman, J. et al. (2017)). El algoritmo de PPO está basado en otro algoritmo conocido como TRPO o Trust Region Policy Optimization (Schulman, J. et al. (2017)).

Tanto TRPO como PPO son algoritmos *on-policy*. Como ya hemos visto, en los algoritmos *on-policy* optimizamos la *policy* del agente con la trayectoria que se ha obtenido a partir de la *policy* más actual. **En este sentido, los métodos que pertenecen a la familia de trust region methods abordan la actualización de la policy diferenciando entre la policy anterior y la policy nueva, calculando el ratio entre ambas.**

El objetivo de TRPO es maximizar todo lo posible la actualización de los parámetros del modelo del agente teniendo en cuenta este ratio, manteniendo que la distancia entre las policies sea menor que un límite definido para asegurar la convergencia del proceso.

De ahí viene la idea de trust región: con la relación entre las policies y los límites establecidos se define una región para asegurarnos que podemos actualizar lo máximo posible la policy siguiendo la dirección más óptima. Para llevar a cabo este proceso, la comparación entre *policies* se realiza por medio de **KL-divergence**, ya que es una medida de distancia entre distribuciones de probabilidad:

$$KL(\Theta_1, \Theta_2) = E[\pi_{\Theta_1}(\cdot | s) \| \pi_{\Theta_2}(\cdot | s)]$$

El reto con TRPO es que en su función de coste utiliza matrices **hessianas**, o sea, se basa en segundas derivadas para la optimización durante el proceso de aprendizaje. Este hecho provoca que el coste computacional y conceptual sea una barrera de uso. PPO, como veremos en la sección sobre el pseudocódigo de este algoritmo, suaviza este obstáculo conceptual manteniendo unos resultados espectaculares en multitud de retos diferentes, lo que le ha llevado a convertirse en uno de los algoritmos del estado del arte y de referencia dentro del aprendizaje por refuerzo.

n comparación con otros algoritmos, la característica que destacaremos de este es la forma en **la que optimiza la distribución de probabilidades**. Además, al igual que A2C y A3C, aprovecha la ejecución multi-proceso con un conjunto de *workers* para aumentar la distribución de datos disponibles para el aprendizaje del agente. Aunque utilice agentes en paralelo, no llega a aplicar la misma estrategia de A2C y A3C, sino que **estos workers solamente llevan a cabo tareas de recolección**, en ningún caso varían el modelo o *policy* que se está utilizando.

Como los algoritmos de la familia de Policy Gradients, **PPO puede trabajar en espacios de acciones discretos y continuos**.

5.3.1 Pseudocódigo del algoritmo PPO

A continuación vamos a analizar el pseudocódigo del algoritmo de PPO:

Algorithm 1 PPO-Clip

- 1: Input: initial policy parameters θ_0 , initial value function parameters ϕ_0
- 2: **for** $k = 0, 1, 2, \dots$ **do**
- 3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
- 4: Compute rewards-to-go \hat{R}_t .
- 5: Compute advantage estimates, \hat{A}_t (using any method of advantage estimation) based on the current value function V_{ϕ_k} .
- 6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg \max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \min \left(\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_k}(a_t | s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.

- 7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg \min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^T \left(V_{\phi}(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.

- 8: **end for**

Figura 12. Pseudocódigo del algoritmo Proximal Policy Optimization (PPO) para su versión Clip. Recuperado de Schulman, J. et al. (2017). Proximal Policy Optimization Algorithms. Cornell University-ArXiv e-print.

Al continuar trabajando con el mismo esqueleto de algoritmo desde el ejemplo de Actor-Critic, podemos resaltar las siguientes similitudes:

- La ejecución se controla mediante un bucle de iteraciones, para definir cuándo termina una trayectoria. La trayectoria está formada por las transiciones, que, a su vez, están compuestas de estado, acción, recompensa y siguiente estado.
- Se calculan los *discounted rewards* (en este caso, llamados *rewards to go*) y las funciones de ventaja que luego se utilizarán en la función de coste.

Se usará un conjunto de *workers* ejecutándose en paralelo. Como PPO pertenece a la familia de Policy Gradients y, por tanto, hace uso de una estrategia *on-policy*, esto implica que durante una secuencia de *steps* se almacenará la trayectoria de transiciones utilizando la última versión de la *policy* de nuestro modelo. Esto se realizará por cada proceso que se esté ejecutando, de tal forma que cada *worker*, con la última versión del modelo, recolectará todas las transiciones de su propia trayectoria. Hay que recalcar que estos procesos solo hacen labores de recolección de estas trayectorias, en ningún caso hacen modificación alguna del modelo global.

A la hora de realizar la selección de datos para llevar a cabo la actualización del modelo, PPO cambia en comparación con sus antecesores. **En vez de utilizar todos los datos disponibles en la trayectoria, se ejecutan una serie de epochs** (como se hace en entrenamientos de *deep learning*) **seleccionando un conjunto aleatorio de datos de entre todos los datos recolectados en los procesos**. O sea, de entre todas las trayectorias recolectadas, se selecciona aleatoriamente un *batch* de datos para llevar a cabo varias iteraciones con el fin de optimizar la versión del modelo global usado por los agentes.

Esta variación, junto con una mayor recolección de trayectorias, implica un uso más rico de los datos disponibles, ya que aprovecha el hecho de tener un mayor número de trayectorias disponibles y, a la vez, se incluye la variabilidad de utilizar transiciones de ejecuciones diferentes, disminuyendo el sesgo de utilizar todos los datos pertenecientes a una misma trayectoria.

La función de coste es también muy específica en comparación con otros enfoques dentro de la familia de Policy Gradients. Partimos de que el modelo de los agentes sigue proporcionando dos salidas diferentes, una para la distribución de las probabilidades de las acciones y otra para la estimación del *value*. También destaca el uso de la función de ventaja como factor de ponderación de las acciones seleccionadas. **La diferencia aparece en la forma en la que se optimiza la distribución de probabilidades de las acciones.**



Realmente, hay dos versiones diferentes de PPO, dependiendo de la estrategia utilizada en la función de coste: PPO-penalty y PPO-clip. En esta versión del pseudocódigo veremos la estrategia de PPO-clip, ya que es más directa y fue la primera versión presentada por OpenAI.



Ejemplo

En vez de utilizar funciones de coste similar a sus predecesores, **PPO realiza una comparación, o cálculo del ratio, entre la distribución de probabilidades que se está calculando en el momento actual con su versión anterior:**

$$\min\left(\frac{\pi(a_t | s_t; \Theta)}{\pi(a_t | s_t; \Theta_k)} A^{\pi_{\Theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\Theta_k}}(s_t, a_t))\right)$$

donde $g(\epsilon, A) = (1 + \epsilon) A$ si $A \geq 0$

$$g(\epsilon, A) = (1 - \epsilon) A$$
 si $A < 0$

A partir de este ratio se obtiene la diferencia entre ambas distribuciones para poder optimizar en la dirección oportuna. Además, nótese que nos quedamos con el valor mínimo entre un rango con unos límites fijados:

$$\min\left(\frac{\pi(a_t | s_t; \Theta)}{\pi(a_t | s_t; \Theta_k)} A^{\pi_{\Theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\Theta_k}}(s_t, a_t))\right)$$

donde $g(\epsilon, A) = (1 + \epsilon) A$ si $A \geq 0$

$$g(\epsilon, A) = (1 - \epsilon) A$$
 si $A < 0$

De aquí viene la etiqueta de clip, ya que la actualización de las *policies* debe estar dentro de estos rangos controlados. Si sobrepasara estos rangos, se tomarían los valores límites para llevar a cabo la actualización. Esta estrategia nos asegura que no actualizamos las *policies* en exceso, evitando afectar negativamente a la convergencia del proceso de optimización.

5.4. Deep Deterministic Policy Gradient

El último algoritmo que veremos de la familia de *model-free* es el algoritmo conocido como DDPG o Deep Deterministic Policy Gradient (Lillicrap, T. et al. (2016)). Este algoritmo es muy interesante por el hecho de ser un **algoritmo híbrido, ya que combina características de la familia de Deep Q-networks con elementos de la familia de Policy Gradients.**



La idea con la que se concibe este algoritmo es la necesidad de trabajar con un espacio de acciones continuo cuando se utiliza un enfoque de Deep Q-networks.

Si recordamos el funcionamiento del algoritmo de Deep Q-networks, en el momento de la selección de una acción necesitaba realizar un análisis de todas las acciones disponibles para calcular la recompensa esperada en cada caso y así seleccionar el caso que la maximizara. Esto es inviable si pensamos en términos de un espacio continuo, apareciendo la opción de combinar Deep Q-Networks con Policy Gradients.

La combinación surge de usar un enfoque Actor-Critic en el que el Actor se corresponde con un modelo basado en Policy Gradients capaz de ofrecer como salida valores en una escala continua, mientras que el Critic será un modelo basado en Deep Q-networks para estimar los valores de recompensa esperados.

El algoritmo de DDPG es ampliamente usado en dominios donde el espacio de acciones es continuo, normalmente en proyectos de control o robótica. En los últimos años ha perdido terreno en comparación con otras técnicas como PPO, pero sigue siendo una solución referente dentro del aprendizaje por refuerzo para este tipo de escenarios.

5.4.1 Pseudocódigo del algoritmo Deep Deterministic Policy Gradient

Veamos el pseudocódigo del algoritmo de DDPG para resaltar las partes más interesantes:

Algorithm 1 DDPG algorithm

```

Randomly initialize critic network  $Q(s, a|\theta^Q)$  and actor  $\mu(s|\theta^\mu)$  with weights  $\theta^Q$  and  $\theta^\mu$ .
Initialize target network  $Q'$  and  $\mu'$  with weights  $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$ 
Initialize replay buffer  $R$ 
for episode = 1, M do
    Initialize a random process  $\mathcal{N}$  for action exploration
    Receive initial observation state  $s_1$ 
    for t = 1, T do
        Select action  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  according to the current policy and exploration noise
        Execute action  $a_t$  and observe reward  $r_t$  and observe new state  $s_{t+1}$ 
        Store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $R$ 
        Sample a random minibatch of  $N$  transitions  $(s_i, a_i, r_i, s_{i+1})$  from  $R$ 
        Set  $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$ 
        Update critic by minimizing the loss:  $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$ 
        Update the actor policy using the sampled policy gradient:

```

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

```

        Update the target networks:
         $\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$ 
         $\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$ 
end for
end for

```

Figura 13. Pseudocódigo del algoritmo Deep Deterministic Policy Gradient (DDPG). Recuperado de Lillicrap, T. et al. (2016). Continuous Control with Deep Reinforcement Learning. ICLR.

La forma de combinar Policy Gradients con Deep Q-networks que comentábamos al comienzo de la sección podemos observarla justo al comienzo de la implementación del pseudocódigo.

Se definen cuatro modelos diferentes:

- a. Un modelo basado en una arquitectura Deep Q-networks para la implementación del *Critic*, encargado de ofrecer un valor de bondad a las acciones seleccionadas.
- b. Un modelo basado en una arquitectura Policy Gradients determinista para el Actor, encargado de seleccionar la acción a ejecutar en el estado actual.
- c. Dos modelos que harán las labores de *target networks*, como veíamos en el algoritmo de Deep Q-network, para favorecer la estabilidad durante el proceso de entrenamiento.

La estrategia que usa DDPG es una estrategia off-policy, como en Deep Q-networks. Por ello, tenemos que definir una estructura de datos o *buffer* que nos sirva para almacenar la experiencia que se va acumulando durante la interacción entre el agente y el entorno.

Observamos cómo en este ejemplo existen dos bucles para ir controlando la ejecución, a nivel de episodios e iteraciones. Justo al comienzo del bucle de iteraciones se aprecian un par de novedades en cuanto al proceso de exploración y la selección de acciones.

Al trabajar con un modelo que devuelve un valor determinista, **la selección de acciones se hace directamente del modelo sin necesidad de usar una distribución de probabilidad aleatoria**. El factor que nos facilita la exploración, principalmente en las primeras ejecuciones, lo obtenemos con la función de ruido que acompaña la selección de las acciones, en el pseudocódigo representada como N . Esta función incluirá pequeñas perturbaciones para variar los valores de la acción a ejecutar y así favorecer la exploración. En el artículo de referencia, esta función está basada en un proceso de **Ornstein-Uhlenbeck**.

Desde este punto hasta la función de coste, la estructura es la misma que la que vimos en el algoritmo de Deep Q-networks: ejecutamos la acción, observamos la información que nos devuelve el entorno, almacenamos la transición y seleccionamos un conjunto de datos aleatorios de la memoria para proceder a actualizar los modelos del agente.

En la función de coste observamos la primera diferencia en el cálculo de los valores objetivo y_i . En el segundo término, donde en Deep Q-networks nos quedábamos con el valor máximo esperado de entre todas las acciones posibles, nos quedamos con la acción que nos devuelve la *target Actor* en el siguiente estado:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}))$$

Con esta variable y_i tenemos la estimación del *Critic* en el momento actual de la ejecución, la cual utilizaremos, como en el caso del algoritmo de Actor-Critic, a modo de comparativa del error para nuestro modelo Q como si fuera una optimización de un problema de regresión.

Para la estimación del Actor, la función de coste que utilizaremos será la misma que en la familia de Policy Gradients. El factor multiplicativo de la acción será el valor obtenido de la función Q y la acción corresponderá al valor devuelto por el modelo del *Critic*. Hay que destacar que en este caso no trabajamos con probabilidades de las acciones, sino que la propia distribución son los valores continuos que puede tomar el espacio de acciones, de tal forma que se optimiza directamente el uso o no de estos valores dependiendo del valor de Q .

5.5. Caso de uso model-based: AlphaGo

En este apartado vamos a estudiar el único ejemplo basado en una solución model-based, AlphaGo (Silver, D. et al. (2016)). La aplicación de un enfoque *model-based* se debe a que, al ser un juego de tablero, podemos desarrollar un agente de aprendizaje por refuerzo conociendo todas las reglas y dinámicas del juego y, además, teniendo en todo momento toda la información de la partida. Todo este contexto extra es el que usará el agente para poder estimar mejor las acciones que va tomando.

Más allá de los resultados obtenidos, hay que tener en cuenta que el juego del Go es uno de los más complicados que existen en la actualidad. Aun teniendo unas reglas sencillas, la cantidad de posibles situaciones del tablero, del orden de 10^{170} , equivale a un número mayor que átomos en el universo, provocando que, desde un punto de vista computacional, sea un reto muy complejo.

En AlphaGo encontramos no solo una combinación de algoritmos de refuerzo con modelos de deep learning, sino que también utiliza otro enfoque del aprendizaje automático como el aprendizaje supervisado. Su desarrollo comienza con un entrenamiento supervisado a partir de un histórico de partidas y, a partir de este conocimiento, continúa evolucionando el modelo del agente con los algoritmos de aprendizaje por refuerzo. Entraremos en detalle en estas fases en el siguiente apartado del capítulo.

Esta combinación de técnicas es una de las responsables de que AlphaGo obtuviera una capacidad nunca vista antes en este juego, no solamente con un nivel experto en cuanto a la probabilidad en la toma de decisiones, sino también en la creación de nuevos movimientos y estrategias.

Durante los últimos años han ido apareciendo otras versiones de AlphaGo con diferentes enfoques desde el punto de vista del proceso de entrenamiento. Es el caso de AlphaZero (Silver, D. et al. (2017)), una versión que ha mejorado de manera sustancial todos los resultados conseguidos por AlphaGo hasta ese momento.

Su principal diferencia es que el agente de AlphaZero fue entrenado a partir de jugar partidas consigo mismo, solamente conociendo las reglas de la partida, sin tener en cuenta un primer aprendizaje supervisado a partir de un histórico de partidas.

5.5.1 Análisis de la solución AlphaGo

Este apartado será diferente respecto a los apartados anteriores sobre el pseudocódigo de los algoritmos. Al ser una combinación de diferentes técnicas, vamos a centrarnos en las partes importantes del proceso global y así entender cómo se lleva a cabo el proceso de aprendizaje y de entrenamiento del agente desde un punto de vista de alto nivel.

En primer lugar, los datos de entrada de la solución se corresponden con una serie de variables a partir del estado de la partida, como las posiciones de las fichas para cada jugador y las distintas alternativas de movimientos. Estos datos son representados en diferentes capas siguiendo una estructura visual, para así poder tener ventaja del uso de modelos de redes convolucionales. A continuación se muestra el listado de estas variables:

| Feature | # of planes | Description |
|----------------------|-------------|---|
| Stone colour | 3 | Player stone / opponent stone / empty |
| Ones | 1 | A constant plane filled with 1 |
| Turns since | 8 | How many turns since a move was played |
| Liberties | 8 | Number of liberties (empty adjacent points) |
| Capture size | 8 | How many opponent stones would be captured |
| Self-atari size | 8 | How many of own stones would be captured |
| Liberties after move | 8 | Number of liberties after this move is played |
| Ladder capture | 1 | Whether a move at this point is a successful ladder capture |
| Ladder escape | 1 | Whether a move at this point is a successful ladder escape |
| Sensibleness | 1 | Whether a move is legal and does not fill its own eyes |
| Zeros | 1 | A constant plane filled with 0 |
| Player color | 1 | Whether current player is black |

Figura 15. Listado de variables de entrada en la solución de AlphaGo. Recuperado de Silver, D. et al. (2016). Mastering the game of Go with Deep neural networks and Tree search. *Nature*.

Estos datos de entrada se utilizarán en el conjunto de modelos que hemos mencionado previamente. La solución de AlphaGo está desarrollada en un flujo de tres niveles. **En el primer nivel se entrena dos modelos diferentes, una supervised policy network y una rollout policy.** Como su nombre indica, la *supervised policy network* es el modelo entrenado de manera supervisada, con un histórico de unos 30 millones de partidas de Go. El objetivo de este modelo es tener un modelo base con un buen nivel de precisión para poder estimar los movimientos que hay que realizar en un estado determinado. Acorde a los resultados mostrados en el artículo de referencia, este primer modelo tiene un *baseline* por encima de un 55 % de victorias.

En cuanto a la *rollout policy*, su arquitectura es diferente, ya que trabaja con un conjunto reducido de patrones de la partida con el objetivo de seleccionar una acción en un tiempo mucho menor que la *supervised policy network*. La precisión no es tan buena (solo un 27 % de precisión aproximadamente), **pero la idea es poder seleccionar más acciones en un tiempo casi marginal.**

La segunda fase se corresponde con el entrenamiento de la reinforcement learning policy network, es decir, el modelo entrenado a partir del algoritmo de aprendizaje por refuerzo centrado en decidir qué acción tomar. La arquitectura de este modelo es igual que la del modelo supervisado, tomando al comienzo los pesos ya entrenados. A partir de este momento, el resto del entrenamiento se realiza mediante partidas contra versiones pasadas de este mismo modelo.



Esta estrategia hace que el agente vaya mejorando en base a jugar contra las mejores versiones del pasado de sí mismo. La elección de la versión contra la que se enfrenta es seleccionada de manera aleatoria, para evitar situaciones de sobreajuste.

Durante el entrenamiento de la *reinforcement learning policy network*, la recompensa utilizada se corresponde con:

- 0 en cualquier instante de tiempo.
- 1 si el jugador gana la partida.
- -1 si el jugador pierde la partida.

Como esta recompensa puede presentar problemas al dar una respuesta tan tardía, lo que se hace es, dependiendo de cómo termine la partida, usar el valor de 1 o -1 para ponderar las acciones seleccionadas durante toda la trayectoria. Es decir, una vez un jugador ha ganado o perdido, todas las acciones que ha tomado durante la partida son ponderadas adecuadamente. Finalmente, este modelo será el utilizado para la estimación de las probabilidades de los movimientos que el agente llevará a cabo.

La última de las fases se centra en la **evaluación de la posición** en la que se encuentra la partida. Esta evaluación se realiza por medio de un modelo conocido como **value network**, que devolverá un valor numérico respecto a si el jugador ganará la partida o no a partir del estado actual.



Aunque hayamos presentado los diferentes modelos en diferentes fases, en realidad *reinforcement learning policy network* y *value network* se corresponden con el enfoque Actor-Critic que hemos estudiado previamente. De hecho, la arquitectura del modelo será la misma y lo que obtendremos serán dos salidas diferentes, una para la distribución de probabilidades de las acciones y otra para el valor del estado actual.

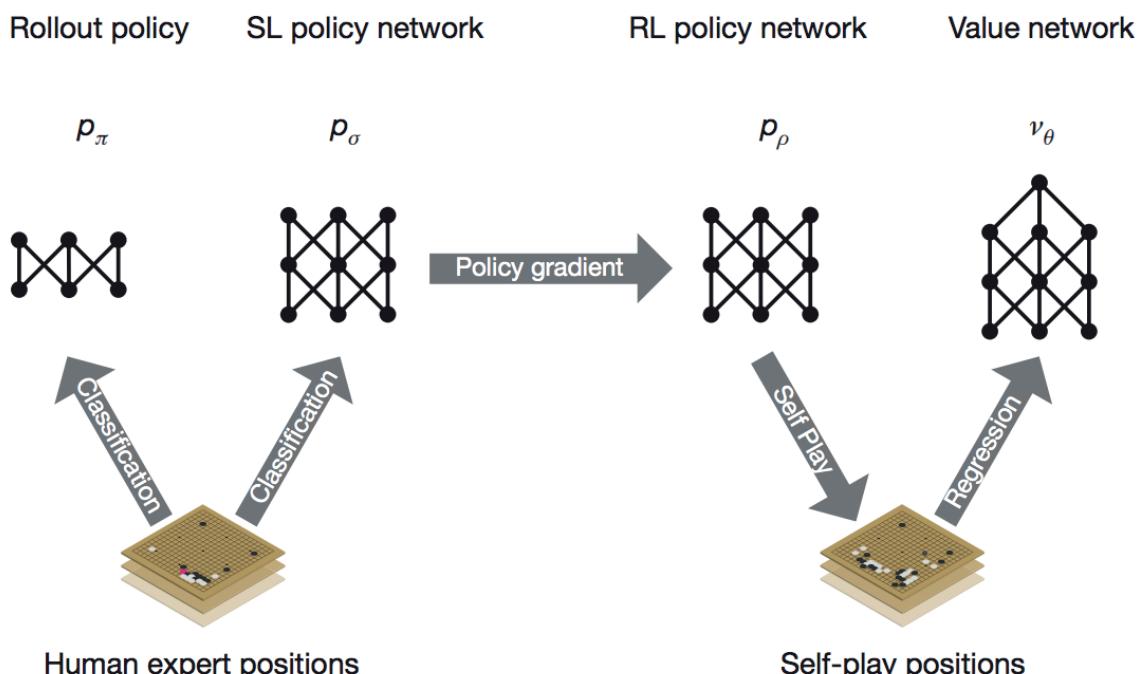


Figura 16. Fases de entrenamiento y modelos utilizados en AlphaGo. Recuperado de Silver, D. et al. (2016). Mastering the game of Go with Deep neural networks and Tree search. *Nature*.

Una vez tenemos todos los modelos que utilizaremos durante la ejecución nos faltaría aprovechar el conocimiento que tenemos del entorno para poder enriquecer la toma de decisiones. **Esto se produce utilizando una búsqueda en árbol basada en simulaciones de Montecarlo**, que será combinada con los modelos antes definidos.

En cada simulación, AlphaGo realizará las siguientes tareas:

1. Se elige la arista o camino que devuelve el valor máximo esperado a partir de la acción seleccionada. También se introduce un valor extra calculado a partir de una estimación a priori de la probabilidad de tomar ese camino.
2. Si el siguiente estado puede ser explorado, se utiliza la *reinforcement learning policy network* para almacenar las probabilidades de cada acción en el caso de que se tomen esos caminos. Estas probabilidades se corresponden con la estimación mencionada en el primer punto.
3. Cuando la simulación termina, el estado terminal se evalúa teniendo en cuenta la *value network* en el estado actual y la recompensa obtenida.
4. Por último, todos los valores Q de las acciones tomadas durante la simulación se actualizan a partir de los valores del *value* y la recompensa.

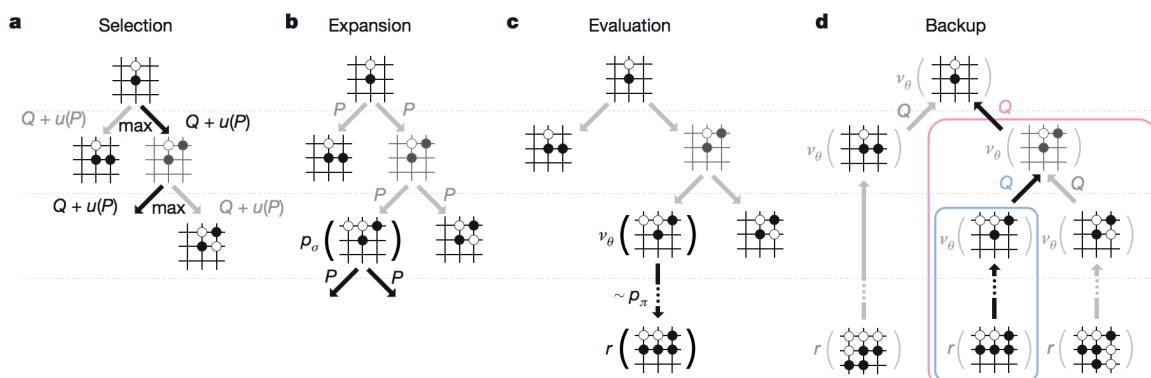


Figura 17. Tareas realizadas por AlphaGo en cada simulación de la búsqueda en árbol. Recuperado de Silver, D. et al. (2016). Mastering the game of Go with Deep neural networks and Tree search. *Nature*.

El análisis presentado muestra la complejidad en el diseño de la solución de AlphaGo para poder aprovechar todas las características de los distintos enfoques dentro del aprendizaje automático y el aprendizaje por refuerzo. Durante la asignatura se complementará este análisis con el detalle a nivel de implementación, para así tener la fotografía completa de la solución propuesta.

5.5.2 Comentarios sobre AlphaGo

Después del entrenamiento y el perfeccionamiento de la solución, los resultados obtenidos por AlphaGo son impresionantes. Se recomienda consultar el artículo de referencia para conocer información más detallada a partir de los *benchmarks* adjuntados. Por otro lado, también es muy recomendable la visualización del documental de Google DeepMind sobre el desarrollo de este proyecto, que puede ser encontrado en los enlaces de interés de este manual.

Por último, al ser un caso de uso real, es interesante analizar las necesidades computacionales para poder desarrollar una solución de este tipo. Según el equipo del proyecto, para llevar a cabo una combinación efectiva de la búsqueda basada en Montecarlo y los modelos de redes neuronales, AlphaGo ejecuta las simulaciones en varias CPU en paralelo, y la ejecución de los modelos en GPU también en paralelo. La versión final usa:

- 40 hilos para las simulaciones de la búsqueda en 48 CPU
- 8 GPU

Además, existe otra versión distribuida para poder exprimir al máximo los recursos disponibles y aumentar los tiempos de entrenamiento. En este escenario, los recursos computacionales se disparan a:

- 40 hilos de búsqueda en 1200 CPU
- 76 GPU

Queda a la vista las restricciones computacionales y de coste que un proyecto de este tipo arroja para poder reproducir los resultados presentados. Justamente, este punto fue comentado en el Capítulo 1 del manual para hacer ver el obstáculo que puede presentar las soluciones de aprendizaje por refuerzo dependiendo del entorno y la complejidad de la solución propuesta.

5.6. Resumen

En este capítulo se ha llevado a cabo un repaso de algunos de los algoritmos más destacados dentro del aprendizaje por refuerzo en la actualidad. Estos algoritmos son, a su vez, ejemplos que podemos identificar dentro de las clasificaciones que vimos en el Capítulo 3.

Relacionando los algoritmos con las clasificaciones, todos los algoritmos estudiados podemos encontrarlos dentro de retos *model-free* exceptuando AlphaGo, que estaría en el grupo *model-based*. En este punto es importante recordar que las clasificaciones basadas en modelo se refieren a la existencia de un modelo del entorno, por lo que podemos usar combinaciones de diferentes técnicas en el caso de *model-based*.

En el caso de la clasificación basada en estrategia, todos los algoritmos que hemos estudiado pertenecen al grupo de *on-policy* exceptuando Deep Deterministic Policy Gradient que sería un ejemplo *off-policy*.

En el caso concreto de cada algoritmo, podemos destacar dos objetivos principales. El primero de ellos es entender cómo se lleva a cabo el proceso de aprendizaje, resaltando las variaciones en las distintas evoluciones dentro de una familia. En segundo lugar, cómo realizan la gestión de la experiencia acumulada.

Hemos hecho más hincapié en la familia de Policy Gradients, ya que en los últimos años ha habido una gran evolución en las mejoras de sus versiones. También por su capacidad para trabajar en espacios de acciones continuos. Partiendo del algoritmo base en el Capítulo 4, Policy Gradients, hemos hecho un repaso de los principales cambios en sus sucesores, como Actor-Critic, A2C/A3C y PPO. A destacar el cambio de perspectiva en la función de coste en el caso de PPO y los algoritmos basados en *trust region*.

Por último, hemos analizado la solución de AlphaGo, que ha sido uno de los hitos dentro de la inteligencia artificial en los últimos años. Al ser una solución *model-based*, su implementación aprovecha el conocimiento del entorno para combinar modelos basados en Actor-Critic con técnicas de optimización en la toma de decisiones. En este caso, usando búsqueda óptima en árboles basada en Montecarlo.



Glosario

Algoritmo de planificación

Los algoritmos de planificación forman parte de la rama de la inteligencia artificial de planificación automática. Una de las características principales de estos algoritmos es la definición de una secuencia de acciones reguladas por precondiciones y poscondiciones que se deben cumplir durante su ejecución. Estas condiciones están relacionadas con características del entorno y del objetivo a alcanzar.

API

Del inglés *application process interface*, una API se corresponde con un catálogo de funciones y procedimientos que sirven para la comunicación entre diferentes sistemas o servicios. En aprendizaje por refuerzo, usaremos el término API para definir las funciones disponibles para la interacción entre el agente y el entorno.

Aprendizaje *from scratch*

Llamamos aprendizaje *from scratch* al tipo de aprendizaje que ocurre desde cero, comenzando con un enfoque totalmente aleatorio al no tener ninguna referencia inicial. Está muy relacionado con el concepto de prueba y error y el tipo de aprendizaje que los humanos tenemos al comienzo de nuestra vida.

Benchmark

Un *benchmark* se corresponde con una batería de experimentos que tiene como objetivo la comparación de ciertas variables o indicadores a partir de diferentes soluciones a un mismo problema. En el caso del aprendizaje por refuerzo, algunas variables comunes en *benchmarks* son el uso de recursos, tiempos de entrenamiento, número de episodios o recompensa obtenida durante el experimento.

Big data

Big data se corresponde con el conjunto de patrones, técnicas y tecnologías necesarias para poder almacenar, consultar y procesar grandes conjuntos de datos. Es un concepto muy ligado al aprendizaje automático, ya que la unión de ambos enfoques es la responsable de la explosión de las soluciones basadas en inteligencia artificial en los últimos años.

Búsqueda en árboles

Una búsqueda en árbol es un tipo de heurística relacionada con la simulación de decisiones a partir del estado actual, para poder estimar cuál es la mejor acción a tomar en el siguiente estado. Al ejecutar simulaciones, la búsqueda en árbol está fuertemente ligada a procesos de simulación, como es el caso de Montecarlo.

Cadenas de Markov

Una cadena de Markov se corresponde con una serie de estados asociados a una probabilidad de transición entre ellos. Una de sus características más representativas es que la decisión de pasar de un estado a otro depende únicamente de la información disponible en el estado actual.

Canales RGB

En visión por computador, los canales RGB son las capas de información que se corresponden con los valores de píxeles para los colores rojo, verde y azul. A partir de la composición de estos píxeles se obtiene toda la gama de colores conocida. Técnicamente, se corresponde con tres dimensiones en la última posición de la composición de un *frame*.

Deep learning

El *deep learning* o aprendizaje profundo es un conjunto de técnicas dentro del aprendizaje automático. Una característica fundamental es que las aproximaciones que propone están basadas únicamente en el concepto de red neuronal. Las arquitecturas de *deep learning* son soluciones muy versátiles que se adaptan a una amplia gama de dominios y tipos de datos, fruto de la combinación y diseño a partir de las piezas disponibles en el mapa conceptual de las redes neuronales.

Discount factor

En aprendizaje por refuerzo, es el factor que se corresponde con la importancia que le damos a las recompensas obtenidas en el corto, medio y largo plazo. Tiene un impacto directo en la estrategia que el agente aprende durante su entrenamiento.

Discounted rewards

En aprendizaje por refuerzo, cuando se realiza la estimación de la recompensa esperada, se hace un cálculo de las recompensas desde el final de la trayectoria hasta el principio, en modo inverso a como fueron recolectadas las transiciones. Esta forma de estimación permite crear la recurrencia necesaria de todas las recompensas disponibles en un estado determinado de la ejecución. En este cálculo se tiene en cuenta el *discount factor* que se haya fijado para el experimento.

Exploration-Exploitation trade-off

El equilibrio entre exploración y explotación se refiere a una línea de trabajo dentro del aprendizaje por refuerzo enfocada en la búsqueda de la mejor relación entre ambos términos en la resolución de un problema. Encontrar este punto de equilibrio es un reto importante, ya que la definición de estos procesos está expuesta a la configuración de hiperparámetros y a una componente aleatoria en la ejecución.

Feedback

En aprendizaje por refuerzo, el *feedback* es la información que el entorno devuelve al agente cuando este toma una acción en un momento dado de la ejecución. Normalmente, se corresponde con toda la información disponible (observaciones, variables extra, recompensa, etc.), aunque en algunas ocasiones el *feedback* se puede referir directamente a la recompensa obtenida.

GPU

Del inglés *graphical processing unit*, una GPU es la pieza que se encarga de todo el procesamiento gráfico en un ordenador o servidor. Por su configuración y diseño, que le permite procesar mucha cantidad de datos en forma de matriz, es uno de los elementos hardware por excelencia para entrenamientos de modelos de *deep learning*.

Inferencia

La inferencia es el escenario en el que un modelo de aprendizaje automático está entrenado y listo para usarse. Tiene una relación directa con el concepto de puesta en producción. En aprendizaje por refuerzo, decimos que el agente está en inferencia una vez ha sido entrenado.

Información raw

La información *raw* o en crudo es un concepto muy relacionado con el *big data* y el procesamiento y limpiado de los datos. En cualquier solución basada en datos, siempre es necesario llevar a cabo una fase de procesamiento y normalización que asegure que la información puede ser usada por un modelo de aprendizaje automático. En aprendizaje por refuerzo, este concepto se corresponde con el tipo de datos que obtenemos en una observación, el cual necesita ser preprocesado y adaptado acorde al modelo del agente y al tipo de solución.

Inteligencia Artificial

Rama de las ciencias de la computación enfocada en el desarrollo de soluciones de una manera similar a como lo haría un humano. Es por ello que esta rama se centra en resolver retos cognitivos o complejos para los que, actualmente, no se ha conseguido encontrar una solución con técnicas más tradicionales. Algunos ejemplos de líneas de trabajo dentro de la inteligencia artificial serían sistemas expertos, visión por computador, robótica o heurísticas.

KL-divergence

KL-divergence o la divergencia de Kullback-Leibler es una medida de distancia entre dos distribuciones de probabilidad. La distancia se interpreta en términos de similitud, de ahí que se use como comparación de este tipo de distribuciones. Hay que destacar que la divergencia KL no es simétrica, siendo dependiente de la distribución que se tome como referencia.

Laboratorio y Producción

En inteligencia artificial, cada vez es más común diferenciar las ejecuciones que se realizan durante la fase de desarrollo de aquellas tareas una vez la solución está lista para usarse. Normalmente, la fase de desarrollo se define como laboratorio, ya que se corresponde con una serie de experimentos, validación de hipótesis y analítica de datos. En cambio, una vez la solución está desarrollada se dice que se pone en producción, ya que estará preparada para producir y ofrecer el valor para la que ha sido concebida.

Matriz hessiana

En el ámbito matemático, la matriz hessiana se corresponde con las segundas derivadas parciales de las combinaciones de las variables de una función.

(Meta)heurística

Una metaheurística es un método heurístico usado para resolver problemas de computabilidad, principalmente en situaciones donde no se puede implementar una solución óptima. Por ello, su objetivo es encontrar la mejor solución posible en un tiempo viable.

Método voraz

Un método voraz se puede interpretar como un tipo de heurística a corto plazo. El objetivo de estos algoritmos es tomar la acción que maximiza el objetivo en el siguiente estado, sin tener en cuenta la estrategia a largo plazo.

Minibatch

Al igual que en *deep learning*, un *minibatch* de datos es un conjunto de instancias seleccionadas a partir del conjunto total disponible. La selección de estas instancias depende de la configuración del experimento, pudiendo ser totalmente aleatoria, una selección de los elementos más nuevos, etc.

Proceso Ornstein-Uhlenbeck

Proceso estocástico usado para estimar variaciones en un modelo manteniendo una correlación finita respecto al tiempo. En aprendizaje por refuerzo, se usa en el algoritmo de Deep Deterministic Policy Gradient con el objetivo de añadir ruido a la selección de acciones y así aumentar la probabilidad de exploración.

Robótica

La robótica es una intersección de varias ramas de estudio relacionadas con la ingeniería y la tecnología. Su relación con la inteligencia artificial surge desde los comienzos, muy apoyada en el ideal de dotar a los robots de características y habilidades humanas. Actualmente, tanto el *deep learning* como el aprendizaje por refuerzo tienen cada vez más presencia en soluciones de robótica, justamente por aportar esas características humanas a nivel cognitivo y de funcionalidad.

TPU

Del inglés tensor processing unit, una TPU es una pieza hardware diseñada para procesar estructuras de datos basadas en tensores, o lo que es lo mismo, vectores de datos con más de dos dimensiones. Justo este es el caso que se da en *deep learning*, de ahí que la TPU tenga cada vez más peso en desarrollos y entrenamientos costosos cuando se usan este tipo de técnicas.

Trayectoria

En aprendizaje por refuerzo, la trayectoria se refiere a la secuencia de ejecución durante un número finito de *steps*, es decir, todos los estados, acciones y recompensas que han ido surgiendo a partir de la interacción agente-entorno durante este intervalo finito.

Visión por computador

Es un grupo de retos dentro de la inteligencia artificial relacionados, principalmente, con tareas de clasificación, segmentación o detección en imágenes. En los últimos años, este campo se ha visto impactado positivamente por las soluciones basadas en técnicas de *deep learning*, en concreto las redes convolucionales.

Worker

En aprendizaje por refuerzo, un *worker* será todo agente que se esté ejecutando en paralelo a otro grupo de agentes, ya sea con el objetivo de entrenar un modelo global o enfocado en la recolección de experiencia para el proceso de entrenamiento.



Enlaces de interés

Spinning-up RL

Repositorio educativo de contenidos relacionados con Deep Reinforcement Learning, liderado por la empresa de inteligencia artificial OpenAI. Su contenido es muy variado, incluyendo artículos científicos, tutoriales y ejemplos de implementación de soluciones entre otros.

<https://spinningup.openai.com/>

Papers with Code

Agregador de artículos sobre aprendizaje automático. Para cada artículo, ofrece explicaciones, estadísticas y enlaces a repositorios de código, con ejemplos de implementación del artículo en cuestión. Además, indexa y categoriza los conjuntos de datos utilizados en los experimentos mostrados en los artículos.

<https://paperswithcode.com/>

Google DeepMind y OpenAI

Dos de las empresas referencia dentro del aprendizaje por refuerzo y de soluciones a retos de inteligencia artificial general. En ambos enlaces se pueden encontrar multitud de recursos y proyectos que forman parte de la base de este manual y del estado del arte de la inteligencia artificial actual.

<https://deepmind.com/>

<https://openai.com/projects/>

AlphaGo: The movie

Documental sobre el desarrollo de la solución AlphaGo por parte de Google DeepMind.

https://deepmind.com/research/case-studies/alphago-the-story-so-far#alphago_the_movie

Bibliografía



Libro completo:

Barto, A. y Sutton, R. (1992). *Reinforcement Learning: An Introduction*. The MIT Press.

GoodFellow et al. (2016). *Deep Learning book*. The MIT Press.

Artículos de revistas en línea:

Arulkumaran, K. et al. (2017). A Brief Survey of Deep Reinforcement Learning. *IEEE Signal processing magazine-Arxiv e-print*.

Clemente, A. et al. (2017). Efficient Parallel Methods for Deep Reinforcement Learning. *Cornel University-Arxiv e-print*.

Kiran, B. R. et al. (2021). Deep Reinforcement Learning for Autonomous Driving. *Cornel University-Arxiv e-print*.

Konda, V. R. (2000). Actor-Critic algorithms. *NIPS conference*.

Lample, G. et al. (2017). Playing FPS games with Deep Reinforcement Learning. *AAAI*.

Lepenioti, K. et al. (2020). Prescriptive analytics: Literature review and research challenges. *International Journal of Information Management, ScienceDirect*.

Lillicrap, T. et al. (2016). Continuous Control with Deep Reinforcement Learning. *ICLR*.

Mnih, V. et al. (2016). Asynchronous methods for Deep Reinforcement Learning. *Cornell University-Arxiv e-print*.

Mnih, V., Kavukcuoglu, K. et al. (2015). Human Level Control Through Deep Reinforcement Learning. *Nature*.

Nguyen, T. T. . et al. (2019). Deep Reinforcement Learning for Multi-Agent Systems: A review of Challenges, Solutions and Applications. *Cornel University-Arxiv e-print*.

OpenAI, Andrychowicz, M. et al. (2018). Learning Dexterous In-Hand Manipulation. *Cornel University-Arxiv e-print*.

Schulman, J. et al. (2016). High-dimensional continuous control using generalized advantage estimation. *Cornell University-Arxiv e-print*.

Schulman, J. et al. (2017). Trust region policy optimization. *Cornel University-Arxiv e-print*.

Schulman, J. et al. (2017). Proximal Policy Optimization algorithms. *Cornell University-Arxiv e-print*.

Silver, D. et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*.

Silver, D. et al. (2017). Mastering the game of Go without human knowledge. *Nature*.

Sutton, R. et al. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. *NIPS Conference*.

Watkins, C. J. C. H. y Dayan, P. (1992). Q-learning. *Mach Learn* 8, 279–292.



Autor

Adrián Colomer Graner
Gabriel Enrique Muñoz Ríos