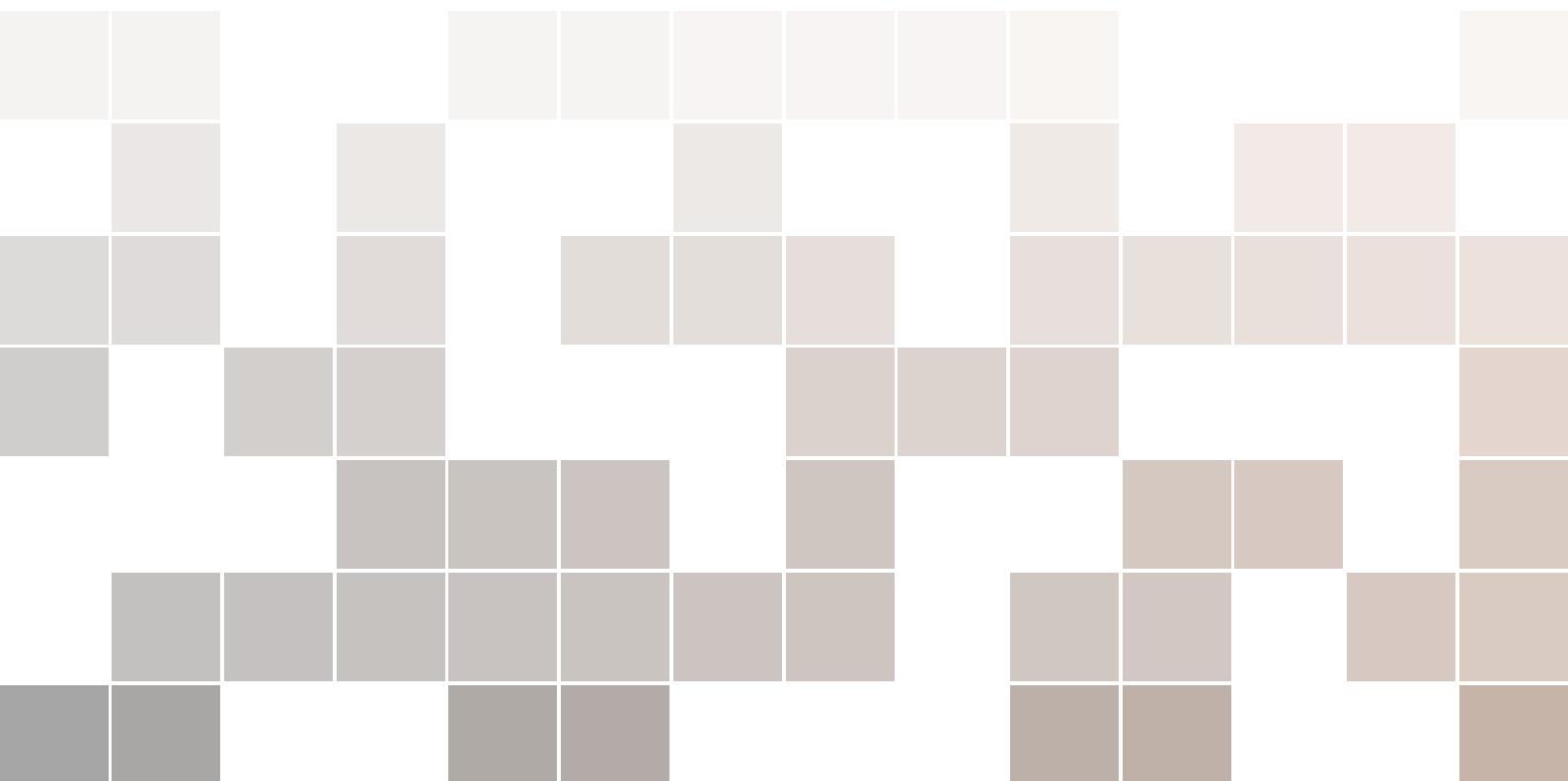


Aprendizaje profundo por refuerzo para la resolución de problemas.

Técnicas actuales

Alejandro Campoy Nieves

Tutor: Juan Gómez Romero



Copyright © 2020 Alejandro Campoy

Índice general

0.1	Resumen	9
-----	---------	---

I Introducción

1	Introducción	13
1.1	Aprendizaje por refuerzo profundo en la actualidad	16
1.2	Objetivos	16
1.3	Planificación	17

II Descripción de Tecnologías

2	Redes Neuronales	23
2.1	Funciones de activación (h)	24
2.2	Funciones de salida (h')	26
2.3	Funciones de coste	26
2.4	Algoritmos de optimización	27
3	Aprendizaje por refuerzo(RL)	31
3.1	Componentes del RL	32
3.1.1	Ciclo de interacción	33
3.2	Procesos de Decisión de Markov(MDPs)	34
3.2.1	Función de transición	34
3.2.2	Función de recompensa	35
3.2.3	Descuento	35

3.3	Política	37
3.3.1	Función de estado-valor	37
3.3.2	Función de acción-valor	38
3.3.3	Ecuación de Bellman	38
3.3.4	Función de acción-ventaja	39
3.3.5	Evaluando una política	39
3.3.6	Mejorando una política	40
3.4	Exploración vs Explotación	41
3.5	Aprendiendo a evaluar y mejorar políticas	42
3.5.1	Método de Montecarlo	43
3.5.2	Métodos de Diferencia Temporal(SARSA)	45
3.5.3	Q-Learning	48
4	Aprendizaje por Refuerzo Profundo(DRL)	51
4.1	Métodos Basados en Valor: Deep Q-Network(DQN)	51
4.2	Métodos basados en política: REINFORCE	55
4.3	Métodos actor-critic	57

III

Desarrollo y Experimentación

5	OpenAI baselines	63
6	Entornos Open AI Gym	65
6.1	MountainCar-v0	66
6.2	MountainCarContinuous-v0	68
6.3	Entorno más complejo (aun sin decidir)	68
7	Entorno de desarrollo	69
7.1	Metodología de experimentación	70
8	Experimentación: Resultados Obtenidos	71
8.1	MountainCar-v0	71
8.1.1	DQN	71
8.1.2	PPO2	80
8.1.3	DDPG	82
8.2	Otro entorno más complejo (Aún sin decidir)	87
8.2.1	DQN	87
8.2.2	PPO2	87
8.2.3	DDPG	87

IV

Estado del arte

9	AlphaGO	93
10	AlphaStar	95

11	Conclusiones finales	99
11.1	Posibles mejoras	99
11.2	Conclusiones	99
	Bibliografía	101
	Artículos	101
	Libros	104

A	Configuración en la nube	107
A.1	Configuración de la Infraestructura	107
A.2	Configuración de la Máquina	112
B	Uso de la Máquina Virtual	117
B.1	Archivos generados en los logs	119
B.2	Visualizar la información de los logs	121
C	Stable Baselines	123
C.1	Generar un Modelo	123
C.2	Cargar un modelo	126

Índice de figuras

1.1	Categorización de tipos de aprendizajes en Machine Learning.	14
1.2	Estructura de conceptos dentro de la Inteligencia Artificial.	15
2.1	Esquema simplificado de una red neuronal. [1]	23
2.2	Función sigmoide en plano bidimensional. Extraída de origen [11].	25
2.3	Función ReLU en plano bidimensional. Extraída de origen [11].	26
2.4	Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de origen [8].	28
3.1	Esquema de componentes del RL. Imagen extraída y posteriormente modificada de origen [48].	32
3.2	Ilustración gráfica del factor de descuento sobre la recompensa. Extraído de origen. [25]	36
3.3	Esquema Método Montecarlo. Tabla función extraída de origen. [40]	44
3.4	Esquema Método de diferencia temporal(SARSA). Tabla función extraída de origen. [40]	47
3.5	Esquema Método Q-Learning. Tabla función extraída de origen. [40]	49
4.1	Esquema red neuronal, acciones discretas finitas. Imagen extraída de origen. [40]	52
4.2	Esquema red neuronal, acciones continuas infinitas. Imagen extraída de origen. [40]	53
4.3	Esquema Método Deep Q-Learning. Extraída de origen. [3] [40]	55
4.4	Metodología basada en políticas. Extraída de origen. [40]	56
4.5	Esquema de metodología actor-critic. Extraída de origen. [40]	57
6.1	Agente entrenando conducción autónoma en entorno simulado de GTA V. Extraída de vídeo en Youtube. [38]	66
6.2	Representación gráfica del problema MountainCar v0 de OpenAI Gym.	67
7.1	Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada	
70		

8.1	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i>	72
8.2	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 8.1). Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i>	73
8.3	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i>	74
8.4	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i>	75
8.5	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 8.4). Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i>	75
8.6	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i>	76
8.7	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i>	76
8.8	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 8.7). Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i>	77
8.9	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i>	77
8.10	Porcentaje de tiempo empleado en explorar para algoritmo DQN en <i>MountainCar-v0</i>	79
8.11	Resumen general de progreso de algoritmo DQN en <i>MountainCar-v0</i>	80
8.12	Recompensa obtenida con el algoritmo PPO2 en <i>MountainCar-v0</i> , independientemente de la semilla iteraciones y otros parámetros.	81
8.13	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 3 y en <i>MountainCar-v0</i>	83
8.14	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3 y en <i>MountainCar-v0</i>	83
8.15	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 13 y en <i>MountainCar-v0</i>	84
8.16	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 13 y en <i>MountainCar-v0</i>	84
8.17	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 46 y en <i>MountainCar-v0</i>	85
8.18	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 46 y en <i>MountainCar-v0</i>	85
8.19	Gráfica general obtenida durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3, 13 y 46 simultáneamente y en <i>MountainCar-v0</i>	86
8.20	Gráfica comparativa de los tres algoritmos: DQN, PPO2 y DDPG en <i>MountainCar-v0</i>	87
A.1	Iniciando asistente para creación de máquina virtual.	108
A.2	Creando una máquina virtual.	108
A.3	Configurando la máquina virtual para el proyecto.	109
A.4	Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [13]	110
A.5	Indicando Sistema operativo y memoria SSD a la máquina virtual.	111
A.6	Comprobando que la máquina virtual ha sido creada con éxito.	112
A.7	Entrando en la máquina virtual.	112
A.8	Muestra de uso de GPU al entrenar con los baselines.	115
A.9	Pasando el script a la máquina virtual para abastecerla.	116
B.1	Probando agente entrenado en entorno simulado <i>MountainCar-v0</i>	119

0.1 Resumen

El Aprendizaje por Refuerzo o, en inglés, Reinforcement Learning (RL) es un área dentro de la Inteligencia Artificial y más concretamente del Aprendizaje Automático que estudia la forma en la que un agente puede resolver una tarea mediante una experimentación repetitiva y asignación de recompensas a esas tareas que realiza. Es el concepto de “ensayo y error” que los humanos utilizamos para aprender en muchos de los problemas que nos plantea la vida en nuestro día a día.

Recientes estudios han mostrado que los avances en Aprendizaje Profundo o Deep Learning (DL) han dado lugar a redes neuronales capaces de estimar esas recompensas en función de las decisiones tomadas y optimizar las acciones del agente. A esto se le llama Aprendizaje Profundo por Refuerzo o Deep Reinforcement Learning (DRL).

Estas técnicas, relativamente nuevas, han demostrado ser extraordinariamente efectivas, superando incluso a la inteligencia humana en muchos ámbitos; por ejemplo, en la resolución de juegos como Go, con el agente AlphaGo, o en el StarCraft II, con el agente AlphaStar entre muchos otros.

En este trabajo se describirán los fundamentos del DRL, se estudiarán los aspectos prácticos de implementación para la resolución de problemas de videojuegos, se construirán e ilustrarán algunos agentes y, finalmente, se tratará de dar una visión más clara del horizonte de esta tecnología. En otras palabras, hasta donde han sido capaces de llegar las organizaciones más punteras del mundo en este sector y qué se podría esperar de ellas en el futuro.



Introducción

1	Introducción	13
1.1	Aprendizaje por refuerzo profundo en la actualidad	
1.2	Objetivos	
1.3	Planificación	



1. Introducción

La **Inteligencia Artificial** (IA) es una de las ramas o disciplinas de la informática relativamente más jóvenes que existen a día de hoy, nació en la década de 1960. Es el concepto más abstracto, genérico y amplio que podemos encontrar; se trata de una combinación de técnicas que nos permiten emular el comportamiento y capacidades de los seres humanos para resolver problemas de cualquier tipo. Incluyendo en este concepto cosas como planificación, reconocimiento de objetos, sonidos, hablar, traducir, etc.^[18]

Por tanto, la Inteligencia Artificial aborda todo lo que se encuentra dentro de “imitar” el razonamiento y capacidades cognitivas de un ser humano. Cuando hacemos uso de algoritmos y técnicas matemáticas para poder llevar a cabo ésto, de tal manera que el programa es capaz de auto-perfeccionarse a medida que obtiene más información, estamos hablando de técnicas de **Machine Learning** dentro de la propia IA. Son, pues, técnicas matemáticas para construir algoritmos de decisión ante problemas determinados y que son capaces de mejorarse a sí mismos de forma autónoma. Estas técnicas se implementan en computadores, normalmente de potencia considerable, para que sean capaces de decidir acciones dentro de un entorno determinado con la finalidad de cumplir un objetivo preestablecido.

En el caso de un videojuego, este objetivo podría ser pasarse un nivel, por ejemplo. En el caso de un juego competitivo, sería ganar al rival o rivales. En el caso de que el problema sea de clasificación, clasificar todos los casos que aparezcan correctamente, etc.

Dentro del Machine Learning se pueden encontrar una gran cantidad de técnicas más específicas tales como Árboles de Decisión, Support Vector Machine (SVM), Clustering, regresión Logística, etc. Estas técnicas pueden estar clasificadas principalmente en tres grandes grupos o paradigmas: [19]

- **Aprendizaje supervisado:** Basado en un set de ejemplos que han sido etiquetados

previamente con la respuesta o acción que debería dar el modelo para considerar que lo ha hecho correctamente. Por tanto, necesitan de la atención de los humanos para poder entrenarse, ya que los datos con los que entrena deben incluir la etiqueta que indica cuando aciertan o se equivocan conforme están aprendiendo, siendo éstas predefinidas.

- **Aprendizaje no supervisado:** La idea es que estos algoritmos de aprendizaje sean capaces de mejorar solo a partir de la entrada, realizando una búsqueda de patrones o estructuras dentro de los datos con los que debe tomar decisiones, sin necesidad de etiquetas que definen como de buenas son las salidas que brinda.
- **Aprendizaje por refuerzo:** Es similar al aprendizaje no supervisado, en el sentido de que no necesita de la supervisión de un ser humano en las decisiones que va tomando. Sin embargo, en lugar de analizar los datos de entrada en búsqueda de un orden, se centra en mejorar las recompensas que el entorno le devuelve con las decisiones que toma. Sería, por ejemplo, la forma en la que un humano aprende a montar en bici. En caso de que se caiga, sabría detectar qué acciones han hecho que falle en su objetivo de ir de un punto a otro y acabe cayendo. En caso de que llegue al lugar esperado, se reforzaría las acciones que haya considerado buenas para poder cumplir ese objetivo de la forma que lo ha conseguido. Por tanto, esto requiere una experimentación del agente sobre qué cosas están bien hacerlas y qué cosas no en momentos determinados.

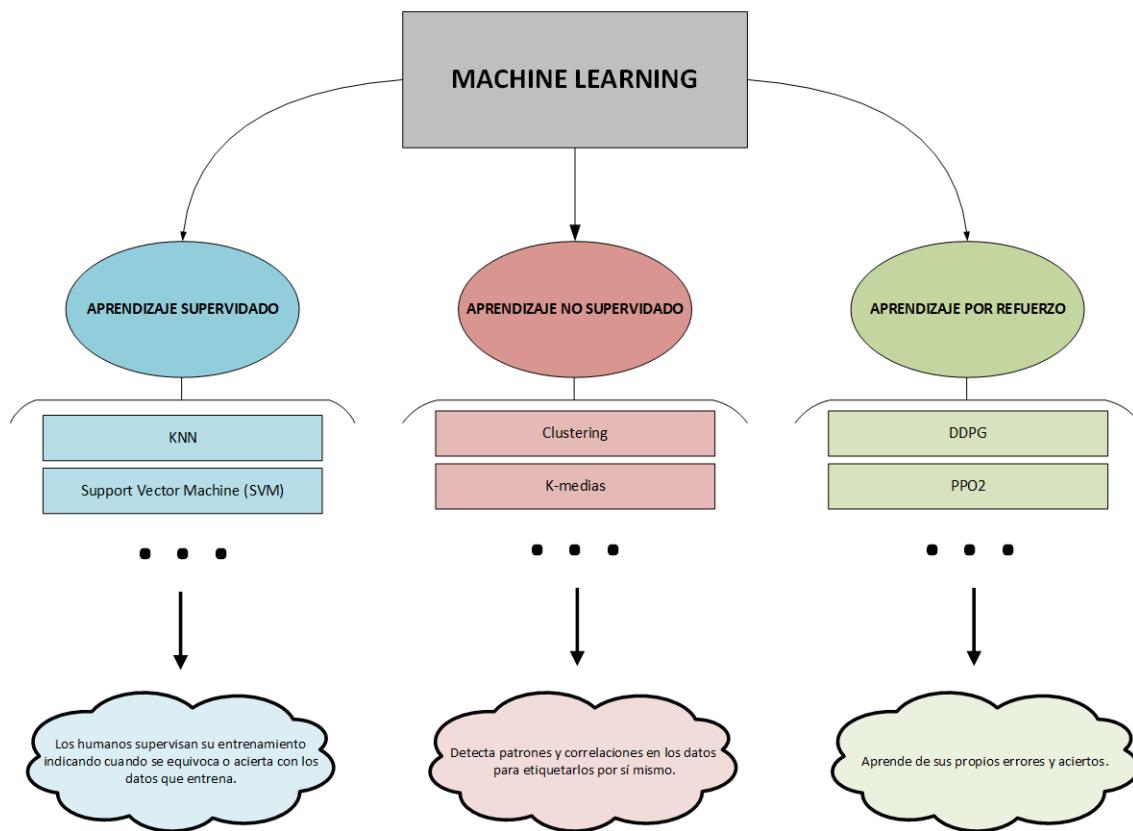


Figura 1.1: Categorización de tipos de aprendizajes en Machine Learning.

Dentro del Machine Learning y todo lo que esto abarca, existe un conjunto de técnicas concretas que hacen uso de **redes neuronales artificiales** que se compone de un número de niveles jerárquicos, es lo que se denomina **Deep Learning**. Hay que decir que, aunque se haga uso de ese nombre, aun desconocemos la forma de trabajar exacta del cerebro humano.

Estas técnicas son aproximaciones o ideas generales de como se cree que funciona nuestro desarrollo del razonamiento. Para que se entienda con un ejemplo, aprendimos y nos inspiramos a volar observando los pájaros, aunque los aviones no utilicen exactamente las mismas técnicas que ellos.

Estas redes neuronales son adecuadas en determinados problemas en los que no necesitamos saber **cómo piensa el agente**, simplemente queremos conseguir la **mejor respuesta** posible, sin más. Son usados cuando se dispone de mucha fuerza de computación y datos no estructurados (reconocimiento de imágenes, por ejemplo). Las redes neuronales pueden funcionar tanto en *aprendizaje supervisado* como en *aprendizaje por refuerzo*. Podemos indicarle los ejemplos con los que se quiere que aprenda y las salidas que debería de dar, o bien permitirle que explore el mismo y evalúe sus propias decisiones una vez obtiene recompensas negativas o positivas como fruto de las mismas, esto dependerá de la naturaleza del problema a resolver y de los datos que deba manejar.^[26]

Cuando se usa Deep Learning dentro del paradigma de aprendizaje por refuerzo, es llamado **Aprendizaje por refuerzo profundo** y es una combinación que ha sido demostrada **muy eficaz** y potente para la resolución de algunos problemas tales como juegos de mesa, videojuegos, etc.

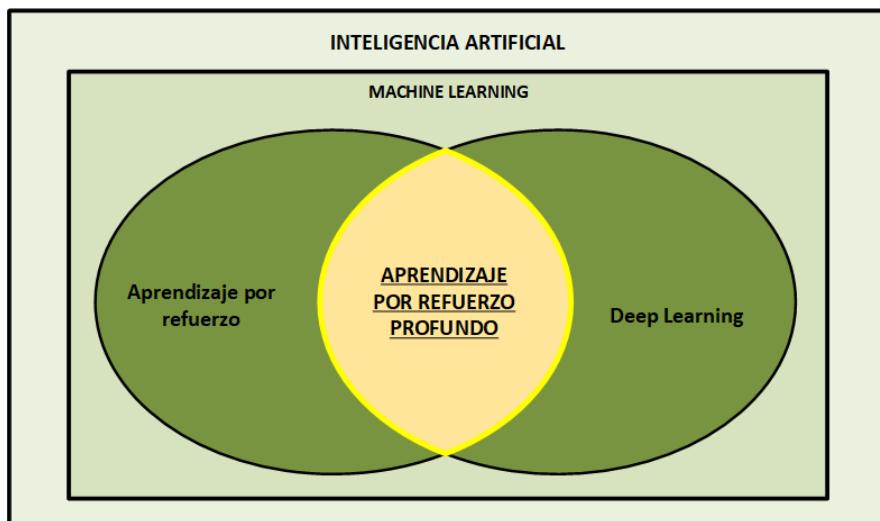


Figura 1.2: Estructura de conceptos dentro de la Inteligencia Artificial.

Tanto mi tutor, Juan Gómez Romero, como yo, hemos decidido investigar de una forma más exhaustiva sobre todo lo que significa el **Aprendizaje por refuerzo profundo**. Tanto en el Grado de Ingeniería Informática como en el Máster, la información que se aportaba sobre este tema era simplemente un par de frases dando su definición. Nunca hemos abordado o resuelto, tanto de forma teórica como práctica, problemas en los que la mejor metodología que se puede usar dentro del Machine Learning sea esa. Como sí ha ocurrido con AlphaGo o AlphaStar, los cuales serán analizados en este trabajo. Personalmente, me

resulta muy interesante a nivel académico debido al total desconocimiento desde el que parto como estudiante y sobre el cual me gustaría avanzar y ver las posibilidades que ofrece.

1.1 Aprendizaje por refuerzo profundo en la actualidad

El aprendizaje por refuerzo profundo ha sido demostrado como la mejor tecnología actual para resolver ciertos problemas o, de forma más concreta, algunos juegos y videojuegos competitivos. Son el caso de **AlphaGo** y **AlphaStar**, elaborados por la empresa **DeepMind** de Google.^[15]

En el caso de AlphaGo, fue capaz de ganar a uno de los mejores, sino el mejor, jugador de GO del mundo actualmente llamado Lee Sedol. Consiguió ganar 4 partidas frente a 1 del campeón mundial.^{[42] [28]}

En el caso de AlphaStar, supuso un paso más allá en el avance de este campo, creando un agente capaz de colarse entre el 0,2% de los mejores jugadores del mundo. Se pasó de un problema con tiempo discretizado (por turnos) en el que se conocía en todo momento el entorno completo (conjunto de fichas en juego), a un videojuego en tiempo real (inexistencia de turnos) en el que el jugador solo tienen como entrada de información una porción del entorno (es como ver solo una parte del tablero). Eso sin contar con la complejidad de la información de entrada como conjunto de posibles acciones que se pueden realizar en cada momento.^{[47] [24]}

En este Trabajo fin de Máster se va a tratar de arrojar algo de luz a este paradigma, prácticamente desconocido a nivel académico. Como se trata de un conjunto de técnicas punteras que aún se están investigando, desarrollando y perfeccionando, este trabajo se va a centrar en explicar de una forma básica las técnicas más comunes que usan empresas como DeepMind, aplicándolas a entornos o problemas más simples para poder experimentar con las mismas.¹

Una vez hecho esto, vamos a enfocarlo y a tratar de explicar de una forma algo más abstracta cómo funcionan las arquitecturas basadas en aprendizaje por refuerzo profundo de estos agentes que suponen la actual cumbre de este tipo de tecnologías.

1.2 Objetivos

Los objetivos principales que se van a tratar de abordar en este trabajo son:

1. Entender los **conceptos principales** de la Inteligencia Artificial, concretamente todo lo relacionado con el aprendizaje por refuerzo profundo.
2. Entender, de una forma clara, el **funcionamiento y metodología** que siguen las **técnicas** más importantes de este campo.
3. Descubrir las posibilidades que nos da **gym** ^[30] para la construcción de **entornos o simulaciones** en los que entrenar a nuestros agentes sin la necesidad de espacios

¹Hay que tener en cuenta que estas empresas invierten muchos recursos y tiempo a estas investigaciones, por lo que no se puede abarcar de una forma tan ambiciosa de manera académica, por desgracia.

físicos reales para ciertos problemas.

4. Saber aprovechar los recursos en la nube para una mayor eficiencia en los entrenamientos de los agentes, concretamente en la plataforma de **Google Cloud**.
5. Saber aplicar las diferentes técnicas en los distintos entornos mencionados anteriormente y saber ajustarlos al máximo.
6. Monitorizar el proceso de aprendizaje y la experiencia obtenida por el agente a medida que realiza intentos en el entorno con la finalidad de validar su progreso y determinar **cuando el agente está mejorando a partir de sus aciertos y errores**.
7. Analizar, desde el punto de vista científico y técnico, problemas considerados ya resueltos gracias al paradigma de aprendizaje profundo por refuerzo tales como **AlphaGo** y **AlphaStar**.
8. Tener una visión básica y general del progreso actual en este campo y a lo que podría llegar en el futuro.
9. Reflexionar sobre las conclusiones que se pueden extraer del cumplimiento de los objetivos anteriores y considerar posibles mejoras futuras del trabajo realizado durante este proyecto.

1.3 Planificación

Descripción de Tecnologías

2	Redes Neuronales	23
2.1	Funciones de activación (h)	
2.2	Funciones de salida (h')	
2.3	Funciones de coste	
2.4	Algoritmos de optimización	
3	Aprendizaje por refuerzo(RL)	31
3.1	Componentes del RL	
3.2	Procesos de Decisión de Markov(MDPs)	
3.3	Política	
3.4	Exploración vs Explotación	
3.5	Aprendiendo a evaluar y mejorar políticas	
4	Aprendizaje por Refuerzo Profundo(DRL)	51
4.1	Métodos Basados en Valor: Deep Q-Network(DQN)	
4.2	Métodos basados en política: REINFORCE	
4.3	Métodos actor-critic	

Introducción

En el Grado de Ingeniería Informática de la Universidad de Granada, más concretamente en la especialidad de Computación y Sistemas Inteligentes, así como en el Máster profesionalizante, hemos podido ver y experimentar con las redes neuronales profundas y ver las posibilidades que estas ofrecen.

Del mismo modo, hemos podido comprobar que llega un momento en el que estos tipos de agentes se “**estancan**” en su aprendizaje cuando no alcanzan a resolverlos a la perfección, de tal manera que por más información etiquetada que se le aporte no consigue aprender nada nuevo o, incluso se **sobreajustan**. Quizás pueden ser mejorados un poco más cambiando la arquitectura de las capas neuronales que utiliza, pero si el modelo ya está bastante bien ajustado se traduce en mucho tiempo de pruebas y cambios para muy poca recompensa en los resultados, si es que se consiguen.

En definitiva, esta tecnología para ciertos problemas puede quedarse algo corta o no conseguir aprender lo suficiente como para encontrarse satisfecho con los resultados, como sería el caso de jugar bien a **Go**. Aunque ya hablaremos de ese tema.[\[21\]](#)

Se puede entender mejor con el siguiente ejemplo, el cual encuentro muy apropiado ya que la Inteligencia Artificial trata de imitar la metodología del aprendizaje humano. Si queremos ser los mejores jugando al Ajedrez, está bien comenzar a leer libros y analizar partidas de jugadores profesionales; aprender patrones y técnicas comunes así como una noción básica del juego. Esto sería equivalente a las redes neuronales con información supervisada.

Sin embargo, los humanos llegan a un punto en el que por más libros que lean o partidas que observan, no son capaces de mejorar más. ¿Por qué ocurre esto? Esos jugadores profesionales que escriben esos libros tienen un conocimiento interno fruto de la **experiencia** que no es fácil aclarar o expresar en el papel con lenguaje natural(o un dataset, para la red neuronal), es un plus que se adquiere solamente con la **práctica**.

Por ello, los humanos que a parte de “estudiar” (aprendizaje supervisado), practican y experimentan posibles situaciones de juego (aprendizaje por refuerzo), pueden llegar a conseguir ese conocimiento extra que los hace únicos y posiblemente mejores que el resto de sus adversarios.

Esta es la filosofía y el origen del nacimiento del aprendizaje por refuerzo profundo. Haciendo uso de las redes neuronales para manejar esa experiencia que el agente adquiere a medida que acierta o erra en sus intentos, el agente será capaz de aprender tanto de los aciertos como de los errores que comete e ir actualizándose a versiones mejores de sí mismo de forma autónoma.

En esta parte estudiaremos de una forma **abstracta y teórica** tanto las redes neuronales como los tipos de algoritmos de aprendizaje por refuerzo que utilizaremos para resolver los distintos problemas que se plantearán en el futuro a lo largo de este trabajo.

De esta forma iremos adquiriendo los conocimientos y conceptos básicos que necesitaremos para luego entender cómo funcionan en la práctica y cuáles son las causas de los buenos y malos resultados que vayamos obteniendo.

2. Redes Neuronales

Como hemos mencionado anteriormente, estas técnicas se apoyan firmemente en el uso de las redes neuronales. Son técnicas que ya conocemos del Grado y el Máster respectivamente, aunque considero que no está mal repasarlo de forma resumida. [23] [39] [7] [41] [40]

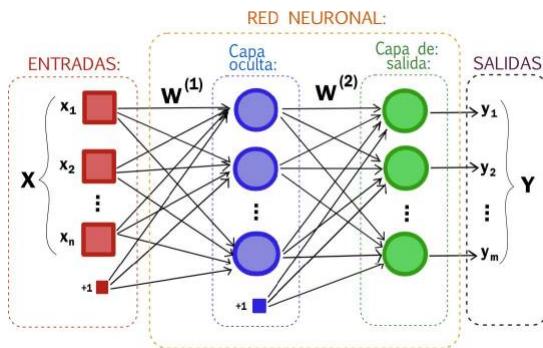


Figura 2.1: Esquema simplificado de una red neuronal. [1]

Las redes neuronales están formadas por un conjunto de nodos (neuronas), que están conectadas entre sí, tal y como podemos ver en la figura 2.1, por ejemplo.

Estos nodos se encuentran organizados por **capas**. La **primera capa** está asociada a la entrada o *input*, la **última capa** a la salida que devuelve la red y el resto de capas que se encuentran entre la primera y la última son conocidas como **capas ocultas o intermedias**.

La entrada es un **vector de características** del problema que queremos resolver. Por ejemplo, si estamos intentando decidir el siguiente movimiento a realizar en un tablero de ajedrez, el vector de características estaría formado por elementos que indicasen la pieza que hay en cada casilla respectivamente (o indicar que no hay ninguna). Es un ejemplo, ya que no sería la única manera de describir el tablero para una red neuronal. Por tanto, la

capa de entrada tendrá tantas neuronas como elementos el vector de características; cada neurona recibe una de las características consideradas.

Las capas ocultas, pudiendo ser una o más capas, no tienen una conexión directa con el entorno como hemos visto anteriormente. En su lugar, la entrada que recibe es la salida de la capa anterior(capa de entrada en este caso). La salida que devuelvan los nodos de esta capa serán utilizados como entrada en la capa siguiente. ¿Para que sirve esto? La respuesta resumida es que si solo tuviéramos la capa de entrada y salida, las entradas serían demasiado independientes entre sí a la hora de influir en la salida seleccionada entre todas las posibles. En general, los problemas del mundo real son mucho más complejos que eso y es necesario que la red sea capaz de detectar patrones e interdependencias entre los distintos valores de las entradas. Estas capas ocultas ayudan a dar esa **riqueza** al proceso.[11]

La capa de salida recoge los datos de la última capa oculta y los procesa para dar una respuesta definitiva en la red. La forma en la que da esta salida depende de diversos factores de la propia capa que veremos un poco más adelante.

Las conexiones de los nodos de una capa a otra son definidos por los **pesos**(w_i). Estos pesos son utilizados por los nodos para dar la salida siguiente, tal y como se ve en la figura 2.1.

2.1 Funciones de activación (h)

Cada neurona no transmite la entrada que recibe a las siguientes de forma directa. Antes de hacer esto, procesa dicha entrada. Esto se realiza mediante la **función de activación**. Estas pueden ser de muchos tipos y determina cuando la neurona se **excita o no** en función de las entradas que recibe junto con su ponderación (ya que su entrada se compone de las salidas de las neuronas de la capa anterior)[11].

Se han propuesto muchos tipos de funciones de activación en este campo, vamos a mencionar las más habituales o las que más hemos utilizado durante la carrera y el Máster:

- **Sigmoide:** Las neuronas sigmoide se comportan sumando todas las entradas ponderadas, es decir, aplicándoles sus respectivos pesos, para luego utilizar ese único valor (z) de la siguiente forma:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Se hace de esta manera para que dicha función tenga la siguiente forma, si la dibujamos en un plano bidimensional:

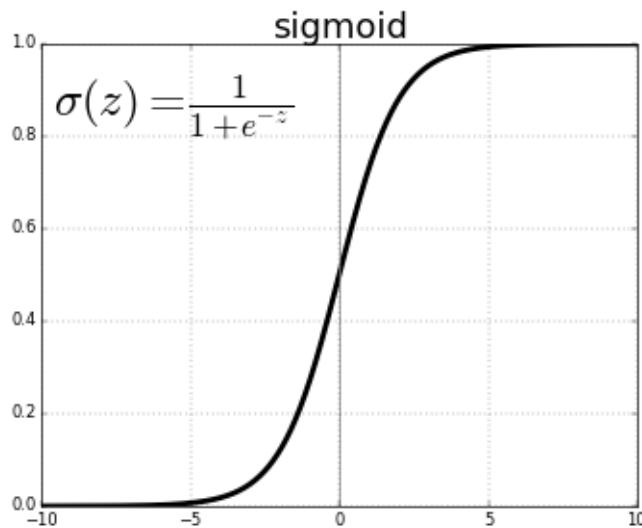


Figura 2.2: Función sigmoide en plano bidimensional. Extraída de origen [11].

En la figura 2.2, vemos como la salida se acota entre los valores 0 y 1 independientemente de las entradas que pueda recibir. El valor neutro de la entrada z (valor 0), corresponde con un valor intermedio de la salida(valor 0.5). A medida que la entrada es positiva y más alta, la salida se acerca al valor 1, mientras que cuando la entrada es negativa y menor, la salida se acerca al valor 0.

Esto hace que no haya tanta diferencia entre un conjunto de entradas muy altas a otras que no lo son tanto, por ejemplo, ya que en cualquier caso al ser valores positivos la neurona se va a excitar y devolver un 1. Lo mismo ocurre para las entradas negativas y la salida 0.

- **Rectified linear unit(ReLU):** Esta función es actualmente más usada que la sigmoide. Principalmente se debe a la existencia de más capas en las redes neuronales. Los pesos de las neuronas sigmoide son mucho más difíciles de actualizar debido al **problema de desaparición del gradiente**, aunque no vamos a entrar en detalles con eso. La función es:

$$R(z) = \max(0, z)$$

Esta función es más sencilla de entender incluso que la anterior. Básicamente si el valor de la sumas ponderadas de las entradas es menor que 0, se devuelve el valor 0. Si esa suma ponderada de entrada es positiva, se devuelve tal cual:

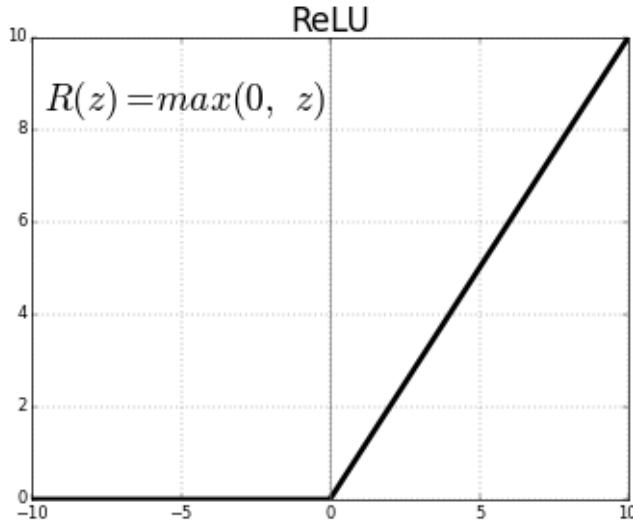


Figura 2.3: Función ReLU en plano bidimensional. Extraída de origen [11].

En general, se utilizan estos tipos de funciones no lineales debido a que permiten que las redes detecten esas relaciones no lineales entre entradas y salidas. Cosa que es muy frecuente a la hora de resolver todo tipo de problemas en nuestro día a día (clasificación de dígitos manuscritos, por ejemplo). La mayoría de problemas que podemos resolver con estas técnicas no se van a ajustar bien a funciones lineales, hay que tenerlo siempre en cuenta.

2.2 Funciones de salida (h')

Las funciones de salida son utilizadas en la **última capa de la red neuronal**. Tratan de utilizar los valores que han llegado a este punto de la red y transformarlos en una **respuesta final** al problema que está tratando de resolver. La más frecuente y una de las más utilizadas es la **función exponencial normalizada o softmax**. Suele ser utilizada en problemas de clasificación con salidas discretas, ya que podemos obtener una **distribución de probabilidades** entre las posibles salidas:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

Para j con valores entre 1 y K . La idea, para este tipo de problemas con un número limitado de salidas posibles, es que la última capa de la red neuronal tenga tantos nodos o neuronas como posibles salidas. Cada neurona representaría una posible respuesta de la red y el valor que devuelve cada una de ellas sería la probabilidad de que la respuesta que representa sea la correcta.

La potencia de esto es que la red neuronal no solo aporta una salida (aquella neurona con valor más alto), sino que indica una distribución de probabilidades de todas ellas.

2.3 Funciones de coste

Es muy posible, y sobretodo durante el proceso de aprendizaje, de que la red neuronal se **equivoque** al dar una salida para una entrada determinada. Hay que buscar un

modo de comparar la salida dada por la red de la salida real o salida que debería haber dado.

Las **funciones de coste o de pérdida** son utilizadas una vez la red neuronal da una salida definitiva. Trata de evaluar o, mejor dicho, cuantificar el error que ha tenido. Hay muchas formas de analizar o evaluar este error, cada una con sus pros y contras, vamos a ver un par de ellas:

- **Error cuadrático medio(MSE):** Mide el promedio de los errores al cuadrado. En otras palabras, la diferencia de el valor que debería estimar de lo que estima realmente:

$$MSE = \frac{1}{n} \sum_n^{j=1} (d_j - y_j)^2$$

Siendo n el número de salidas que ha dado, d las predicciones e y las salidas correctas. Esta función tiene el inconveniente de que los errores más altos afectan mucho al promedio.

- **Entropía cruzada:** Es un cálculo que consiste en la suma negativa del producto del logaritmo de cada componente de las salida predicha y el componente de las salida real que debería dar:

$$H(p, q) = - \sum_x p(x) \log(q(x))$$

El objetivo de cualquier red neuronal debe ser el de **minimizar** esta función de pérdida, ya que es sinónimo de una mayor frecuencia de acierto en las salidas que aporta. Dicho de otra manera, el objetivo es buscar la combinación de pesos en la red que hagan que las salidas en la función de error sea la mínima posible de forma general para cualquier caso que se le plantee.

Decidir una función de pérdida es **muy importante** a la hora de entrenar la red en un futuro, ya que dependiendo de su diseño puede hacer más difícil su optimización o menos (por tanto, que lleguemos a un conjunto de pesos determinado para la red u otros diferentes). Dependiendo de la naturaleza del problema que queramos resolver y de cómo se interpreta la entrada para la red.

Esta es una de las tareas del programador, no solo saber un lenguaje de programación, sino decidir y probar aquellas cosas en las que tienen un motivo para creer que pueden funcionar mejor que otras.

2.4 Algoritmos de optimización

Hasta ahora, todo lo que se ha explicado de las redes neuronales es diseñado y establecido de antemano (las capas, número de neuronas por capa, las funciones de activación y salida que van a tener, función de coste, etc). Solo hay una cosa que va a variar en dichas redes neuronales a lo largo de su aprendizaje: la **actualización de sus pesos** en las conexiones entre sus respectivas capas.

Los **algoritmos de optimización** buscan concretamente esto. Usando los valores de error que devuelve la función de perdida, determina como se van a actualizar los pesos con la finalidad de que la red neuronal mejore en sus cálculos de salidas futuras. De nuevo, hay

muchos algoritmos para realizar esta tarea, algunos de ellos son:

- **Gradiente descendente Estocástico:** Es una técnica genérica que sirve para minimizar cualquier función. Con el uso de derivadas, es capaz de encontrar la combinación de pesos óptima o muy buena para devolver el mejor punto de esa función.

El principal problema aquí es que estamos hablando de redes neuronales que pueden tener cientos de miles de parámetros, por lo que no es viable hacer este tipo de cálculos a lo largo del tiempo.

Por ello, esta técnica se diferencia del gradiente descendente tradicional en que utiliza solo una muestra aleatoria de todo el conjunto de entradas en cada iteración para actualizar los pesos en la función de perdida. Esto mete algo de aleatoriedad en el aprendizaje y evita sobreajuste y estancamiento en el proceso. Además, también se puede incluir una organización de estas muestras en **mini lotes** (mini batches), haciendo el entrenamiento aún más rápido al tener la posibilidad de paralelizar estas operaciones.

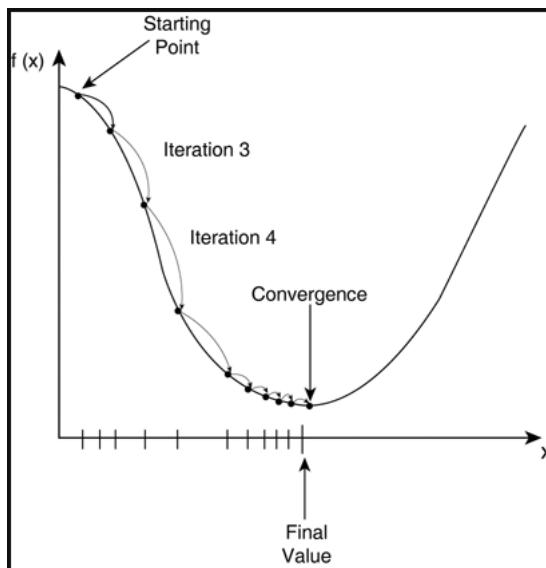
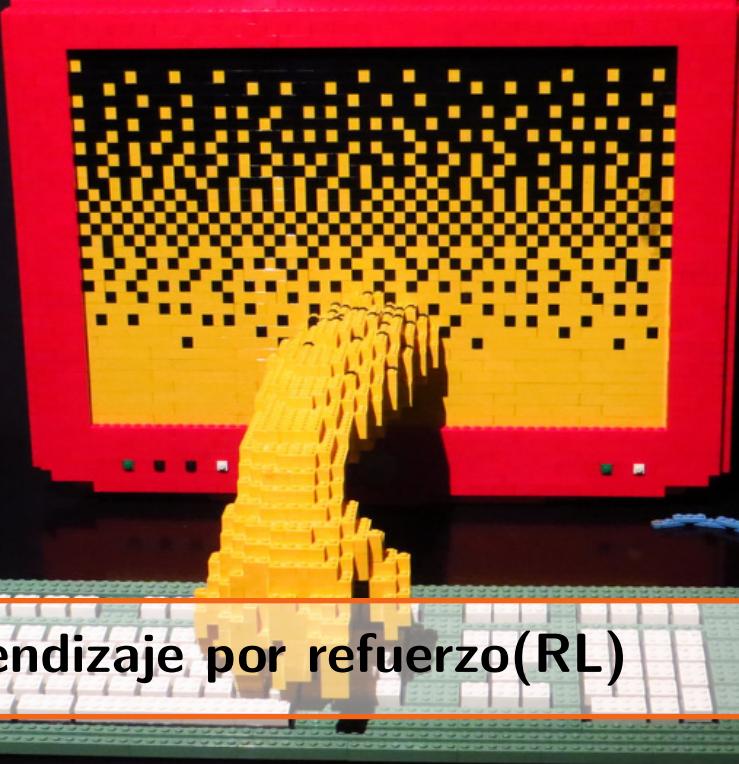


Figura 2.4: Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de origen [8].

Si observamos la figura 2.4, vemos que es un ejemplo muy sencillo. Debemos tener en cuenta que sólo tiene un parámetro (x). El caso es que en redes neuronales tenemos una función que puede tener cientos de miles de parámetros, dependiendo de la red, y que esto ni siquiera es mostrable en una gráfica ya que espacialmente no entendemos más allá de tres dimensiones (2 parámetros más el valor de salida). Por eso es importante el muestreo de entradas, para hacerlo viable computacionalmente.

- **Adam:** El algoritmo Adam (Momentum Adaptable) es uno de algoritmos de optimización más usados. No vamos a entrar en detalle, simplemente decir que esta basado en gradientes de primer orden, es computacionalmente eficiente, pocos requisitos de memoria y es muy adecuado para problemas de redes con un gran número de parámetros y datos.

En definitiva, las redes neuronales son tan amplias como artículos tiene publicados al respecto y experimentos se han realizado desde su aparición. Solo se pretende dar una vista general de su arquitectura y como funcionan.



3. Aprendizaje por refuerzo(RL)

También es muy importante tener conocimientos con respecto a las técnicas de aprendizaje por refuerzo(RL) para desarrollar los algoritmos que veremos y utilizaremos más adelante. Comenzaremos por explicar los fundamentos del RL, los componentes que lo forman y definen un problema de este tipo, junto con algunos de sus conceptos básicos y metodología genérica, para luego centrarnos en algunas de las técnicas y estrategias más utilizadas y populares en la actualidad.

El mundo es muy **complejo** y los retos que nos plantea también, a veces incluso más de lo que imaginamos. Cuando tratamos de llevar esos problemas al RL, nos encontramos con agentes que deben aprender en entornos con un espacio de estados y posibles acciones muy grande.

La metodología de aprendizaje es **secuencial**. En muchos de los problemas que existen hay consecuencias en las decisiones que no se manifiestan inmediatamente, se demoran en el tiempo. Esto hace que sea complicado definir cómo de buena o mala es una decisión y es uno de los principales retos del RL.

Debemos ser conscientes de que siempre tendremos que lidiar con la **incertidumbre**, es algo inevitable. Los problemas son tan complejos que es imposible tener todos los casos posibles en cuenta de manera específica. Esa incertidumbre es la que da sentido a la **exploración**, tratando de buscar el equilibrio con la **explotación** para ofrecer respuestas de la mejor calidad posible en un rango de tiempo viable.

Expicaremos todos los conceptos y componentes que conforman los problemas a partir de un marco de trabajo matemático llamado **Procesos de Decisión de Markov(MDPs)**. El cual nos permitirá modelar virtualmente cualquier problema complejo de una forma que los agentes de RL puedan interactuar y aprender a resolverlos. [25] [40] [3]

3.1 Componentes del RL

Los dos principales componentes del RL son el **entorno** y el **agente**. El entorno es una representación del problema planteado, el agente es un tomador de decisiones, y por tanto, una solución. Una de las diferencias más distintivas que tiene RL con otras técnicas de Machine Learning es que el agente **interactúa**, tratando de influenciar el entorno rea- lizando **acciones**, mientras que el entorno **reacciona** ante ellas, generando nuevos **estados**.

- El espacio de estados, el conjuntos de todos los posibles que existen, puede ser finito o infinito dependiendo del problema que sea. Cada estado estará formado por un número **finito** de variables que lo describe, siempre debe ser finito.
- El espacio de acciones puede variar de un estado a otro, a ese subconjunto se denota como $A(s)$ siendo s el estado del entorno en ese momento. Esto sucede debido a la existencia de problemas en los que, dado un estado, hay acciones que sencillamente no se pueden realizar en ese momento por el agente. Por ejemplo, en ajedrez no es posible mover una pieza mientras que se está en jaque, a no ser que ese movimiento revierta ese estado de amenaza por parte del oponente. El conjunto de acciones también puede ser finita o infinita, pero cada una de ellas está compuestas por una o más variables finitas, igual que los estados.

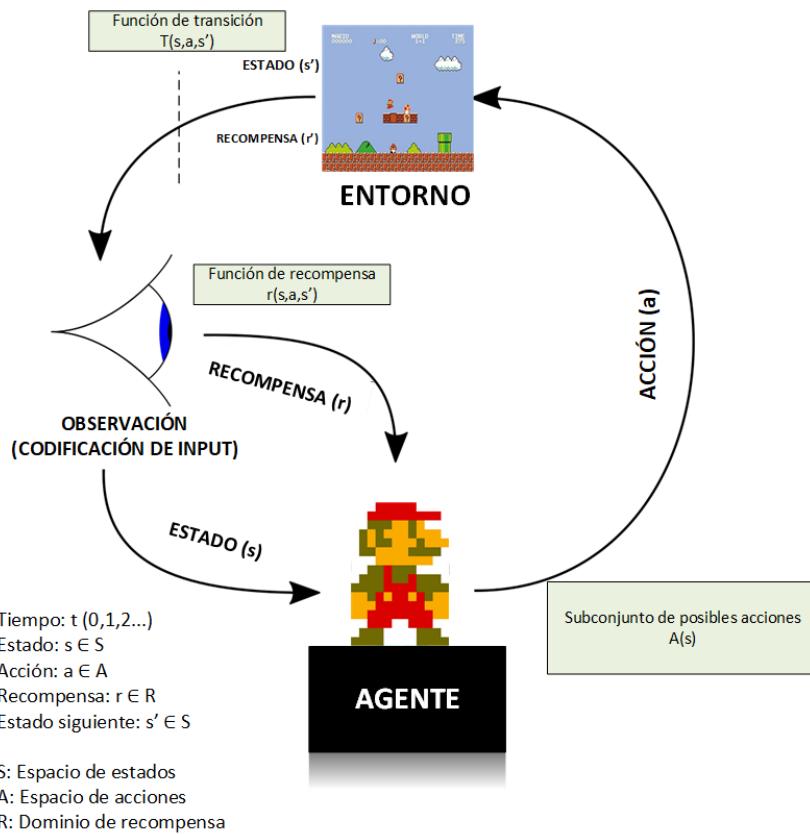


Figura 3.1: Esquema de componentes del RL. Imagen extraída y posteriormente modificada de origen [48].

Cuando el entorno reacciona, quiere decir que se produce un cambio que, a su vez, repercute en observaciones diferentes por parte del agente en el futuro. La **recompensa** es

una parte fundamental en esta representación, dependiendo de la calidad de esa estimación, el agente realizará un mejor aprendizaje de su propia experiencia. La secuencia que se produce entonces a partir del esquema de la figura 3.1 sería:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots$$

Es común representar malas acciones con un valor negativo en la recompensa. Entonces, conceptualmente, se trataría de una **penalización** en su lugar. Sin embargo, la comunidad de RL siempre utiliza la palabra recompensa para definir tanto las buenas decisiones como las malas, refiriéndose directamente a su valor numérico.

En la comunidad de RL también se suele utilizar estado del entorno y observación como conceptos intercambiables. Sin embargo, tiene un detalle que puede establecer una diferencia entre ambos conceptos. La **observación** es una interpretación del estado. Dependiendo del problema, es posible que el agente solo pueda interpretar una parte del estado, que sería lo que ocurre con AlphaStar, por ejemplo.

Cuando el agente realiza una acción de su espacio de acciones, puede desencadenar una reacción por parte del entorno que haga cambiar su estado entre su espacio de estados. La función que es responsable de esta **transición** es llamada **función de transición**.

Después de esa transición, el entorno ofrece un nuevo estado, que indica una nueva observación del agente. El entorno también puede ofrecerle una señal de **recompensa**. La función encargada de esto es la **función de recompensa**.

Las funciones de transición y recompensa son las que establecen el **modelo** del entorno como tal. Es importante tener estos conceptos claros antes de continuar, dado que es el marco a partir del cual representaremos cualquier problema para poder entrenar y formar cualquier modelo.

3.1.1 Ciclo de interacción

Las interacciones entre el agente y el entorno se produce en una serie de ciclos secuenciales, tal y como se aprecia en la figura 3.1. Cada uno de esos ciclos recibe el nombre de **time-step**. Es una unidad de tiempo en la que se completa cada ciclo de interacción, el cual depende del problema específico.

En cada *time-step*, el agente observa el entorno, realiza una acción y recibe una nueva observación y recompensa. La experiencia recolecta estos datos precisamente; siendo el conjunto de observación, acción, recompensa y nueva observación una **tupla de experiencia**. Más adelante hablaremos de este tema con mayor detenimiento.

Los **episodios** pueden entenderse como cada intento de cumplir el objetivo final. Por ejemplo, si hablásemos de AlphaGo; el agente que juega a GO, un episodio correspondería a una partida completa contra su oponente.

Definido de una manera más formal, un episodio es una secuencia de estados del entorno, acciones realizadas y recompensas obtenidas que finalizan con un **estado terminal**. Ya sea fracasando o teniendo éxito en su objetivo. Un agente puede tardar uno o más episodios en aprender una tarea concreta o conocimiento que le ayude a completar su objetivo. También

puede darse el caso de que se tenga que definir un **problema continuo** el cual no tenga estado terminal. Para ese caso deberemos simular un estado terminal que normalmente se establece limitando un **máximo** de *time-steps*, veremos esto con detenimiento un poco más adelante también.

La sumatoria de todas la recompensas habiendo finalizado un episodio es denominado **recompensa acumulada** y con ella puede saberse de forma implícita si ha conseguido un estado terminal favorable o no para el agente. Por ejemplo, si la recompensa acumulada es positiva o negativa, aunque no sería la única forma y depende de la definición del problema concreto en el marco MDPs.

3.2 Procesos de Decisión de Markov(MDPs)

El MDPs es un marco de trabajo que es utilizado para describir un entorno para el aprendizaje por refuerzo. Prácticamente todos los problemas de RL pueden ser formalizados en MDPs o en una de sus extensiones, definiendo el formato de cada unos de sus componentes y funciones que hemos visto en el apartado 3.1.

Se parte de una **suposición** muy importante; las probabilidades del siguiente estado, dado el estado actual y la acción, es totalmente **independiente** de las interacciones realizadas hasta el momento. Esta propiedad no contextual de los efectos de una acción en el entorno es conocida como la **propiedad de Markov**: La probabilidad de moverse de un estado s a otro estado s' , dada la misma acción y estado, es la misma independientemente de todos los estados previos y desarrollo del episodio encontrados hasta ese punto.

$$P(s'|s, a) = P(s', r|s, a, S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2}, \dots)$$

Siendo S el espacio de estados, A el espacio de acciones, t el *time-step* actual, r la recompensa devuelta por el entorno, $s' = S_{t+1}$, $s = S_t$ y $a = A_t$.

3.2.1 Función de transición

Define las consecuencias de una acción en el entorno. Aquí es donde se aplica la propiedad de Markov, siendo la finalidad que devuelva un conjunto de probabilidades a estados de transición dado solamente el estado y acción actual. Vemos el sentido de esta propuesta, ya que trata de simplificar este tipo de cálculos. Se define de la siguiente forma:

$$\tau(s, a, s') = p(s'|s, a) = P(S_t = s'|S_{t-1} = s, A_{t-1} = a)$$

Por cuestiones obvias, esperamos que la suma de todas las probabilidades de cada uno de los posibles estados siguientes del entorno que pueden darse como fruto de una acción del agente sea 1:

$$\sum p(s'|s, a) = 1, \forall s \in S, \forall a \in A(s), s' \in S$$

Dependiendo del problema que se esté definiendo, la complejidad de la función de transición para calcular estas probabilidades variará. Si se trata de un problema totalmente **determinístico**, solo habrá un posible estado con valor 1 cuando se realice una acción. Si el problema es **estocástico**, existe un factor de cierta aleatoriedad en la reacciones del entorno que deben calcularse para todos los estados que pueden sucederse a partir de una

acción en un estado concreto, la **propiedad de Markov** trata de simplificar precisamente estos casos y hacerlos más predecibles.

3.2.2 Función de recompensa

Ofrece una señal numérica como magnitud de bondad a las transiciones que se producen. En robótica es muy común añadirle a este valor un coste en tiempo que reduce la bondad en función de éste. Esto se hace debido a que el objetivo no es ser solo eficaz, sino eficiente en las tareas que hay que realizar.

Existen numerosas formas de calcular la recompensa, depende de forma directa con el problema que se trata de resolver. Entonces, esta dependencia del problema hace que pueda definirse de varias formas en función de los datos que utilicemos para su cálculo. Puede ser $R(s, a, s')$ o $R(s, a)$ o incluso simplemente $R(s)$. No obstante, lo más común normalmente es utilizar el estado actual, acción actual y estado siguiente. La función se define de la siguiente forma:

$$\begin{aligned} r(s, a) &= \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] \\ r(s, a, s') &= \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] \\ R_t &\in R \end{aligned}$$

Con R nos referimos al dominio que tiene las señales de recompensa.

En muchos problemas es importante calcular la **recompensa acumulada**, o llamada en inglés **return**. Se trata simplemente de la suma de todas las recompensas que se han ido calculando durante un episodio:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$

Siendo T el número total de *time-steps* del episodio.

3.2.3 Descuento

Como ya se mencionó en el apartado 3.1.1 la unidad de medida para el tiempo en MDPs son los **time-steps** o pasos de tiempo en español. Cada unidad hace referencia a un ciclo de interacción del agente con el entorno y el tiempo real que tarda depende del problema en cuestión.

Los **episodios** hacen referencia a todos los *time-steps* sucedidos hasta llegar a un estado terminal, lo cual indica que su número de ciclos es finito. Sin embargo, también podemos encontrarnos ante problemas continuos los cuales nunca terminan; no existe un estado terminal para ellos. Podemos encontrarnos ante un caso híbrido, en el que hay un número de *time-steps* finitos en caso de que se llegue a un estado terminal satisfactorio y que nunca se llegué a un estado terminal si no consigue resolverse el problema, o viceversa. Para esos dos últimos casos, es conveniente establecer un **límite** de *time-steps* con los que parar manualmente el proceso una vez se alcanzan. En la comunidad de RL lo definen como tipos de **horizonte**.

Ante la posibilidad de estas prolongaciones largas de interacción entre agente y entorno, se necesita una forma de **descontar** ese tiempo de la recompensas que se obtienen mediante su función de recompensa acumulada G_t . Es una forma de dar más valor a las recompensas

tempranas que a las tardías, y que el agente se optimice teniendo en cuenta esto. De lo contrario, podría realizar acciones para optimizarse a muy largo plazo, demorarse demasiado en el tiempo.

Comúnmente se suele utilizar un valor real positivo por debajo de 1 para exponencialmente ir descontando ese valor de las futuras recompensas. Este valor es denominado **factor de descuento o gamma(γ)**.

Elegir este valor de forma adecuada no es genérico, ni tiene unas recomendaciones explícitas. Al tratarse de un hiperparámetro, puede ser totalmente distinto de un caso a otro dependiendo del problema. Es importante encontrar un valor bueno, ya que un cambio en este influye de manera considerable en la capacidad de aprendizaje del agente.

También sirve como reductor de varianza en las estimaciones. Dado que el futuro es incierto, y cuánto más *time-steps* miramos hacia delante, mayor factor estocástico y varianza tendrán las estimaciones. El factor de descuento ayuda a este hecho al ir reduciendo el grado en el que las recompensas afectan y estabilizan el aprendizaje:

Effect of discount factor and time on the value of rewards

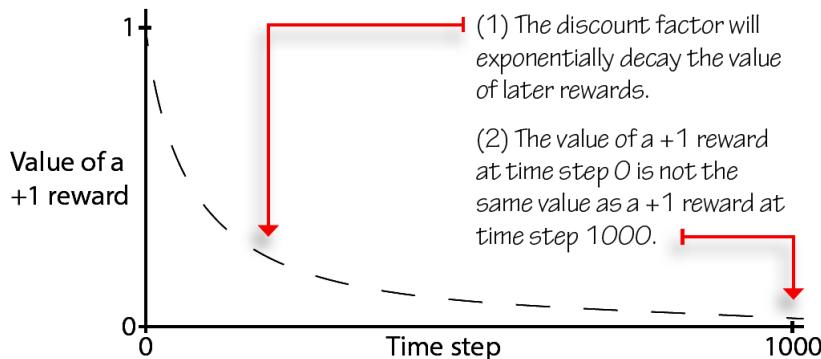


Figura 3.2: Ilustración gráfica del factor de descuento sobre la recompensas. Extraído de origen. [25]

Matemáticamente, la **recompensa acumulada** quedaría de la siguiente forma incluyendo el descuento:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T$$

Podemos simplificar esta ecuación y tener una más general:

$$G_t = \sum_{k=0}^T \gamma^k R_{t+k+1}$$

Estos son los componentes y conceptos del los MDPs. Existe muchas extensiones y variables que se han desarrollado para adaptarlo de una manera más fiel y ajustada a un conjunto de problemas con características concretas. Por ejemplo, para problemas con observaciones parciales del entorno(POMDP), para problemas con acciones, estados y recompensas de valores continuos... No es objetivo de este proyecto ver todas las posibles variantes que la comunidad científica ofrece sobre este aspecto.

3.3 Política

Ya hemos definido MDPs el cual es un motor y una arquitectura con su definición de componentes para representar los problemas de RL. En esta sección, pasamos de hablar de cómo **definir** un problema a cómo **soltar** un problema en MDPs. En términos generales, se necesita un **sistema** capaz de realizar una **secuencia de acciones** que trate de conseguir la mejor recompensa acumulada posible. Esto es lo que se conoce como **política** y es denotada como π .

Se trata de una función que devuelve una acción dado un estado del entorno. Una política puede ser cualquier función, no tiene por qué ser necesariamente “inteligente”. Por ejemplo, una función **aleatoria** puede usarse como política, generalmente mala, pero una política al fin y al cabo. En un lado opuesto, una política puede ser una **red neuronal**(DRL), algo mucho más complejo que una función aleatoria.

Al igual que los propios problemas de MDPs, las políticas pueden ser tanto **estocásticas** como **deterministas**. Dada una observación s , en las políticas estocásticas tendremos como salida con distribución de probabilidades de todas las salidas(acciones) posibles, mientras que en una política determinista tendremos una única salida.

A continuación, veremos una serie de funciones que, junto con la política, nos permite describir, evaluar y mejorar el comportamiento de la misma, estas son:

- Función de *estado-valor*
- Función de *acción-valor*
- Función de *acción-ventaja*

3.3.1 Función de estado-valor

Uno de los principales requisitos que hay que conseguir es una forma fiable de **comparar** las políticas. Decidir entre varias cual es mejor para resolver un problema complejo no es algo sencillo. Para este cometido, se utiliza una función llamada **función de estado-valor**:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

Esta función calcula una **previsión** de la recompensa acumulada a partir de un estado(s) bajo una política concreta(π). En otras palabras, estima como de bien se va a comportar una política a partir de un estado del entorno.

Si detallamos la función anterior, definiendo de una forma más específica la recompensa acumulada, tenemos lo siguiente:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right]$$

Teniendo esta función podemos comparar políticas diferentes para un mismo problema, partiendo de un idéntico estado para ambas:

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s)$$

Se asume que existe una política óptima (π_*), lo cual quiere decir que no existe otra política mejor que esa y el objetivo es aproximar una política lo máximo posible a ésta:

$$\pi_* \geq \pi' \forall \pi'$$

3.3.2 Función de acción-valor

En lugar de comparar una previsión de recompensas acumuladas a partir de un mismo estado para dos políticas diferentes, esta **función de acción-valor** trata de valorar la realización de una acción a en un estado s . Si tenemos una buena función que determine esto, podría ser de gran utilidad a la hora de decidir entre acciones y de esta manera **mejorar** las políticas.

Entonces, trata de estimar la recompensa acumulada de una política π después de haber realizado una acción a en un estado s :

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi [G_t | S_t = s, A_t = a] \\ q_\pi(s, a) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \end{aligned}$$

Vemos que la nomenclatura es muy similar a la función *estado-valor*, dado que el objetivo es el mismo solo que teniendo en cuenta una acción en ese estado antes de calcular esa recompensa acumulada. Esto puede ayudar a considerar el **dinamismo** del entorno y como cambia la expectativa de recompensa final acumulada obtenida a partir de las acciones que se realizan.

3.3.3 Ecuación de Bellman

La función de *estado-valor* puede ser expresada **recursivamente**. Esto es conocido como **Ecuación de Bellman**:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi [G_t | S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned}$$

Partimos de lo aprendido hasta el momento; la recompensa acumulada es la suma de la recompensa hasta el fin del episodio, junto con la ponderación del valor de descuento exponencial(γ). Si expresamos la acumulación de recompensa como la primera recompensa obtenida más la acumulación de recompensas siguientes multiplicadas por gamma(elevado a 1 por calcularse a partir del estado siguiente), podemos volver a obtener una función *estado-valor* y, por tanto, una función **recursiva**.

Esto resulta bastante interesante, ya que nos percatamos de que el valor de un estado depende recursivamente de los valores posibles de otros muchos estados futuros, incluyendo los estados originales a los que se llegará en el momento de llevarlo a la práctica.

Los algoritmos pueden iterar en esta ecuación y resolver el valor de estado para un entorno completo. A su vez, se puede extraer una conclusión a partir de este concepto. En una idílica política óptima, la función *estado-valor* siempre será igual a la función de *acción-valor* para la acción a que estime el mejor valor en s (la mejor acción posible), ya que partimos del supuesto de que siempre elige la mejor acción al ser la mejor política:

$$\begin{aligned} v_{\pi_*}(s) &= \max(q_{\pi_*}(s, a)) \\ a &\in A(s) \end{aligned}$$

Recordemos que $A(s)$ hace referencia al conjunto de acciones posibles en el estado s .

3.3.4 Función de acción-ventaja

Esta es una función que se deriva de las dos anteriores. También conocida como **función de ventaja**, es la diferencia de recompensas acumuladas estimadas por la función *acción-valor* de una acción a en un estado s y la función *estado-valor* del estado s bajo la política π :

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s)$$

Esta función determina como de bueno es coger la acción a en lugar de seguir la política π .

3.3.5 Evaluando una política

Utilizando estas funciones podemos evaluar el nivel de bondad de una política cualquiera definida en un problema MDPs. Ya se mencionó que utilizando la función *estado-valor* era posible decidir cuando una política es mejor que otra.

Explicamos la nomenclatura y el concepto de esta función. Ahora vamos a explicar una de las técnicas utilizadas para definirla dado un problema MDPs y una política que resuelve dicho problema.

Este algoritmo es conocido como **Evaluación iterativa de política**. Consiste en calcular la función *estado-valor* de una política recorriendo el espacio de estados(S) y mejorando dichas estimaciones de forma **iterativa**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a) [r + \gamma v_k(s')]$$

La variable k hace referencia a la iteración que está ejecutando. Vayamos por partes, $v_0(s)$ será la asignación inicial de estimaciones por parte de la función *estado-valor* para cada estado $s \in S$ con la política que estamos evaluando, serán valores arbitrarios y 0 en caso de que el estado sea terminal para cada estado $s \in S$.

$\pi(a|s)$ hace referencia a la probabilidad de ejecutar una acción a en el estado s por parte de la política. Para simplificar, asumamos una política determinística, por lo que el valor será directamente 1.

s' y r hacen referencia al siguiente estado y recompensa obtenida tras realizar dicha acción por parte de la política. Como ya se ha mencionado anteriormente, una acción a puede desencadenar más de una s' diferente, dependiendo de si el problema es estocástico o no, por eso se habla de probabilidad de s' y r sabiendo el estado s y acción a realizada.

Esta segunda parte calcula el valor del estado s como la suma ponderada de la recompensa y el valor estimado para el siguiente estado s' con su correspondiente descuento en caso de tenerlo.

Esta función la podemos ejecutar iterativamente de tal forma que estaremos aproximando el valor de $v_\pi(s)$ a su valor real para cada estado del problema. Este valor converge en

el **infinito**. En la práctica, deberemos establecer un **umbral**(θ) en el que si el valor nuevo no tiene una diferencia absoluta mayor que θ , se considere por aproximado correctamente y finalice.

Al tenerse en cuenta ese factor de **aleatoriedad** en reacciones del entorno ante las acciones, el valor que vamos a ir convergiendo para cada estado será la recompensa acumulada esperada que obtendremos si ejecutamos muchos episodios con esa política desde ese estado. En la realidad no se obtiene esa recompensa de manera exacta a partir de ese estado y esa política. No olvidemos que estamos hablando de **estimaciones**.

Nos damos cuenta que en el momento que hagamos más de una iteración, estaremos estimando recompensas a partir de estimaciones de recompensas. Esto es conocido como **bootstrapping** y es muy utilizado como técnica dentro de RL y DRL.

3.3.6 Mejorando una política

El siguiente paso es utilizar estos valores de evaluación de políticas a nuestro favor para poder **mejorarlas**. El ejemplo más sencillo de como hacerlo y menos eficaz sería generar muchas políticas aleatorias, evaluarlas e ir quedándonos con la mejor.

Lamentablemente, los problemas que se resuelven con estas técnicas suelen ser demasiado complejos como para alcanzar patrones y abstracción de conocimiento de una política de forma aleatoria.

Existen formas mucho más elegantes y mejores para inducir una mejora más eficiente. Vamos a hablar de la utilización de la función *acción-valor* para conseguir este objetivo.

Con esta función obtendremos las estimaciones de recompensas para cada acción a posible en cada estado s del entorno. Entonces, podemos utilizar esa información para establecer una **guía** en la modificación de la política.

Si en el caso anterior calculábamos la función *estado-valor* a partir de un algoritmo iterativo, veremos como calcular la función *acción-valor* a partir de la primera y MDP, denominado **algoritmo de mejora de política**:

$$\pi'(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')]$$

Recordemos que γ es la tasa de descuento comprendida entre 1 y 0 y que s' hace referencia al estado siguiente de s tras realizar la acción a . Básicamente consiste en buscar aquellas acciones que nos lleven a estados siguientes con la estimación de *estado-valor* más alta existente.

Cogiendo estas acciones que nos llevan a estados con estimaciones *estado-valor* más alto en lugar de la acción por la que optaría la política π en sí misma, estamos creando una nueva política denotada como π' , la cual vemos que sigue una estrategia **voraz** o greedy, dado que siempre sigue el mismo patrón en la selección de la siguiente acción.

Recordemos también que la función *estado-valor* calcula una estimación sumando todas las recompensas estimadas de todos los siguientes estados posibles (ecuación de Bellman en el que se explica esta recursividad), esto es lo que se utiliza para cada acción en un estado

concreto y nos quedamos con el máximo existente. Como consecuencia directa, esta función también es una aproximación que tratamos que sea lo más leal a la realidad posible.

A partir de este concepto, nos percatamos de que existe un **ciclo** que podemos repetir mientras que la política siga mejorando:

1. Calculamos la función *estado-valo*r con el **algoritmo iterativo de evaluación** para una política π_i en un problema MDP.
2. Calculamos la función *acción-valo*r con el **algoritmo de mejora de política** el cual parte de la función de *estado-valo*r previamente calculada.
3. Utilizando la función *acción-valo*r, podemos generar una nueva política.
4. Recalculamos la función *estado-valo*r de esta nueva política y repetimos el proceso siempre que haya un cambio significativo positivo en las estimaciones de la misma(mayor que θ).

Llegará un momento en el que la política converge y no se obtiene nada nuevo y mejor utilizando este proceso.

3.4 Exploración vs Explotación

Todo lo explicado hasta ahora referente a la evaluación y mejora de las políticas funciona muy bien cuando sabemos de antemano como va a reaccionar el entorno a cada una de las acciones que se realizan en el mismo por parte del agente.

Sabiendo exactamente la **función de transición** y la **función de recompensa** del problema MDP podemos obtener una **política óptima** sin la necesidad de que el agente interactúe con el entorno, dado que podemos estudiar las posibilidades que ofrece el problema de forma directa. Por desgracia, la mayoría de problemas planteados no son tan predecibles, ni podemos asumir la **propiedad de Markov** para simplificar la transiciones de los entornos.

Sabiendo lo que sabemos llegados a este punto, ahora es necesario buscar una manera de que el agente pueda aprender mientras lida con la **incertidumbre** en las transiciones de estados del entorno. Es lo que se conoce como **aprendizaje de ensayo y error**.

Esto plantea una serie de inconvenientes nuevos a la hora de mejorar la política del agente. Hay que decidir, dado un entorno s , si **explota** su propio conocimiento estimando las recompensas como hemos visto hasta ahora para obtener su mejor acción a para esa política π o si, como medida alternativa, decide **explorar** nuevos posibles desenlaces del entorno realizando acciones no óptimas bajo su política actual para generar nuevas interacciones sobre las que aprender.

Al no tener una función de transición perfectamente definida, se añade el problema de no saber si las estimaciones que tenemos sobre el valor de los estados y acciones son suficientemente buenas.

La respuesta es el **equilibrio**. La exploración va a permitir obtener un conocimiento que va a permitirnos ser más efectivos en las estimaciones de valores y, por tanto, en la explotación de la política.

Existen varias forma de introducir exploración en los problemas:

- **Estrategias de exploración aleatoria:** Es el más popular. El agente explota su política la mayoría de veces, y en algunas ocasiones explora utilizando una acción aleatoria. Es conocido como ϵ -*Greedy*, siendo ϵ un valor muy pequeño que representa la probabilidad de explorar en lugar de explotar.
- **Estrategias de exploración optimistas:** Es más sistemático que el anterior. Incrementa la preferencia de estados en los que hay un mayor factor de incertidumbre(son desconocidos por parte del agente), pudiendo provocar que el agente realice acciones con mayor probabilidad de transitar a estos estados en lugar de utilizar la acción óptima para su política dada.
- **Estrategias de exploración de información de espacio de estado:** Trata de enriquecer la información aportada por los estados del entorno, codificando un grado de incertidumbre como parte de dichos estados. Es una forma de diferenciar por parte del agente qué estados han sido explorados y cuales no.

Existe una gran cantidad de opciones a la hora de administrar la explotación y exploración en un agente, incluso variantes de las explicadas. Es uno de los principales retos a resolver por parte de RL.

3.5 Aprendiendo a evaluar y mejorar políticas

En el apartado 3.3.5 vimos la *Evaluación iterativa de política*, la cual nos permitía evaluar un problema MDP de forma matemáticamente directa sin la necesidad de que el agente tuviera que interactuar con el entorno previamente para optimizar sus funciones valor.

Vuelvo a insistir, uno de los principales desafíos del RL reside en que los agentes en la gran mayoría de problemas no pueden ver el MDP subyacente que rige el problema y el entorno. Más concretamente, las funciones de transición y de recompensa son **ruidosas**, lo cual hace al entorno mucho más hostil y que no pueda ser evaluado de forma directa, a no ser que exploremos sobre el mismo. A esto se le llama **problema de predicción**.

Cuánto mejor seamos capaces de **estimar** las recompensas futuras, a pesar de la incertidumbre de estos problemas, mejor capacidad tendremos para **mejorar** nuestro agente por medio del feedback que recibe de su propio desempeño con el entorno(**aprendizaje de ensayo-error**).

Entonces, a modo de condensación de conceptos, vamos a hacer hincapié en los siguientes:

- **Recompensa(r):** Es el valor que devuelve el entorno después de realizar una acción sobre él mismo, viene dado por su función de recompensa definida. Es uno de los datos más relevantes del RL, pero no es el valor que tratamos de maximizar, ya que no permite aprender conocimientos y estrategias con buenos planteamientos a medio

o largo plazo por parte del agente.

- **Recompensa acumulada(G_t):** Finalizado un episodio, es la suma de recompensas que se han ido produciendo en cada uno de sus *time-steps*. Si tiene un sistema de descuento con γ las recompensas irán perdiendo relevancia a medida que el episodio se prolonga más. Este dato es más representativo para el conocimiento que estamos tratando de conseguir, ya que tiene en cuenta la secuencia de acciones completas y estrategias a largo plazo. No obstante, de nuevo, no es el valor que estamos tratando de optimizar para el agente. Un agente que trate de obtener la recompensa acumulada más alta posible puede desarrollar una política con comportamientos ruidosos (si el episodio se prolonga en el tiempo puede obtener recompensas más altas). Es posible que se consiga una política con muy buen rendimiento, pero lo más probable es que no sea así.
- **Función de valor:** Hace referencia a la **estimación** de recompensa acumulada que se obtiene de antemano. Por lo que es un valor que se consigue durante cada *time-step* del episodio y no necesitamos que finalice para conocerlo.

Éste último es el valor que tratamos de **optimizar**. Al ser la mayoría de problemas estocásticos y no saber con certeza las transiciones de estados del entorno, tener unas medias altas de estimaciones para una acción dada ayuda a **generalizar** ese conocimiento a pesar del propio ruido del entorno.

3.5.1 Método de Montecarlo

El objetivo final, como bien sabemos, es conocer el valor de una política; cómo de buena es. Dicho de otra forma, el objetivo es estimar la función *estado-valor* $v_\pi(s)$ de la política π lo mejor posible.

El **Método de Montecarlo** consiste, en términos generales, en ejecutar un número considerable de episodios con la política a evaluar de tal manera que se almacenan cientos de trayectorias posibles de estados y acciones. Después se calcularía el **promedio** de recompensas acumuladas para cada estado.

La principal ventaja de este método es la sencillez de su implementación. Una vez finaliza un episodio, se almacena cada estado, acción, recompensa y estado siguiente obtenido como una **tupla de experiencia**. Cuando juntamos todas las tuplas de experiencia de un episodio(desde su estado inicial hasta el terminal), tenemos lo que se denomina como una **trayectoria**.

MÉTODO DE MONTECARLO:

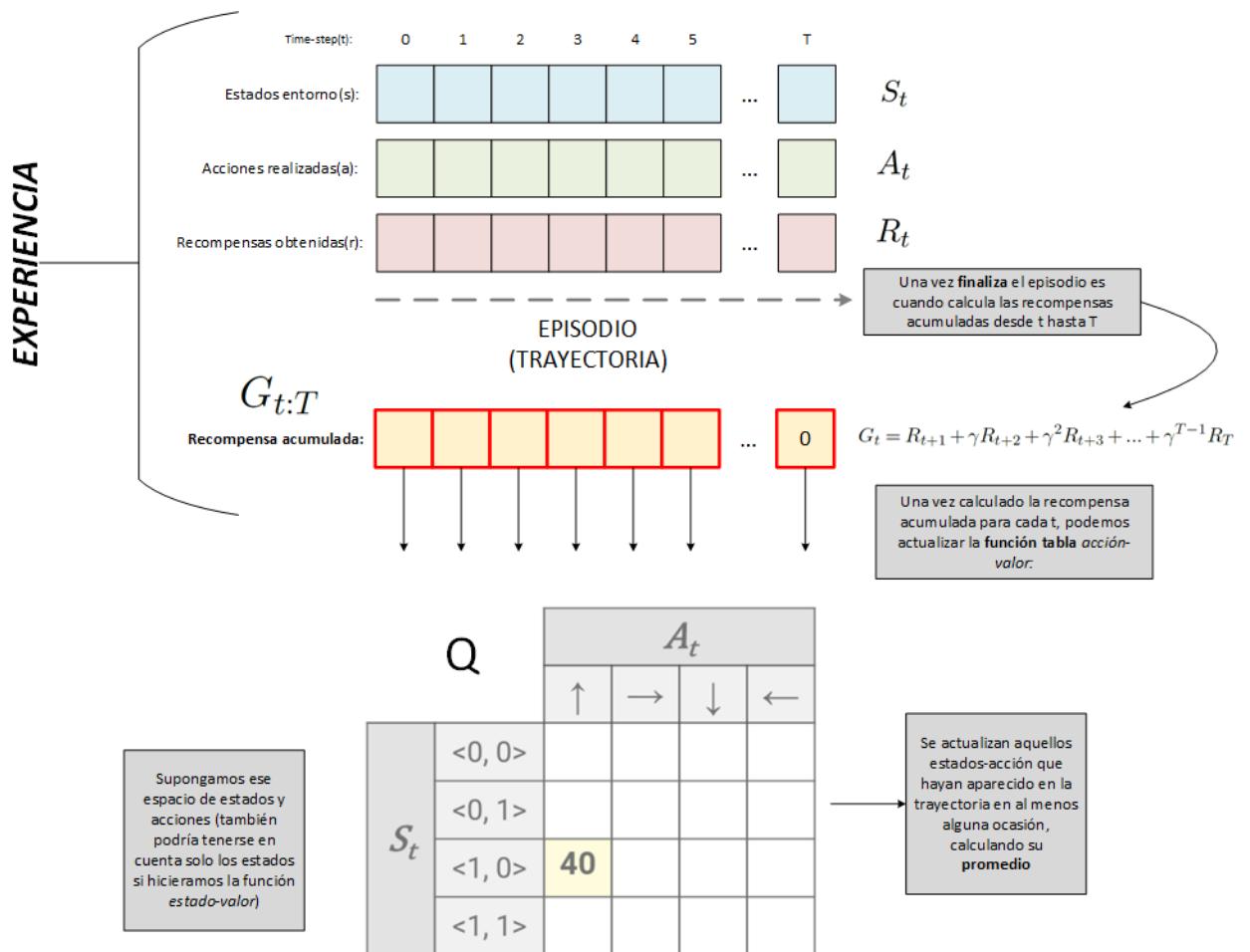


Figura 3.3: Esquema Método Montecarlo. Tabla función extraída de origen. [40]

Una vez se obtiene una trayectoria, calculamos el valor de **recompensa acumulada** que hubo desde cada estado hasta su estado final o, lo que es lo mismo, finalización del episodio, aplicando descuentos de forma exponencial como hemos hecho siempre. Entonces, tendremos una recompensa acumulada diferente por cada uno de los *time-steps* del episodio. Este valor se denota como $G_{t:T}$ siendo t el *time-step* actual y T el último *time-step* del episodio.

Ya se ha terminado de formar la trayectoria. El siguiente paso es estimar la función *estado-valor* simplemente calculando el promedio de recompensa acumulada obtenida en cada estado de la trayectoria, ya que un estado puede repetirse en más de una ocasión dado un episodio. Entonces, la función simplemente consiste en almacenar un valor promedio por cada estado diferente del entorno, se trata de una **función tabla** tal y como podemos observar en la figura 3.3.

También se puede realizar este método para calcular la función *acción-valor* ($q_\pi(s, a)$).

La forma de hacerlo es igual, solo que para los promedios se tendría en cuenta los estados y acciones juntos $\langle a, s \rangle$. Dicho de otra forma, se diferenciaría entre dos estados iguales en los que se realizó acciones diferentes. Cada actualización de la función valor nos permite modificar la política y tratar de mejorárla, como ya se ha explicado en otras ocasiones.

```
for i = {0, ..., n_episodes}
    Calcular trayectoria del episodio
    Actualizar la función estado-valor o acción-valor
    Obtener política mejorada a partir de esta nueva función
```

¿Cómo se calcula ese promedio una vez tenemos formada una trayectoria completa? Se puede hacer de dos maneras, y ambas son demostradas y aceptadas como buenas:

1. Tomar la **media** de los valores G para cada estado $\langle s \rangle$ o estado-acción $\langle s, a \rangle$ dependiendo de si estamos hablando de $v_\pi(s)$ o de $q_\pi(s, a)$ respectivamente.
2. En lugar de calcular la media, quedarse con el **primer valor G** encontrado para cada estado o estado-acción, de nuevo dependiendo de la función que estemos calculando.

Como ya mencionamos en el apartado 3.4, la exploración es muy importante para los problemas de RL. Por ello, en lugar de la política π podemos utilizar la variante ϵ -greedy. Con probabilidad ϵ generamos una acción aleatoria y la acción respondida por π en caso contrario.

El principal requisito que debemos tener en cuenta, directamente deducible de el funcionamiento de este método, es que solo podemos actualizar la función valor cuando el agente **finaliza** cada episodio, y no durante. Aunque existe una forma de poder actualizarse y mejorar paso a paso en su lugar.

3.5.2 Métodos de Diferencia Temporal(SARSA)

La principal desventaja del método de Montecarlo reside en su baja precisión en los problemas que superan una cierta complejidad de estados y acciones con alta incertidumbre de las transiciones de los mismos.

Esto se debe a que $G_{t:T}$; la recompensa acumulada desde el punto en el que se estudia hasta el final del episodio con el que se está aprendiendo, suele tener mucha varianza debido a este problema. Es decir, El **ruido** del propio problema hace que un estado o estado-acción en una trayectoria tenga una recompensa acumulada muy distinta en otra trayectoria futura que genere.

La forma de combatir este ruido para Montecarlo son los datos, recolectar toda la experiencia y trayectorias que pueda, para tratar de obtener mejores promedios y compensar de alguna manera el ruido.

El **Método de Diferencia Temporal(SARSA)** propone sustituir $G_{t:T}$ por la suma de la recompensa actual que recibe el agente del entorno R_{t+1} tras realizar su acción A_t con la estimación de función valor que es capaz de realizar con su política actual π y ponderada con el factor de descuento γ , ya que se trata de una interacción siguiente a la que estamos estudiando. De nuevo, se puede utilizar tanto para calcular la función *estado-valor*,

utilizada para evaluar políticas, como su función *acción-valor*, utilizada generalmente para mejorar dichas políticas:

$$\begin{aligned} v_{\pi}(S_t) &\leftarrow v_{\pi}(S_t) + \alpha [R_{t+1} + \gamma v_{\pi}(S_{t+1}) - v_{\pi}(S_t)] \\ q_{\pi}(S_t, A_t) &\leftarrow q_{\pi}(S_t, A_t) + \alpha [R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) - q_{\pi}(S_t, A_t)] \end{aligned}$$

El valor α se usa para ponderar el **cambio** de la función valor. En cada paso de entrenamiento si $\alpha = 1$ no se tiene en cuenta el valor previo de la función. Si $\alpha = 0$, se mantiene el valor previo y no experimenta una actualización de la función en ese paso.

Ese **cambio** del que estamos hablando consiste en sumarle a la función actual la diferencia de ella misma con el nuevo sistema en el que se tiene en cuenta la recompensa actual y estimación a partir del estado siguiente s' con esa función.

Podemos ir calculando las nuevas funciones valor durante cada *time-step* y recompensa que obtiene sin tener que esperar a que finalice un episodio, aunque no es la única ventaja.

La principal consecuencia de este hecho y diferencia entre SARSA y MonteCarlo es que conseguimos tener una **varianza** mucho menor. Esto ocurre porque SARSA depende solo de una acción y recompensa que se obtiene en ese mismo instante, el resto del valor es estimado con su función valor actual.

Estas estimaciones, dependiendo en el estado de desarrollo que se encuentren, van considerando cada vez mejor el ruido del problema que está tratando de solventar. Cosa que no ocurre con $G_{t:T}$, ya que es específico de la trayectoria y experiencia con la que esté trabajando en ese momento.

La conclusión final es que los métodos de diferencia temporal convergen más rápido que Montecarlo en términos generales.

MÉTODO DE DIFERENCIA TEMPORAL (SARSA):

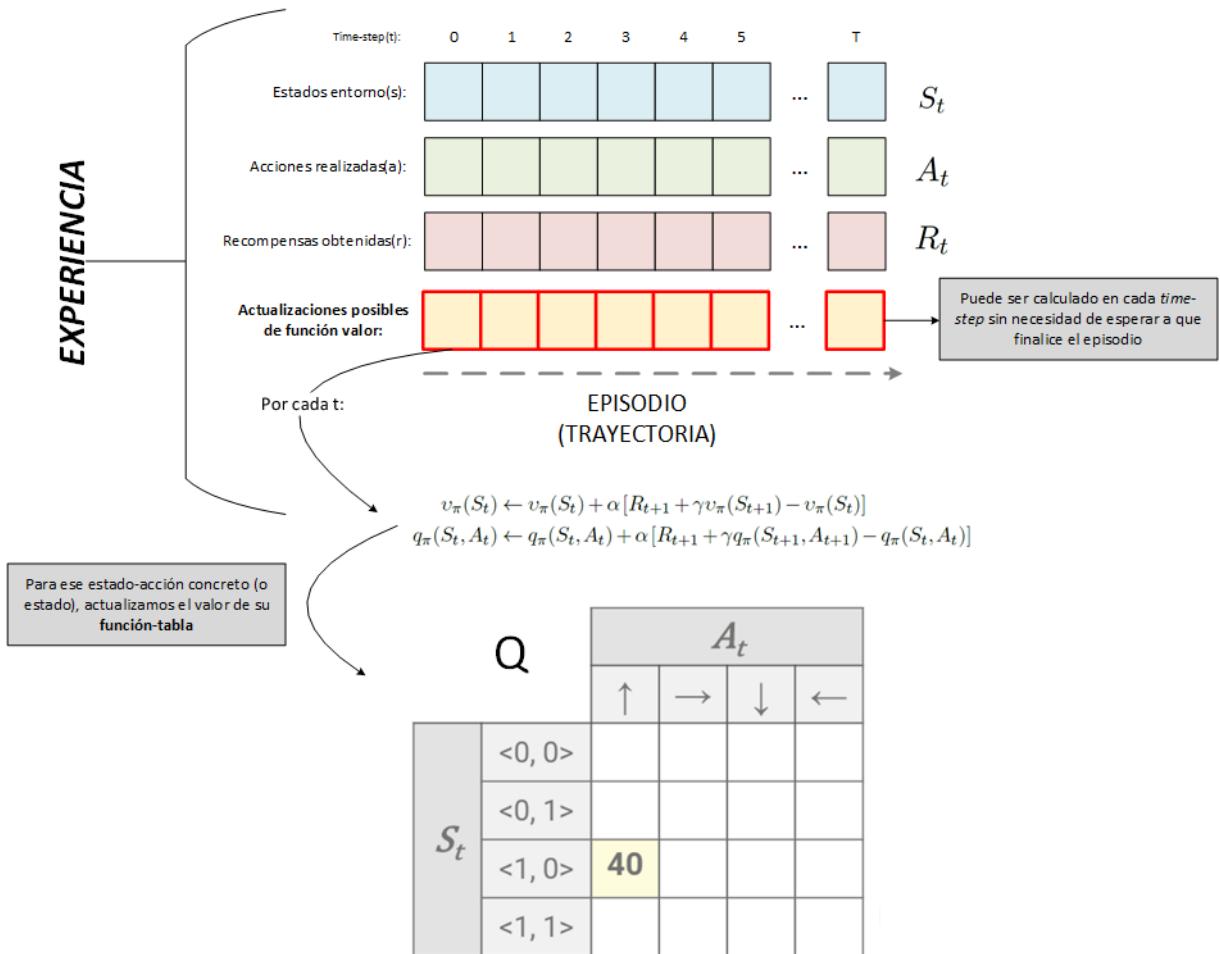


Figura 3.4: Esquema Método de diferencia temporal(SARSA). Tabla función extraída de origen. [40]

Calcular estimaciones a partir de estimaciones es un concepto que ya ha sido mencionado; estamos hablando de **bootstrapping**, muy común en el mundo de RL como podemos observar.

También es posible establecer **puntos intermedios** entre Montecarlo y SARSA. No se actualiza las funciones valor ni cuando finaliza un episodio, ni de forma inmediata cada vez que se produce un *time-step*. En su lugar, actualizamos cada *n* interacciones. Durante esas *n* interacciones utilizaremos la recompensa acumulada $G_{t:t+n}$, para el resto de acciones, hasta el supuesto final del episodio, volvemos a recurrir a las estimaciones de recompensas(bootstrapping):

$$G_{t:t+n} + v_{\pi}(S_{t+n+1})$$

$$G_{t:t+n} + q_{\pi}(S_{t+n+1}, A_{t+n+1})$$

No es objetivo de este trabajo entrar en mayor detalle con estos algoritmos debido a

que es suficiente para entender su arquitectura y funcionalidad dentro de los problemas de RL.

3.5.3 Q-Learning

Se trata de un modelo **libre de política** y con **bootstrapping**. Aproxima directamente la política óptima. Esto quiere decir que el agente puede actuar aleatoriamente y aún así encontrar la función valor y política óptima, veremos como es esto posible. Se trata de una propuesta diferente conceptualmente a las dos mostradas anteriormente, vamos a explicar cada parte con mayor detalle.

Si volvemos al ejemplo de SARSA, se trata de un modelo conocido como **On-Policy**, ya que depende de su propia política para poder mejorarse, por lo que es capaz de aprender de sus propios errores y aciertos. Esto las hace excelentes, sin embargo, esto hace al mismo tiempo que no sea capaz de aprender de “los errores de otro”, cosa que hacemos los humanos constantemente.

Por ejemplo, si alguien que está delante de nosotros tropieza con una piedra, generalmente vamos a aprender a evitarla antes de experimentar un tropiezo. Los modelos **On-Policy** son muy buenos, pero no aprenden fuera de la experiencia que hayan generado ellos mismos, por cuestiones matemáticas obvias vistas en las fórmulas anteriores.

El aprendizaje **Off-Policy**, por otra parte, es capaz de hacerlo. Puede aprender de una política que es diferente de la política generadora de experiencia. A esto es a lo que nos referimos con **libre de política** para el Q-Learning. En este tipo de aprendizaje tenemos dos tipos de políticas funcionando:

- **Política de comportamiento:** Utilizada para generar experiencia, interactuando con el entorno de forma directa.
- **Política objetivo:** Es la política sobre la que se realiza el aprendizaje y mejoras.

Durante las trayectorias de episodios con experiencia, tendremos en las estimaciones por *time-step* lo siguiente:

$$Q_{\pi}(S_t, A_t) \leftarrow Q_{\pi}(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a \in A(S_t)} Q(S_{t+1}, a) - Q(S_t, A_t) \right)$$

Tenemos una función *acción-valor* que denominaremos $Q(s, a)$. Para estimar las recompensas futuras en cada uno de los pasos de una trayectoria, hacemos una operación muy similar a SARSA, ya que también realiza un aprendizaje paso a paso y no episodio a episodio como Montecarlo. La principal diferencia viene dada en la **actualización** de la política. En SARSA, utilizábamos la diferencia de esa función con la suma de recompensa actual y estimación siguiente. En esta ocasión la estimación siguiente, es otra política que elige la mejor acción que considera Q (que es $a \in A(s)$), en lugar de elegir la acción con la que realmente se está produciendo esa experiencia (que es A_t).

Q-LEARNING:

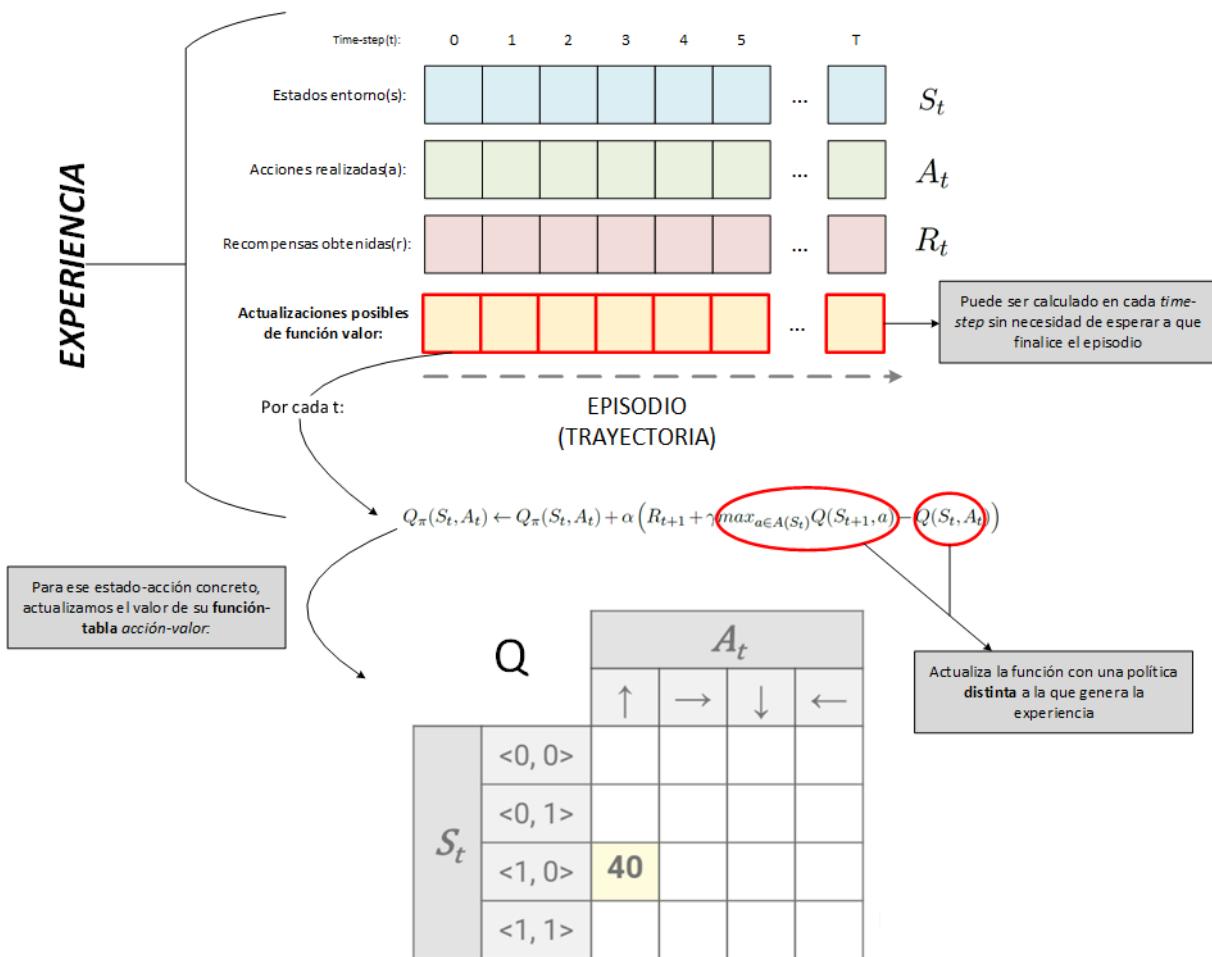


Figura 3.5: Esquema Método Q-Learning. Tabla función extraída de origen. [40]

Para la exploración, podemos seguir utilizando ϵ -greedy(Q) a la hora de realizar las estimaciones de recompensa con las acciones que tendría la política objetivo.

Existe algunas variantes como **Double Q-Learning**, la cual se basa en tener dos políticas objetivo en lugar de una. Es una estimación cruzada que ha mostrado mejores resultados en ciertas ocasiones, aunque no entraremos en más detalles con esta parte.

Como conclusión final, destacar la **independencia** de los problemas a resolver con las técnicas de RL vistas en estos apartados. Definiendo cada uno de los **componentes** del problema, como vimos en el apartado 3.1, estas técnicas ya pueden entrenar agentes directamente, dado que tiene la forma de obtener experiencia y tratarla. Es una de las grandes ventajas de RL.



4. Aprendizaje por Refuerzo Profundo(DRL)

Hemos visto lo más importante y necesario de las técnicas de RL; su arquitectura y funcionamiento. Si recordamos el apartado 3.3, una política es una función que puede ser de cualquier forma, siendo una función totalmente aleatoria la más sencilla a la hora de devolver salidas ante el estímulo que ofrece el entorno del problema.

Con los métodos de Montecarlo, y sobretodo SARSA y Q-Learning (basado en SARSA), vimos formas más sofisticadas de utilizar el feedback del entorno para construir nuevas políticas a partir del aprendizaje por ensayo-error. Sin embargo, estos métodos simplemente guardaban un valor estimado por cada estado-acción diferente del entorno, en eso se basaban esas funciones valor de la política.

¿Y si en lugar de utilizar eso usamos como política redes neuronales? Si representamos las funciones como un conjunto grande de pesos y una red, estamos hablando del DRL. Las técnicas son muy parecidas a las explicadas hasta ahora, sin embargo, contamos con una nueva tecnología para estimar los valores de función que puede llegar a ser más potente. Sobretodo para abordar problemas complejos, no solo por su espacio de estados y acciones, sino por la cantidad de variables o información que puede tener cada uno de esos estados y acciones, sin contar que esta información puede ser continua y no discreta que directamente no puede ser recordada por una función tabla.

Vamos a ver algunas de las formas más importantes en las que se ha conseguido integrar el DL dentro del RL, para conseguir alcanzar estas ventajas respecto al RL original.

4.1 Métodos Basados en Valor: Deep Q-Network(DQN)

Es el primer algoritmo DRL que vamos a ver. DQN es uno de los algoritmos más populares que hay dentro de este ámbito, siendo el inicio de una serie de investigaciones e innovaciones que marcaron un antes y un después en el mundo del RL. Con esta técnica fue la primera vez que se pudo crear un agente en ATARI capaz de resolverlo a partir de

los píxeles crudos, de la propia imagen. Esto resulta muy interesante, las observaciones que el agente hace del problema es exactamente igual a las que hace un humano, observar los píxeles de una imagen, en lugar de pensar anteriormente un formato de input específico para las observaciones del entorno.

Es una estrategia **basado en valor**(*value-based*). Esta estrategia parte de la técnica de RL Q-Learning visto en el apartado 3.5.3, utilizando como funciones optimizables redes neuronales, vistas en el apartado 2, lo que es comúnmente conocido como **Deep Q-Learning**.

El principal cambio reside en su función Q, modelándose como una **función no lineal**(redes neuronales), lo cual permite trabajar con estados y acciones continuos como ya hemos comentado anteriormente.

Sabemos que para esta estrategia se tienen dos políticas diferentes; la generadora de experiencia y la objetivo. Sin embargo, esto no equivale a dos redes neuronales distintas. La arquitectura de la red neuronal como tal es la misma, lo que se cambia es el **conjunto de pesos** que define su comportamiento. Teniendo dos conjuntos de pesos diferentes, uno para cada cometido que hemos explicado.

Hay varias formas de plantear esa red neuronal. Si las acciones posibles del problema son discretas(no infinitas). Podemos crear una red neuronal que tenga tantas neuronas en la capa de entrada como variables una observación del estado s , y por otro lado, que la capa de salida tenga tantas neuronas(y por tanto,outputs) como acciones posibles del problema. Representado cada una de esas salidas $Q(s, a; \theta) \forall a \in A$.

Para elegir una acción, podríamos coger la que tuviera un valor estimado más alto o aplicar una distribución de probabilidad para favorecer la exploración(esta decisión sería parte de la política π). Podemos observar dicha explicación en la figura 4.1.

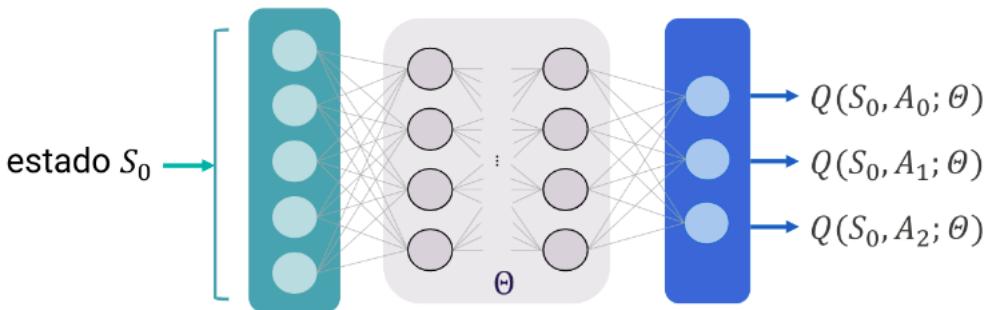


Figura 4.1: Esquema red neuronal, acciones discretas finitas. Imagen extraída de origen. [40]

El símbolo θ hace referencia al conjunto de pesos.

Por otra parte, podemos enfrentarnos a un problema que tenga acciones continuas y, por tanto, sean infinitas. La opción aquí sería incluir la acción como entrada de la red. Entonces, la salida sería $Q(s, a; \theta)$ para esa acción concreta. Si queremos Q para otra acción, la parte de entrada de s sería la misma y habría que modificar a . Al ser el número de

acciones infinitas, no podemos calcular todas las posibles acciones a partir de la entrada s como en el caso anterior. Lo vemos a continuación en la figura 4.2.

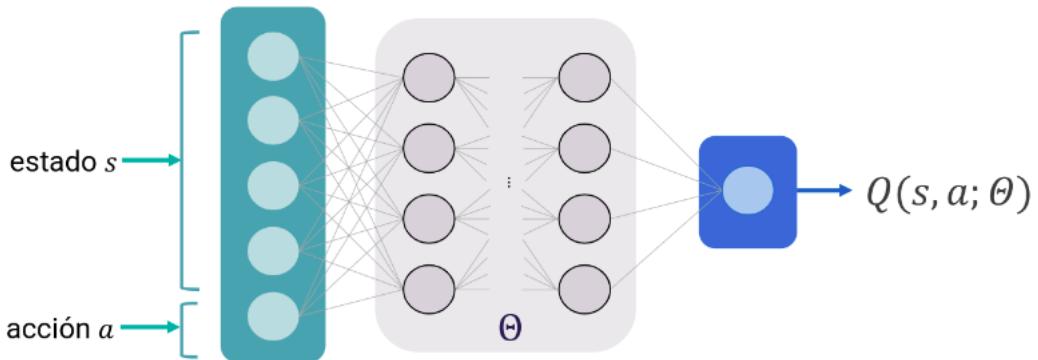


Figura 4.2: Esquema red neuronal, acciones continuas infinitas. Imagen extraída de origen. [40]

En definitiva, el principal cambio entre Q-Learning y Deep Q-learning es la **sustitución** de la función-valor tabla por una red neuronal, con el potencial que esta nos ofrece frente a la anterior. ¿Pero como repercute dicho cambio en el proceso de entrenamiento?

Q-Learning permitía entrenarse paso a paso conforme experimentaba utilizando el *método de diferencia temporal*(SARSA). No obstante, aquí se propone guardar una base de datos o también denominado **memoria de experiencias**. Esto hace que la experiencia pueda utilizarse en **cualquier orden y más de una vez**.

Esta adaptación trata de combatir dos problemas principales que tienen las estrategias basadas en valor(*value-based*): en primer lugar, mejorar la **exploración** de dependencias entre zonas, ya que la experiencia se genera de forma secuencial. Por otra parte, con el método Q-Learning tradicional, no podíamos utilizar varias veces un paso o pasos de la experiencia que fuesen interesantes para el modelo.

Si recordamos el apartado 2, las redes neuronales necesitan, entre otras cosas, una **función de error o pérdida** para posteriormente poder optimizar sus pesos con una **función de optimización** que se aproveche de esta información.

El problema es que no sabemos la respuesta óptima(o etiqueta original cuando hablábamos de aprendizaje supervisado). Es parte del paradigma del aprendizaje por refuerzo, existe tal incertidumbre y ruido en el entorno que es imposible en la mayoría de ocasiones saber la respuesta óptima, y la exploración (aprendizaje ensayo-error) es la única forma de poder acercarnos o aproximarnos a la mejor respuesta posible en cada una de las situaciones que nos plantea el entorno.

Una vez más, hay que **estimar** la respuesta óptima para poder tener una función de error para nuestra red neuronal, esto sería el entrenamiento del conjunto de pesos para nuestra política objetivo:

$$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha [R_1 + \gamma \max_{a \in A} Q(S_1, a) - Q(S_0, A_0)]$$

Con esa formulación podemos estimar la recompensa acumulada real que esperaríamos del entorno al final del episodio, a partir de un estado, acción , siguiente estado y recompensa, como ya hemos visto.

Cuanto mejor aproximemos Q, mejor será nuestra función de error para la red neuronal en sí. Por ejemplo, podríamos utilizar el error cuadrático(*MSE*):

$$MSE = \mathbb{E}_\pi[(Q(S, A) - Q(S, A; \theta))^2]$$

Entonces, se debe producir una **actualización de los pesos** en la red ($\Delta\Theta$) denotada con la siguiente expresión

$$\begin{aligned}\Delta\Theta &= \Delta w = -\eta \frac{\delta E}{\delta w} \\ \frac{\delta E}{\delta w} &= 2(Q(S, A) - Q(S, A; \Theta)) \frac{\delta Q(S, A; \Theta)}{\delta w} \\ \frac{\delta Q(S, A; \Theta)}{\delta w} &= \nabla_\Theta Q(S, A; \Theta) \\ \Delta\Theta &= -2\eta(Q(S, A) - Q(S, A; \Theta))\nabla_\Theta Q(S, A; \Theta)\end{aligned}$$

Si simplificamos y aplicamos la expresión de Q-Learning que ya sabemos:

$$\Delta\Theta = \alpha [R + \gamma \max_{a \in A} Q(S', a; \Theta)] \nabla_\Theta Q(S, A; \Theta)$$

Toda esta formulación matemática describe cómo se va a realizar esos cambios en los pesos que mencionamos a partir de cada tupla de experiencia. Vemos que se aplica el **gradiente de los pesos** para dicho cometido, parecido al que se mencionó en el apartado 2.4, solo que la formulación es diferente por el hecho de combinarlo con Q-Learning.

Podemos aplicar una estrategia de *mini-batches* (mini-lotes) extraido de la memoria de experiencias; esto quiere decir subconjuntos para realizar entrenamientos de los pesos.

Estamos actualizando una aproximación a partir de una aproximación. Una vez más, aparece el ya conocido concepto de **bootstrapping**. Por ello, se utilizan dos redes Θ , lo cual significa dos políticas diferentes, identificativo de la metodología Q-Learning.

En resumen, los pesos Θ^- , los cuales forman la red de la **política objetivo**(\bar{Q}), se actualizan con menor frecuencia a partir de los pesos(Θ) que forman la red *función-valor* que a su vez describe la **política generadora de experiencia**(Q).

$$\Delta\Theta = \alpha(R + \gamma \max_{a \in A} \bar{Q}(S', a; \Theta^-) - Q(S, A; \Theta))\nabla_\Theta Q(S, A; \Theta)$$

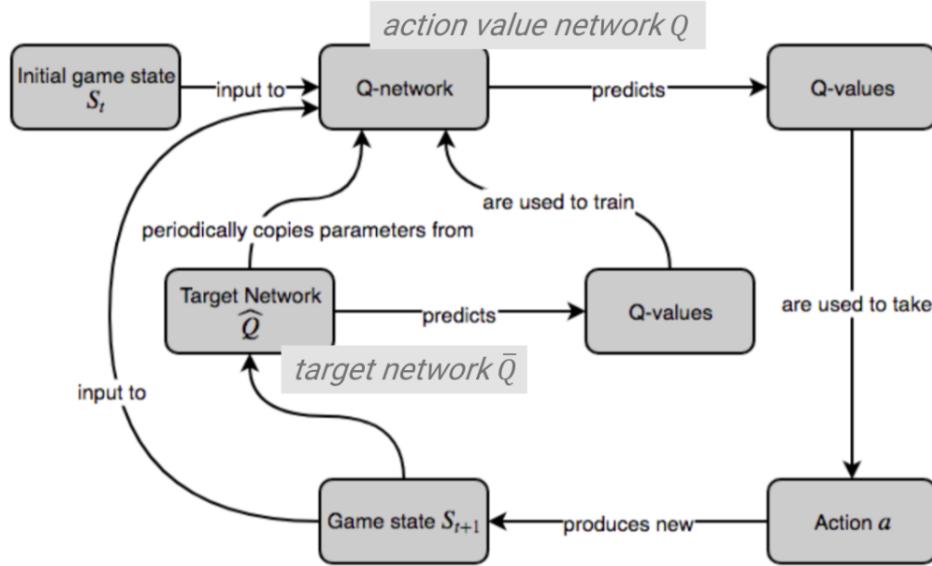


Figura 4.3: Esquema Método Deep Q-Learning. Extraída de origen. [3] [40]

En el apartado 3.5.3 se habló de las ventajas de tener dos políticas en lugar de una, eso se mantiene. La diferencia de todo esto es que buscamos las actualizaciones de los dos conjuntos de pesos de ambas políticas en lugar de cambiar valores en una función-tabla.

Al igual que hablamos de *Double Q-Learning*, existe el **Double DQN**, tratando de mejorar el inconveniente de **sobreajuste** muy común en los métodos basados en valor, ya que la política objetivo elige siempre ese máximo. También existen otras propuestas como el uso de la **ventaja** visto en el apartado 3.3.4, junto con otras muchas variaciones de esta estrategia, aunque no son objetivo de este trabajo.

4.2 Métodos basados en política: REINFORCE

En los métodos basados en valor, el objetivo principal era aprender a evaluar las políticas, minimizando la diferencia o perdida entre el valor predicho y el valor objetivo.

Por otro lado, en los **Métodos basados en política** o **métodos de política gradiante** tratamos de optimizar directamente una política parametrizada, obteniendo una **distribución de probabilidades** del espacio de acciones. Para ello utilizamos una política de gradientes.

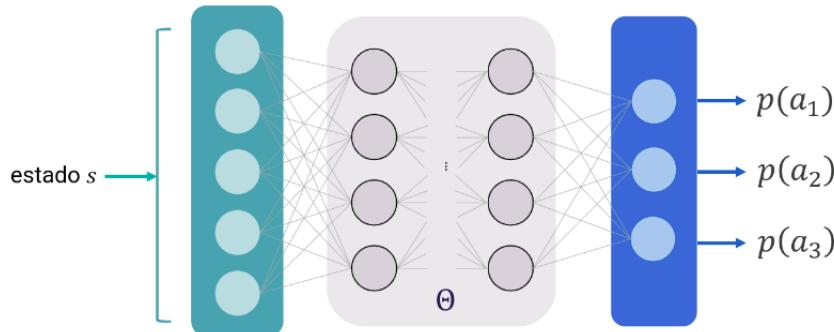


Figura 4.4: Metodología basada en políticas. Extraída de origen. [40]

La principal ventaja que nos ofrece esta metodología es el hecho de que no suponga un inconveniente el **espacio de acciones** del problema. Si teníamos que elegir el valor máximo entre muchas estimaciones(una por cada acción), antes había que estimar cada una de ellas. Si había muchas acciones posibles(o esa acción es continua), la cantidad de estimaciones también era realmente alta, y por lo tanto, limitaba bastante a DQN en problemas de ese tipo. Mucha de la información calculada no era utilizada realmente.

Los métodos basados en política, por otro lado, pueden aprender más fácilmente esas **políticas estocásticas**, lo cual deriva en otros tipos de ventajas. Como aprender mejor de entornos parcialmente conocidos o no ser tan dependiente de la **suposición de Markov**. Además, no hay ese exceso de información que mencionábamos.

Estos métodos también suelen **converger más rápidamente** que los basados en valor, ya que fijamos directamente los parámetros de la red neuronal y estos se van modificando suavemente con los gradientes, por lo que nos aseguramos alcanzar, como mínimo, una **optimización local**. Los métodos basados en valor suelen ser algo más impredecibles en ese sentido.

El algoritmo REINFORCE tradicional se basa en una red neuronal la cual es entrenada **al final de cada episodio** con los valores que devuelve **Montecarlo** y que ya conocemos, calculando su valor de pérdida y optimización a partir de estos, utilizando el optimizador Adam mencionado en el apartado 2.4.

$$U(\Theta) = \sum_x P(x; \Theta) R(x)$$

$$\nabla_{\Theta} U(\Theta) \approx \frac{1}{K} \sum_{i=1}^K \sum_{t=1}^n \nabla_{\Theta} \log_{\pi_{\Theta}}(A_t^{(i)} | S_t^{(i)}) R(x^{(i)})$$

$U(\Theta)$ es conocida como la función “**máximo beneficio**” es la que estamos tratando de maximizar. Generamos K trayectorias $x^{(i)}$ de longitud n según la política π_{Θ} . Calculamos la recompensa R^i por cada una de las trayectorias y actualizamos el conjunto de pesos Θ para estimar el gradiente de esa función de máximo beneficio de la que hablábamos.

Existen muchas variantes como VPG aunque, de nuevo, no es el objetivo de este trabajo.

4.3 Métodos actor-critic

Llegados a este punto, conocemos los métodos basados en valor y basados en políticas. Sabemos cuales son las ventajas principales de cada uno de ellos. No obstante, vamos a centrarnos en los inconvenientes de cada uno.

De forma resumida, REINFORCE se basa en **actuar**, ya que calcula las probabilidades de todas las acciones para que nos conduzcan a un buen resultado final. Esto tiene el principal inconveniente de que si una trayectoria conduce a un mal resultado final, puede **contener buenas acciones** que no están siendo reforzadas en su actualización de pesos(recordemos que se actualiza al final de cada episodio).

Por otro lado, DQN se centra en **estimar** los valores de recompensas que nos vamos a encontrar en el futuro a partir de un estado y una acción realizada en el entorno. El principal inconveniente de éste es que calcula las estimaciones a partir de estimaciones, lo cual introduce **sesgo** en el aprendizaje.

La propuesta que se va a analizar a continuación es una **combinación de las dos anteriores**, de ahí su nombre **actor-critic**(una parte actúa y la otra estima). Entonces, hace uso de dos redes:

- **Actor:** Similar a la red REINFORCE.

- **Critic:** Similar a la red DQN.

La principal ventaja es que **acelera** el aprendizaje, al mismo tiempo que **evita sesgos** en sus estimaciones, por lo que lo hace más estable.

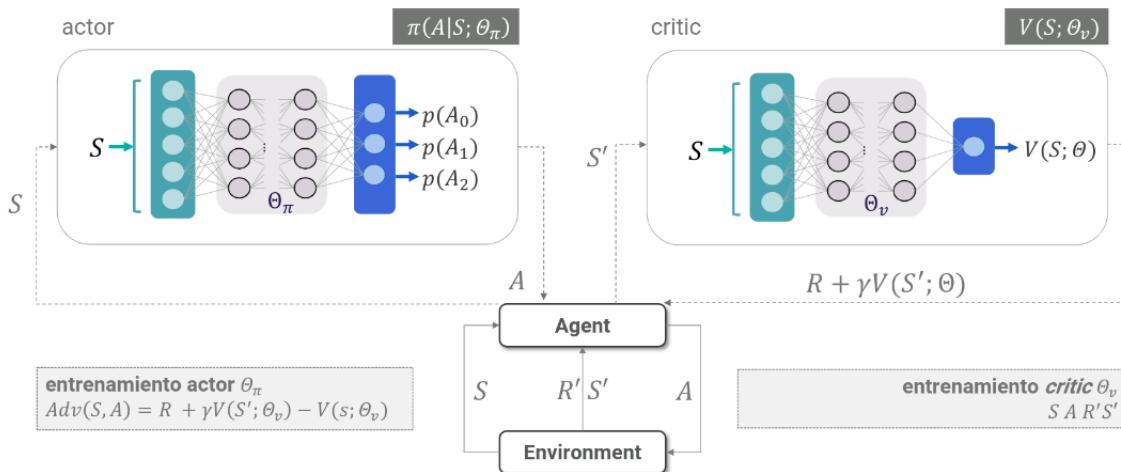


Figura 4.5: Esquema de metodología actor-critic. Extraída de origen. [40]

El actor recibe un estado s , que a su vez provoca una acción a por parte del mismo. Esta acción es realizada en el entorno, transitando a un estado nuevo s' , ese estado es analizado por el crítico el cual realiza la estimación, recogida por el agente de nuevo.

Lo más importante es que el actor aprende teniendo en cuenta las estimaciones del crítico, el cual aprende de forma directa de la experiencia generada. La explicación detallada de esta técnica depende directamente del algoritmo concreto que estemos hablando como, por

ejemplo, A3C (aprendizaje asíncrono) o A2C(aprendizaje síncrono), aunque no entraremos en mayor detalle.

Desarrollo y Experimentación

5	OpenAI baselines	63
6	Entornos Open AI Gym	65
6.1	MountainCar-v0	
6.2	MountainCarContinuous-v0	
6.3	Entorno más complejo (aún sin decidir)	
7	Entorno de desarrollo	69
7.1	Metodología de experimentación	
8	Experimentación: Resultados Obtenidos	71
8.1	MountainCar-v0	
8.2	Otro entorno más complejo (Aún sin decidir)	

Introducción

Hasta ahora, hemos explicado desde un punto de vista más **teórico** los conceptos que vamos a manejar durante el desarrollo de este trabajo. Ahora, daremos un enfoque **práctico** de todo el esfuerzo que se ha dedicado en este proyecto. Las explicaciones serán realizadas a partir de entornos específicos.

Utilizaremos para ello las librerías de **OpenAI** para aplicar estos algoritmos en **Python** y algunos entornos de **Gym** en los que le daremos la posibilidad a nuestros agentes de “ensayar” su tareas.

Por experiencia, las redes neuronales son computacionalmente costosas en términos generales, sin contar con las técnicas de aprendizaje por refuerzo que utilizaremos. Por ello, se hará uso de una **máquina virtual**, aprovechando los **recursos en la nube** para contar con una mayor potencia y, por tanto, reducir los tiempos de entrenamientos necesarios para nuestros agentes. Utilizaremos la plataforma de **Google Cloud**.

Por último, veremos el resultado final. La eficacia del funcionamiento de los agentes obtenidos junto con los resultados de la monitorización de su aprendizaje. Estos datos serán mostrados, analizados y extraeremos las conclusiones de este proceso y objetivo final conseguido.



5. OpenAI baselines

Las técnicas DRL vistas en el capítulo anterior serán usadas e ilustradas de las librerías de **OpenAI**. Se trata de un laboratorio de investigación con sede en San Francisco, California. Su misión y línea de trabajo general consiste en utilizar la inteligencia artificial en beneficio de la humanidad. No solamente por ellos, sino por cualquiera, ya que brindan sus algoritmos, técnicas e implementaciones de código abierto a cualquiera que tenga la voluntad y la curiosidad para realizar sus propias investigaciones. [29].

Podemos ver los avances que han obtenido en su página web [35], siendo en general sobre inteligencia artificial, pero sobretodo relacionada con aprendizaje por refuerzo y redes neuronales.

Es una de las últimas propuestas para crear una estandarización de las implementaciones de algoritmos DRL. Sin embargo, no es la única que existe, es difícil decidir cuál es la mejor librería de todas las posibles. Se pueden encontrar otras alternativas como *RLlib*, *Stable Baselines*, *TensorForce*, *KerasRL*... Entre otras cuantas más.

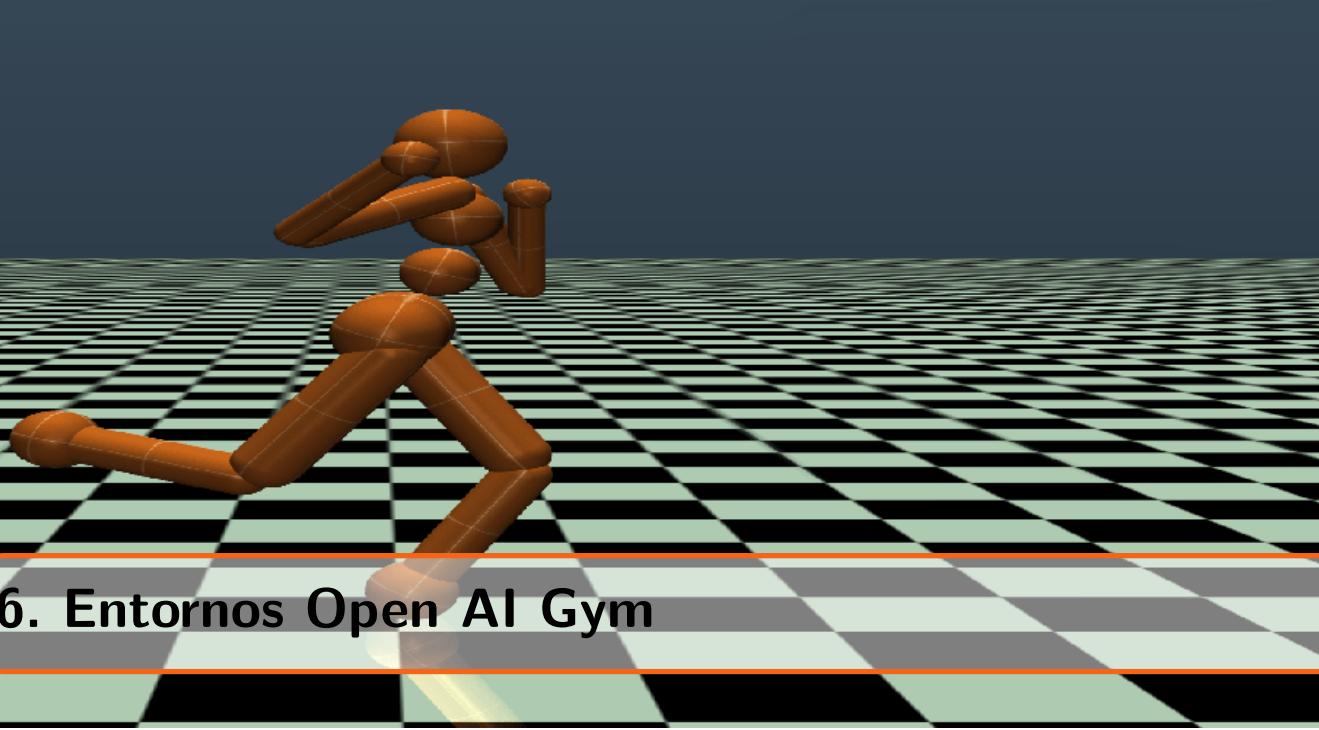
Uno de los principales motivos por los que se ha elegido a *OpenAI baselines* es su comunidad activa, la documentación y detalles de los algoritmos que implementa y la incorporación de ejemplos sencillos con la finalidad de facilitar el entendimiento de uso a los más nuevos.

Sus implementaciones están desarrolladas en Python, el cual es un lenguaje muy potente y ha demostrado buen desempeño en otros campos de IA, sin mencionar la legibilidad de su código y lo fácilmente modificable que resulta. Además, incluye funcionalidad propia para la recogida de información del proceso de entrenamiento y librerías para el tratamiento y visualización de la misma.

La compatibilidad con entornos Gym y la gran variedades de adaptaciones de otros

entornos hace que sea muy abarcable con una gran cantidad de problemas, incluso de algunos que no han sido desarrollado por la propia OpenAI, lo veremos más adelante.

Por último, destacar su estado del arte, teniendo un gran abanico de los algoritmos más novedosos, aunque en este trabajo solo se vaya a analizar varios de ellos.



6. Entornos Open AI Gym

En este apartado, se va a explicar los problemas y **entornos** elegidos para el desarrollo del proyecto.

Podríamos utilizar los algoritmos explicados para resolver **problemas del mundo real**. Por desgracia, puede ser que en la mayoría de casos nos encontremos con una serie de inconvenientes que haga más complicado poder llegar siquiera a entrenar un agente.

Por una parte, está el **presupuesto**. Supongamos un agente que sea un robot humanoide y que el problema a resolver sea obtener la capacidad de correr en entornos hostiles(piedras, agujeros en el suelo, terrenos irregulares,etc.) a una buena velocidad y sin caerse o perder el equilibrio por parte del mismo. No sería nada barato tener un agente con los sensores y características suficientes, no estaría al alcance de cualquier persona.

Por otra parte, hay que tener en cuenta el **riesgo** que supone equivocarse en un entorno real. El riesgo en este caso, sería la posibilidad de que el robot se averíe en el momento que cometa errores que le hagan caer.

Por suerte, existe una alternativa. La respuesta a estos inconvenientes son los **entornos simulados**. Simular ese mismo entorno en un **mundo virtual** y entrenar esa inteligencia desde ahí, con un agente y sensores también virtuales. Normalmente las grandes compañías dedicadas a esto parten de esos entornos antes de probarlo con robots físicos en un entorno físico, lo cual les ahorra muchos errores, dinero y tiempo.

Existe, por ejemplo, un software que permite entrenar futuros sistemas de conducción automáticos desde el entorno creado en el videojuego de RockStars, *Grand Theft Auto V*. Esto evitaría el riesgo de experimentar con peatones, vehículos e infraestructura urbana real.



Figura 6.1: Agente entrenando conducción autónoma en entorno simulado de GTA V. Extraída de vídeo en Youtube. [38]

En concreto, nosotros vamos a utilizar unos entornos predefinidos, de nuevo, por OpenAI. La librería que los implementa y ejecuta es llamada **Gym**. Se trata de un conjunto de herramientas desarrolladas en Python, al igual que los *baselines*, que brinda soporte de entornos simulados de distintos tipos para que podamos realizar pruebas con nuestros agentes.

Hay una gran diversidad de entornos, algunos son muy simples, otros más complejos; algunos no son fieles al mundo real tratando de hacerlos más simples, y otros tratan de simular la realidad lo mejor posible.

En definitiva, vamos a trabajar con dos entornos, uno simple y otro algo más complejo, con los que probar y analizar las técnicas mencionadas en los apartados anteriores. [30] [36]

6.1 MountainCar-v0

Este es un entorno muy sencillo y simple que nos va a permitir familiarizarnos tanto con los *baselines* como con *Gym*, entendiendo cómo funcionan y cómo se utilizan para crear nuestras propias pruebas y agentes con los métodos de aprendizaje por refuerzo profundo.

Una **carreta** se encuentra en una pista unidimensional, teniendo únicamente una dirección y sus dos sentidos como libertad de movimiento. La carreta se encuentra entre dos “montañas”. El objetivo es conseguir subir a la montaña que se encuentra a su derecha.

El problema es sencillo, pero no tanto como llega a aparentar, resulta que el motor que tiene esa carreta **no es suficientemente potente** como para poder subir la cuesta de esa montaña.

Solo existe una forma de resolverlo; conduciendo de un lado a otro para conseguir generar el impulso suficiente o, dicho de otra forma, un mayor momento lineal o momentum con el que la carreta conseguirá subir hasta la cima.

La idea es partir de un agente que **sabe el objetivo**, pero **no cómo se consigue**, ya que solo se le proporcionará los datos que se recogen del entorno (input), las posibles salidas que puede realizar (outputs posibles) y las recompensas en función de las acciones que realice en cada momento(rewards).



Figura 6.2: Representación gráfica del problema MountainCar v0 de OpenAI Gym.

Podemos ver los detalles técnicos en el repositorio oficial de OpenAi para Gym en Github [37]. Las **observaciones** del entorno, o lo que es lo mismo, el *input* que recibe el agente, es un contenedor con **dos datos** por observación:

- **position:** Posición en la que se encuentra la carreta. Los valores que puede alcanzar se encuentran entre $-1,2$ y $0,6$ que correspondería a los límites de izquierda y derecha en la figura 6.2 respectivamente.
- **velocity:** Alcanza valores comprendidos entre $-0,07$ y $0,07$. Representa la velocidad de la carreta en esa observación. Si el valor es negativo sería en el sentido izquierdo y si es positivo en el sentido derecho.

Por otro lado, las **acciones** o decisiones posibles que puede tomar en cada *time-step* son:

- **0 → push left:** Esta acción representada por el valor 0 activa el motor de la carreta para que trate de moverse hacia la izquierda.
- **1 → no push:** Desde el momento en el que realiza esta acción el motor permanecerá apagado hasta que realice otra acción.
- **2 → push right:** Esta acción activa el motor de la carreta para que trate de moverse hacia la derecha.

Que la acción sea *push left* no quiere decir que en la observación siguiente esté moviéndose hacia la izquierda, ya que se tiene en cuenta, no solo el motor, sino la pendiente en la que se encuentra y la velocidad que tenía en ese momento. El entorno implementa esa **física básica** que habría en el mundo real.

Ahora hablemos de la **recompensa**. La **función de recompensa** podría ser pensada

de diferentes formas. En este caso, se ha decidido que la recompensa para su entorno sea -1 en cada *time-step* si no se encuentra en la meta y 1 en caso de alcanzarla.

También tenemos que hablar del **estado inicial** del entorno. Será con la carreta **sin velocidad** y en una posición aleatoria entre -0,6 y -0,4.

Se entiende como un **estado final** en el entorno cuando han ocurrido 200 *timesteps* sin que la carreta haya alcanzado la meta o cuando la haya alcanzado en menos de esos 200 pasos.

La recompensa acumulada, por tanto, será -200 cuando no consigue su objetivo y un valor negativo mayor que -200 cuando si lo alcanza.

6.2 MountainCarContinuous-v0

El problema planteado es exactamente igual que el anterior. La diferencia reside en que las acciones que el agente puede realizar no son discretas (*push left*, *push right* o *no push*), sino continuas.

En otras palabras, la acción se representa con un **valor real**; siendo el 0 equivalente al *no push*, valores positivos trata de mover la carreta hacia la derecha y valores negativos hacia la izquierda.

La otra diferencia reside en su definición de **función de recompensa**, siendo la recompensa acumulada igual al valor 100 menos la suma cuadrática de los *time-steps* sucedidos hasta alcanzar la meta. Añadir un número máximo de *time-steps* que pueden sucederse es altamente recomendable para evitar que suceda un episodio infinito en el que no logra alcanzar la meta.

Este entorno ha sido utilizado específicamente para el algoritmo DDPG, recordemos que este algoritmo solo funciona con entornos continuos y no discretos, por lo que esta adaptación era necesaria.[33].

6.3 Super Mario Bros



Google Cloud

7. Entorno de desarrollo

Tener a disposición una mayor cantidad de recursos para poder realizar una mayor cantidad de entrenamientos y pruebas de agentes en una menor cantidad de tiempo es un factor muy beneficioso para este proyecto.

Por ello, se tomó la decisión de utilizar **infraestructura de la nube**. Tras un análisis del mercado, se ha decidido utilizar **Google Cloud** para montar una máquina virtual que pudiéramos utilizar para entrenar y guardar agentes en los entornos explicados.

Existen un buen número de plataformas que nos ofrecen una gran diversidad de servicios en la nube de buena calidad. Los usuarios solo deben de preocuparse de usarlo(y pagar lo que corresponda), dejando el resto de ámbitos a la propia empresa que abastece.

Estos ámbitos son la administración de recursos, disponibilidad, integridad de los datos, etc. En muchos aspectos estarán, como organización, mejor preparados para dar los servicios. Por ejemplo, en cuestión de seguridad, con casi total certeza, serán más robustos y estarán mejor preparados que lo que puede estar un usuario de forma individual.

De forma general y resumida, podemos decir que **Google Cloud Platform** ofrece una mejor **potencia de procesamiento**, mientras que **Amazon Web Services**(AWS) destaca más en la **capacidad de memoria**. Para nuestro proyecto interesa que el servicio de procesamiento sea lo mejor posible, ya que para los entrenamientos que realizaremos serán esenciales.

Otro de los puntos fuertes de Google Cloud es la escalabilidad y equilibrio de carga que ofrece, que permite ajustar la infraestructura de una forma más exacta a nuestras necesidades. Los precios de Google son más competitivos, aunque ofrece menos servicios que AWS. En este caso, interesa mejores precios y menos servicios siempre que los que necesitemos se encuentren.

Para más detalles de su configuración y utilización, se encuentra en los apéndices de esta documentación([A](#) y [B](#)).

7.1 Metodología de experimentación

La máquina cuenta con una mayor potencia, como es lógico, que un ordenador personal corriente. Sin embargo, la máquina virtual tiene una desventaja. Al comunicarse por SSH desde una terminal, no se dispone de la interfaz gráfica(gnome de linux) que *baselines* necesita para probar el agente de una manera visual.

Entrenamos los modelos y obtenemos los logs en la máquina virtual. Después, se traen de vuelta al equipo personal con el comando SCP para posteriormente visualizarlos desde la interfaz. El coste computacional viene casi en su totalidad en el entrenamiento, por lo que es el uso concreto que hacemos con los recursos en la nube.

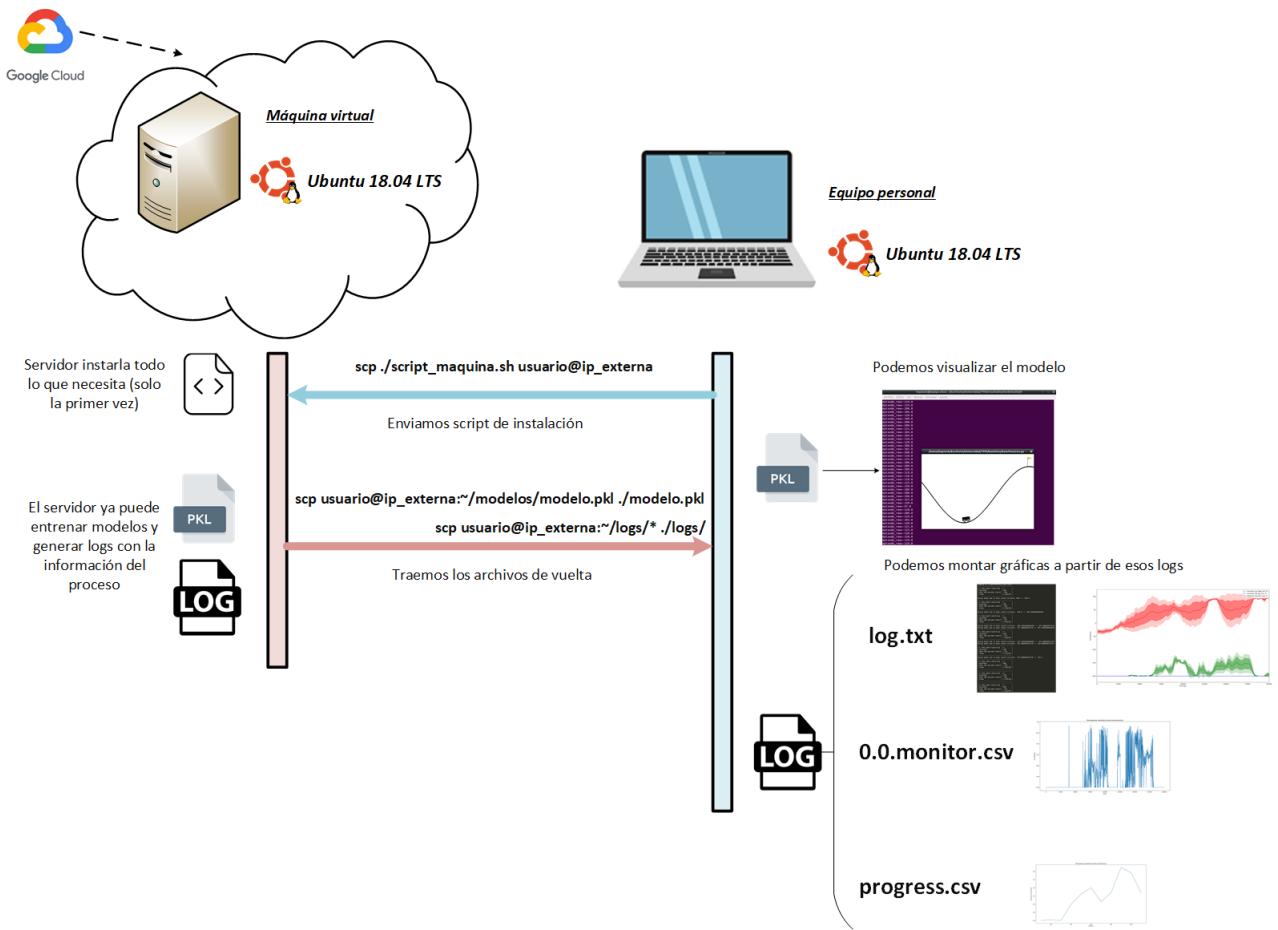
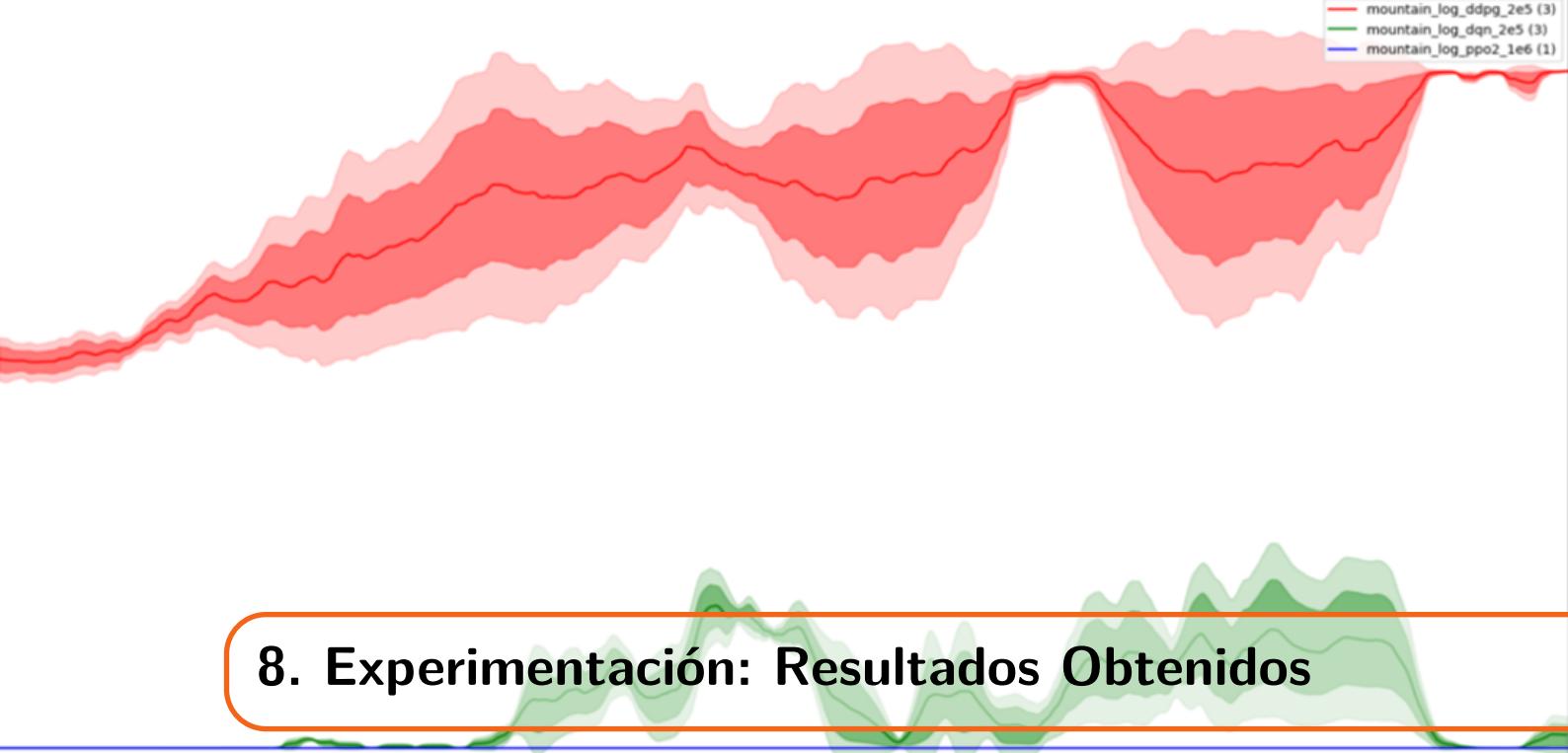


Figura 7.1: Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada



8. Experimentación: Resultados Obtenidos

En este apartado, vamos a mostrar los resultados que hemos obtenido de los algoritmos, entornos de gym y entorno de trabajo que hemos explicado en este proyecto.

Para más información sobre los logs y su visualización en el proceso de aprendizaje de los agentes que veremos a continuación, se puede consultar el apéndice de esta documentación([B.1](#) y [B.2](#)).

8.1 MountainCar-v0

Comenzamos con el problema más sencillo que hemos utilizado (apartado [6.1](#)). A pesar de su simplicidad, podemos entender que dependiendo del problema, y de como lo interpretamos para el agente, esto puede hacer que algunas técnicas no sean efectivas o que den resultados mejores o peores. Lo veremos a continuación.

8.1.1 DQN

He realizado diferentes pruebas de entrenamiento. Voy a mostrar aquellos modelos que me han parecido más interesantes. También realicé el mismo entrenamiento haciendo uso de semillas diferentes(3, 13 y 46). De esta forma, podía reentrenar el modelo en caso de que hubiera algún error y podía valorar la importancia de la aleatoriedad en cada técnica.

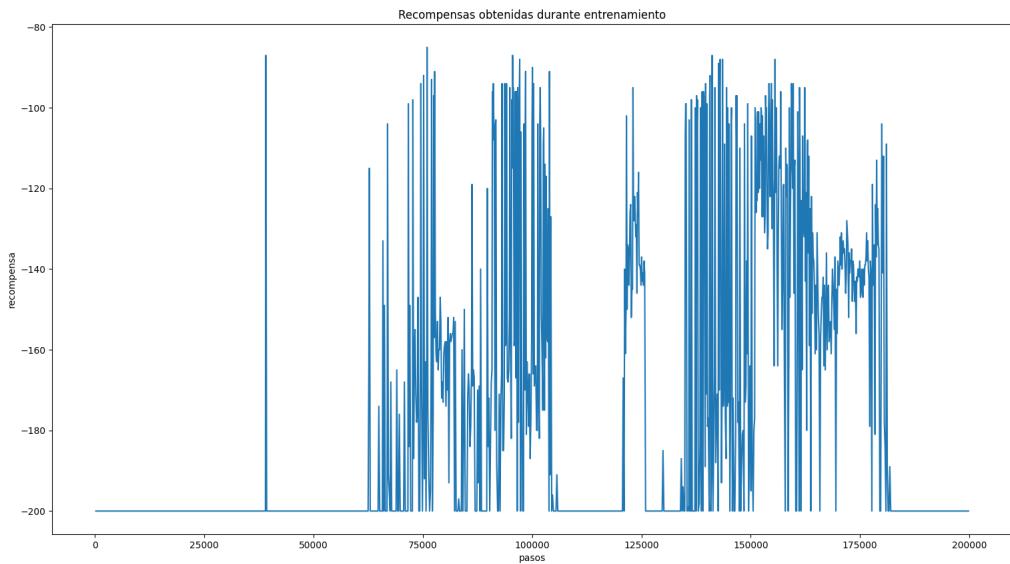


Figura 8.1: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 3 y en *MountainCar-v0*

Esta es la primera gráfica que aprendí a generar a partir de los *logs*. Con ella podemos observar las recompensas que fue obteniendo el agente en cada *timestep* que realizaba de experimentación. En el caso de la figura 8.1, vemos que consiguió resolverlo tempranamente en un caso aislado, además en pocos pasos, ya que tardó unos 90 *timesteps*(recompensa de -90). Lo cual quiere decir que aprovecho el momentum muy bien para conseguir subir.

Sin embargo, por como es la gráfica, todo apunta a que fue un caso **casual**. Realmente comienza a tener más aciertos y de una forma más continua y estable posteriormente, como se puede apreciar en los picos más juntos en las otras zonas de la gráfica. Esto me hace pensar que denota un mejor entendimiento por parte del agente de como tiene que **comportarse** para conseguir su objetivo, resolverlo con **suerte** más que con conocimiento de lo que está haciendo no podemos considerarlo como éxito, aunque tampoco estoy diciendo que la primera vez lo resuelva de forma 100 % aleatoria ya que es prácticamente imposible.

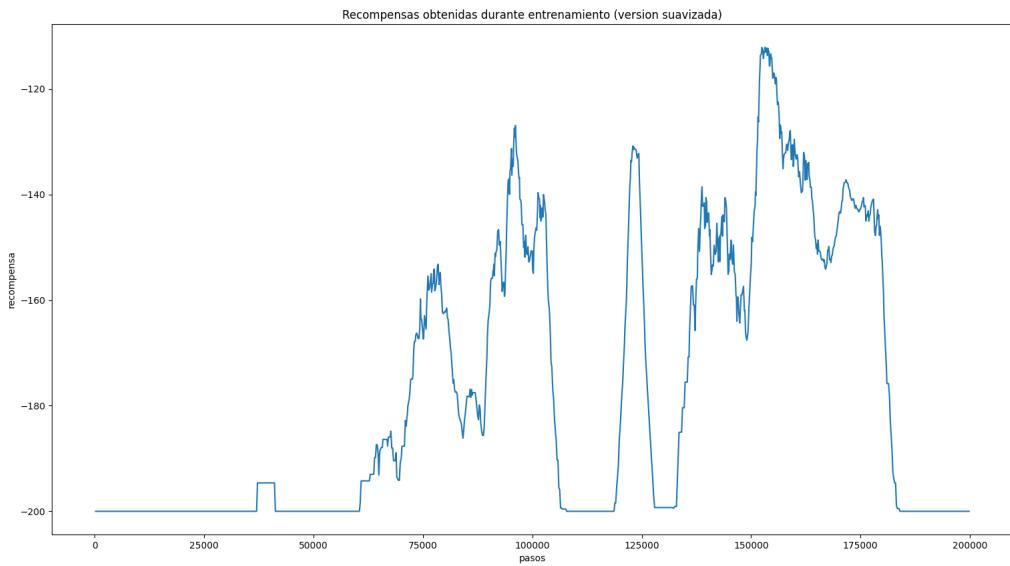


Figura 8.2: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 8.1). Algoritmo DQN, semilla 3 y en *MountainCar-v0*

El siguiente objetivo que busqué fue el tratar, de alguna manera, de suavizar aquellas subidas que no estaban tan motivadas por un buen conocimiento. Por ello, construimos una gráfica a partir de la anterior, solo que **suavizada**. Cada punto tiene un valor que depende de los valores que tiene alrededor, son influidos por sus vecinos.

El resultado es la figura 8.1. Fijémonos en el primer éxito que tiene. Ahora la subida en la recompensa es muy pequeña en comparación de las que hay más adelante, justo lo que sospechábamos.

Podemos ir más allá. En lugar de utilizar un suavizado de los datos de las recompensas, utilizar directamente las medias obtenidas por episodios, de los cuales hablamos en el apartado B.1

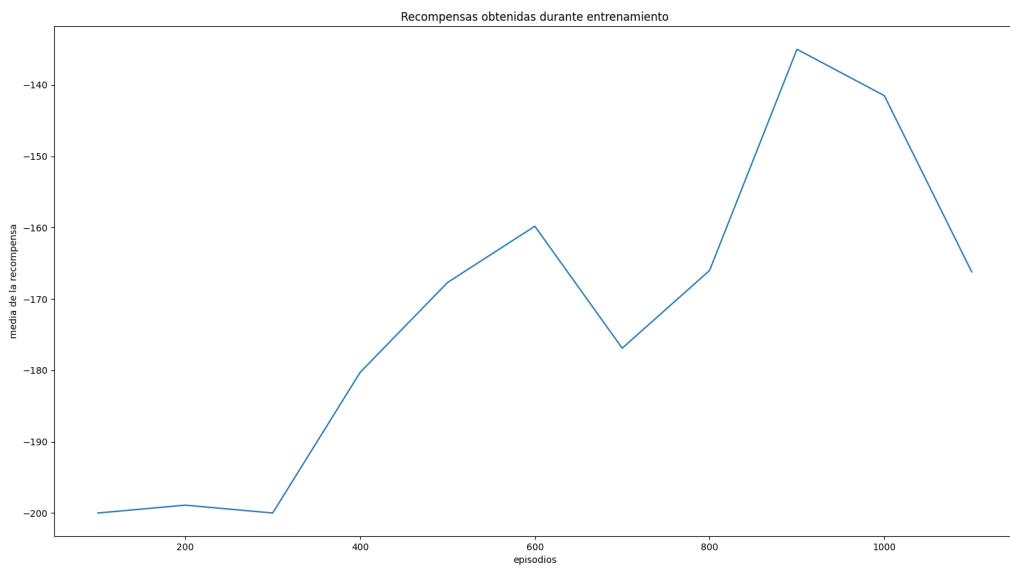


Figura 8.3: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 3 y en *MountainCar-v0*.

La figura 8.3, tiene el mismo objetivo que lo explicado anteriormente, dar más valor a aquellos éxitos que son resultado de un aprendizaje real y menos de la aleatoriedad, solo que aún más extremo.

Vemos que ahora la primera subida apenas es apreciable directamente, sin embargo, las otras dos zonas que tenía muchos éxitos son aún mas escarpadas. Como resumen de estas pruebas para la semilla 3 en DQN y *MountainCar-v0*, llegamos a la conclusión de que durante el entrenamiento pasa por **dos etapas**. Hay una primera subida general de los resultados, que luego se desploma, seguramente porque no consigue actualizarse correctamente a partir de esos éxitos y se actualiza a una versión peor de sí mismo. No obstante, luego es capaz de recuperarse cuando vuelve a comenzar a fallar y consigue resultados aún mejores. Por último, comienza otro desplome de su comportamiento, por suerte el algoritmo hace actualizaciones del modelo siempre que se considera que mejora y se va quedando guardada la **última mejor versión** registrada.

Hice las mismas pruebas para las semillas 13 y 46, muestro las gráficas a continuación:

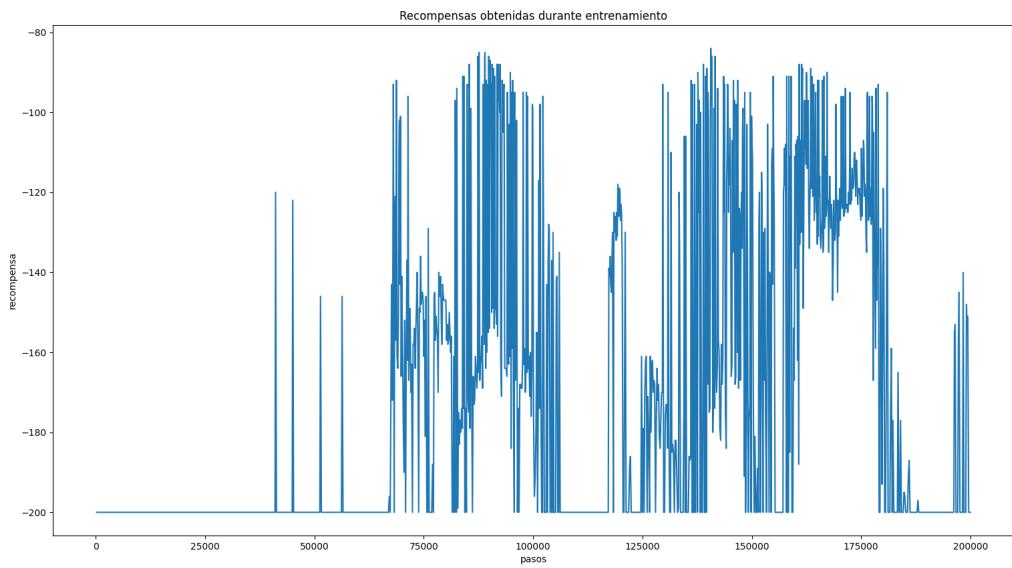


Figura 8.4: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

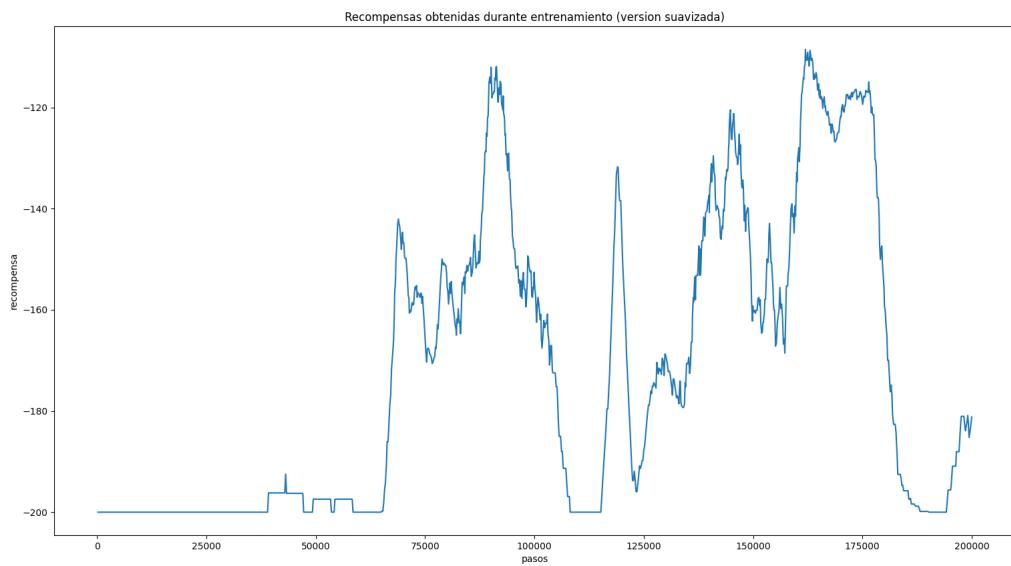


Figura 8.5: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 8.4). Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

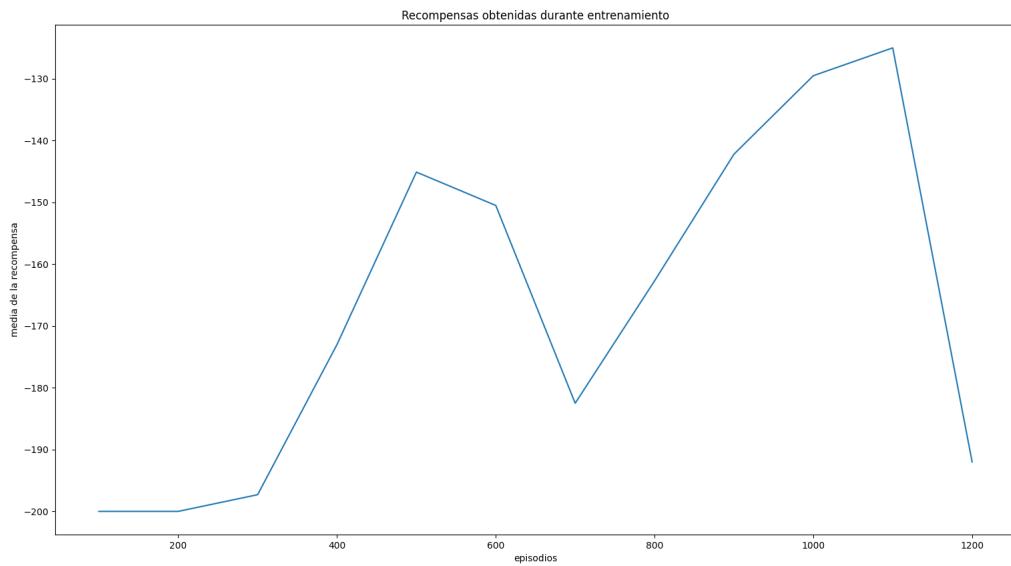


Figura 8.6: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

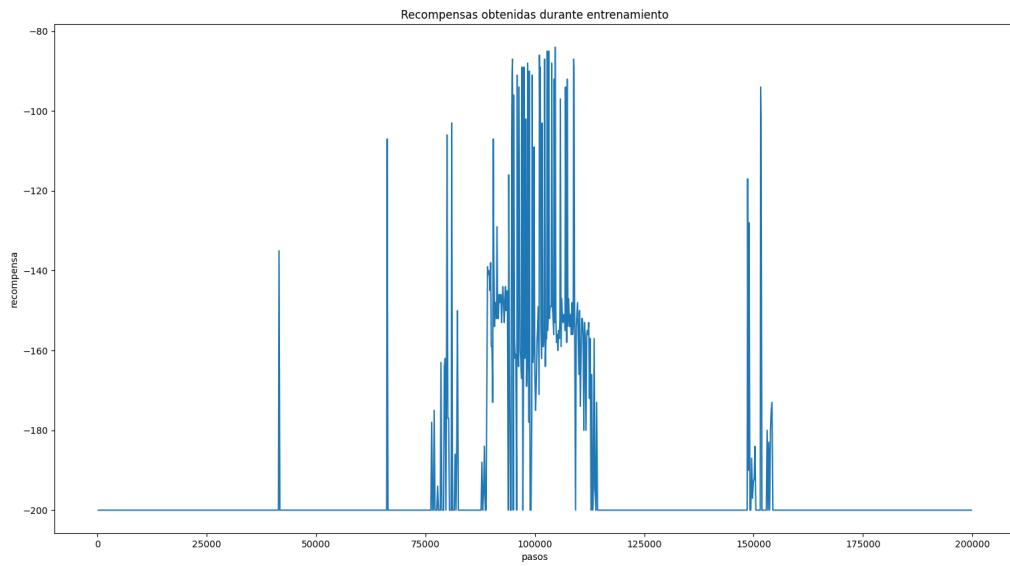


Figura 8.7: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

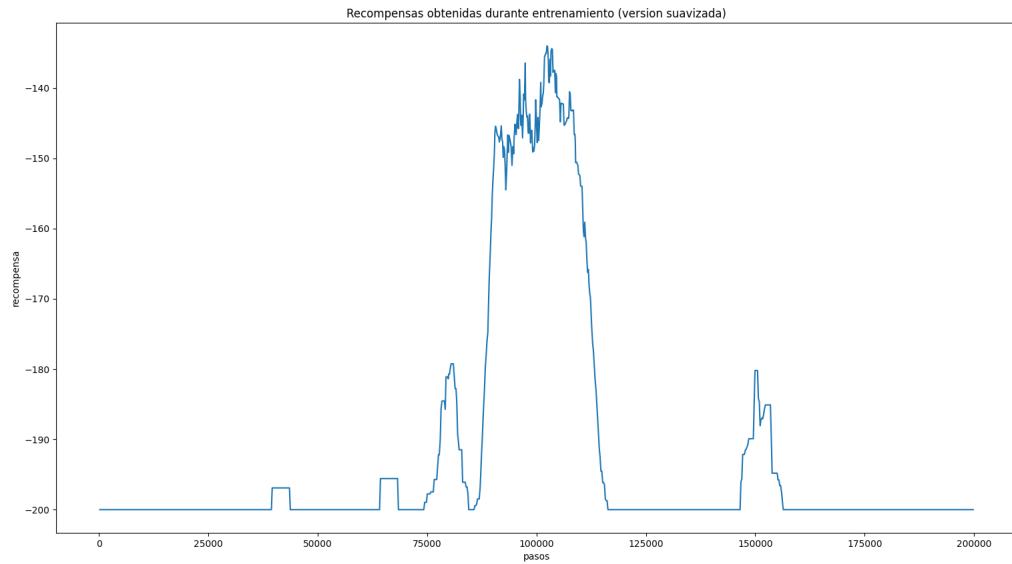


Figura 8.8: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 8.7). Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

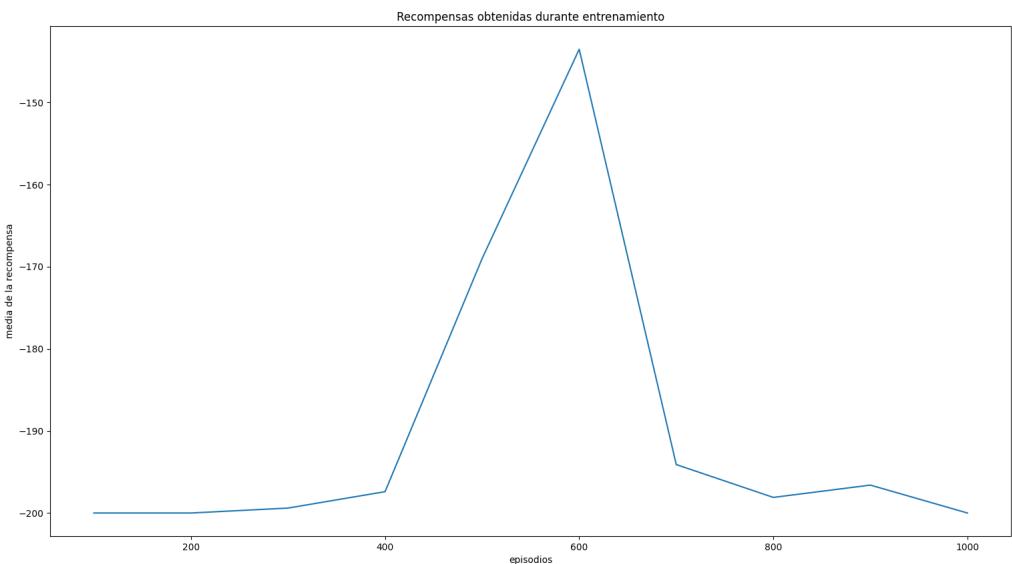


Figura 8.9: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

Vemos que en la semilla 13 tiene una **estructura similar**, mientras que la semilla 46 solo tiene una subida significativa más o menos a mitad del entrenamiento.

Cuando vi los resultados llegué a la siguiente conclusión: nos encontramos ante un problema sencillo, el cual no tiene muchas posibilidades a nivel de acciones para que el agente pueda realizar. Una vez es bueno en su entorno, con las siguientes actualizaciones es mucho más **probable que empeore** a que mejore. En este caso, solo hay una forma de hacer las cosas bien para llevar la carreta a su meta, una vez aprendida, la exploración

solo va a dar resultados negativos.

Por esa misma razón pienso que se suceden estos ciclos de subida y bajada en las recompensas. Es capaz de aprender de sus errores, pero una vez comienza a acertar no es capaz de aprender de sus éxitos de una forma tan buena. Hago hincapié en que no es preocupante, ya que siempre se guarda el modelo del mejor agente durante todo el entrenamiento, no el que queda cuando termina.

También podemos llegar a pensar; si estos algoritmos son capaces de reforzar las buenas acciones para que las repita en situaciones similares, ¿por qué en este caso no es así? Otras de las conclusiones que he sacado a partir de estos experimentos, es que considero que el sistema de **recompensas** que utiliza OpenAI para este entorno **no es óptimo** para esta técnica.

En el apartado **6.1** mencionamos que la recompensa de este entorno es siempre -1 a no ser que alcance la meta, en ese paso el valor de la recompensa es 1. Creo que este sistema no ayuda al agente de ninguna manera a determinar si sus acciones son buenas o no y solo se puede dejar llevar por el resultado final para aprender. Entonces, el agente no puede diferenciar entre buenas y malas acciones. En su lugar, refuerza todas las acciones que ha realizado cuando tiene éxito, aunque dentro tenga decisiones malas y quita relevancia a todas las decisiones tomadas durante un fracaso, aunque en ese intento haya tomado algunas buenas decisiones en realidad.

Estaría bien que la recompensa fuera positiva cuando se detectase que la carreta está aprovechando el momentum para subir, y negativa en caso contrario. No obstante, no siempre vamos a saber la mejor manera de resolver los problemas para un entorno dado, y menos cuando son mucho más complejos que éste.

Entonces, en el momento que comienza a tener éxito, sigue explorando un 2% de su tiempo como veremos a continuación. Como el problema es muy sencillo y solo hay una manera de hacerlo bien, ese 2% de exploración puede hacer que el agente erre en su intento y, como consecuencia, desfavorezca al resto de decisiones buenas que ya había aprendido a tomar.

Creo que es un ejemplo magnífico para entender la gran **importancia** en la **definición del problema** que se trata de resolver. Dependiendo de esa definición, no solo la representación de la recompensa, sino el formato de input, output y ventajas, en caso de utilizarlas, puede hacer que una técnica consiga funcionar mejor o peor al procesar la experiencia que recoge. Es como elegir la mejor función de pérdida para una red neuronal, esa decisión puede cambiarlo todo.

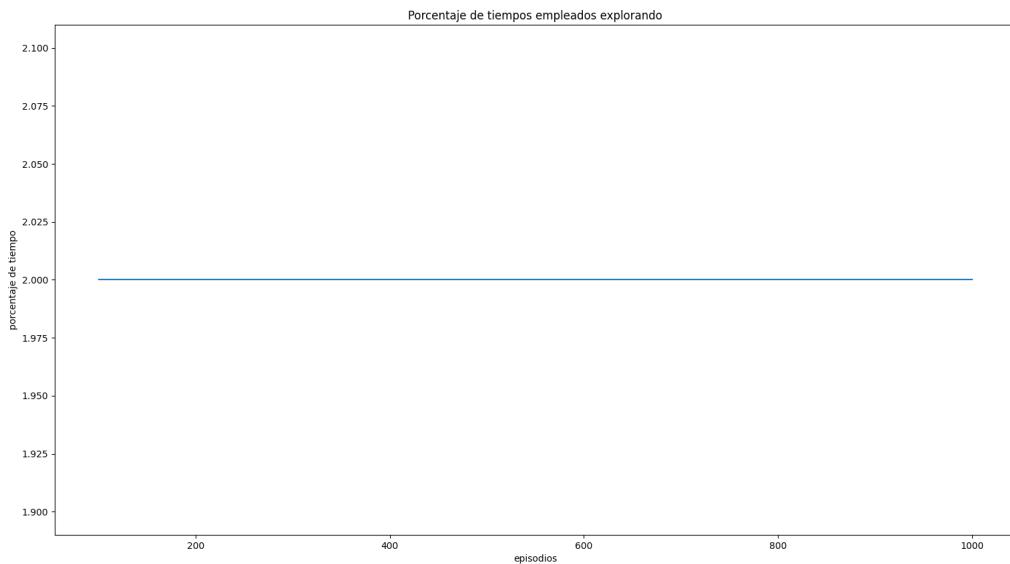


Figura 8.10: Porcentaje de tiempo empleado en explorar para algoritmo DQN en *MountainCar-v0*.

En la figura 8.10 vemos el porcentaje de tiempo empleado por el agente para explorar. Como ya hemos mencionado, no es mucho debido a la simplicidad del entorno y que explorar otras alternativas a las que el agente piensa no va a ser bueno. Sería interesante incluso implementar un sistema que redujera ese porcentaje a 0 cuando se superara una cierta recompensa. Estoy convencido de que esto mejoraría aún más los resultados o, al menos, estabilizaría más el aprendizaje. Por desgracia, no he tenido el suficiente tiempo para hacerlo.

¿Hasta qué punto influye la aleatoriedad en el progreso de su aprendizaje con esta técnica? Por esta cuestión es por a que he ejecutado la misma técnica con semillas diferentes. Podemos hacer una gráfica que resuma todos los datos que hemos visto anteriormente en una sola. Mezclando los tres modelos diferentes y plasmarlos como uno solo:

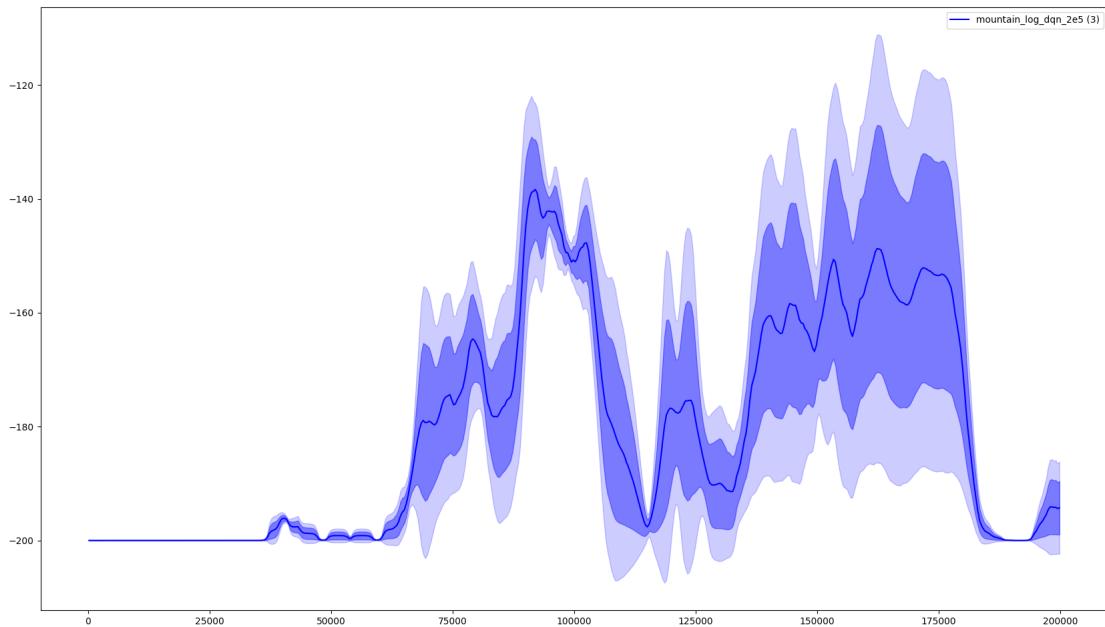


Figura 8.11: Resumen general de progreso de algoritmo DQN en *MountainCar-v0*.

El tono más claro muestra la desviación estándar de los datos. El tono más oscuro el error en la estimación de la media. En otras palabras, la desviación estándar dividida por la raíz cuadrada del número de semillas que tenemos. También incluye un suavizado para que la gráfica sea más estable.

La forma más simple de interpretarlo para que se entienda lo que acabo de explicar; cuanto más estrecha sea las áreas sombreadas alrededor de la línea, más coincide el progreso en ese punto para los tres modelos diferentes que hemos visto.

Entonces, tenemos zonas más relevantes en la gráfica de la figura 8.11. Al principio coincide de forma perfecta que los tres modelos no son capaces de cumplir su objetivo, luego comienzan a tener algunos éxitos puntuales. La primera mejora sustancial de los datos es bastante común en el mismo punto del aprendizaje y después es un poco más variable de la aleatoriedad. Sin embargo, al final, hay una gran coincidencia de nuevo; todos empeoran hasta no ser capaces ni siquiera de cumplir el objetivo de alcanzar la meta y luego comienzan a remontar, pero aquí se acaba la gráfica. Esto me hace pensar que las conclusiones expresadas anteriormente son bastante firmes en cuanto a tiempo, sucediéndose en momentos similares en tres casos distintos.

Como conclusión, diría que esta gráfica es muy útil para saber qué puntos del aprendizaje, como la primera mejora real de resultados o el empeoramiento final, no son dependientes del azar, sino que vienen muy determinados directamente por el algoritmo y el entorno. Por otro lado, también nos ayuda a entender qué partes del entrenamiento son más influenciables por el azar, como hemos podido observar.

8.1.2 PPO2

Con esta técnica para este entorno no tenemos mucho que decir, por desgracia. Los resultados obtenidos fueron un completo **fracaso**. Por alguna razón, no consiguió siquiera

resolver el problema que se le planteaba ni una sola vez. Se probó a quintuplicar la cantidad de *timesteps* de entrenamiento (hasta el millón), no hubo ningún cambio.

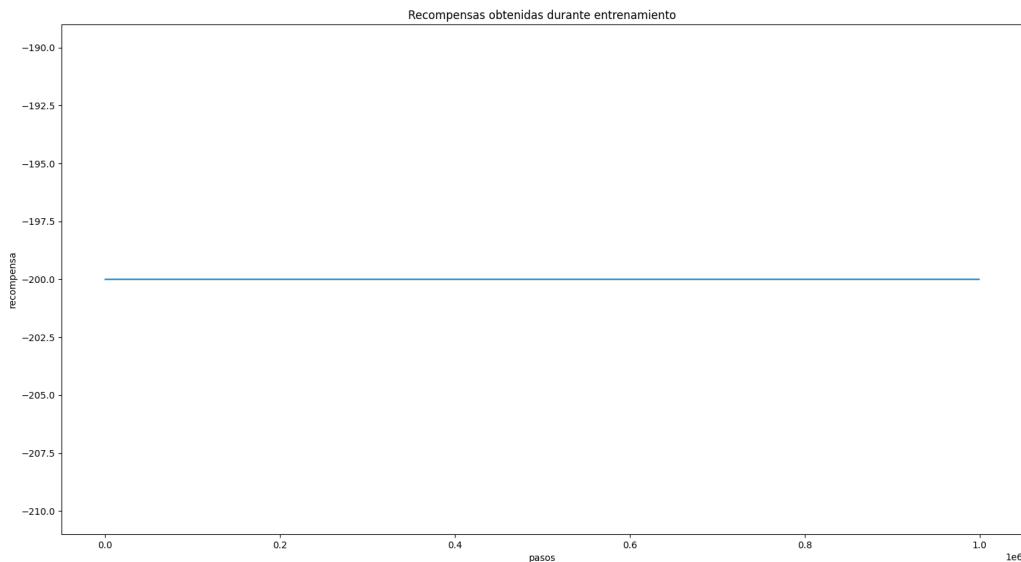


Figura 8.12: Recompensa obtenida con el algoritmo PPO2 en *MountainCar-v0*, independientemente de la semilla iteraciones y otros parámetros.

Tras realizar algunas pruebas y leer un poco las opiniones de los demás por Internet, conseguí darme cuenta de lo que estaba ocurriendo. El **problema** reside de nuevo en la forma de **representar la recompensa** por parte del entorno de Gym para el agente que estamos entrenando, el cual le viene especialmente mal a PPO2.

MountainCar solo consigues un valor positivo en la recompensa cuando alcanzas la meta, y -1 para el resto de casos. Recordemos que PPO2 es un algoritmo basado en actualizar los parámetros de la política utilizando el gradiente para ello.

Conseguir el momentum necesario para que suba la carretilla por la montaña es algo que rara vez va a ocurrir con un agente totalmente aleatorio, sin partir de un agente con una mayor base aunque introduzca aleatoriedad en su proceso.

Cuando consiga llegar a la meta, si es que ocurre en algún momento, no creo que con una sola actualización de sus parámetros para reforzar las acciones que ha realizado sea suficiente como para acertar mucho más en el futuro, por lo que el agente va a seguir atascado en ese proceso de nunca conseguir su objetivo

En definitiva, no cuenta con esa **memoria** que tiene DDPG, por ejemplo. Además, no valora lo prometedores que son las acciones en la solución como si hace DQN o DDPG, centrándose únicamente en los resultados una vez se consiguen, cosa que nunca llega (ver apartado 3 y ??).

Gracias a estos resultados, fui capaz de aprender que PPO2 funcionará mejor cuando queramos mejorar un modelo que ya es capaz de alcanzar el objetivo (obtenido con otra

técnica) o, al menos, que sea factible resolverlo a menudo con un agente totalmente aleatorio, no siendo este el caso.[22]

8.1.3 DDPG

Este algoritmo ha dado problemas desde el principio para poder ejecutarlo, aunque debo reconocer que ha merecido la pena dado que ha sido el que mejor resultados ha logrado. Como mencionamos en el apartado ??, este algoritmo trata de explotar los puntos fuertes de *Q-Learning* y de *Política de gradientes* al mismo tiempo. En este caso, se ha notado.

Vamos a comentar primero los problemas que he encontrado a la hora de poder utilizarlo. Resulta que los *baselines* de OpenAI **fallan** a la hora de guardar el modelo entrenado. Si no podemos guardar el modelo, no nos sirve de nada.

Estuve investigando mucho por Github, estudiando implementaciones forkeadas de otros usuarios que estaban tratando de solucionarlo. Incluso traté de modificar las implementaciones yo mismo, llegando a conseguir que guardase el modelo, aunque no guardaba parámetros importantes referentes al ruido, explicado en este trabajo. Haciendo que el modelo guardado no fuese el mismo que el entrenado, perdiendo parte de su esencia y, por tanto, calidad.[43]

Se trata de un error que llevan arrastrando desde 2017. No debe ser trivial resolverlo con esas librerías, dado que hay unos cuantos usuarios discutiendo sobre ello y creen que la mejor opción es utilizar otras implementaciones de distintas fuentes.

Finalmente, terminé cediendo y buscando otras alternativas. Encontré **Stable Baselines** y me pareció la mejor opción, dado que podía seguir usando parte de la configuración realizada de TensorFlow y tarjeta gráfica sin apenas hacer nada.

Resultados

Vamos a ver los resultados obtenidos con este algoritmo. Como he mencionado anteriormente, la información de *progress.csv* no va a poder ser mostrada. Aunque este inconveniente no nos afecta a la hora de saber el desempeño referente a calidad de resultados por parte del agente, que es lo que más nos interesa.

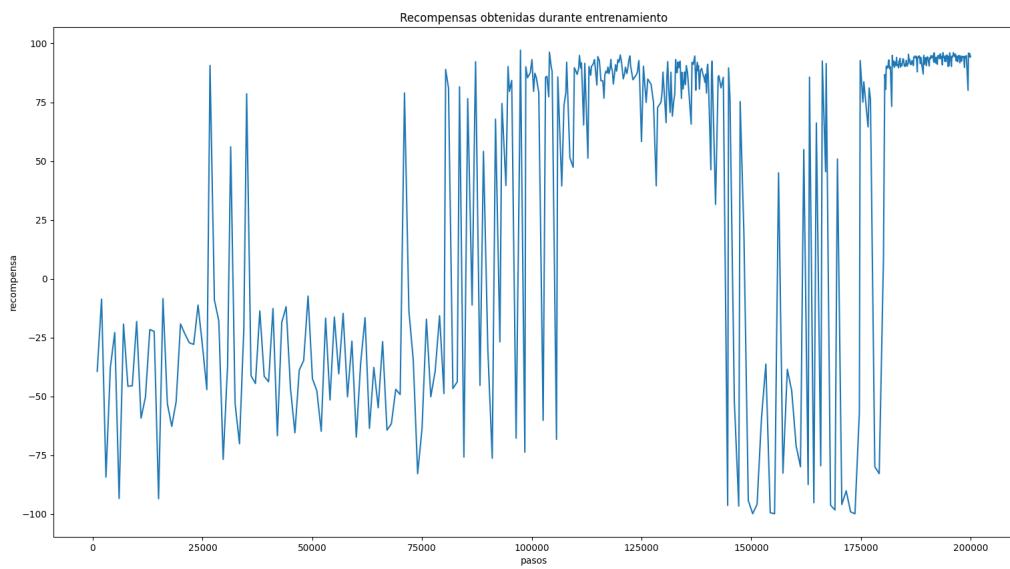


Figura 8.13: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 3 y en *MountainCar-v0*.

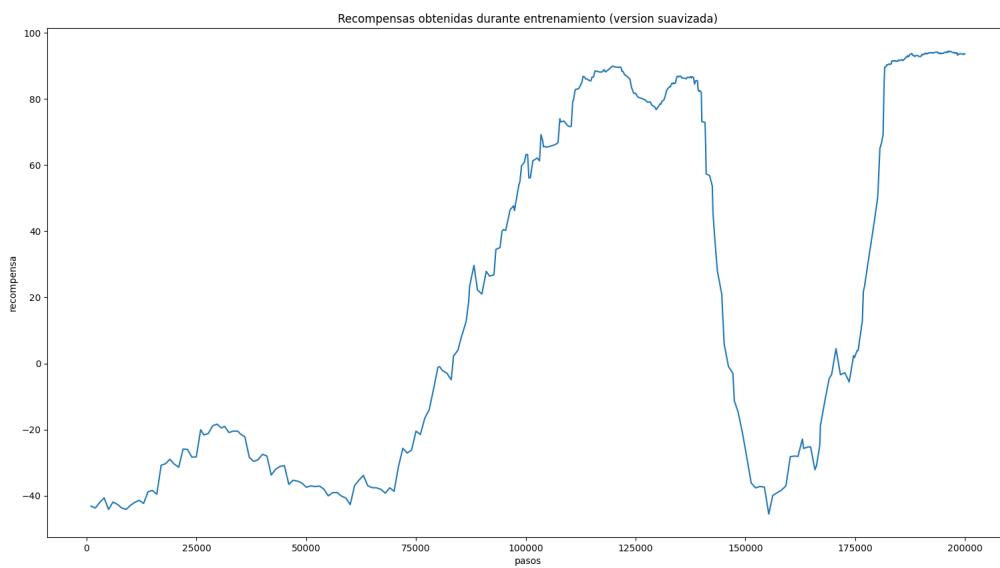


Figura 8.14: Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3 y en *MountainCar-v0*.

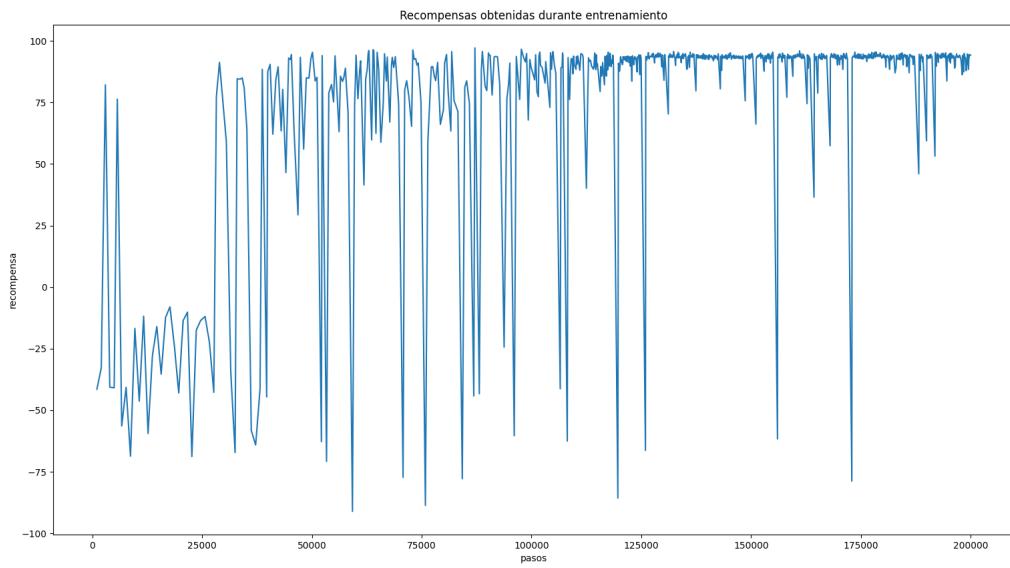


Figura 8.15: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 13 y en *MountainCar-v0*.

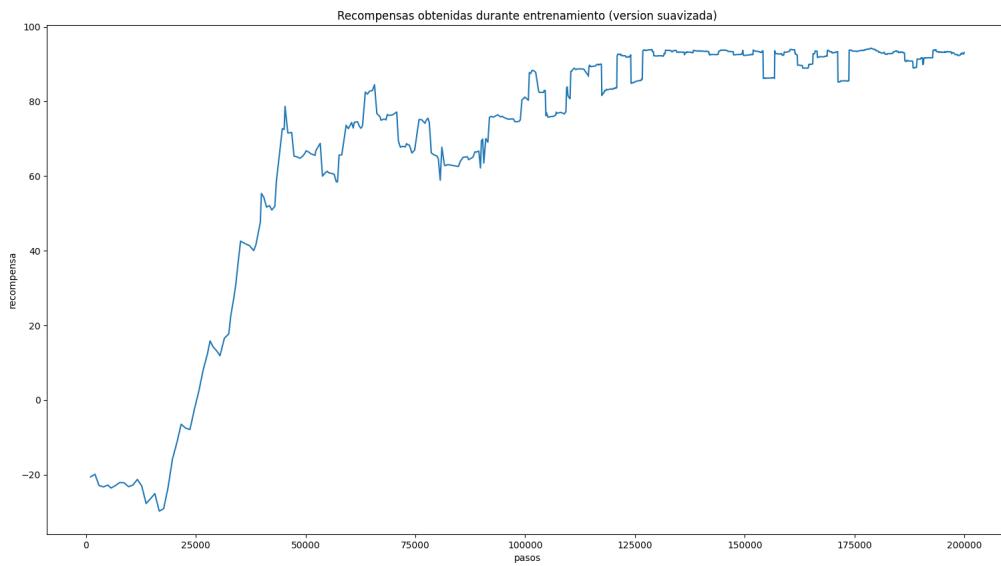


Figura 8.16: Gráfica de recompensas obtenidas durante entrenamiento (suavizadas). Algoritmo DDPG, semilla 13 y en *MountainCar-v0*.

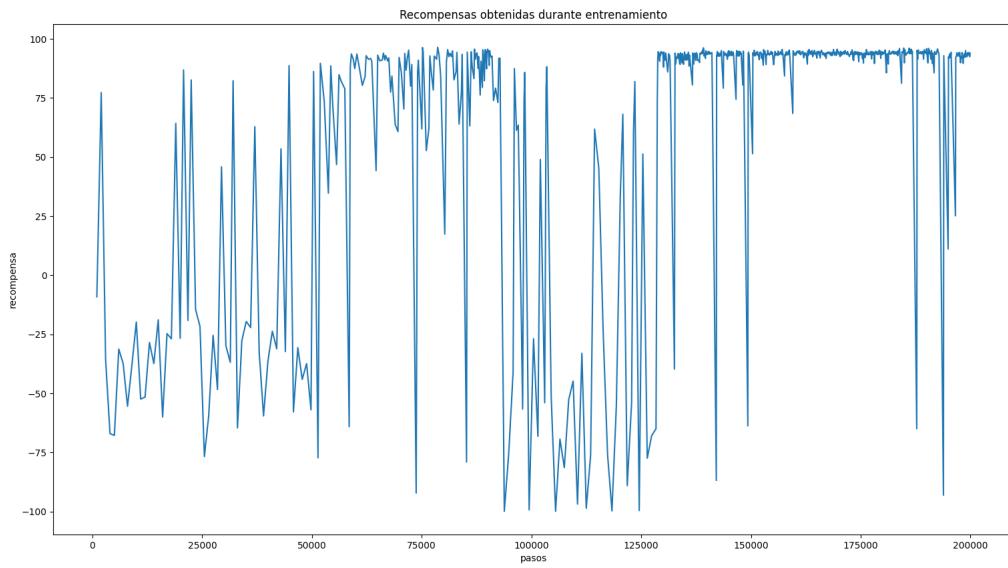


Figura 8.17: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 46 y en *MountainCar-v0*.

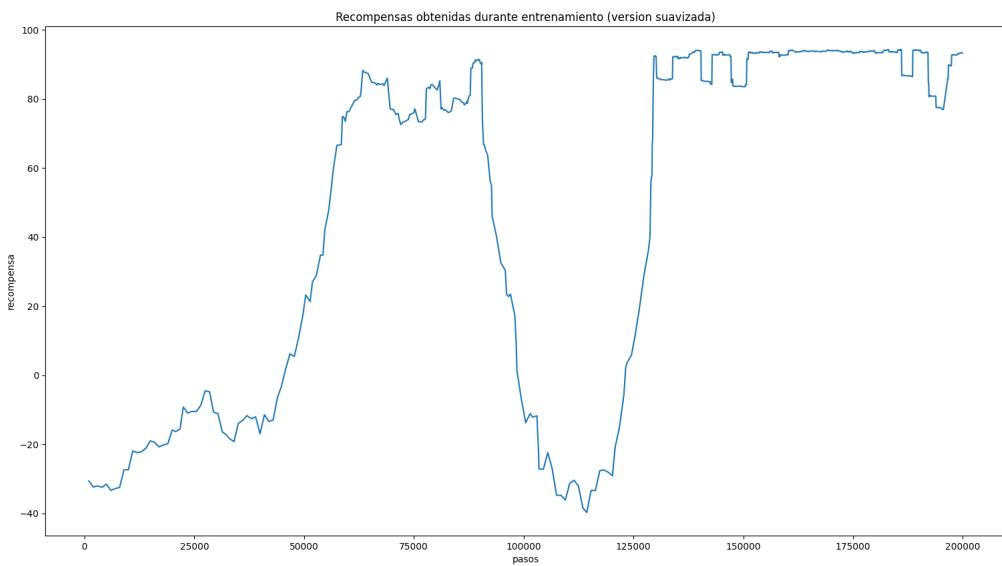


Figura 8.18: Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 46 y en *MountainCar-v0*.

Recordemos que este algoritmo se está ejecutando en la versión continua de MountainCar-v0 visto en el apartado 6.2. Por ello, el rango de recompensas no es el mismo al calcularse de diferente forma.

A simple vista, vemos que las gráficas son mucho más estables que en DQN, sobretodo para la semilla 13 en la figura 8.16. Recordemos que tiene dos partes, una política que decide cual es la siguiente acción y un Q-Learning en otra red neuronal que evalúa como de buena es esa decisión y supervisa su desempeño.

En cuestiones de resultados ha conseguido **mejores resultados** que DQN, resolviendo más veces y en menos pasos el momentum de la carreta para alcanzar la cima. Creo que DDPG resuelve mejor el problema que teníamos cuando el agente comenzaba a acertar gracias a su **ruido**.

Si recordamos el apartado ??, el ruido es introducido directamente en los parámetros de la red neuronal y no en las acciones posibles. Esto quiere decir que ese ruido no tiene porqué modificar de forma directa las acciones que realiza, simplemente un poco su comportamiento.

Creo que el nuevo **sistema de recompensas** también influye en la forma de aprender del agente una vez entiende como se resuelve el problema, ya que afecta de forma directa en la experiencia. Aunque vemos que puede experimentar alguna caída, se recupera rápidamente.

Vamos a ver como afecta la aleatoriedad a este algoritmo:

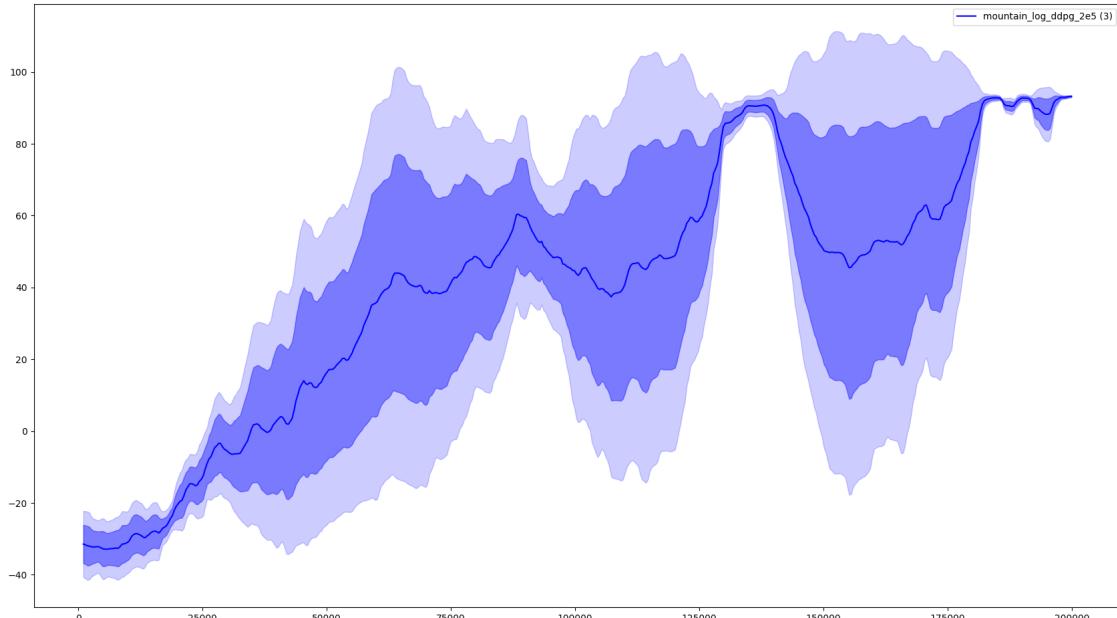


Figura 8.19: Gráfica general obtenida durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3, 13 y 46 simultáneamente y en *MountainCar-v0*.

Vemos que el ruido que implementa este algoritmo afecta en prácticamente todo el proceso de aprendizaje excepto en dos zonas muy concretas. En general las caídas de eficacia no suelen tener un orden estricto, aunque en los tres modelos cuadra bastante bien el punto a partir del cual comienzan a estabilizarse en los resultados que ofrece.

Por último, se me ocurrió la idea de plantear los resúmenes de los tres algoritmos en una misma gráfica. Sin embargo, para este entorno no tiene mucho sentido; DQN y DDPG tienen escalas de recompensas diferentes, mientras que PPO2 no ha dado buenos resultados. Igualmente voy a mostrarla, así podemos apreciar el progreso de cada uno indistintamente de la escala de recompensas.

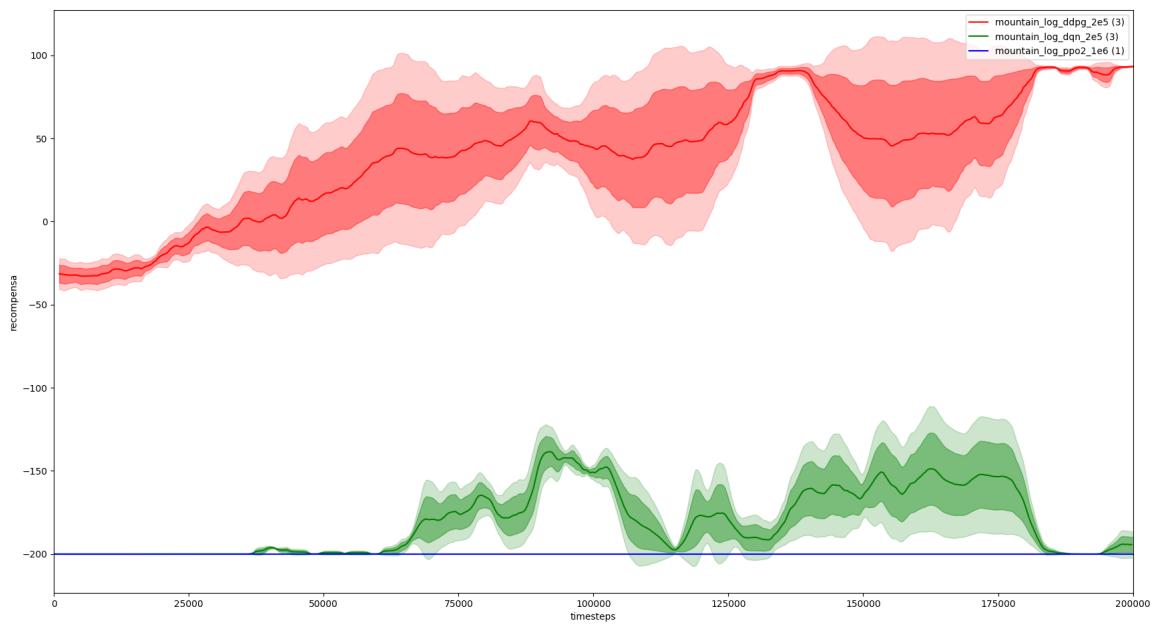


Figura 8.20: Gráfica comparativa de los tres algoritmos: DQN, PPO2 y DDPG en *MountainCar-v0*.

8.2 Otro entorno más complejo (Aún sin decidir)

8.2.1 DQN

8.2.2 PPO2

8.2.3 DDPG

IV

Estado del arte

9	AlphaGO	93
10	AlphaStar	95

Introducción

Hasta ahora hemos podido entender, desarrollar y probar algunas técnicas de aprendizaje por refuerzo profundo. Es posible que a estas alturas hayas podido llegar a pensar, ¿qué utilidad tiene esto? ¿De qué sirve crear un agente capaz de jugar muy bien a un juego?

Quizás es cierto que resolver videojuegos o juegos competitivos en general no aporta nada útil a la humanidad como tal. Pero hay que entender que, la curiosidad es lo que nos ha hecho tan grandes. Las investigaciones y avances tecnológicos que nacen de querer resolver estos problemas pueden conducirnos a cosas más interesantes y ambiciosas en el futuro. Escalando a partir de los avances que se van produciendo gracias a este trabajo.

AlphaGo Zero

Discovering new knowledge

9. AlphaGO



10. AlphaStar

V

Posibles mejoras y conclusiones

11	Conclusiones finales	99
11.1	Posibles mejoras	
11.2	Conclusiones	
	Bibliografía	101
	Artículos	
	Libros	



11. Conclusiones finales

11.1 Posibles mejoras

11.2 Conclusiones



Bibliography

Articles

- [1] Claudio Amorim. “Imagen de red neuronal”. En: . (consultado en 2020). <https://artfromcode.wordpress.com/2017/04/18/red-neuronal-en-python-con-numpy-parte-1/> (véase página 23).
- [2] Anaconda. “Who is Anaconda?” En: *Página oficial* (2020). <https://www.anaconda.com/about-us> (véase página 114).
- [4] Stable Baselines. “DDPG”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/modules/ddpg.html> (véase página 125).
- [5] Stable Baselines. “Examples”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/examples.html> (véase página 125).
- [6] Stable Baselines. “Welcome to Stable Baselines docs! - RL Baselines Made Easy”. En: *Github Docs* (2020). <https://stable-baselines.readthedocs.io/en/master/> (véase página 123).
- [8] Fernando Sancho Caparrini. “Ejemplo de gradiente descendente estocástico.” En: *Imagen* (consultado en 2020). <http://www.cs.us.es/~fsancho/?e=165> (véase página 28).
- [9] Shiyu Chen. “Tutorial: Installation and Configuration of MuJoCo, Gym, Baselines”. En: *www.chenshiyu.top* (2019). <https://www.chenshiyu.top/blog/2019/06/19/Tutorial-Installation-and-Configuration-of-MuJoCo-Gym-Baselines/> (véase página 115).
- [11] Irene Alvarado engineer y creative technologist. “redes Neuronales”. En: *Github* (Consultado en 2020). https://ml4a.github.io/ml4a/es/neural_networks/ (véanse páginas 24-26).
- [12] FileInfo. “.PKL File Extension”. En: *Página oficial* (2020). <https://fileinfo.com/extension/pkl> (véase página 118).

- [13] gcpping. "Measure your latency to GCP regions". En: *gcpping.com* (2020). <http://www.gcpping.com/> (véase página 110).
- [14] Docs Github. "Installation". En: *Stable-Baselines Official* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/install.html> (véase página 115).
- [15] Google. "DeepMind". En: *Página web oficial* (2020). <https://deepmind.com/> (véase página 16).
- [16] Google. "Descripción general de la herramienta de línea de comandos de gcloud". En: *Google Cloud CLI* (2020). <https://cloud.google.com/sdk/gcloud?hl=es-419> (véase página 112).
- [17] Red Hat. "Provisioning Ansible". En: *Red Hat Ansible* (2020). <https://www.ansible.com/use-cases/provisioning> (véase página 113).
- [18] Iberdrola. "¿Qué es la Inteligencia Artificial?" En: <https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial> (2020) (véase página 13).
- [20] Linuxize. "How to Use SCP Command to Securely Transfer Files". En: *Página de Linuxize* (2019). <https://linuxize.com/post/how-to-use-scp-command-to-securely-transfer-files/> (véase página 115).
- [21] Ludoteka. "Go". En: *Ludoteka.com* (Consultado en 2020). <http://www.ludoteka.com/juego-go.html> (véase página 21).
- [22] LupusPrudens. "PPO struggling at MountainCar whereas DDPG is solving it very easily. Any guesses as to why?" En: *Reddit* (2019). https://www.reddit.com/r/reinforcementlearning/comments/9o8ez0/ppo_struggling_at_mountaincar_ (véase página 82).
- [23] James McCaffrey. "Clasificación y predicción con el uso de redes neuronales". En: *Microsoft docs* (2016). <https://docs.microsoft.com/es-es/archive/msdn-magazine/2012/july/test-run-classification-and-prediction-using-neural-networks> (véase página 23).
- [24] Marcos Merino. "La inteligencia artificial AlphaStar se proclama 'gran maestro' de Starcraft II en igualdad de condiciones frente a los humanos." En: *Xataka* (2019). <https://www.xataka.com/inteligencia-artificial/inteligencia-artificial-alphastar-se-proclama-gran-maestro-starcraft-ii-igualdad-condiciones-frente-a-humanos> (véase página 16).
- [26] Carlos García Moreno. "¿Qué es el Deep Learning y para qué sirve?" En: *Indra Blog Neo* (2020). <https://www.indracompany.com/es/blogneo/deep-learning-sirve> (véase página 15).
- [27] David Naranjo. "¿Cómo instalar los drivers de video Nvidia en Ubuntu 18.04?" En: *Ubunlog* (2020). <https://ubunlog.com/como-instalar-los-drivers-de-video-nvidia-en-ubuntu-18-04/> (véase página 115).
- [28] Netflix. "Alphago". En: *Documental* (2017). <https://www.alphagomovie.com/> (véase página 16).
- [29] OpenAI. "About OpenAI". En: *Página web oficial* (2020). <https://openai.com/about/> (véase página 63).
- [30] OpenAI. "Gym". En: *Página web oficial* (2020). <https://gym.openai.com/> (véanse páginas 16, 66).
- [31] OpenAI. "https://gym.openai.com/envs/MountainCar-v0/". En: *Entornos Gym* (2020). <https://gym.openai.com/envs/MountainCar-v0/>.

- [32] OpenAI. “Loading and visualizing results (open in colab)”. En: *Github* (2020). <https://github.com/openai/baselines/blob/master/docs/viz/viz.ipynb> (véase página 122).
- [33] OpenAI. “MountainCarContinuous v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCarContinuous-v0> (véase página 68).
- [34] OpenAI. “OpenAI baselines”. En: *Repositorio Github oficial* (2020). <https://github.com/openai/baselines> (véanse páginas 114, 117).
- [35] OpenAI. “Progresos de OpenAI”. En: *Página web oficial* (2020). <https://openai.com/progress/> (véase página 63).
- [36] OpenAI. “Table of environments”. En: *Github* (2020). <https://github.com/openai/gym/wiki/Table-of-environments> (véase página 66).
- [37] OpenAi. “Detalles técnicos del entorno MountainCar v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCar-v0> (véase página 67).
- [38] Craig Quiter. “DeepDrive Universe”. En: *Youtube* (2017). https://www.youtube.com/watch?time_continue=26&v=X4u2DC0LoIg&feature=emb_logo (véase página 66).
- [40] Juan Gómez Romero. “Curso sobre Aprendizaje Profundo por Refuerzo en ECI 2019”. En: *Github* (2019). <https://github.com/jgromero/eci2019-DRL> (véanse páginas 23, 31, 44, 47, 49, 52, 53, 55-57).
- [41] Juan Gómez Romero. “Sistemas Inteligentes para la Gestión de la Empresa. Tema 4: Modelos avanzados Deep Learning”. En: *Universidad de Granada* (2019) (véase página 23).
- [42] Crustian Rus. “‘AlphaGo’ es el documental de Netflix que mejor explica lo que supuso la victoria de la IA de Google al campeón de Go.” En: *Xataka* (2018). <https://www.xataka.com/cine-y-tv/alphago-es-el-documental-de-netflix-que-mejor-explica-lo-que-supuso-la-victoria-de-la-ia-de-google-al-campeon-de-go> (véase página 16).
- [43] Sumsamkhan. “Saving and restoring DDPG agent”. En: *Github issue* (2017). <https://github.com/openai/baselines/issues/162> (véase página 82).
- [44] Tensorflow. “Why TensorFlow”. En: *Página oficial* (2020). <https://www.tensorflow.org/about> (véase página 114).
- [45] Jacques Thibodeau. “Cómo solicitar aumento de cuota de GPU en Google Cloud”. En: *it-swarm* (2017). <https://www.it-swarm.dev/es/google-cloud-platform/como-solicitar-aumento-de-cuota-de-gpu-en-google-cloud/833474100/> (véase página 111).
- [46] Kapil Varshney. “How to Setup Ubuntu 16.04 with CUDA, GPU, and other requirements for Deep Learning”. En: *Medium* (2018). <https://medium.com/@kapilvarshney/how-to-setup-ubuntu-16-04-with-cuda-gpu-and-other-requirements-for-deep-learning-f547db75f227> (véase página 115).
- [47] Oriol Vinyals. “AlphaStar: Mastering the Real Time Strategy Game StarCraft II - Oriol Vinyals”. En: *Youtube* (2019). <https://www.youtube.com/watch?v=3UdH3lPF7nE> (véase página 16).
- [48] Wikipedia. “Gráfica de aprendizaje por refuerzo.” En: *Imagen* (consultado en 2020). https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo (véase página 32).

Books

- [3] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning*. Editado por . MIT Press, 2018 (véanse páginas 31, 55).
- [7] Fernando Berzal. *Redes Neuronales y Deep Learning*. Editado por . <https://sites.google.com/view/redes-neuronales/>. Amazon, 2019 (véase página 23).
- [10] Stable Baselines Contributors. *Stable Baselines Documentation*. Editado por Release 2.10.1a0. <https://readthedocs.org/projects/stable-baselines/downloads/pdf/master/>. Página oficial, 2020 (véase página 123).
- [19] Max Pumperla y Kevin ferguson. *Deep Learning and The Game of Go*. Manning, 2019 (véase página 13).
- [25] Miguel Morales. *Deep Reinforcement Learning*. Editado por Manning. Grokking, 2018 (véanse páginas 31, 36).
- [39] Ismael Pérez Roldán. *Clasificación de Obras de Arte Por Estilo Artístico Usando Redes Neuronales Convolucionales*. http://oa.upm.es/56163/1/TFG_ISMAEL_PEREZ_ROLDAN.pdf. Universidad Politécnica de Madrid, 2019 (véase página 23).

VI

Apéndice

A	Configuración en la nube	107
A.1	Configuración de la Infraestructura	
A.2	Configuración de la Máquina	
B	Uso de la Máquina Virtual	117
B.1	Archivos generados en los logs	
B.2	Visualizar la información de los logs	
C	Stable Baselines	123
C.1	Generar un Modelo	
C.2	Cargar un modelo	

A. Configuración en la nube

Mi tutor, Juan Gómez Romero, me proporcionó 50 dólares en esta plataforma para que pudiera montar y utilizar una máquina virtual a mi gusto, en función de mis necesidades. A continuación, voy a explicar como monté esa máquina virtual que he estado utilizando para realizar la experimentación de este proyecto.

Hay varios factores a tener en cuenta y solucionar. En primer lugar, decidir la región para el servicio de infraestructura. Aprender a utilizar el cliente de Google Cloud Platform (en el Máster había usado otras plataformas como Microsoft Azure, por lo que no tenía pleno dominio). También hay que tener en cuenta la **configuración** de la máquina para que funcionase los *baselines*, *Gym*, etc. Puede presentar diferencias con la configuración realizada en local en mi ordenador personal. Hay que configurar los drivers y las librerías necesarias para hacer uso de la GPU de la máquina.

Trataré de explicar el proceso de la forma más detallada posible y mencionar los problemas con los problemas que fueron surgiendo. Además, incluiré las referencias de aquellos artículos que favorecieron a las soluciones planteadas, por supuesto.

A.1 Configuración de la Infraestructura

Comenzamos con la plataforma de Google Cloud. Se ha utilizado el servicio llamado *Compute Engine* el cual ofrece un asistente para creación de *máquinas virtuales*.

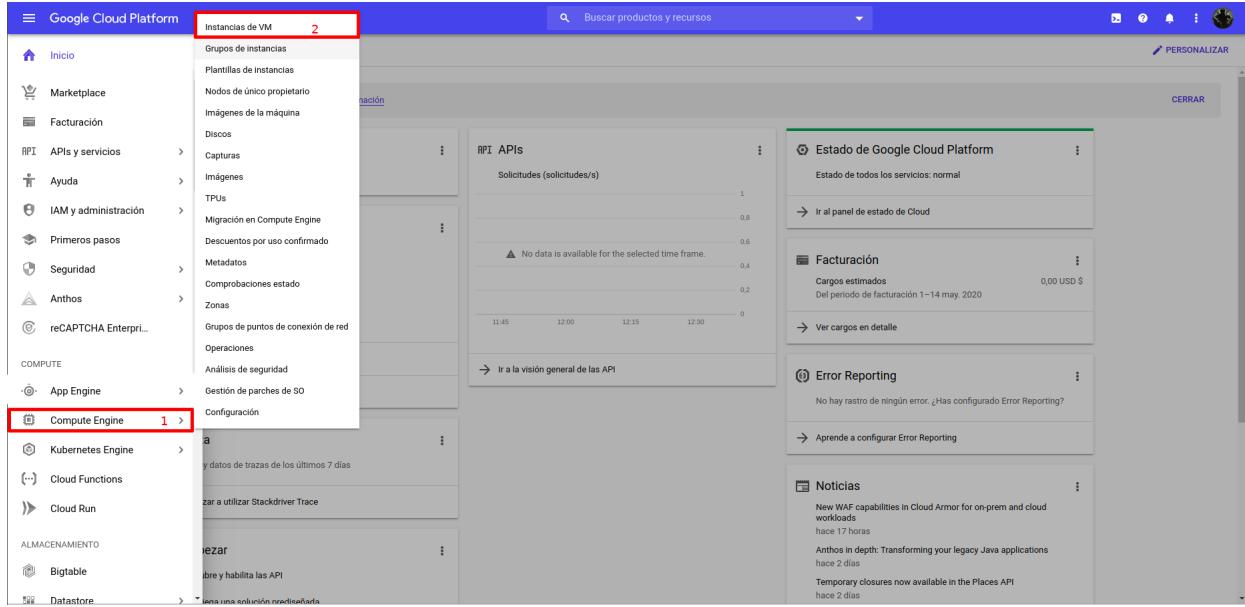


Figura A.1: Iniciando asistente para creación de máquina virtual.

A continuación nos aparecerá las máquinas que tenemos montadas actualmente en la plataforma. En caso de no tener ninguna nos aparecerá directamente la opción de crear una nueva.

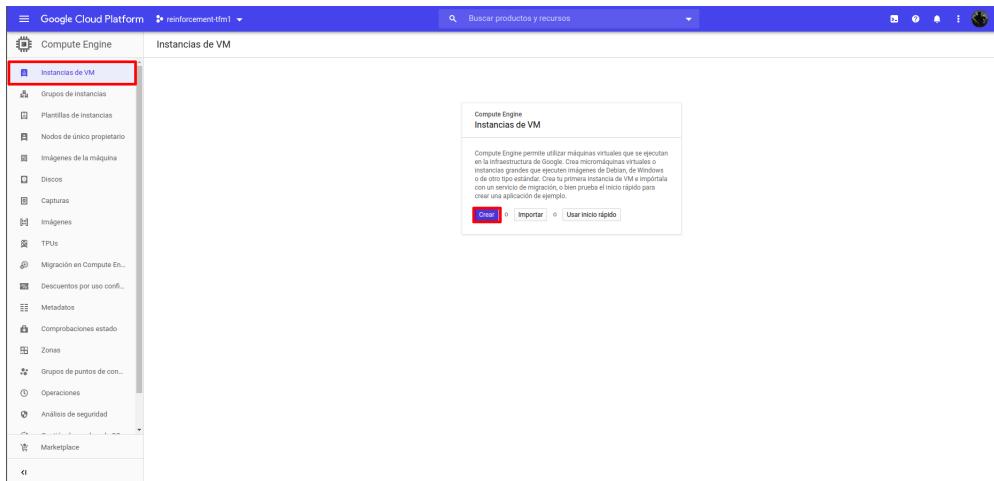


Figura A.2: Creando una máquina virtual.

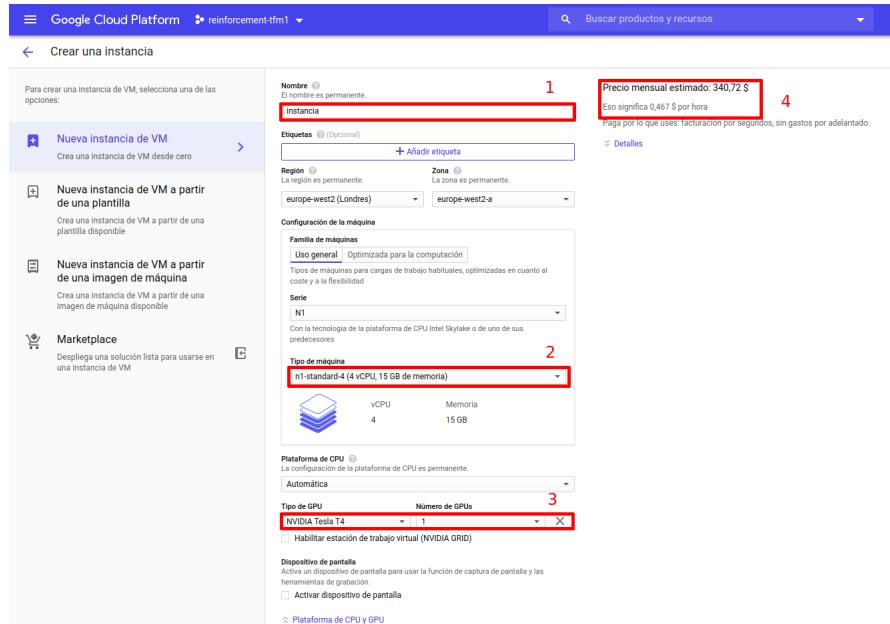


Figura A.3: Configurando la máquina virtual para el proyecto.

Llegados a este punto debemos tener en cuenta diferentes factores a la hora de detallar la máquina que vamos a crear a partir de la infraestructura de Google Cloud. En primer lugar damos nombre a la máquina, el cual nos servirá para identificarla dentro de la plataforma, si lo hacemos de forma remota debemos utilizar la IP externa, como es obvio.

Tenemos que decidir la **región** en la que van a estar los recursos de la máquina. Decidir la región es algo bastante importante debido a que la latencia de la conexión y recursos disponibles a solicitar depende directamente de ello.

Para tener en cuenta la latencia existe un servicio de Google que nos brinda exactamente esa información:

REGION	MEDIAN LATENCY
Global HTTP Load Balancer global	46ms
Belgium europe-west1	52ms
London, UK europe-west2	55ms
Frankfurt, Germany europe-west3	57ms
Netherlands europe-west4	57ms
Zurich, Switzerland europe-west6	57ms
Hamina, Finland europe-north1	86ms
Montreal, Canada northamerica-northeast1	162ms
Northern Virginia, USA us-east4	200ms
South Carolina, USA us-east1	201ms
Iowa, USA us-central1	202ms
Salt Lake City, USA us-west3	247ms

Figura A.4: Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [13]

Como podemos observar en la captura realizada en la figura A.4, la región con mejor latencia es la de **Bélgica**. Sin embargo, finalmente se ha utilizado la región de **Londres** tal y como se puede ver en la figura A.3. La mayoría de veces Bélgica tenía problemas de disponibilidad para la GPU e incluso, en algunas ocasiones, daba problemas una vez la máquina comenzaba a funcionar. Igual para cuando este proyecto esté terminado, ese inconveniente no existe.

En definitiva, se busca un **equilibrio** entre buena conexión y buen abastecimiento de recursos. En cuanto a esos recursos, se ha establecido un total de 4 CPU's Intel con 15GB de memoria. Recordemos que algunas técnicas mencionadas en apartados anteriores como el gradiente descendente estocástico puede beneficiarse de la **programación paralela**(apartado 2.4). En principio, esas 4 unidades son suficientes, pudiendo realizar una escalada vertical si es necesario solicitando mayor infraestructura. Este es una de las grandes ventajas de la **nube** visto en el Máster de Ingeniería Informática.

Uno de los componentes más importantes del que podemos beneficiarnos son las **tarjetas gráficas**(GPU). En la figura A.3, tenemos una **Nvidia Tesla T4**. En un principio esto también puede dar problemas, y es que, al parecer, es necesario **solicitar una cuota** previamente a Google para poder acceder a este servicio de solicitud de tarjetas gráficas.

Hay que tener esto en cuenta si no tenemos la posibilidad de incluir tarjetas gráficas en nuestras máquinas virtuales.[45]

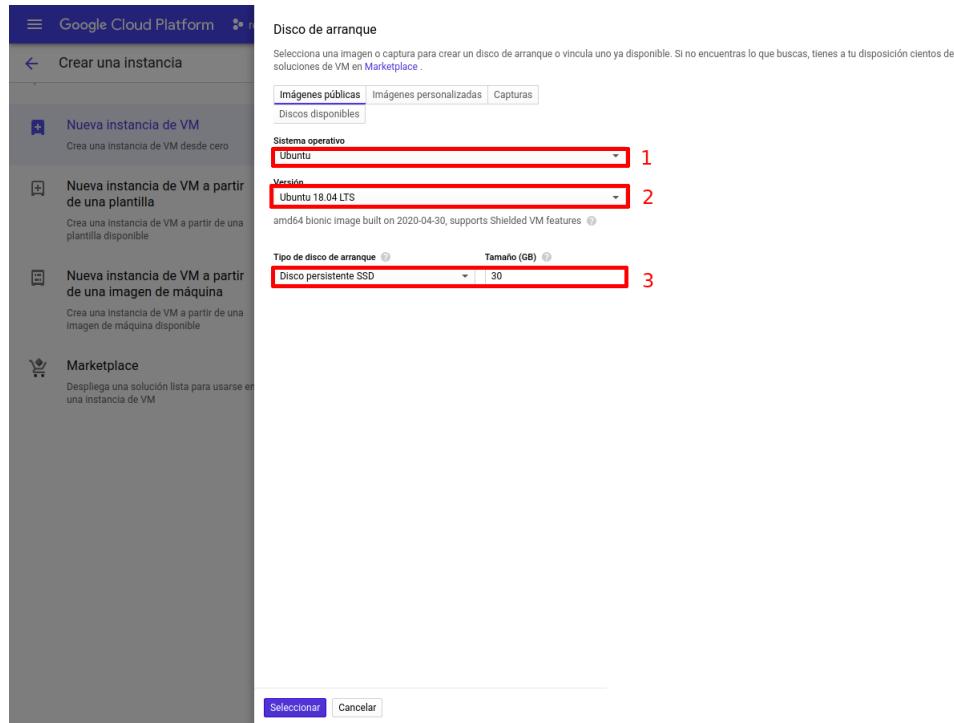


Figura A.5: Indicando Sistema operativo y memoria SSD a la máquina virtual.

El asistente también nos brinda la posibilidad de **seleccionar la imagen del Sistema Operativo** con el que va a funcionar nuestra máquina. En el momento que se realizó la configuración en local, fue con *Ubuntu 18.04 LTS*. Con esa imagen tenía una mejor noción de los problemas e inconvenientes que podrían surgir, así como instalación de drivers necesarios. Por ello, se tomó la decisión de utilizar la misma imagen, además de añadirle **30 GB de SSD** para mejorar el rendimiento de la máquina. se trato de hacer con la versión minimalista de esa imagen, pero daba una gran cantidad de errores.

Una vez creada la máquina, en la misma sección de *Compute Engine* y *máquina virtuales* debe de aparecernos lo siguiente:

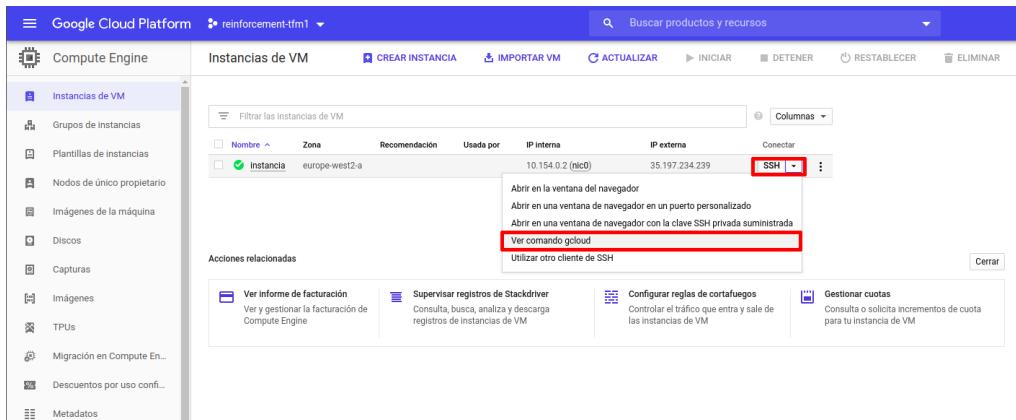


Figura A.6: Comprobando que la máquina virtual ha sido creada con éxito.

A.2 Configuración de la Máquina

Hasta ahora, hemos estado detallando la configuración de infraestructura en la nube para la creación de la máquina virtual. Vamos a explicar la **configuración** de esa máquina para poder hacer los experimentos con las distintas técnicas de aprendizaje por refuerzo explicadas en este trabajo.

Lo primero que deberíamos tratar de hacer es **conectarnos** a la máquina que hemos creado. Para ello hay varias formas. Yo recomiendo, por su sencillez, utilizar la línea de comandos cliente de Google Cloud [16]. Simplemente nos logeamos desde esos comandos y así podemos utilizar directamente el comando de conexión que aparece en la opción de la figura A.6.

Realiza una comunicación **SSH**, haciendo uso de las claves asimétricas que previamente Google Cloud ya ha administrado entre nuestro equipo y la máquina virtual al estar utilizando *gcloud* con nuestra cuenta. Es muy cómodo una vez entendemos como funciona el CLI.

Una vez dentro tendremos un terminal, como el que se muestra a continuación:

```
(base) hapneck@Equipo-Alex:~$ gcloud beta compute ssh --zone "europe-west2-a" "Instancia" --project "reinforcement-tfm1"
Warning: Permanently added 'compute.393979946370914032' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.3.0-1018-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Thu May 14 15:49:59 UTC 2020

 System load:  0.26      Processes:          145
 Usage of /:   4.9% of 28.90GB   Users logged in:     0
 Memory usage: 1%
 Swap usage:  0%

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

hapneck@Instancia:~$ _
```

Figura A.7: Entrando en la máquina virtual.

Esta máquina tiene la imagen de *Ubuntu 18.04 LTS* virgen, sin ningún paquete o librería adicional. Ahora tenemos que pasar al **aprovisionamiento** de esa máquina para que tenga todas las herramientas que vamos a necesitar. Si en el Máster ya vimos algunas herramientas, muy útiles para ello, como **Ansible** [17], es cierto que tenía sentido cuando había que administrar e instalar una gran cantidad de máquinas al mismo tiempo, así como realizar escalados horizontales si era necesario(en un servicio web, por ejemplo).

No obstante, no es necesario para este caso, ya que se va a contar con una única máquina y que, en cualquier caso, necesitaría un escalado vertical y no horizontal si llegase ese momento. Por ello, se opta directamente por un método más “tradicional” aunque efectivo para nuestras necesidades; los **scripts bash** de Ubuntu.

Simplemente se ha desarrollado un script con todos los comandos necesarios, el cual posteriormente se envía al servidor y ejecuta desde allí. Ese script fue perfeccionándose a medida que se encontraban problemas en la configuración del mismo:

```
#!/bin/bash

#Primeras instalaciones
sudo apt update && sudo apt install -y cmake libopenmpi-dev python3-dev zlib1g-dev

wget https://repo.anaconda.com/archive/Anaconda3-2020.02-Linux-x86_64.sh

sh Anaconda3-2020.02-Linux-x86_64.sh

source .bashrc

conda update -n base conda

conda update anaconda

#Creamos entorno virtual para python 3.6
conda create --name TFM python=3.6

#Entramos en dicho entorno
conda activate TFM

#Tenemos que hacer que ubuntu reconozca la tarjeta grafica Tesla de la VM instalando sus
# drivers para ubuntu 18.04
wget http://us.download.nvidia.com/tesla/410.129/nvidia-diag-driver-local-repo-ubuntu1804
-410.129_1.0-1_amd64.deb
sudo dpkg -i nvidia-diag-driver-local-repo-ubuntu1804-410.129_1.0-1_amd64.deb
sudo apt-key add /var/nvidia-diag-driver-local-repo-410.129/7fa2af80.pub
sudo apt update
sudo apt install -y cuda-drivers

#Pra comprobar que los drivers estan bien
sudo apt install -y ubuntu-drivers-common

#instalamos pip
sudo apt install -y python3-pip

#instalamos git
```

```

sudo apt install -y git

#Clonamos el repositorio de openAI baselines y entramos en el
git clone https://github.com/openai/baselines.git

cd baselines

#Tenemos que instalar el tensorflow compatible con uso de GPU
pip install tensorflow-gpu==1.14

#Instalamos todo baselines
pip install -e .[all]

#Instalamos tambien stable-baselines como alternativa a openai baselines
pip install stable-baselines[mpi]

#Para cuando da error en los test con el video_recorder.py
sudo apt install -y ffmpeg

pip install pytest

#Reiniciamos la VM, lo cual significa que nos va a tirar la conexión y vamos a tener que volver
# a entrar por SSH
sudo reboot

```

En primer lugar, hacemos unas primeras instalaciones de librerías que va a utilizar *OpenAI baselines* y *Gym*. Recordemos que estas implementaciones están hechas en Python.

El siguiente paso es instalar una herramienta que nos permita crear entornos virtuales en los que tener un entorno de trabajo en Python adecuado para los *baselines*. En este caso, se ha utilizado **Anaconda** [2]. Tras muchos intentos y pruebas, se consiguió crear una configuración que funcione. Con este **control de versiones** de Python y paquetes instalados, nos permite tener más de un entorno de trabajo, por si un problema de DRL necesitara un aprovisionamiento diferente.

Los siguientes comandos son simplemente para actualizar conda y crear el entorno con una versión de Python 3.6, la versión que no dio ningún problema en este caso.

Recordemos que configuramos una tarjeta gráfica bastante potente para la máquina (apartado A.1). Por defecto, las librerías de *baselines* no van a ser capaces de reconocer esa GPU, aunque sean específicas para uso de la misma. Es necesario instalar los **controladores** para esa tarjeta concreta, así *Tensorflow* puede utilizarla, es la librería que utiliza los *baselines* para las redes neuronales profundas.[44].

Antes de instalar los *baselines* es importante instalar *TensorFlow*, como mencionamos hace un momento. Es necesario instalar la versión para GPU, de lo contrario no la utilizará aunque hayamos instalado los controladores. La versión 1.14 es bastante estable (la versión que aparece en la guía de OpenAI[34]). Hay algunas guías con otras versiones, aunque no han funcionado en este caso, los motivos exactos son desconocidos.

Clonamos el repositorio Github de donde extraemos los *baselines* de *OpenAI* [34]. Entramos en la carpeta y con el comando pip comenzamos a instalar las librerías de Python

necesarias que incluye. Importante poner la opción `-e ./all` aunque no venga en su guía, ya que de lo contrario los test con pytest futuros no funcionarán correctamente, quizás se trate de un error que hay que solucionar.

Hacemos la instalación de *stable-baselines*^[14] para poder usarlo concretamente en el algoritmo DDPG.

Nos podemos encontrar con otro problema que no es mencionado en la guía de instalación de OpenAI. Resulta que hace uso de unas librerías alojadas en un archivo llamado *video_recorder.py*. El caso es que este archivo da error cuando ejecutamos los test para comprobar la instalación. Para solucionarlo, tenemos que hacer una instalación en el sistema del paquete *ffmpeg*, de lo contrario no funcionará.

Por ultimo instalamos *pytest*. Con esto podremos ejecutar los test que incluye los *baselines* para comprobar que la instalación de todo ha finalizado correctamente. Tras hacer todo lo mencionado en este script, no deberíamos encontrarnos con ningún problema.

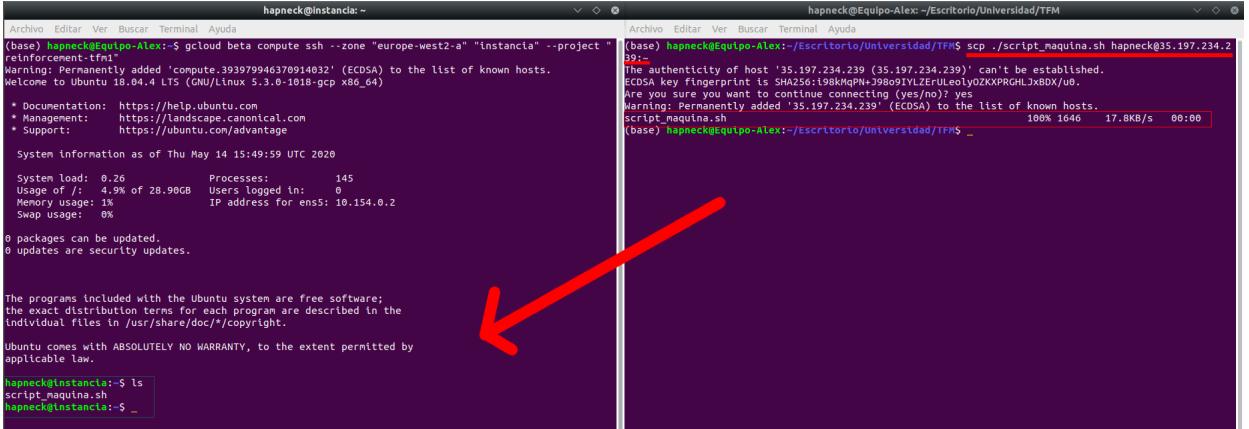
Es importante *reiniciar* la máquina virtual cuando finalicemos. Esto se hace concretamente por los controladores de la tarjeta gráfica. Si no lo hacemos, los controladores no funcionarán y la GPU no será utilizada cuando ejecutemos los algoritmos. A continuación, se muestra un ejemplo de uso de GPU en la figura A.8 de cuando se está entrenando y cuando no. [9][27][46]

(base) hapneck@Equipo-Alex:~\$ nvidia-smi		(TFM) hapneck@Equipo-Alex:~\$ nvidia-smi					
Tue May 19 17:23:44 2020		Tue May 19 17:26:59 2020					
		Sin entrenar					
		Entrenando					
+-----+ NVIDIA-SMI 415.27 Driver Version: 415.27 CUDA Version: 10.0 +-----+ GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC +-----+ Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M. +-----+ 0 GeForce GTX 1050 Off 00000000:01:00.0 Off N/A +-----+ N/A 45C P0 N/A / N/A 583MiB / 4040MiB 0% Default +-----+		+-----+ NVIDIA-SMI 415.27 Driver Version: 415.27 CUDA Version: 10.0 +-----+ GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC +-----+ Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M. +-----+ 0 GeForce GTX 1050 Off 00000000:01:00.0 Off N/A +-----+ N/A 44C P0 N/A / N/A 670MiB / 4040MiB 4% Default +-----+		+-----+ Processes: GPU Memory +-----+ GPU PID Type Process name Usage +-----+ 0 5478 G /usr/lib/xorg/Xorg 273MiB +-----+ 0 5679 G /usr/bin/gnome-shell 159MiB +-----+ 0 29862 G ...AAAAAAAAACAAAAAAA= --shared-files 147MiB +-----+		+-----+ Processes: GPU Memory +-----+ GPU PID Type Process name Usage +-----+ 0 2447 C python 39MiB +-----+ 0 5478 G /usr/lib/xorg/Xorg 291MiB +-----+ 0 5679 G /usr/bin/gnome-shell 179MiB +-----+ 0 29862 G ...AAAAAAAAACAAAAAAA= --shared-files 147MiB +-----+	
+-----+ NVIDIA-SMI 415.27 Driver Version: 415.27 CUDA Version: 10.0 +-----+ GPU Name Persistence-M Bus-Id Disp.A Volatile Uncorr. ECC +-----+ Fan Temp Perf Pwr:Usage/Cap Memory-Usage GPU-Util Compute M. +-----+ 0 GeForce GTX 1050 Off 00000000:01:00.0 Off N/A +-----+ N/A 44C P0 N/A / N/A 670MiB / 4040MiB 4% Default +-----+							
+-----+ Processes: GPU Memory +-----+ GPU PID Type Process name Usage +-----+ 0 5478 G /usr/lib/xorg/Xorg 273MiB +-----+ 0 5679 G /usr/bin/gnome-shell 159MiB +-----+ 0 29862 G ...AAAAAAAAACAAAAAAA= --shared-files 147MiB +-----+		+-----+ Processes: GPU Memory +-----+ GPU PID Type Process name Usage +-----+ 0 2447 C python 39MiB +-----+ 0 5478 G /usr/lib/xorg/Xorg 291MiB +-----+ 0 5679 G /usr/bin/gnome-shell 179MiB +-----+ 0 29862 G ...AAAAAAAAACAAAAAAA= --shared-files 147MiB +-----+					

Figura A.8: Muestra de uso de GPU al entrenar con los baselines.

Luego solo fue necesario enviar el script desde el equipo local a la máquina que tenemos en la nube. Para ello se puede utilizar el comando **SCP**(secure copy), el cuál funciona también por encima de la tecnología SSH [20]. Posteriormente será utilizado en sentido

opuesto para descargar los agentes entrenados en la máquina al equipo local.



The screenshot shows two terminal windows side-by-side. The left window is titled 'hapneck@instancia:' and shows the command 'gcloud beta compute ssh --zone "europe-west2-a" "instancia" --project reinforcement-tfm'. It also displays system information for Thursday, May 14, 2020, including load average, memory usage, and swap usage. The right window is titled 'hapneck@Equipo-Alex:' and shows the command 'scp ./script_maquina.sh hapneck@35.197.234.239'. A red arrow points from the left terminal's command area to the right terminal's progress bar, indicating the transfer of the script.

```
(base) hapneck@Equipo-Alex:~$ gcloud beta compute ssh --zone "europe-west2-a" "instancia" --project reinforcement-tfm
Warning: Permanently added 'compute.393979946578914032' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.3.0-1018-gcp x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

System information as of Thu May 14 15:49:59 UTC 2020

System load: 0.26      Processes:          145
Usage of /: 4.9% of 28.90GB  Users logged in:   0
Memory usage: 1%          IP address for ens5: 10.154.0.2
Swap usage:  0%

0 packages can be updated.
0 updates are security updates.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

hapneck@instancia:~$ ls
script_maquina.sh
hapneck@instancia:~$ _
```

```
(base) hapneck@Equipo-Alex:~/Escritorio/Universidad/TFM$ scp ./script_maquina.sh hapneck@35.197.234.239:
The authenticity of host '35.197.234.239 (35.197.234.239)' can't be established.
ECDSA key fingerprint is SHA256:19BkWqPNwJ0869fYLZfUleOlyDZXKXRGMlxbDX/u0.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.197.234.239' (ECDSA) to the list of known hosts.
script_maquina.sh                                         100% 1646    17.8KB/s   00:00
```

Figura A.9: Pasando el script a la máquina virtual para abastecerla.

B. Uso de la Máquina Virtual

Con lo explicado en los apartados anteriores, tenemos lo que necesitamos para comenzar a realizar los experimentos en los diferentes entornos y con las diferentes técnicas explicadas en este trabajo. Pero antes de empezar, detallaremos como utilizar estas librerías y herramientas de *OpenAi baselines* y *Gym*.

En la misma guía de Github[34], hay ejemplos de uso. Primero tenemos que asegurarnos de que nos encontramos en el entorno llamado TFM de Anaconda, ya que es donde tenemos instaladas todas las herramientas.

```
> conda activate TFM
```

La forma de usar *baselines* si vamos a utilizar los algoritmos tal y como están diseñados es muy sencillo y solo tenemos que preocuparnos de los parámetros que necesita:

```
> python -m baselines.run --alg=deepq --env=MountainCar-v0 --num_timesteps=1e6
```

Los parámetros que vemos en este comando son **-alg**, el cuál es utilizado para definir el algoritmo con el que queremos entrenar nuestro agente, *deepq* es el algoritmo DQN visto en el apartado 4.1.

Luego tenemos el parámetro **-env**. Con este parámetro definimos el entorno Gym que va a utilizarse para entrenar el agente. Tenemos la lista de entornos disponibles en los repositorios de OpenAI Gym como ya se ha mencionado en otras ocasiones. En este caso, aparece el entorno *MountainCar-v0*, el entorno mencionado en el apartado 6.1.

Por último, está el parámetro **-num_timesteps** el cual se utiliza para definir las interacciones que va a tener disponibles el agente para entrenarse. Hay que definir este parámetro atendiendo siempre al entorno, ya que dependiendo de éste puede que un episodio

ocupe de media más o menos pasos.

Estos serían los parámetros principales que debemos atender a la hora de usar los *baselines*. Sin embargo, el agente entrenará y no nos devolverá resultados, ni se guardará el agente, ni datos referentes al proceso de aprendizaje. Por ello, es necesario añadir algunos parámetros más, dependiendo de lo que queramos hacer exactamente:

```
> python -m baselines.run --alg=deepq --env=MountainCar-v0 --num_timesteps=1e6
--seed=3 --save_path=./modelo.pkl --log_path=./logs_mountain/
```

Si no **guardamos** el modelo que hemos entrenado, no servirá de nada el proceso de entrenamiento. Con el parámetro **–save_path** indicamos la ruta relativa en la que queremos guardarla en el formato PKL [12]. Podemos especificar una semilla con **–seed**. Esto lo veremos en la experimentación, pero lo hemos usado para generar varios agentes con la misma técnica con el fin de asegurarnos de que los resultados no han sido solo suerte por parte del agente.

Los *baselines* cuentan con su propio **sistema de monitorización** del aprendizaje. Con el que podemos almacenar datos muy interesantes del proceso realizado durante su entrenamiento. Al igual que para guardar el modelo, especificamos una carpeta en la que almacenar los archivos referentes a estos datos.

Por último, se va a mostrar la manera de **cargar** los agentes guardados anteriormente y la forma de **probarlos** y visualizarlos dentro del entorno gráfico simulado:

```
> python -m baselines.run --alg=deepq --env=MountainCar-v0 --num_timesteps=0
--load_path=./modelo.pkl --play
```

Observamos que para cargar un modelo ya almacenado anteriormente en nuestro equipo tenemos que especificar el parámetro **–load_path**. Es importante seguir indicando el algoritmo y entorno que se está utilizando debido a que no está almacenado de forma explícita en el modelo entrenado, de lo contrario tendrímos problemas al ejecutarlo.

Es imprescindible indicar que el número de *timesteps* es 0, ya que no se trata de un entrenamiento. Finalmente, añadimos el parámetro **–play** al final, para que cuando termine de cargarlo, ejecute el modelo. Si no lo indicamos, no lo probará dentro del entorno simulado y no nos servirá de nada. También podemos utilizar el parámetro **–play** con **–save_path**; al terminar de entrenar y guardarla, lo ejecutará para que podamos verlo.

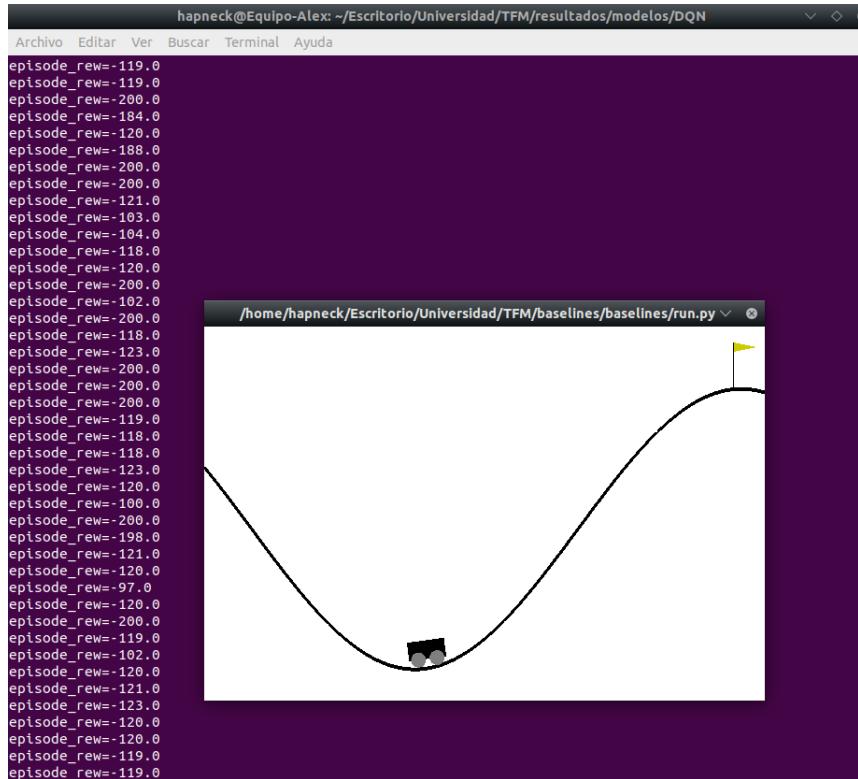


Figura B.1: Probando agente entrenado en entorno simulado *MountainCar-v0*.

Podemos ver en la figura B.1 como algunos episodios tienen una recompensa final por encima de -200. Esto indica que el agente está comenzando a ser capaz de resolver el problema que se le está planteando.

B.1 Archivos generados en los logs

Con los *logs* generados con la funcionalidad que ofrece los *baselines* de OpenAI, tenemos a disposición información que nos puede ayudar a entender el progreso de aprendizaje que ha ido logrando el agente durante el entrenamiento que hemos diseñado para el mismo.

Del mismo modo, puede ser utilizado como una ayuda extra a la hora de determinar si está realmente teniendo una mejora sustancial, en lugar de fijarnos exclusivamente en si es capaz de cumplir el objetivo, ya que dependiendo del problema que se trate puede ser muy ambiguo, o que la cuestión no sea conseguir cumplir el objetivo si no que cada vez lo cumpla de una forma más eficiente.

Obtendremos 3 archivos:

- **log.txt:** La salida por el output estándar(pantalla) del algoritmo es copiado y volcado en este archivo para tenerlo guardado. La salida que nos da por pantalla dependerá del entorno y, sobretodo, algoritmo concreto con el que estemos entrenando. Pero en general suele incluir información cada x episodios referentes a la media de recompensa que lleva, *timesteps* que ha realizado, media de la función de perdida que utiliza para la red neuronal, o el tiempo que ha empleado en explorar, etc. Además, suele indicar cuando actualiza el modelo PKL.

EJEMPLO DE SALIDA PARA DQN:

```
Logging to ./logs/mountain_log-46-2e5/
-----
| % time spent exploring | 2           |
| episodes                | 100          |
| mean 100 episode reward | -200         |
| steps                   | 1.98e+04    |
-----
Saving model due to mean reward increase: None -> -200.0
-----
| % time spent exploring | 2           |
| episodes                | 200          |
| mean 100 episode reward | -200         |
| steps                   | 3.98e+04    |
-----
Saving model due to mean reward increase: -200.0 -> -199.39999389648438
...
```

- **0.0.monitor.csv:** Esta información también dependerá del algoritmo y entorno que estemos usando, en general es referente a los episodios. Por cada episodio, se crea una fila en ese CSV que suele indicarnos la recompensa acumulada que ha obtenido, el número de acciones o *time-steps* que ha realizado y va sumando el tiempo(en segundos) que ha ido transcurriendo durante el entrenamiento episodio a episodio.

EJEMPLO CON DQN:

```
# {"t_start": 1586018142.0617638, "env_id": "MountainCar-v0"}
r,l,t
-200.0,200,3.022039
-200.0,200,3.202508
-200.0,200,3.382297
-200.0,200,3.564011
-200.0,200,3.744724
-200.0,200,4.48549
-200.0,200,5.125517
-200.0,200,5.760974
-200.0,200,6.396026
-200.0,200,7.037994
-200.0,200,7.667753
-200.0,200,8.302286
-200.0,200,8.948372
-200.0,200,9.583497
...
-94.0,94,471.593518 #--> Buen resultado (episodio 810)
-102.0,102,471.938335
-200.0,200,472.619501
```

```
-200.0,200,473.301947  
-200.0,200,473.982054  
-200.0,200,474.639221  
-200.0,200,475.283136  
-200.0,200,475.926098  
-180.0,180,476.504342  
-200.0,200,477.15575  
-183.0,183,477.748613  
-200.0,200,478.397481  
-180.0,180,478.979799  
-175.0,175,479.551284  
-173.0,173,480.109195  
...
```

- **progress.csv:** Cada ciertos episodios(en ejemplo que muestro a continuación cada 100) nos aporta información adicional del proceso de aprendizaje que nos puede ayudar a entender de una forma más abstracta y generalizada como ha ido mejorando el agente. Indicando el tiempo de aprendizaje que destinaba a explorar, la media de recompensas de esos x episodios y los pasos que llevaba en ese momento.

EJEMPLO CON DQN:

```
% time spent exploring,episodes,mean 100 episode reward,steps  
2,100,-200.0,19799  
2,200,-200.0,39799  
2,300,-197.3,59533  
2,400,-173.0,76831  
2,500,-145.1,91341  
2,600,-150.5,106388  
2,700,-182.5,124641  
2,800,-162.7,140909  
2,900,-142.2,155132  
2,1000,-129.5,168086  
2,1100,-125.0,180586  
2,1200,-192.0,199789  
...
```

Como el contenido y columnas exactas que tienen los archivos depende tanto del entorno como del algoritmo que estamos utilizando por parte de *baselines*, en la experimentación(apartado 8.1) se hace uso de estos datos de una forma más específica.

B.2 Visualizar la información de los logs

Los datos que obtenemos de los logs en crudo no nos van a ser de utilidad a la hora de sacar conclusiones a partir de ellos. Por este motivo, es importante buscar una forma de representarlos de una manera más visual y entendible a grandes rasgos.

Baselines ofrece una forma de poder manipularlos más sencilla desde el código. Aunque, por debajo, haremos uso de las típicas librerías de Python cuando tenemos un conjunto de datos, tales como *matplotlib*.

Tiene una guía muy útil para poder hacer uso de esa funcionalidad que queda referenciada al final de este apartado, su uso está bien explicado. Simplemente destacar la necesidad de llamar a campos diferentes de los contenedores que utiliza dependiendo del problema a resolver, dado que en los logs podía haber una información u otra con más o menos campos. Aun así, la forma de usarlos es siempre la misma.[32]

Personalmente, recomiendo usarla, es realmente útil. Principalmente porque no tienes que preocuparte de los diferentes archivos que hemos explicado en el apartado B.1, solo saber llamarlos en función de su contenido. Además, te permite trabajar directamente con directorios y sus subcarpetas para poder representar informaciones de distintas fuentes en una misma gráfica y, de ese modo, poder realizar comparativas de una forma más rápida y sencilla.

C. Stable Baselines

En su documentación oficial[10], así como en su página web[6], encontramos las especificaciones para instalarlo, el cual está incluido en los comandos del script mostrado en la sección A.2.

Se trata de unas implementaciones basadas en *OpenAI baselines*, al ser de código abierto ha surgido esta variante. En estas versiones, existe un mayor esfuerzo en la estabilidad y usabilidad de las técnicas, más que en su innovación. Esta perspectiva es muy acertada, tratando de solucionar o mejorar los errores que van surgiendo por parte de los *baselines* originales, sin tener que preocuparse de tareas investigadoras.

Éste permite una **configuración más profunda** por parte del usuario y, por tanto, algo más compleja. Es una programación un nivel por debajo de los *baselines* originales, los cuales tienen como beneficio principal un **mejor control**.

C.1 Generar un Modelo

Para poder utilizar el algoritmo DDPG con estas librerías, realicé el siguiente script en Python:

```
import os
import argparse
import gym
import numpy as np

from stable_baselines.bench import Monitor

from stable_baselines.ddpg.policies import MlpPolicy
from stable_baselines.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise,
    AdaptiveParamNoiseSpec
```

```

from stable_baselines import DDPG
from stable_baselines.common.callbacks import BaseCallback
from stable_baselines.results_plotter import load_results, ts2xy

class SaveOnBestTrainingRewardCallback(BaseCallback):
    """
    Callback for saving a model (the check is done every "check_freq" steps)
    based on the training reward (in practice, we recommend using "EvalCallback").
    """

    :param check_freq: (int)
    :param log_dir: (str) Path to the folder where the model will be saved.
    It must contains the file created by the "Monitor" wrapper.
    :param verbose: (int)
    """

    def __init__(self, check_freq: int, log_dir: str, save_dir: str, verbose=1):
        super(SaveOnBestTrainingRewardCallback, self).__init__(verbose)
        self.check_freq = check_freq
        self.log_dir = log_dir
        self.save_path = save_dir
        self.best_mean_reward = -np.inf

    #def __init_callback(self) -> None:
    #    # Create folder if needed
    #    if self.save_path is not None:
    #        os.makedirs(self.save_path, exist_ok=True)

    def _on_step(self) -> bool:
        if self.n_calls % self.check_freq == 0:

            # Retrieve training reward
            x, y = ts2xy(load_results(self.log_dir), 'timesteps')
            if len(x) > 0:
                # Mean training reward over the last 100 episodes
                mean_reward = np.mean(y[-100:])
                if self.verbose > 0:
                    print("Num timesteps: {}".format(self.num_timesteps))
                    print("Best mean reward: {:.2f} - Last mean reward per episode: {:.2f}".format(self.
                        best_mean_reward, mean_reward))

            # New best model, you could save the agent here
            if mean_reward > self.best_mean_reward:
                self.best_mean_reward = mean_reward
                # Example for saving best model
                if self.verbose > 0:
                    print("Saving new best model to {}".format(self.save_path))
                    self.model.save(self.save_path)

        return True

#Tratamiento de parametros del script
parser = argparse.ArgumentParser()
parser.add_argument("--num_timesteps", type=float, default=1e6, help="Indica el numero de
    timesteps que va a tener el entrenamiento, por defecto 1e6")
parser.add_argument("--save_path", help="Indica la ruta y el nombre del archivo en el que se
    va a almacenar el modelo.pkl")
parser.add_argument("--log_path", help="Indica la ruta en la que se almacena los logs del

```

```

    proceso de aprendizaje")
parser.add_argument("--env", help="entorno gym que utiliza el modelo")
parser.add_argument("--seed", type=int, help="Semilla con la que se inicia el entrenamiento
    del modelo")
args = parser.parse_args()

env = gym.make(args.env)
# Creamos la carpeta para los logs
if args.log_path:
    log_dir = args.log_path
    os.makedirs(log_dir, exist_ok=True)
# los logs seran guardados en log_dir/monitor.csv
env = Monitor(env, log_dir)

if args.save_path:
    save_dir=args.save_path

# El ruido para DDPG
n_actions = env.action_space.shape[-1]
param_noise = None
action_noise = OrnsteinUhlenbeckActionNoise(mean=np.zeros(n_actions), sigma=float(0.5) *
    np.ones(n_actions))

model = DDPG(MlpPolicy, env, verbose=1, param_noise=param_noise, action_noise=
    action_noise, seed=args.seed)

callback=SaveOnBestTrainingRewardCallback(check_freq=1000, log_dir=log_dir, save_dir=
    save_dir)

model.learn(total_timesteps=args.num_timesteps,callback=callback)

```

Tanto la implementación como la clase *SaveOnBestTrainingRewardCallback*, la cual sirve para guardar los modelos, han sido inspirados a partir de las implementaciones que ellos mismos ofrecen en su página web [4] [5]. Aún así, no es capaz de guardarse de una manera 100% fiel al entrenamiento realizado. No obstante, los resultados obtenidos son buenos y funciona bastante bien.

Este es el script usado para entrenar en la máquina virtual. Incluyendo parámetros que se llaman de la misma forma para que tenga una forma de uso muy similar.

La principal desventaja que se observa con respecto a los *baselines* originales es la **monitorización** del progreso de aprendizaje. Se implementó una forma de registrar todas las recompensas por medio del código, junto con los pasos realizados y tiempo empleado por cada episodio. Esto es equivalente al archivo *0.0.monitor.csv* del apartado B.1.

Para obtener la salida por pantalla en un archivo, equivalente al *log.txt*, simplemente se ha usado los **pipelines** de linux para guardarla en un archivo al mismo tiempo que aparece la información por pantalla para poder visualizar su progreso y que todo está bien mientras entrena:

```
<ejecución script Python> | tee ~/log.txt
```

No obstante, no había una forma clara de poder conseguir la información equivalente del archivo *progress.csv*. La librería no ofrece una forma clara de poder rescatar estos datos

durante su aprendizaje.

C.2 Cargar un modelo

Para cargar los modelos entrenados en el equipo personal y probarlos se fabricó otra implementación a parte:

```
import gym
import time
import argparse

from stable_baselines import DDPG
from stable_baselines.common.evaluation import evaluate_policy

parser = argparse.ArgumentParser()
parser.add_argument("--load_path", help="Indica la ruta en la que se encuentra el modelo a cargar")
parser.add_argument("--env", help="entorno gym que utiliza el modelo a cargar")
args = parser.parse_args()

env=gym.make(args.env)
model = DDPG.load(args.load_path)

# Probando el agente entrenado
obs = env.reset()
while(True):
    #Paramos un poco el tiempo para que de tiempo a visualizarlo
    time.sleep(0.005)
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    if(dones):
        env.reset()
    else:
        env.render()
```

De nuevo, se configuran parámetros con el mismo nombre. Simplemente cargando el modelo, iniciándolo y ofreciéndole la observación del entorno, permitimos que él mismo decida la siguiente acción a realizar.

Gym ofrece una variable que indica cuando se ha llegado a un estado terminal, en ese caso se reinicia el entorno y vuelve a comenzar. Uno de los inconvenientes de probar los modelos es que son **demasiado rápidos** al ejecutarse y es posible que cueste un poco ver como actua, parece que está a cámara rápida.

Al poder generar la implementación, pude mejorar este aspecto. Simplemente haciendo un *sleep*, tal y como se ve en el script, la visualización del agente es más apreciable, podemos modificar la velocidad de ejecución jugando con el valor de este comando.