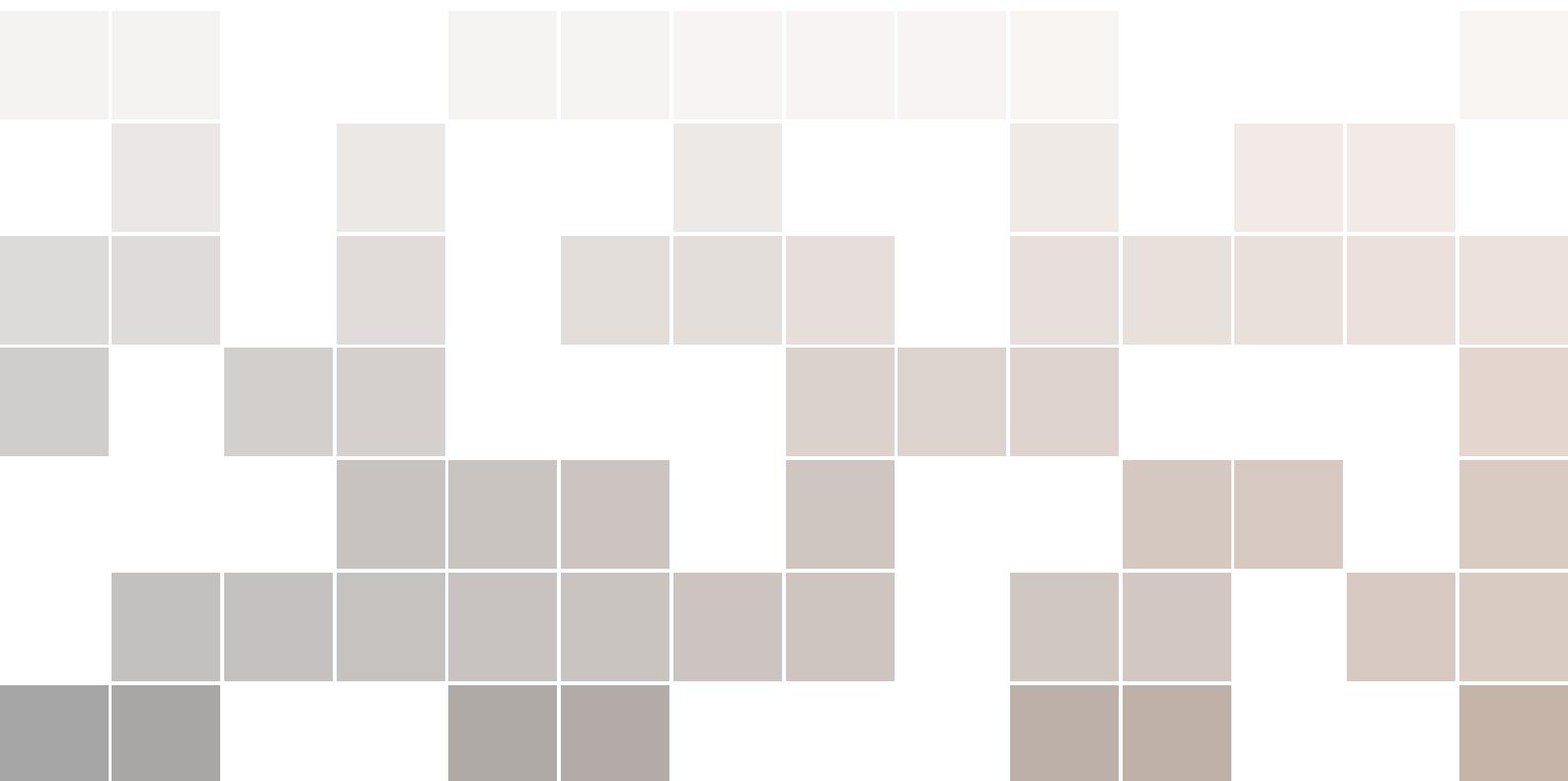


Aprendizaje profundo por refuerzo para la resolución de problemas.

Técnicas actuales

Alejandro Campoy Nieves

Tutor: Juan Gómez Romero



Copyright © 2020 Alejandro Campoy

Índice general

0.1	Resumen	11
I	Introducción	
1	Contexto	15
1.1	Motivación	18
1.2	Objetivos del proyecto	18
1.3	Estructura del proyecto	19
II	Métodos	
2	Redes Neuronales	25
2.1	Funciones de activación (h)	26
2.2	Funciones de salida (h')	28
2.3	Funciones de coste	28
2.4	Algoritmos de optimización	29
2.5	Redes Neuronales Convolucionales (CNN)	31
2.5.1	Capa convolucional	31
2.5.2	Capa de reducción o Pooling	32
2.5.3	Capa Flatten	33
3	Aprendizaje por refuerzo (RL)	35
3.1	Componentes del RL	36
3.1.1	Ciclo de interacción	37

3.2	Procesos de Decisión de Markov (MDPs)	38
3.2.1	Función de transición	38
3.2.2	Función de recompensa	39
3.2.3	Descuento	39
3.3	Política	40
3.3.1	Función de estado-valor	41
3.3.2	Función de acción-valor	42
3.3.3	Ecuación de Bellman	42
3.3.4	Función de acción-ventaja	42
3.3.5	Evaluando una política	43
3.3.6	Mejorando una política	44
3.4	Exploración vs Explotación	45
3.5	Aprendiendo a evaluar y mejorar políticas	45
3.5.1	Método de Montecarlo	46
3.5.2	Métodos de Diferencia Temporal (SARSA)	48
3.5.3	Q-Learning	50
4	Aprendizaje por Refuerzo Profundo (DRL)	53
4.1	Métodos Basados en Valor: Deep Q-Network (DQN)	53
4.2	Métodos basados en política: REINFORCE	58
4.3	Métodos actor-critic	59
4.3.1	Proximal Policy Optimization (PPO2)	60
4.3.2	Deep Deterministic Policy Gradient (DDPG)	62
4.3.3	Asynchronous Advantage Actor-Critic (A3C)	64
4.3.4	Adavtage Actor-Critic (A2C)	66

III

Desarrollo y Experimentación

5	OpenAI baselines	73
6	Entornos Open AI Gym	75
6.1	MountainCar-v0	76
6.2	MountainCarContinuous-v0	78
6.3	Super Mario Bros	78
7	Entorno de desarrollo	83
7.1	Metodología de experimentación	84
8	Experimentación: Resultados Obtenidos	85
8.1	MountainCar-v0	85
8.1.1	DQN	86
8.1.2	DDPG	89
8.1.3	PPO2 y A2C	91
8.1.4	Comparativa de modelos	92

8.2	Super Mario Bros	93
8.2.1	DQN	95
8.2.2	A2C y PPO2	97
8.2.3	Desarrollo de PPO2	100

IV

Posibles mejoras y conclusiones

9	Conclusiones finales	107
9.1	Possibles mejoras	107
9.2	Conclusiones	108
	Bibliografía	109
	Artículos	109
	Libros	113

V

Apéndice

A	Configuración en la nube	117
A.1	Configuración de la Infraestructura	117
A.2	Configuración de la Máquina	122
B	Uso de la Máquina Virtual	127
B.1	Archivos generados en los logs	129
B.2	Visualizar la información de los logs	131
C	Stable Baselines	133
C.1	Generar un Modelo	133
C.2	Cargar un modelo	136

Índice de figuras

1.1	Categorización de tipos de aprendizajes en Machine Learning.	16
1.2	Estructura de conceptos dentro de la Inteligencia Artificial.	17
1.3	Modelo de cascada.	19
1.4	Diagrama de Gantt.	20
2.1	Esquema simplificado de una red neuronal. [1]	25
2.2	Función sigmoide en plano bidimensional. Extraída de [12].	27
2.3	Función ReLU en plano bidimensional. Extraída de [12].	28
2.4	Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de [9].	30
2.5	Aplicación de capa convolucional a imagen de entrada. Imagen extraída de [61].	32
2.6	Aplicación de capa Max-Pooling a mapa de características. Imagen extraída de [53].	33
2.7	Adaptación de parte convolucional a parte densa con uso de capa Flatten. Imagen extraída de [11].	33
3.1	Esquema de componentes del RL. Imagen extraída y posteriormente modificada de [64].	36
3.2	Ilustración gráfica del factor de descuento sobre las recompensas. Extraído de [31].	40
3.3	Esquema Método Montecarlo. Tabla función extraída de [55].	47
3.4	Esquema Método de diferencia temporal (SARSA). Tabla función extraída de [55].	50
3.5	Esquema Método Q-Learning. Tabla función extraída de [55].	51
4.1	Esquema red neuronal, acciones discretas. Imagen extraída de [55].	54
4.2	Esquema red neuronal, espacio de acciones continuo. Imagen extraída de [55].	55
4.3	Esquema Método Deep Q-Learning. Extraída de [3] [55].	57
4.4	La red neuronal es la propia política para los métodos basados en políticas. Extraída de [55].	58
4.5	Esquema de metodología actor-critic. Extraída de [55].	60

4.6	Comportamiento de función objetivo <i>LCLIP</i> para ventajas positivas (izquierda) y ventajas negativas (derecha). El círculo rojo indica el punto inicial a partir del cual se realiza la optimización. Extraída de [26].	62
4.7	Ruido en el espacio de acciones (izquierda) frente a ruido en los parámetros de la red (derecha). Extraída de [34].	63
4.8	Esquema de algoritmo A3C.	65
4.9	Descripción de procedimiento en algoritmo A3C.	66
4.10	Red neuronal compartida para política (actor) y valor de función (crítico). Extraída de [31].	67
4.11	Esquema de algoritmo A2C.	68
6.1	Agente entrenando conducción autónoma en entorno simulado de GTA V. Extraída de vídeo en Youtube. [51]	76
6.2	Representación gráfica del problema MountainCar v0 de OpenAI Gym.	77
6.3	Entorno del videojuego Super Mario Bros.	79
6.4	Entrada del agente en Super Mario Bros.	79
7.1	Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada	
	84	
8.1	Media recompensas acumuladas cada 100 episodios, todas las semillas.	87
8.2	Tiempo empleado en explorar por <i>MountainCar</i> con semilla 113	88
8.3	Resumen general de progreso de algoritmo DQN en <i>MountainCar-v0</i>	89
8.4	Agentes entrenados con DDPG en <i>MountainCar-v0</i>	90
8.5	Agentes entrenados con DDPG en <i>MountainCar-v0</i>	90
8.6	Entropía de política a lo largo de entrenamiento con técnicas A2C y PPO2 en <i>MountainCar-v0</i>	92
8.7	Comparativa de técnicas en <i>MountainCar</i>	93
8.8	Porcentaje de uso CPU en máquina virtual en pruebas con Super Mario Bros (Google Cloud Monitoring).	95
8.9	Recompensas obtenidas durante el entrenamiento DQN con 10 millones de pasos en <i>SuperMarioBros-Nes</i>	96
8.10	Porcentaje de tiempo explorado durante el entrenamiento DQN con 10 millones de pasos en <i>SuperMarioBros-Nes</i>	96
8.11	Media de recompensas durante el entrenamiento A2C y PPO2 con 10 millones de pasos en <i>SuperMarioBros-Nes</i>	97
8.12	Media de pasos por episodio durante el entrenamiento A2C y PPO2 con 10 millones de pasos en <i>SuperMarioBros-Nes</i>	98
8.13	Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C en <i>SuperMarioBros-Nes</i>	99
8.14	Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en <i>SuperMarioBros-Nes</i>	99
8.15	Valores de entropía de las políticas durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en <i>SuperMarioBros-Nes</i> nivel 1-1.	100
8.16	Media de recompensa durante entrenamiento de 100 millones de interacciones con PPO2 en <i>SuperMarioBros-Nes</i>	101
8.17	Media de duración de episodios (pasos) durante entrenamiento de 100 millones de interacciones con PPO2 en <i>SuperMarioBros-Nes</i>	102
8.18	Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en <i>SuperMarioBros-Nes</i>	102
A.1	Iniciando asistente para creación de máquina virtual.	118
A.2	Creando una máquina virtual.	118
A.3	Configurando la máquina virtual para el proyecto.	119

A.4	Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [14]	120
A.5	Indicando Sistema operativo y memoria SSD a la máquina virtual.	121
A.6	Comprobando que la máquina virtual ha sido creada con éxito.	122
A.7	Entrando en la máquina virtual.	122
A.8	Muestra de uso de GPU al entrenar con los baselines.	125
A.9	Pasando el script a la máquina virtual para abastecerla.	126
B.1	Probando agente entrenado en entorno simulado <i>MountainCar-v0</i> .	129

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 mostrarán los resultados obtenidos, tratando de dar una visión general de lo que estos algoritmos son capaces de hacer.



Introducción

1	Contexto	15
1.1	Motivación	
1.2	Objetivos del proyecto	
1.3	Estructura del proyecto	



1. Contexto

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.^[22]

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: ^[25]

- **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 definan 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.

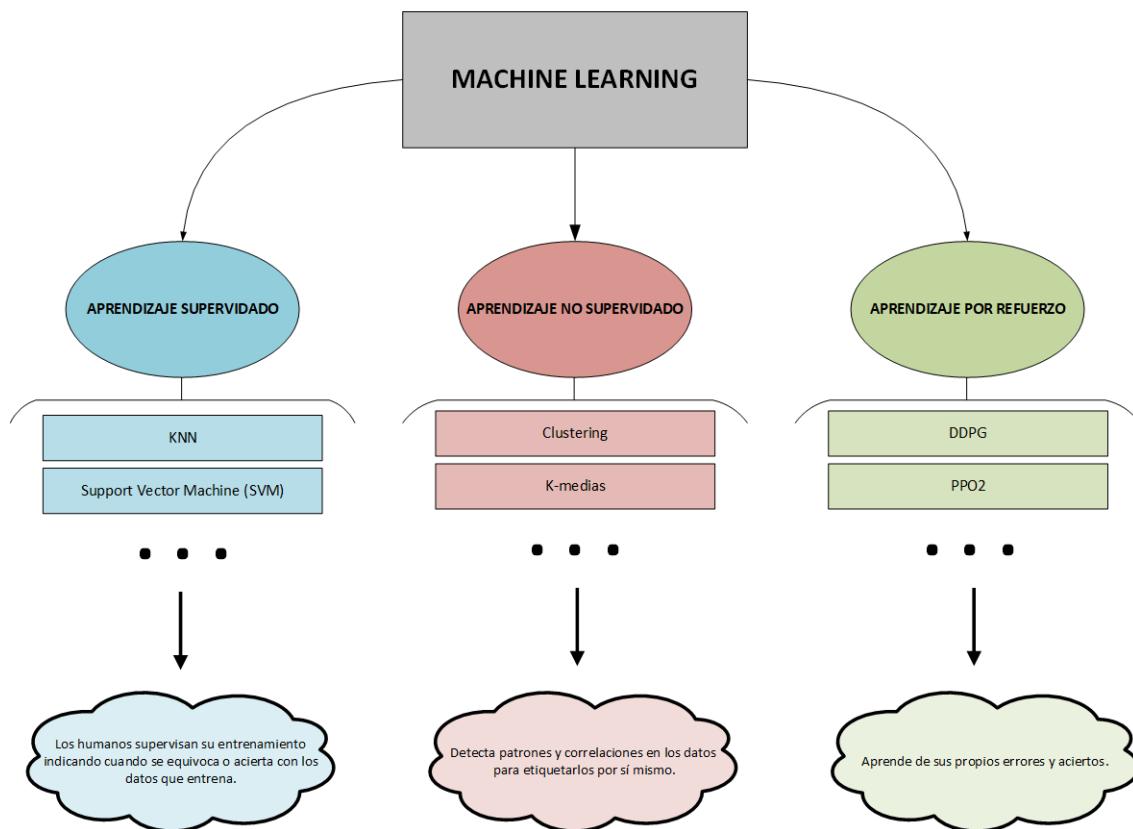


Figura 1.1: Categorización de tipos de aprendizajes en Machine Learning.

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.^[32]

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.

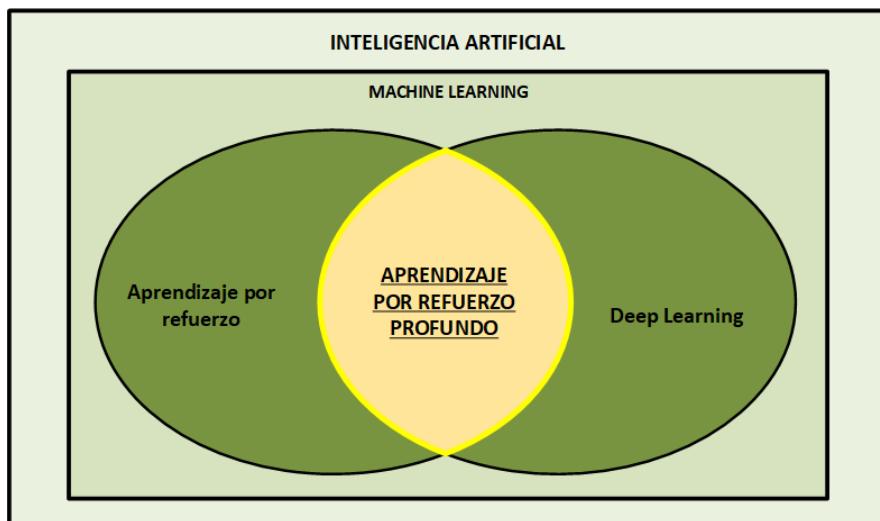


Figura 1.2: Estructura de conceptos dentro de la Inteligencia Artificial.

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, punteros a día de hoy en este campo. 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 Motivación

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.^[16]

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.^[57] ^[33]

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.^[62] ^[30]

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

1.2 Objetivos del proyecto

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

1. Entender los **conceptos fundamentales** del Aprendizaje Automático en los que se sostiene el DRL.
2. Entender, de una forma genérica, el **funcionamiento y metodología** que siguen las **técnicas** más importantes hasta la fecha.
3. Resolver el problema de **control** en un entorno de juegos con DRL. Utilizando la herramienta *Gym* con esta finalidad. ^[40]
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**.

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.

5. Analizar el proceso de aprendizaje y la experiencia obtenida por el agente con la finalidad de validar su progreso y determinar cuando está logrando aprender.
6. 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 Estructura del proyecto

Este proyecto está formado por 9 capítulos repartidos en 4 partes, junto con un apéndice formado por otros 3 capítulos más.

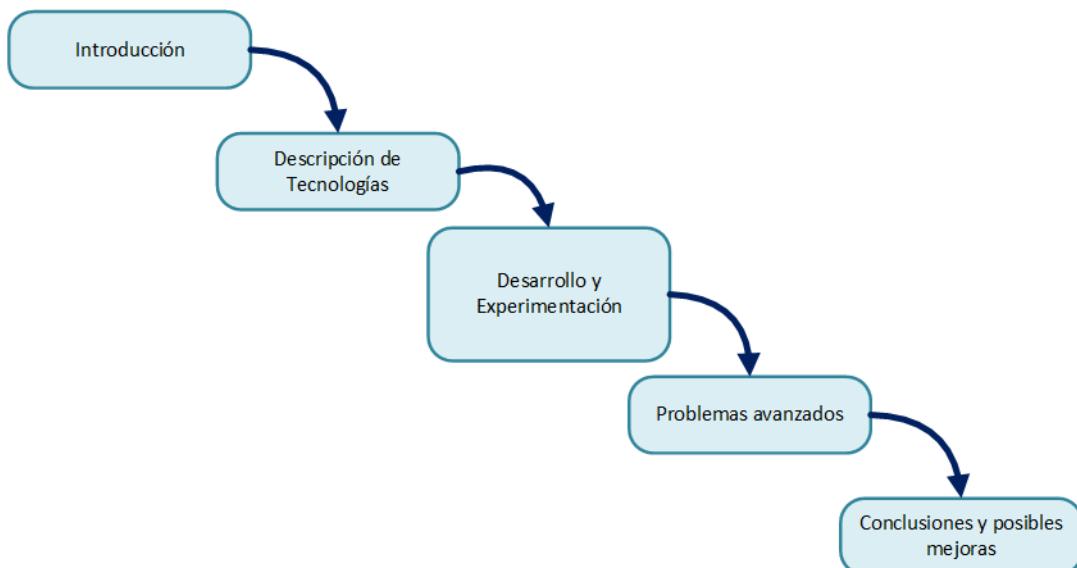


Figura 1.3: Modelo de cascada.

Estas partes son:

- **Introducción:** Se habla del contexto, motivaciones y objetivos que se esperan del proyecto.
- **Descripción de Tecnologías:** Tecnologías que se combinan y usan para resolver problemas de IA, tales como deep learning (DL), aprendizaje por refuerzo (RL) y la combinación de ambas; aprendizaje por refuerzo profundo (DRL).
- **Desarrollo y experimentación:** Se trata las herramientas y recursos actuales que existen para aplicar estas técnicas y se analizan los resultados obtenidos en varios entornos seleccionados.
- **Posibles Mejoras y conclusiones:** Se realiza un análisis crítico del trabajo y se exponen las sensaciones y conceptos adquiridos.
- Además, se ha añadido un apéndice con información relevante sobre **configuración en la nube**, **uso de la máquina virtual** y **Stable Baselines** como herramienta alternativa.

El planteamiento final del proyecto puede observarse de una forma más detallada en el diagrama de Gantt de la figura 1.4:

Aprendizaje por refuerzo profundo para la resolución de problemas - Estructura Proyecto

	Enero 1st	Enero 2nd	Febrero 1st	Febrero 2nd	Marzo 1st	Marzo 2nd	Abril 1st	Abril 2nd	Mayo 1st	Mayo 2nd	Junio 1st	Junio 2nd	Julio 1st	Julio 2nd
Inicios proyecto														
Planificación y extracción de tareas														
Extracción de inconvenientes iniciales														
Análisis de Tecnologías														
Análisis de requisitos														
Redes Neuronales(DL)														
Análisis de los componentes DL														
Aprendizaje por Refuerzo(RL)														
Análisis de los componentes RL														
Técnicas de RL														
Aprendizaje por Refuerzo Profundo(DRL)														
Técnicas de DRL														
Análisis de limitaciones y recursos necesarios														
Análisis de posibles agentes a realizar														
Desarrollo y Experimentación														
Especificación de librerías														
Especificación de entornos														
Especificación de esquema de desarrollo														
Administración de recursos en la nube														
Evaluación de pruebas MountainCar-v0														
Evaluación de pruebas Super Mario Bros														
Evaluación resultados obtenidos														
Estado del arte														
Investigación de tecnologías														
Análisis de AlphaGo														
Análisis de AlphaStar														
Evaluación del campo de investigación actual														
Conclusiones y posibles mejoras														
Documentación														
Ordenamiento de información y diagramas														
Declaración de memoria final														
Preparación de presentación final														

Figura 1.4: Diagrama de Gantt.

Métodos

2	Redes Neuronales	25
2.1	Funciones de activación (h)	
2.2	Funciones de salida (h')	
2.3	Funciones de coste	
2.4	Algoritmos de optimización	
2.5	Redes Neuronales Convolucionales (CNN)	
3	Aprendizaje por refuerzo (RL)	35
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)	
	53	
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**.[28]

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. [29] [54] [8] [56] [55]

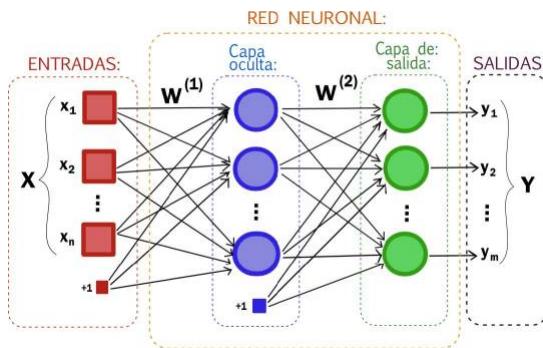


Figura 2.1: Esquema simplificado de una red neuronal. [1]

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 qué 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.[12]

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)[12].

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:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (2.1)$$

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

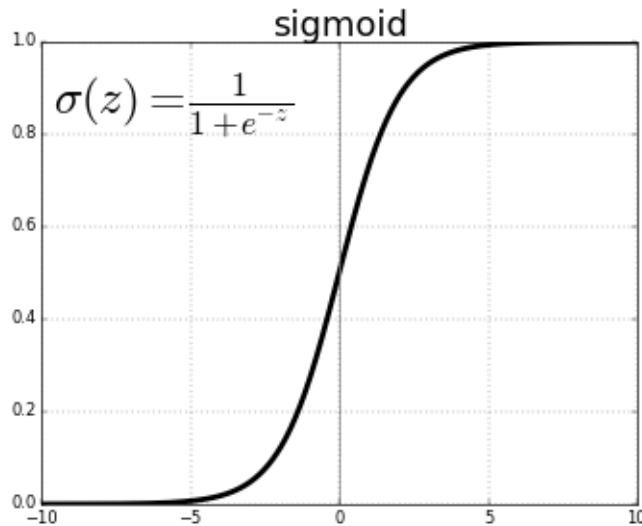


Figura 2.2: Función sigmoide en plano bidimensional. Extraída de [12].

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:

$$R(z) = \max(0, z) \quad (2.2)$$

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:

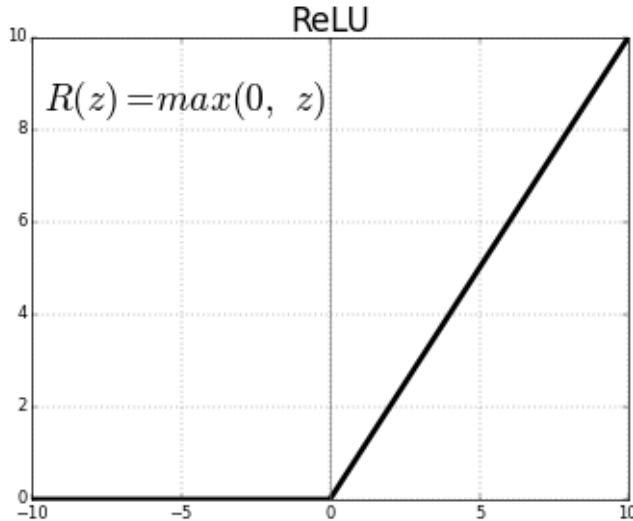


Figura 2.3: Función ReLU en plano bidimensional. Extraída de [12].

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:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}} \quad (2.3)$$

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:

$$MSE = \frac{1}{n} \sum_n^{j=1} (d_j - y_j)^2 \quad (2.4)$$

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:

$$H(p, q) = - \sum_x p(x) \log(q(x)) \quad (2.5)$$

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.

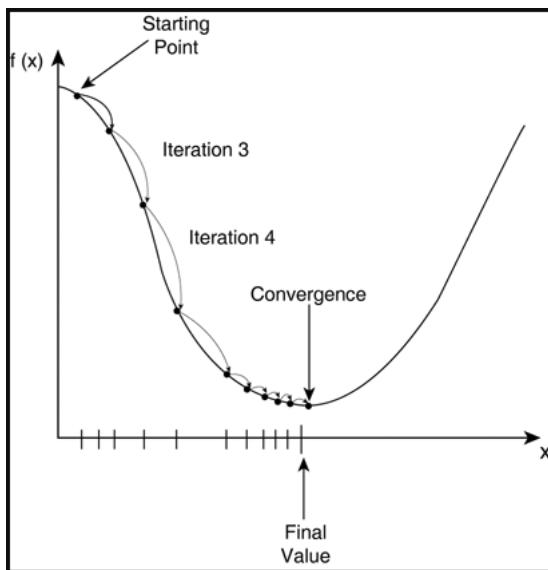


Figura 2.4: Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de [9].

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.

2.5 Redes Neuronales Convolucionales (CNN)

Las redes neuronales convolucionales (CNN) son muy similares a las redes ordinarias. De hecho, se pueden entender como una extensión de éstas. Definimos a las redes neuronales vistas hasta este punto como la parte **densa**. De ellas no se puede saber la metodología de razonamiento, sólo la calidad de los resultados que ofrecen, por lo que es complicado predecir un comportamiento de estas redes.

En las redes convolucionales, se supone que las entradas que recibe son directamente imágenes, por ejemplo, en el reconocimiento de dígitos manuscritos. Este tipo de redes mejoran el tratamiento de estos datos, tanto en rendimiento como en calidad de las respuestas.

Una imagen tiene una dimensión predeterminada, supongamos por ejemplo 800x800. Esto supone un total de 640000 elementos en el vector de características para cada muestra, eso asumiendo que la imagen es en escala de grises. Si es a color, cada pixel tiene una tripleta de valores RGB, lo que supone $640000 * 3 = 1920000$ valores por muestra de entrada. Contar con tantos parámetros desde la capa de entrada ya supone un gran límite para la red, existiendo mayor facilidad y tendencia al sobreajuste.

La parte **convolucional** de la red puede ser entendida como una etapa de **preprocesamiento** de las imágenes para extraer las características más importantes de las mismas, y estas son las que llegarían a la parte **densa** de la red, como se estaba mencionando anteriormente.

Vamos a ver de forma resumida las distintas capas que constituyen esta parte convolucional de la red.

2.5.1 Capa convolucional

Estas capas realizan un proceso llamado **convolución**. Reciben la imagen como entrada y luego aplica sobre ella un **filtro** o **kernel** que nos devuelve una matriz de datos, como la imagen, solo que con una características concretas destacadas. Por ejemplo, un filtro puede ser usado para la detección de bordes de una imagen únicamente.

Ese filtro tiene una **dimensión** (5x5, por ejemplo) lo cual hace que esos mapas de características mencionados reduzcan la cantidad de parámetros (dimensión de la matriz).

Una capa convolucional, a su vez, puede tener más de un filtro, lo que se conoce como **canales** (channels) del mapa de características. La idea es que cada filtro detecte características específicas de la imagen (bordes, patrones, etc).

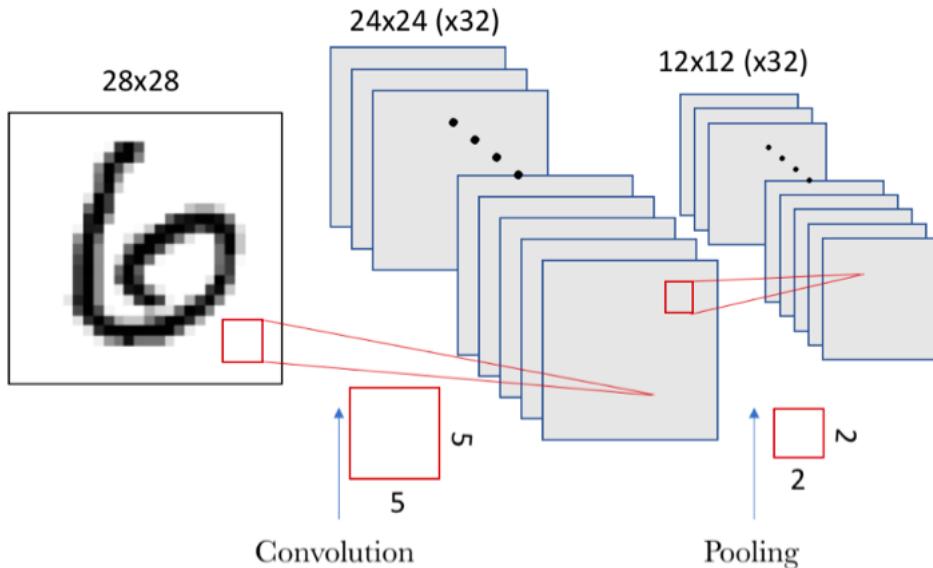


Figura 2.5: Aplicación de capa convolucional a imagen de entrada. Imagen extraída de [61].

En la figura 2.5, tendríamos 32 filtros 5x5 diferentes, dado que obtenemos 32 matrices 24x24 a partir de una sola imagen 28x28.

2.5.2 Capa de reducción o Pooling

Esta capa suele situarse siempre inmediatamente después de una capa convolucional, aunque también es posible realizar más de una capa pooling seguida. Su utilidad, en resumen, consiste en reducir las dimensiones de los mapas de características que se mencionaban anteriormente aún más. Irremediablemente, al realizar este proceso, se pierde información por el camino. Aunque el sentido común pueda determinar en un principio que esto es negativo, generalmente es positivo.

La disminución en el tamaño o dimensión de los mapas de características produce una reducción de sobrecarga en el cálculo en las capas venideras, haciendo viable el procesamiento de imágenes con una resolución mayor. Además, es una representación más **genérica** del problema que se trata de resolver, por lo que implícitamente reduce la tendencia al sobreajuste por parte de la red.

Hablando sobre la operación de reducción concretamente, se suele dividir la imagen en *submarcos* o un conjunto de rectángulos. En cada una de estas *subzonas*, se suelen utilizar la operación de **Max-Pooling**, la cual quiere decir que nos quedamos con el valor más alto, como se aprecia en la figura siguiente:

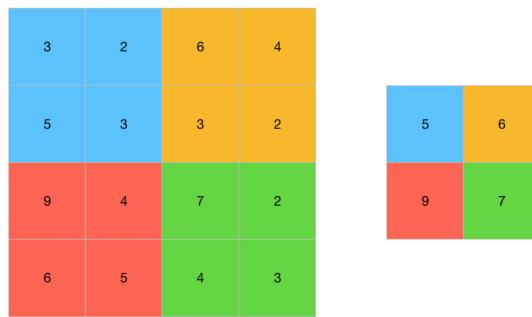


Figura 2.6: Aplicación de capa Max-Pooling a mapa de características. Imagen extraída de [53].

2.5.3 Capa Flatten

Una vez la imagen haya sido tratada en sucesivas capas convolucionales y Pooling, cada vez tendremos más canales de menor dimensión con características más aisladas y específicas del problema. Solo queda conectarla con la parte densa, para tratar esas características como veíamos anteriormente; capa de entrada, capas ocultas o intermedias y la capa de salida. ¿Cómo podemos juntar estas partes?

Ese es el propósito de la **capa Flatten**. Explicado de forma simple, coge todas las características y las concatena en un único **vector de características**. La capa de entrada de la parte densa tendrá una neurona por cada una de estas características.

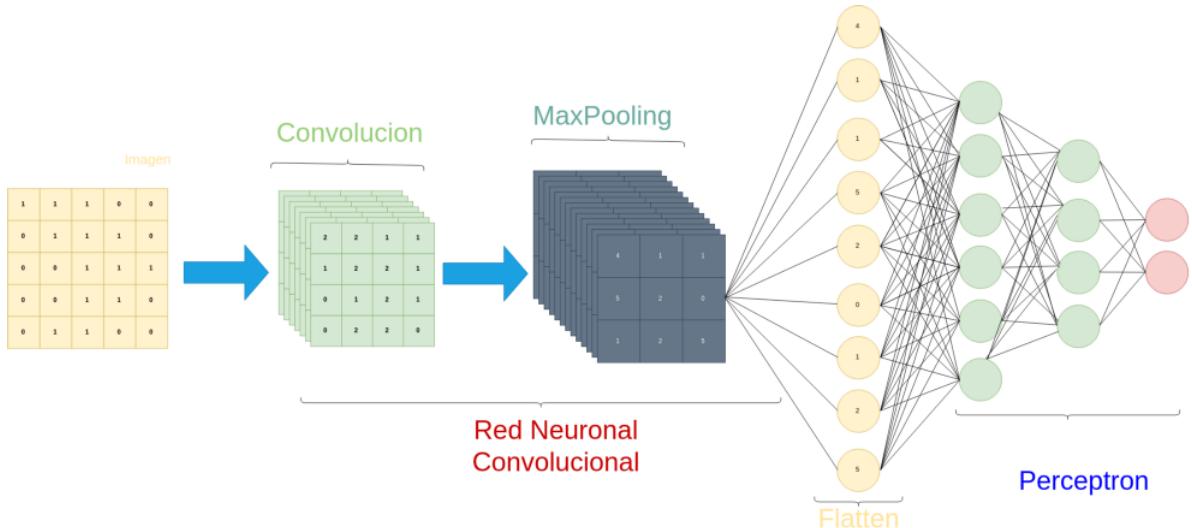


Figura 2.7: Adaptación de parte convolucional a parte densa con uso de capa Flatten. Imagen extraída de [11].



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.

Explicaremos todos los conceptos y componentes que conforman los problemas a partir de un marco de trabajo matemático llamado **Procesos de Decisión de Markov (MDPs)**. Los MDPs nos permitirán modelar un problema de una forma que los agentes de RL puedan aprender a resolverlos. [31] [55] [3]

3.1 Componentes del RL

Los dos principales componentes del RL son el **entorno** y el **agente**. El agente es un tomador de decisiones en el entorno, y por tanto, implementa 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 realizando **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.

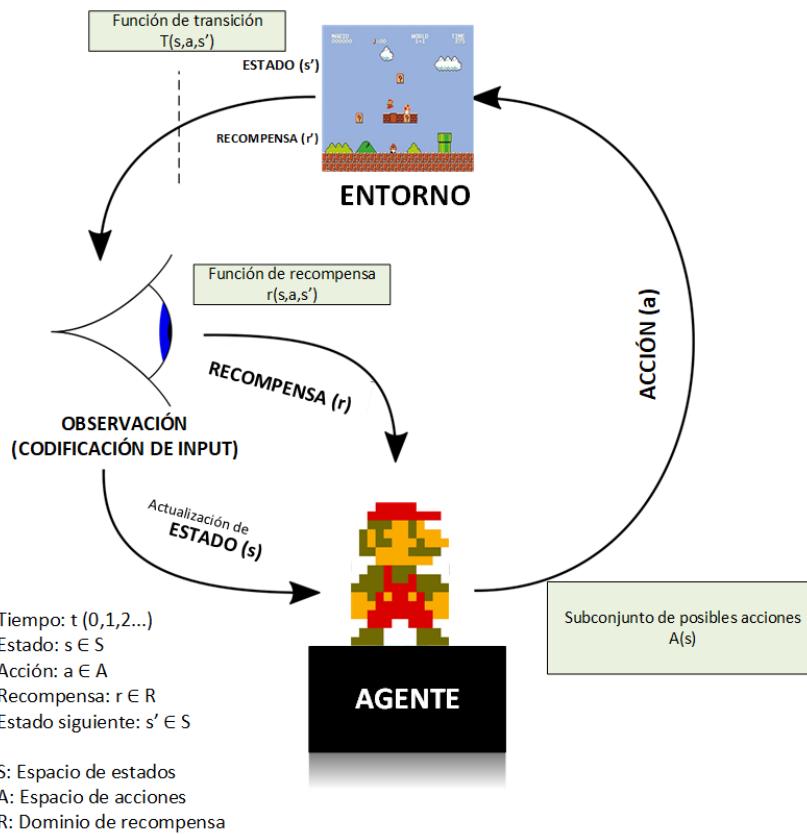


Figura 3.1: Esquema de componentes del RL. Imagen extraída y posteriormente modificada de [64].

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:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \quad (3.1)$$

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.[\[62\]](#) [\[30\]](#)

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**.

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.

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 **paso** o **paso**. Es una unidad de tiempo en la que se completa cada ciclo de interacción, el cual depende del problema específico.

En cada *paso*, 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** (SARS'). 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 pasos**, veremos esto con detenimiento un poco más adelante también.

La sumatoria de todas la recompensas desde un estado hasta el final del episodio es denominado **recompensa acumulada** y con ella, en muchos casos, 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 teórico probabilístico que es utilizado para describir un problema de RL. 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 de alcanzar un estado, dado el estado actual y la acción, es totalmente **independiente** de las interacciones previas. 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.

$$P(s'|s, a) = P(s'|s, a, S_{t-1}, A_{t-1}, S_{t-2}, A_{t-2}, \dots) \quad (3.2)$$

Siendo S el espacio de estados, A el espacio de acciones, t el *paso* actual, $s' = S_{t+1}$, $s = S_t$ 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. Se define de la siguiente forma:

$$\tau(s, a, s') = p(s'|s, a) = P(S_t = s' | S_{t-1} = s, A_{t-1} = a) \quad (3.3)$$

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:

$$\sum p(s'|s, a) = 1, \forall s \in S, \forall a \in A(s), s' \in S \quad (3.4)$$

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.

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:

$$\begin{aligned} r(s, a) &= \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a] \\ r(s, a, s') &= \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a, S_t = s'] \\ R_t &\in R \end{aligned} \quad (3.5)$$

Con R nos referimos al dominio que tiene las señales de recompensa. \mathbb{E} es la esperanza matemática.

En muchos problemas es importante calcular la **recompensa acumulada**, o llamada en inglés **return**. Se trata simplemente de la suma de todas las recompensas a partir del estado s_t que se han ido calculando durante un episodio hasta su final (s_T):

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T \quad (3.6)$$

Siendo T el número total de *pasos* 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 **pasos** 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 *pasos* 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 *pasos* 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 *pasos* 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)**.

También sirve como reductor de **incertidumbre** en las estimaciones. Dado que el futuro es incierto, y cuánto más *pasos* miramos hacia adelante, mayor factor estocástico y variabilidad 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

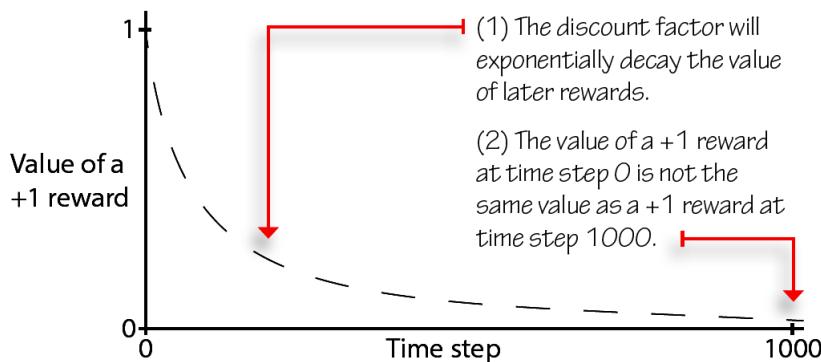


Figura 3.2: Ilustración gráfica del factor de descuento sobre las recompensas. Extraído de [31].

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

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T \quad (3.7)$$

Podemos simplificar esta ecuación y tener una más general:

$$G_t = \sum_{k=0}^T \gamma^k R_{t+k+1} \quad (3.8)$$

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**:

$$v_\pi(s) = \mathbb{E} \pi[G_t | S_t = s] \quad (3.9)$$

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:

$$v_\pi(s) = \mathbb{E} \pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right] \quad (3.10)$$

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

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad (3.11)$$

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:

$$\pi_* \geq \pi' \forall \pi' \quad (3.12)$$

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, la función de *estado-valor* trata de estimar la recompensa acumulada de una política π después de haber realizado una acción a en un estado s y continuar hasta el estado terminal según π :

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi [G_t | S_t = s, A_t = a] \\ q_\pi(s, a) &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \end{aligned} \quad (3.13)$$

Nótese que a no tiene necesariamente que haber sido elegida según π

3.3.3 Ecuación de Bellman

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

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi [G_t | S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned} \quad (3.14)$$

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**.

Se puede extraer una conclusión a partir de este concepto. En una política óptima π_* , 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:

$$\begin{aligned} v_{\pi_*}(s) &= \max(q_{\pi_*}(s, a)) \\ a \in A(s) \end{aligned} \quad (3.15)$$

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 :

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s) \quad (3.16)$$

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-valo*r 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 aproximar la función *estado-valo*r de una política recorriendo el espacio de acciones ($A(s)$) y mejorando dichas estimaciones de forma **iterativa**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] \quad (3.17)$$

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-valo*r 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 para la acción seleccionada y 0 para el resto.

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 de la recompensa y el valor estimado actualmente 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.

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**:

$$\pi'(s) = \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')] \quad (3.18)$$

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-valor* con el **algoritmo iterativo de evaluación** para una política π en un problema MDP (Ecuación 3.17).
2. Calculamos la función *acción-valor* con el **algoritmo de mejora de política** el cual parte de la función de *estado-valor* previamente calculada (Ecuación 3.18).
3. Utilizando la función *acción-valor*, podemos generar una nueva política.

4. Recalculamos la función *estado-valor* de esta nueva política y repetimos el proceso siempre que haya un cambio significativo positivo en las estimaciones de la misma (mayor que θ) (Ecuación 3.18).

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

El método anteriormente comentado es muy dependiente de los valores iniciales de v . En general, si el número de iteraciones es infinito, el método siempre converge, pues equivale aproximadamente a una búsqueda exhaustiva.

La mayoría de problemas que tratan de resolverse con RL son demasiado complejos y se van a necesitar demasiadas iteraciones. En estos casos, este método iterativo no podrá dar con una solución práctica en un tiempo razonable.

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. Se trata de ajustar un **equilibrio** entre ambas partes.

Existen varias formas 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 cuáles no.

3.5 Aprendiendo a evaluar y mejorar políticas

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**).

El objetivo reside en **optimizar** las estimaciones de la función *acción-valor*. 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 **Método de Montecarlo** [31] [55] consiste, en términos generales, en ejecutar un número considerable de episodios con la política a evaluar de tal manera que se disponga de un gran número 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:

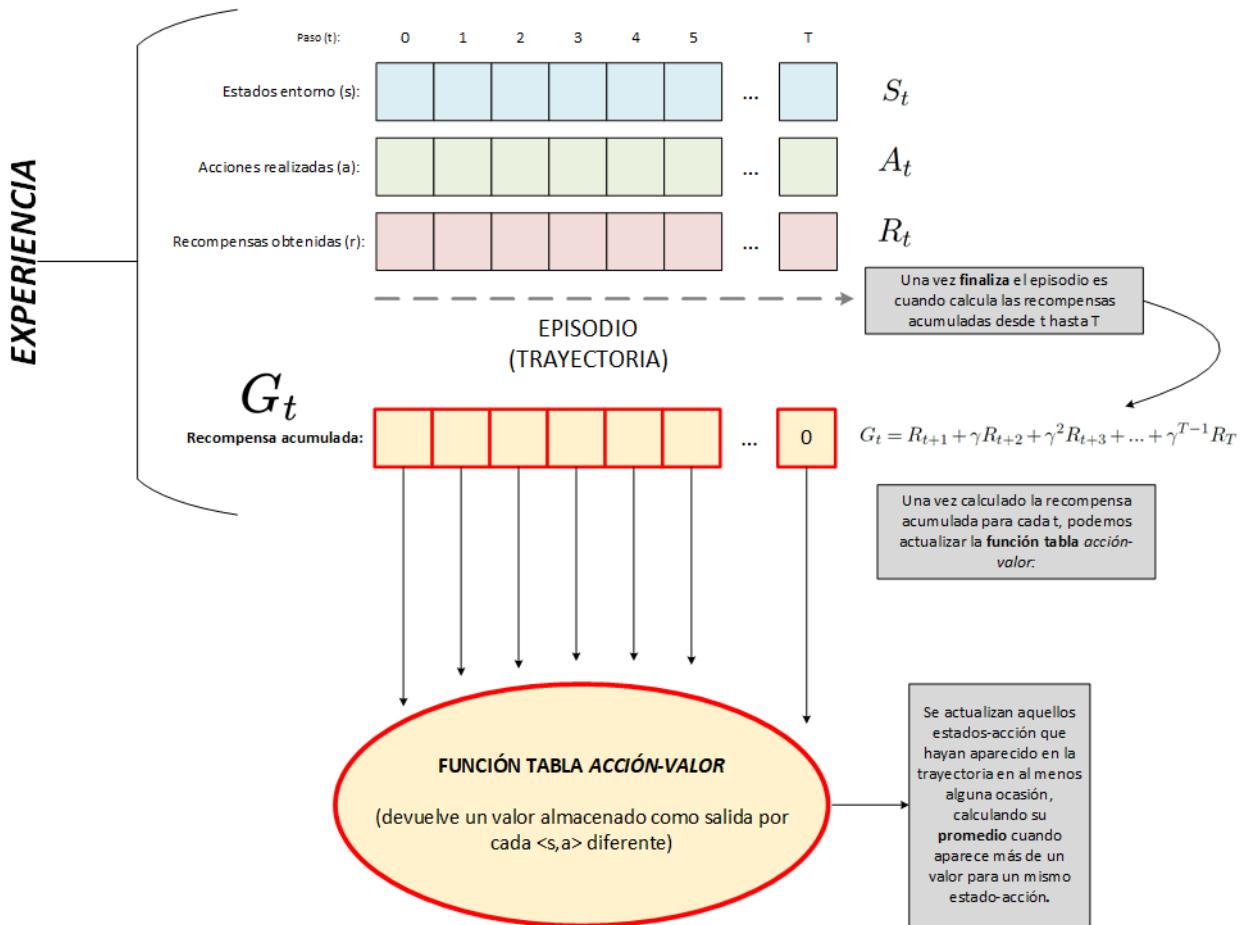


Figura 3.3: Esquema Método Montecarlo. Tabla función extraída de [55].

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 *pasos* del episodio. Este valor se denota como G_t siendo t el *paso* actual.

Ya se ha terminado de formar la trayectoria. El siguiente paso es estimar la función **acción-valor** ($q(s, a)$) simplemente calculando el promedio de recompensa acumulada obtenida en cada estado-acción $\langle s, a \rangle$ de la trayectoria. Un mismo $\langle s, a \rangle$ puede repetirse en más de una ocasión dado un episodio. Entonces, la función simplemente consiste en almacenar un valor promedio por cada estado y acción 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 *estado-valor* ($q \cdot v_\pi(s)$). La forma de hacerlo es igual, solo que para los promedios se tendría en cuenta solo los estados $\langle s \rangle$.

ALGORITMO:

```

for i = {0, ..., n_episodios}
    Calcular trayectoria del episodio
    Actualizar la función acción-valor q
    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-acción $\langle s, a \rangle$.
2. En lugar de calcular la media, quedarse con el **primer valor G** encontrado para cada estado-acción $\langle s, a \rangle$.

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 del 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.

La función *acción-valor* se actualiza al final de cada episodio. Entonces, una acción queda diluida entre todas las de su episodio. Esto ocurre porque el valor estimado de una acción depende mucho de como continúe ese episodio específicamente, ya que de ello dependerá los valores de G_t que como ya sabemos se usan para actualizar la función.

Una acción también queda diluida entre las realizadas en todos los episodios, ya que para actualizar la función *acción-valor* se consideran los valores de todos los episodios anteriores.

Esto, además, hace que este algoritmo resulte poco efectivo para resolver problemas con **sparse rewards**. Es decir, recompensas escasas ofrecidas por el entorno al agente durante el proceso de aprendizaje (como es el caso de MountainCar). [19]

En definitiva, al depender de las recompensas acumuladas que obtiene con una política que partirá siendo mala seguramente, puede hacer que tarde demasiado tiempo en converger en unas buenas estimaciones para la política mejorada.

El **Método de Diferencia Temporal (SARSA)** [31] [55] propone sustituir G_t por la suma de la recompensa actual que recibe el agente del entorno R_{t+1} tras realizar su acción A_t con una estimación del valor acumulado a partir del estado siguiente y ponderada con el factor de descuento γ . 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:

$$q_{\pi}(S_t, A_t) \leftarrow q_{\pi}(S_t, A_t) + \alpha [R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) - q_{\pi}(S_t, A_t)] \quad (3.19)$$

Ese **cambio** del que estamos hablando consiste en sumarle a la estimación actual la diferencia entre su valor actual y la nueva estimación.

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

De esta forma, podemos ir calculando las nuevas funciones valor durante cada *paso* y recompensa que obtiene sin tener que esperar a que finalice un episodio, aunque no es la única ventaja.

Esto hace que el sesgo en las estimaciones sea más alto que en MonteCarlo, aunque aceptable. La principal ventaja es que el algoritmo pueda converger mucho antes. [23]

La conclusión final es que los métodos de diferencia temporal convergen más rápido que Montecarlo en términos generales, aunque el sesgo en las estimaciones sea algo mayor. El proceso de *bootstrapping* que implementan es efectivo, aunque el sesgo debido al uso de estimaciones sea mayor.

MÉTODO DE DIFERENCIA TEMPORAL (SARSA):

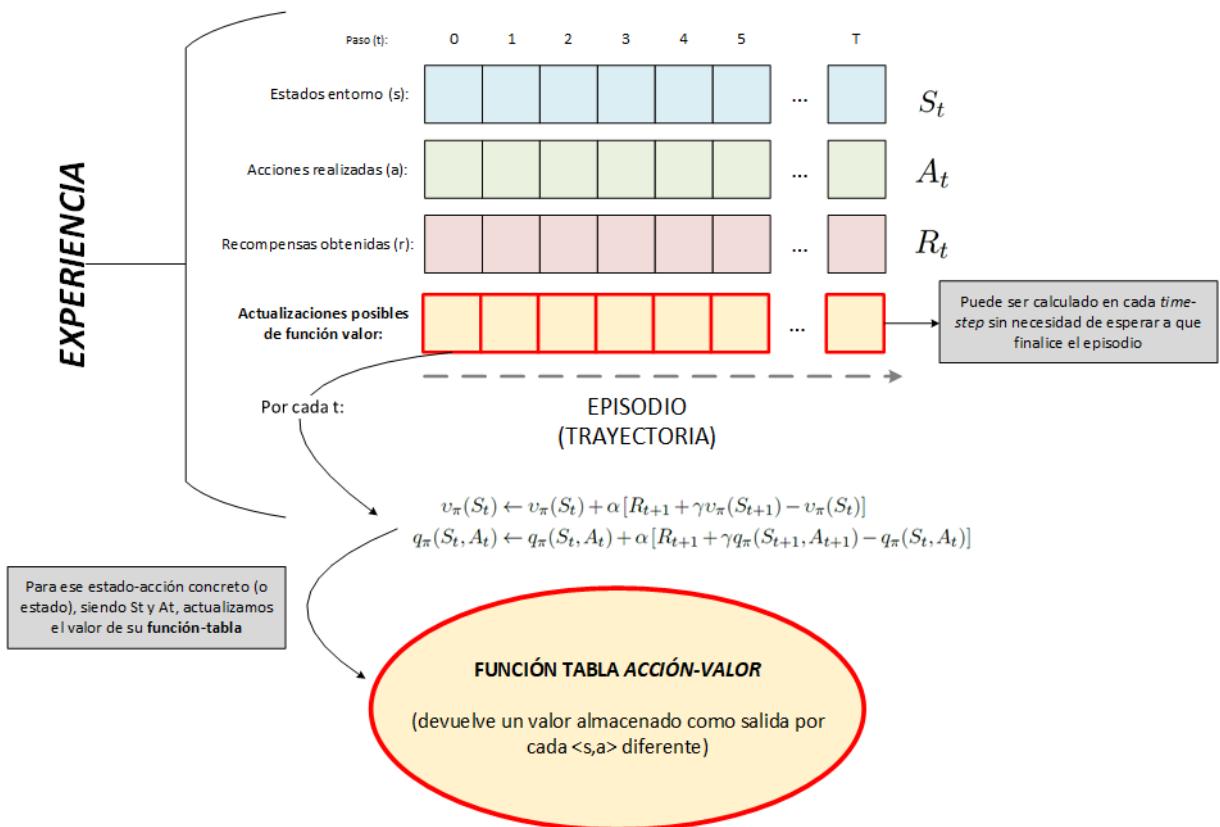


Figura 3.4: Esquema Método de diferencia temporal (SARSA). Tabla función extraída de [55].

También es posible establecer **puntos intermedios** entre Montecarlo y SARSA. No se actualiza las estimaciones de la función valor ni cuando finaliza un episodio, ni de forma inmediata cada vez que se produce un *paso*. 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 acumuladas.

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.

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

$$\begin{aligned} \text{SARSA-MAX} &\rightarrow \max_{a \in A(S_t)} Q(S_{t+1}, a) \\ Q_\pi(S_t, A_t) &\leftarrow Q_\pi(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a \in A(S_t)} Q(S_{t+1}, a) - Q(S_t, A_t) \right) \end{aligned} \quad (3.20)$$

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 estimación con la suma de recompensa actual y estimación en el estado 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:

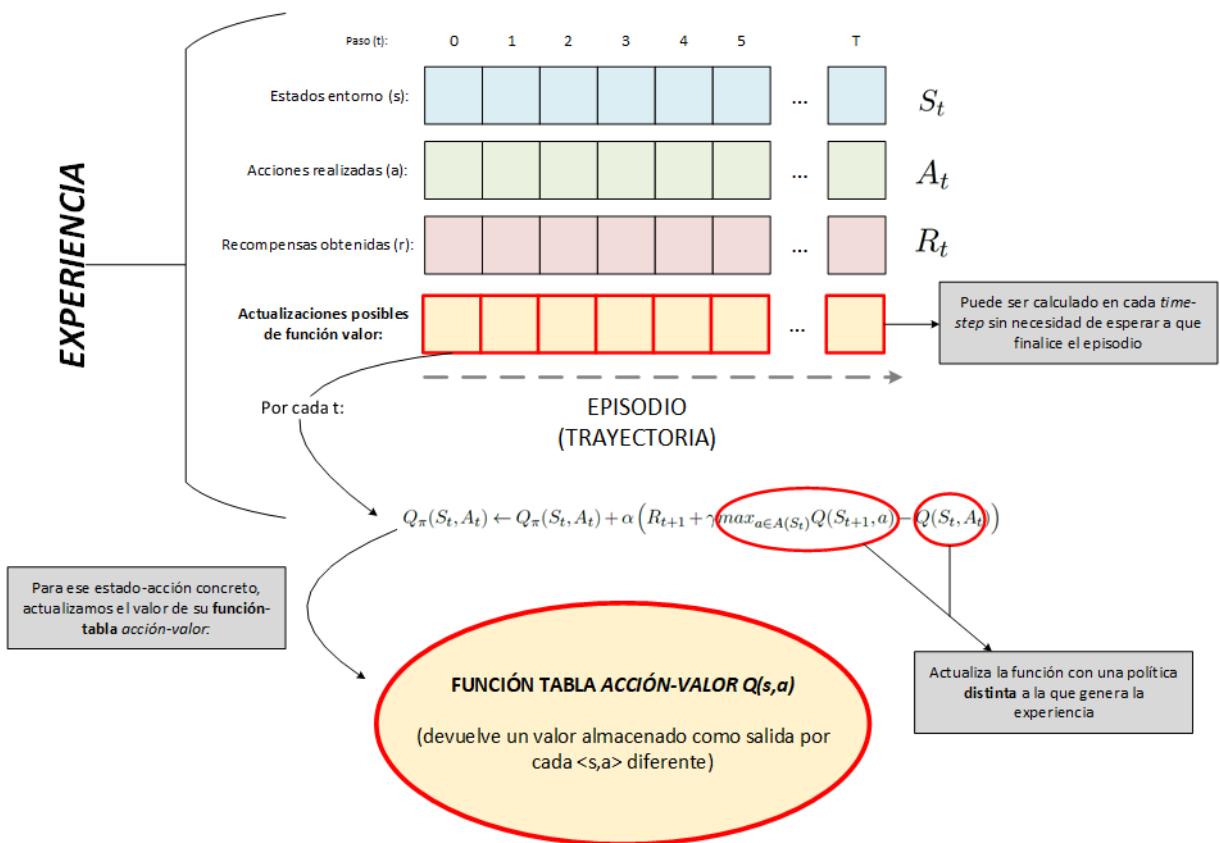


Figura 3.5: Esquema Método Q-Learning. Tabla función extraída de [55].

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** [20], 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.



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. Estos métodos guardaban un valor estimado por cada estado-acción diferente del entorno en una estructura tipo tabla, en eso se basaban esas funciones valor de la política.

¿Y si en lugar de utilizar eso usamos redes neuronales para esas funciones y políticas? 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. Sobre todo para abordar problemas complejos, no solo por su espacio de estados y acciones, sino por la cantidad de variables o información que pueden tener cada uno de ellos. Todo esto sin contar con que el espacio de acciones puede ser continuo, lo que directamente no puede ser abordado 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)

DQN [35] [23] es uno de los algoritmos más populares que hay dentro de este campo, 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 resolverse a partir de los píxeles crudos, de la propia imagen,

combinándose Q-Learning con redes neuronales CNN (apartado 2.5).

Esto resulta muy interesante, las observaciones que el agente hacía 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, construyendo nueva experiencia a partir de ese formato de input, todo esto como resultado de integrar CNN con RL.

Es una estrategia **basado en valor** (*value-based*). 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**.

Las redes neuronales tienen tantas neuronas en la capa de entrada como variables una observación del estado s o igual al vector de características procesadas si ha habido un procesamiento convolutivo previo, como se mencionó anteriormente. Por otro lado, la capa de salida tiene tantas neuronas (*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, aplicar una distribución de probabilidad para favorecer la exploración, o hacer ϵ -greedy, etc (esta decisión sería parte de la política π). Podemos observar dicha explicación en la figura 4.1.

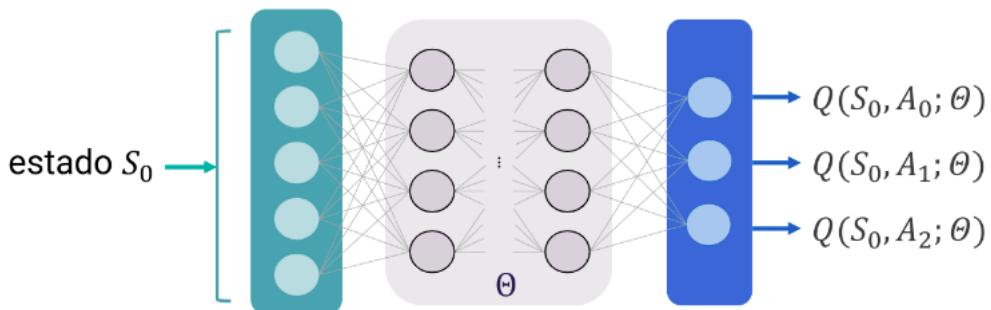


Figura 4.1: Esquema red neuronal, acciones discretas. Imagen extraída de [55].

El símbolo Θ hace referencia al conjunto de pesos.

Por otra parte, podemos enfrentarnos a un problema que tenga acciones continuas y, por tanto, tenga un espacio infinito de valores. Se puede adaptar para DQN incluyendo la acción como entrada de la red (aunque esto no pertenece a DQN base). 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 tratarse de acciones continuas, no podemos calcular todas las posibilidades a partir de la entrada s , como en el caso anterior. Esto hace que DQN no sea apropiada para resolver este tipo de problemas continuos por sí misma:

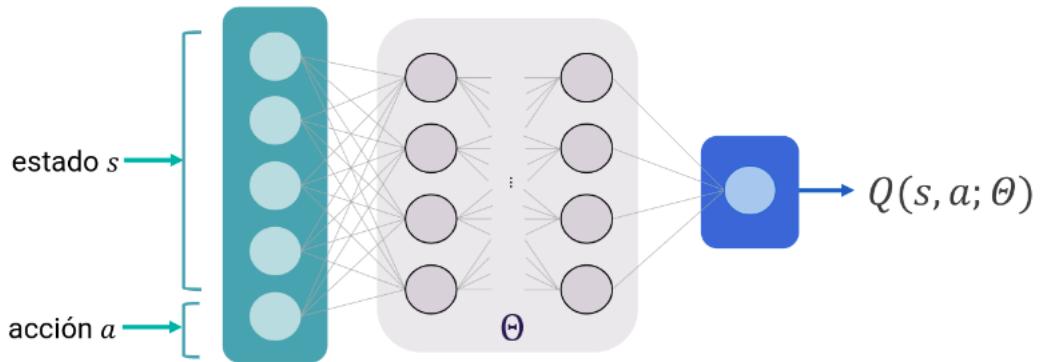


Figura 4.2: Esquema red neuronal, espacio de acciones continuo. Imagen extraída de [55].

Aunque esta adaptación a DQN podría combinarse con otras técnicas para complementarse y que pudiese funcionar bien, como en DDPG (se verá en el apartado 4.3.2). También se podría adaptar **discretizando** el espacio de acciones, pero esto limita y simplifica en exceso la complejidad del problema.

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 eliminar la relación temporal entre acciones con la memoria de experiencia, mejorando de esta manera la **exploración** de dependencias entre zonas, dado que se elimina ese orden 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 complejidad 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:

$$Q(S_0, A_0) \leftarrow Q(S_0, A_0) + \alpha [R_1 + \gamma \max_{a \in A} Q(S_1, a) - Q(S_0, A_0)] \quad (4.1)$$

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.

Las redes neuronales que se incluyen en estas técnicas de RL se componen de las mismas partes y funciones que se explicaron en el apartado 2. Será necesario una **función de perdida**, por ejemplo se podría utilizar el error cuadrático (*MSE*) de la siguiente forma:

$$MSE = \mathbb{E}_{\pi}[(Q(S, A) - Q(S, A; \theta))^2] \quad (4.2)$$

La finalidad es medir la diferencia entre el valor real de Q y el obtenido por la red que se está tratando de mejorar, para luego aplicar **gradiente descendente** y optimizar así la función de error. Se obtendrá una **actualización de los pesos** en la red ($\Delta\Theta$) denotada con la siguiente forma:

$$\begin{aligned} \Delta\Theta &= \Delta w = -\eta \frac{\delta E}{\delta w} \\ \frac{\delta E}{\delta w} &= 2(Q(S, A) - Q(S, A; \Theta)) \frac{\delta Q(S, A; \Theta)}{\delta w} \\ \frac{\delta Q(S, A; \Theta)}{\delta w} &= \nabla_{\Theta} Q(S, A; \Theta) \\ \Delta\Theta &= -2\eta(Q(S, A) - Q(S, A; \Theta))\nabla_{\Theta} Q(S, A; \Theta) \end{aligned} \quad (4.3)$$

Como $Q(S, A)$ no es conocido, aplicamos la expresión de Q-Learning y obtenemos esto:

$$\begin{aligned} Q(S_0, A_0) &\leftarrow Q(S_0, A_0) + \alpha[R_1 + \gamma \max_{a \in A} Q(S_1, a) - Q(S_0, A_0)] \\ \Delta\Theta &= \alpha [R_1 + \gamma \max_{a \in A} Q(S_1, a; \Theta) - Q(S_0, A_0; \Theta)] \nabla_{\Theta} Q(S_0, A_0; \Theta) \end{aligned} \quad (4.4)$$

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) extraído 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 **suelen** utilizar **dos redes Q**. Nos referimos a dos conjuntos de pesos diferentes, no a la arquitectura de la red como tal, la cual es la misma para ambos conjuntos.

Se trata de una variante que tiene como objetivo principal hacer que el algoritmo sea más estable y está propuesta en el artículo original [35]:

- **Red Q:** Es la red que define la función $Q(S, A, \Theta)$.
- **Red objetivo:** Se trata de un estimador imparcial del error medio cuadrático de Bellman utilizado en entrenar la red Q. Esta red objetivo se sincroniza con la red Q después de un predefinido periodo de interacciones, produciéndose un acoplamiento entre las dos redes.

En resumen, los pesos Θ^- , los cuales forman la **red objetivo** (\bar{Q}), se actualizan con menor frecuencia a partir de los pesos (Θ) (realizando copias de esos parámetros) que forman la red **red Q**.

La estimación de recompensa acumulada a partir de un estado siguiente a una acción realizada es aproximada por parte de \bar{Q} y esos valores son utilizados para entrenar la red Q, que a su vez es la que predice las acciones y estados siguientes en el entorno:

$$\Delta\Theta = \alpha(R + \gamma \max_{a \in A} \bar{Q}(S', a; \Theta^-) - Q(S, A; \Theta)) \nabla_\Theta Q(S, A; \Theta) \quad (4.5)$$

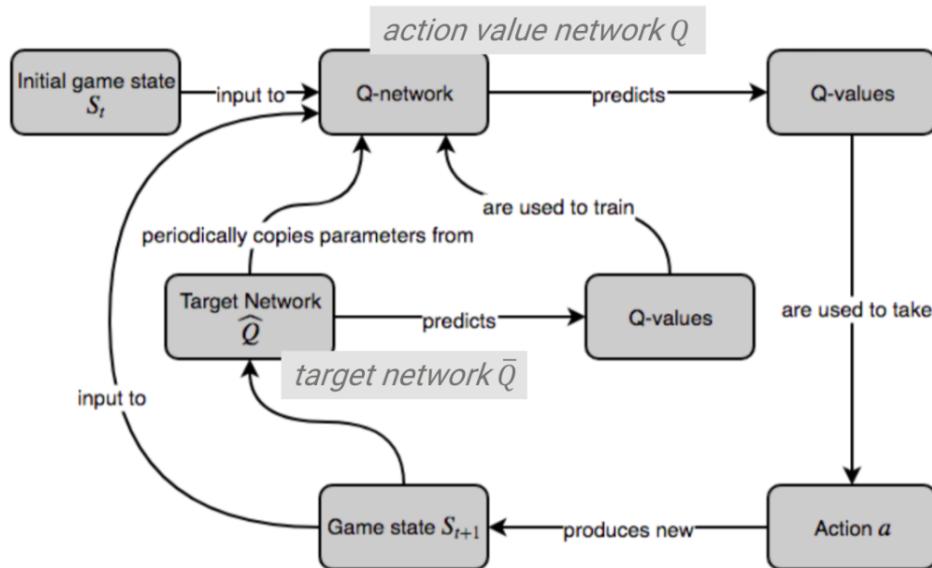


Figura 4.3: Esquema Método Deep Q-Learning. Extraída de [3] [55].

```

for i={0,...,n_episodios}
    for t={0,...,n_entrenamiento}

        Elegir acción con política generadora de experiencia
        Aplicar acción y obtener estado siguiente y recompensa
        Almacenar estado, acción, recompensa y estado siguiente en memoria D

        Muestrear un mini-batch de experiencias de tamaño n_batch de la memoria D
        Obtener salida de predicción de cada experiencia usando política objetivo
        Optimizar red generadora de experiencia con gradiente descendente

        Cada C iteraciones , actualizar red objetivo a partir de la generadora
        de experiencia
    
```

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 era construir una función *acción-valor* $Q(s, a)$ que estimara la recompensa acumulada futura a partir de una acción en un estado del entorno de la mejor forma posible. No se almacena una política explícitamente. En su lugar, la política se encuentra de forma implícita a partir de esta función $Q(s, a)$, cogiéndose la acción con mejor valor estimado (y teniendo en cuenta la estrategia de exploración).

Por otro lado, en los **Métodos basados en política** o **métodos de política gradiente**, la construcción de la política se encuentra representada de forma explícita y es almacenada en memoria durante el entrenamiento ($\pi : s \rightarrow a$) sin necesidad de una función Q .

Entonces, para esta familia de algoritmos, el objetivo no es estimar valores de recompensa a partir de estados y acciones como en los basados en valor. Se quiere recolectar experiencia y después derivar los parámetros a una política óptima (o al menos a optimizaciones locales), utilizando para ello el gradiente.

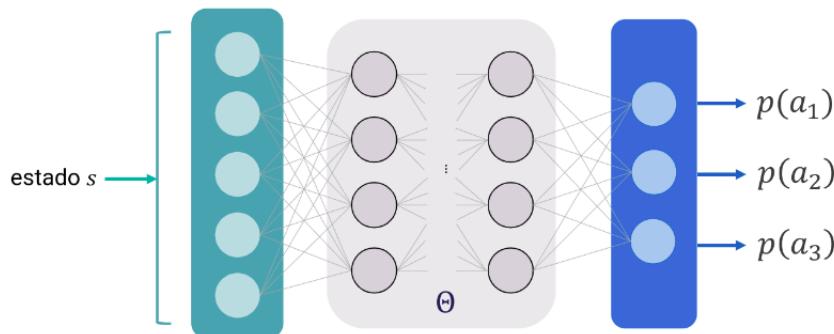


Figura 4.4: La red neuronal es la propia política para los métodos basados en políticas. Extraída de [55].

De esta forma, intenta calcular directamente cuál es la mejor acción dado un estado, ofreciendo como salida una distribución de probabilidades de las acciones, pudiendo usarla directamente como estrategia de **exploración** al no tener siempre por qué seleccionarse la acción con el porcentaje más alto.

La principal ventaja que presenta con respecto a los métodos basados en valor se refleja en los problemas con un espacio de acciones continuos. Como ya mencionamos, DQN tiene limitaciones para trabajar con este tipo de problemas. Sin embargo, estos métodos basados en política pueden ofrecer como salida directamente un valor continuo que determina la acción a realizar. Por otro lado, ofrece menos información que los métodos basados en valor, ya que no se sabe la recompensa que espera al tomar decisiones en cada momento.

El algoritmo REINFORCE [55] [31] [52] 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.

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

$$\nabla_{\Theta} U(\Theta) \approx \frac{1}{K} \sum_{i=1}^K \sum_{t=1}^n \nabla_{\Theta} \log_{\pi_{\Theta}}(A_t^{(i)} | S_t^{(i)}) R^{(i)} \quad (4.6)$$

$U(\Theta)$ es conocida como la función “**máximo beneficio**” y 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 se aplica **gradiente ascendente** para optimizar la función $U(\Theta)$. La derivación que se muestra en las fórmulas anteriores no es trivial.

Existen muchas variantes como Vanilla Policy Gradient (VPG) [37] aunque, de nuevo, no es el objetivo de este trabajo.

While no resuelto

Generar K trayectorias de longitud n según política
 Calcular recompensa obtenida para cada trayectoria
 Actualizar pesos de la red para estimar el gradiente de la
 función "Máximo beneficio"

for i=1,2...

for actor=1,2,...
 Ejecutar política antigua en entorno durante T pasos
 Calcular el valor de ventaja estimadas durante T

Optimizar la función objetivo (L-CLIP) con K épocas y batch-sizes de tamaño M

4.3 Métodos actor-critic

Llegados a este punto, conocemos los métodos basados en valor y basados en políticas. Sabemos cuáles 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 un principal inconveniente, 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 o estado-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:** Trata de seleccionar la mejor acción posible para el estado en el que se encuentra.
- **Critic:** Indica la estimación de recompensa acumulada que se espera a partir del estado actual ($V(S; \Theta_v)$)

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

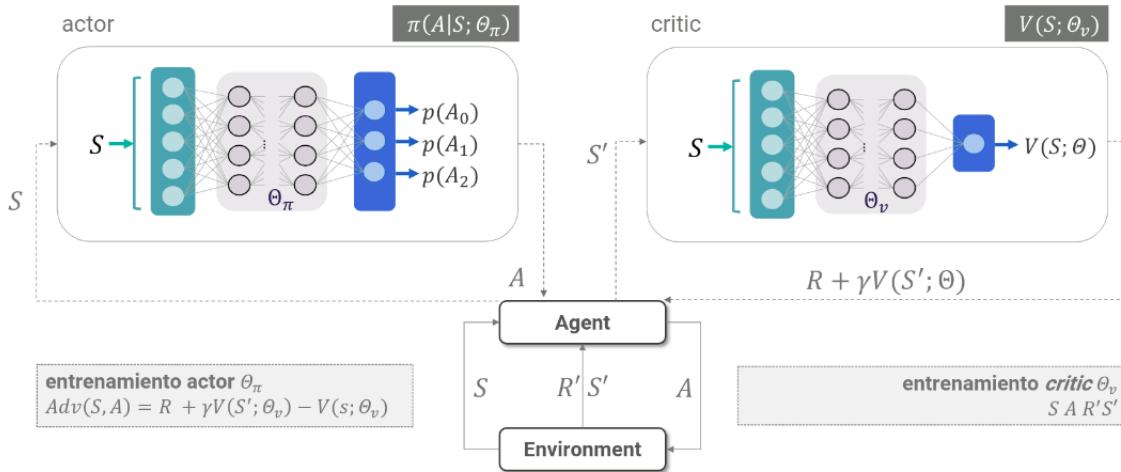


Figura 4.5: Esquema de metodología actor-critic. Extraída de [55].

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 de recompensa acumulada que se espera. Dicho valor es recogido 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 utilizando las ecuaciones de Bellman vistas en el apartado 3.3.3. A continuación se explica las técnicas DDPG y A2C, explicando antes A3C para que se entienda mejor éste último.

4.3.1 Proximal Policy Optimization (PPO2)

Proponen una nueva familia de métodos de **gradiente de políticas** [26] [36] para el aprendizaje por refuerzo, que alternan entre el muestreo de datos a través de la interacción con el entorno (experiencia) y la optimización de una **función objetivo** mediante el **gradiente estocástico ascendente**.

Para esta técnica, se utiliza los valores de ventaja (apartado 3.3.4). La política antigua se ejecuta durante un periodo T (menos de un episodio) en la que se estima esos valores de ventaja. Cuando finaliza, optimiza los pesos de la política en épocas organizadas en *mini-batches* y finalmente se sustituye esta política nueva por la antigua.

Para describir este proceso se debe tener claro unos términos antes. Primero definimos el **ratio de probabilidad**:

$$r_t(\Theta) = \frac{\pi_\Theta(A_t|S_t)}{\pi_{\Theta_{old}}(A_t|S_t)} \quad (4.7)$$

Sirve para considerar como de grande ha sido una actualización en el conjunto de pesos de una política. Cuanto más se aleje $r_t(\Theta)$ del valor 1, más grande será dicho cambio en los parámetros, ya que si introducimos en ese ratio la propia política antigua (sin actualización) obtenemos lo siguiente:

$$r_t(\Theta_{old}) = \frac{\pi_{\Theta_{old}}(A_t|S_t)}{\pi_{\Theta_{old}}(A_t|S_t)} = 1 \quad (4.8)$$

Estamos hablando de optimizar una red neuronal, por lo que la función de pérdida es imprescindible, será nuestra función objetivo:

$$L^{CPI}(\Theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\Theta(A_t|S_t)}{\pi_{\Theta_{old}}(A_t|S_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r_t(\Theta) \hat{A}_t] \quad (4.9)$$

Siendo $L^{CPI}(\Theta)$ la función objetivo, \hat{A}_t denota la ventaja en ese paso t y $\hat{\mathbb{E}}_t$ indica la esperanza matemática empírica con un *mini-batch* finito de pasos. Vemos que un cambio grande de política ($r_t(\Theta)$) penaliza en dicha función.

Esta función es usada en técnicas basadas en política como *Trust Region Policy Optimization* (TRPO) [50]. Su principal problema es que la optimización $L^{CPI}(\Theta)$ lleva a unas actualizaciones de política **excesivamente grandes** si no se establecen restricciones en el proceso.

Por ello, se plantea una función novedosa dentro de esta familia de algoritmos, que en parte se basa en la explicada anteriormente:

$$L^{CLIP}(\Theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\Theta) \hat{A}_t, \text{clip}(r_t(\Theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t) \right] \quad (4.10)$$

Siendo ϵ un hiperparámetro, digamos $\epsilon = 0.2$. A esto se le conoce como **Clipped Surrogate Objective** o **Objetivo sustituto recortado**. La motivación de esta propuesta viene dada por varias razones.

El primer término dentro de \min es el propio L^{CPI} que vimos para TRPO. El segundo término, $\text{clip}(r_t(\Theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t$, modifica la función objetivo “recortando” (clip) el ratio de probabilidad, incitando a que ese ratio no varíe más del intervalo $[1 - \epsilon, 1 + \epsilon]$ para cada actualización de la política.

Finalmente, cogemos el mínimo de estos dos términos; el objetivo recortado y el no recortado. Con esta planificación, solo ignoramos el cambio de ratio de probabilidad cuando hace que el objetivo mejore y se incluye cuando hace que el objetivo empeore. El valor de ventaja sirve para determinar cuando se está mejorando o empeorando dependiendo de si

es positiva o negativa.

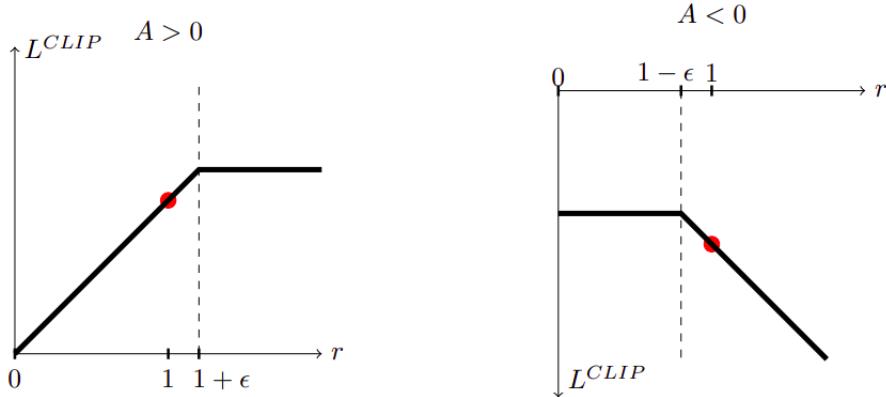


Figura 4.6: Comportamiento de función objetivo L^{CLIP} para ventajas positivas (izquierda) y ventajas negativas (derecha). El círculo rojo indica el punto inicial a partir del cual se realiza la optimización. Extraída de [26].

En cuanto al cálculo de la ventaja en un mini-lote de tamaño T, se realiza de la siguiente forma:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \dots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \quad (4.11)$$

Siendo $\delta_t = r_t + \gamma V(S_{t+1}) - V(s_t)$

Existen otras variantes de PPO que usan como función objetivo el **coeficiente de penalización adaptativo KL**. Sin embargo, no se va a entrar en mayor detalle debido a que es menos utilizado y en general funciona peor que L^{CLIP} según los experimentos realizados.

Esta estrategia presenta un buen equilibrio entre calidad de las actualizaciones de sus parámetros con el tiempo de aprendizaje. Supera a otros métodos de gradiente de políticas online; muestra por muestra, incluyendo además mejoras de simplicidad en la implementación y más generalizables en su uso.

Estos métodos son fundamentales para los avances actuales en el uso de redes neuronales profundas para el control y aprendizaje por refuerzo de locomoción 3D y *Go*, entre otros. Sin embargo, supone un **desafío** porque presentan una sensibilidad notable a la elección del tamaño de esos períodos T que mencionábamos y tamaños de los *mini-batches* anteriormente mencionados. Demasiado pequeño, y el aprendizaje es inconvenientemente lento. Demasiado grande, y la señal se ve demasiado afectada por el ruido (exploración).

4.3.2 Deep Deterministic Policy Gradient (DDPG)

DDPG [63] [34] es un algoritmo que combina las técnicas de actor-critic junto con DQN. No es posible aplicar Q-Learning directamente a un problema con espacio de acciones continuo, es imposible hacer estimaciones de todo ese espacio.

Entonces, se propone una arquitectura actor-critic en la cual la parte crítica es DQN en lugar de $V(s; \Theta)$ (se estima Q). Para ello se utiliza una **memoria de experiencias** al igual que en la propia DQN.

Se dispone de un **actor** el cual se rige por una política $\pi(s; \Theta_\pi)$, en lugar de estimar la probabilidad de cada acción, devuelve directamente la acción seleccionada (un valor numérico de a). La parte crítica es la función Q , puede hacerse para un espacio de acciones continua como se ilustró en la figura 4.2.

Al no tener valores de V , el actor utiliza el Q obtenido con la red crítica dada la acción que realiza y con esa información que devuelve Q se entrena la red actor. Ambas redes son dobles, como DQN, haciendo las actualizaciones más suaves con la ayuda de una red objetivo (*target*).

La arquitectura de DDPG tiene una ventaja para la **exploración**. Puede tratar el problema de exploración independientemente del algoritmo de aprendizaje. Se introduce **ruido** en los pesos de la red directamente, en lugar del espacio de acciones:

$$\pi'(S_t) = \pi(S_t; \Theta_\pi) + N \quad (4.12)$$

Siendo N la representación del ruido añadido en esos parámetros.

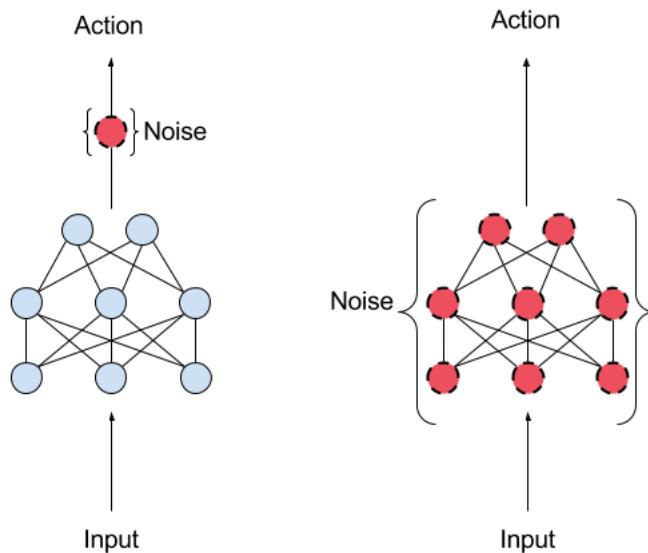


Figura 4.7: Ruido en el espacio de acciones (izquierda) frente a ruido en los parámetros de la red (derecha). Extraída de [34].

Esta técnica ha demostrado mejorar la exploración de los agentes obteniendo mejores resultados. Introducir ruido en los parámetros provoca que la exploración sea consistente a través del paso del tiempo, mientras que agregar el ruido en la acción cuando se hace ϵ -greedy conduce a cambios espontáneos de comportamiento por parte del agente.

Iniciar aleatoriamente la red crítica y actor con un conjunto de pesos diferentes.

Iniciar la red objetivo para la parte crítica
 Iniciar una memoria de experiencia

```
for episodio i=1,...,T
```

```
  Iniciar proceso aleatorio de ruido N  

  Recibir estado inicial de la observación
```

```
  for t=1,...,T
```

```
    Seleccionar acción acorde con la política actor y el ruido de exploración  

    Ejecutar acción y obtener recompensa y estado siguiente  

    Almacenar tupla de experiencia en memoria  

    Muestrear un mini-batch de N transiciones de la memoria de experiencia  

    Obtener etiqueta aproximada con red objetivo  

    Actualizar crítico minimizando función de perdida  

    Actualizar actor utilizando política de gradientes estocástica
```

```
  Actualizar la red objetivo
```

4.3.3 Asynchronous Advantage Actor-Critic (A3C)

A3C [4] [24] es útil para resolver problemas con acciones tanto continuas como discretas. Intenta realizar un procesamiento **concurrente asíncrono** de la arquitectura actor-critic (sección 4.3). El objetivo principal de esta propuesta es tratar de **mejorar el sesgo** de las estimaciones realizadas por esta estrategia al mismo tiempo que se trata de explotar de una forma más eficiente el **procesamiento en paralelo**.

Consiste en tener n agentes diferentes en entornos iguales del problema, cada uno con su propia política y función valor, propio de una arquitectura actor-critic. Llamaremos a cada uno de estos agentes *trabajadores*:

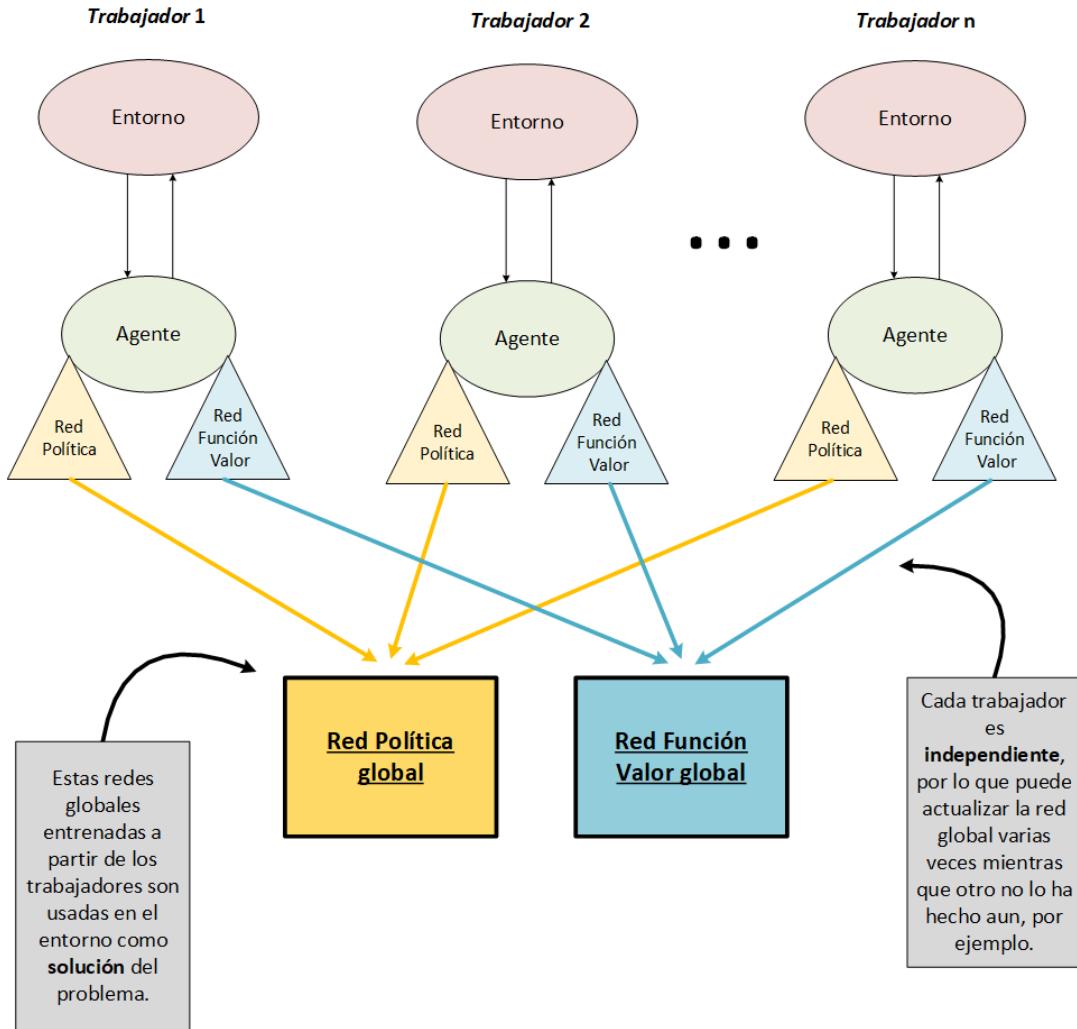


Figura 4.8: Esquema de algoritmo A3C.

La idea consiste en tener a cada uno de estos *trabajadores* funcionando en paralelo sobre copias del mismo entorno y sin ninguna relación entre ellos, trabajando sin coordinarse con el resto. Cuando un *trabajador* termina su ejecución, se tratará de actualizar una **política y función-valor global** a partir de la que tiene éste. Esto decorrelaciona los datos utilizados para el entrenamiento y mejora la convergencia del algoritmo.

La actualización de la política dependerá de la función de optimización que tenga la red neuronal que use en cuestión (Normalmente se usa gradiente descendente estocástico). Para no entrar en mayor detalle, lo denotaremos de la siguiente forma:

$$\Theta = \Theta_{old} + \alpha * \Theta_{trabajador_n} \quad (4.13)$$

Siendo Θ el conjunto de pesos de la red global tras la actualización del *trabajador*, Θ_{old} el conjunto de pesos anterior, α pondera la importancia de la actualización y $\Theta_{trabajador_n}$ el conjunto de pesos del *trabajador* tras su ejecución.

En cuanto a la *función-valor*, su actualización se realizará a partir de $n - pasos$ utilizando valores de ventaja vistos en el apartado 3.3.4. Es decir, se utilizará la sumatoria de recompensas obtenidas en la trayectoria parcial del episodio ejecutado, sumado con la

estimación de recompensa que se espera hasta el final de dicha trayectoria (función-valor actual). La estimación añadirá sesgo, aunque mejorará la velocidad de convergencia en el aprendizaje, como ya se sabe de haber explicado las técnicas MonteCarlo y SARSA, se intenta buscar el mejor equilibrio posible.

Los trabajadores no se esperan entre ellos, cada uno cuando termina de entrenar n -pasos hace su actualización en las redes globales. Por tanto, es muy importante mencionar que A3C tiene diseñado un **mecanismo de bloqueo** para que la concurrencia entre los *trabajadores* no provoque que sus actualizaciones se sobrescriban entre ellas en la red global.

Existen otras muchas variantes de este algoritmo: Actualización en un paso (SARSA), con Q-Learning a un paso, con Q-Learning a varios pasos, etc. No entraremos en mayor detalle en ese sentido. El esquema general del algoritmo es tal y como se describe en la figura siguiente:

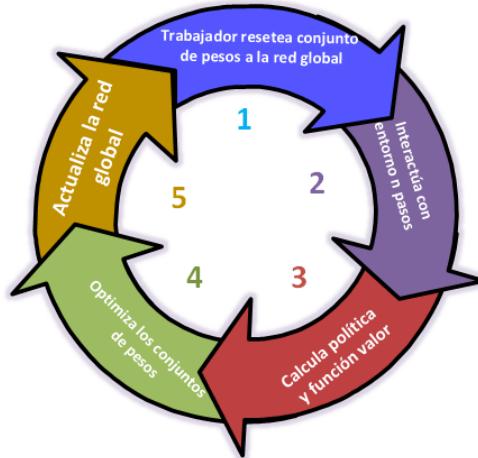


Figura 4.9: Descripción de procedimiento en algoritmo A3C.

4.3.4 Advantage Actor-Critic (A2C)

A2C [4] es, en resumen, una versión **síncrona** a partir de A3C. Según Miguel Morales [31], uno de los cambios que incluye es la utilización de una sola red neuronal y varios conjuntos de pesos para cada actor-crítico por **fusionarlo** todo en una sola red:

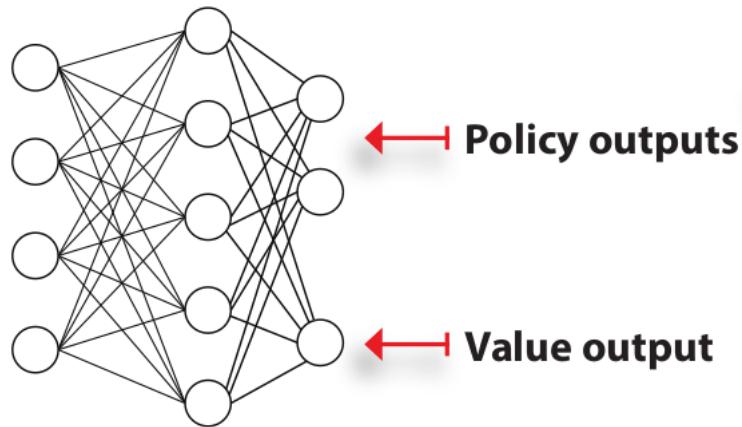


Figura 4.10: Red neuronal compartida para política (actor) y valor de función (crítico). Extraída de [31].

De esta forma, los pesos en las diferentes capas de la red es compartido y utilizado al mismo tiempo tanto por la parte actor como por el crítico, ofreciendo las dos salidas al mismo tiempo. Esto sería utilizado en la red global del mismo modo que se mencionaba en A3C y puede ser particularmente beneficioso cuando se aprende a partir de imágenes ya que la extracción de características se hace una sola vez y puede ser utilizado por el actor y el crítico al mismo tiempo (es útil para el problema de Super Mario Bros que veremos más adelante en el apartado ??).

En este algoritmo también se tiene a un conjunto de *trabajadores* los cuales se ejecutan en el mismo entorno. La diferencia reside en que se espera a que todos los *trabajadores* terminen sus ejecuciones antes de realizar una **única actualización** de la red global cada vez a partir de todos estos.

Cada trabajador calculará su conjunto de pesos ($\Theta_{trabajador_n}$) y lo enviará a la red global mencionada anteriormente. La red global esperará a que todos los *trabajadores* envíen sus correspondientes conjunto de pesos. Entonces, hará un promedio de todos ellos, se actualizará con ese promedio, se inicia los pesos de todos los trabajadores con ese conjunto de pesos y comenzará el proceso de nuevo.

se descubrió empíricamente que A2C produce mejores resultado que A3C generalmente. Según la publicación en el blog de OpenAI, los investigadores no están completamente seguros de si la asincronía beneficia el aprendizaje o la exploración.

Según ese blog también, A2C resulta más rentable que A3C cuando se usan máquinas con una sola GPU (como será nuestro caso) ya que trabaja con lotes de datos. Mientras que A3C explotará mejor la CPU a medida que tengamos mas *trabajadores* funcionando simultáneamente.

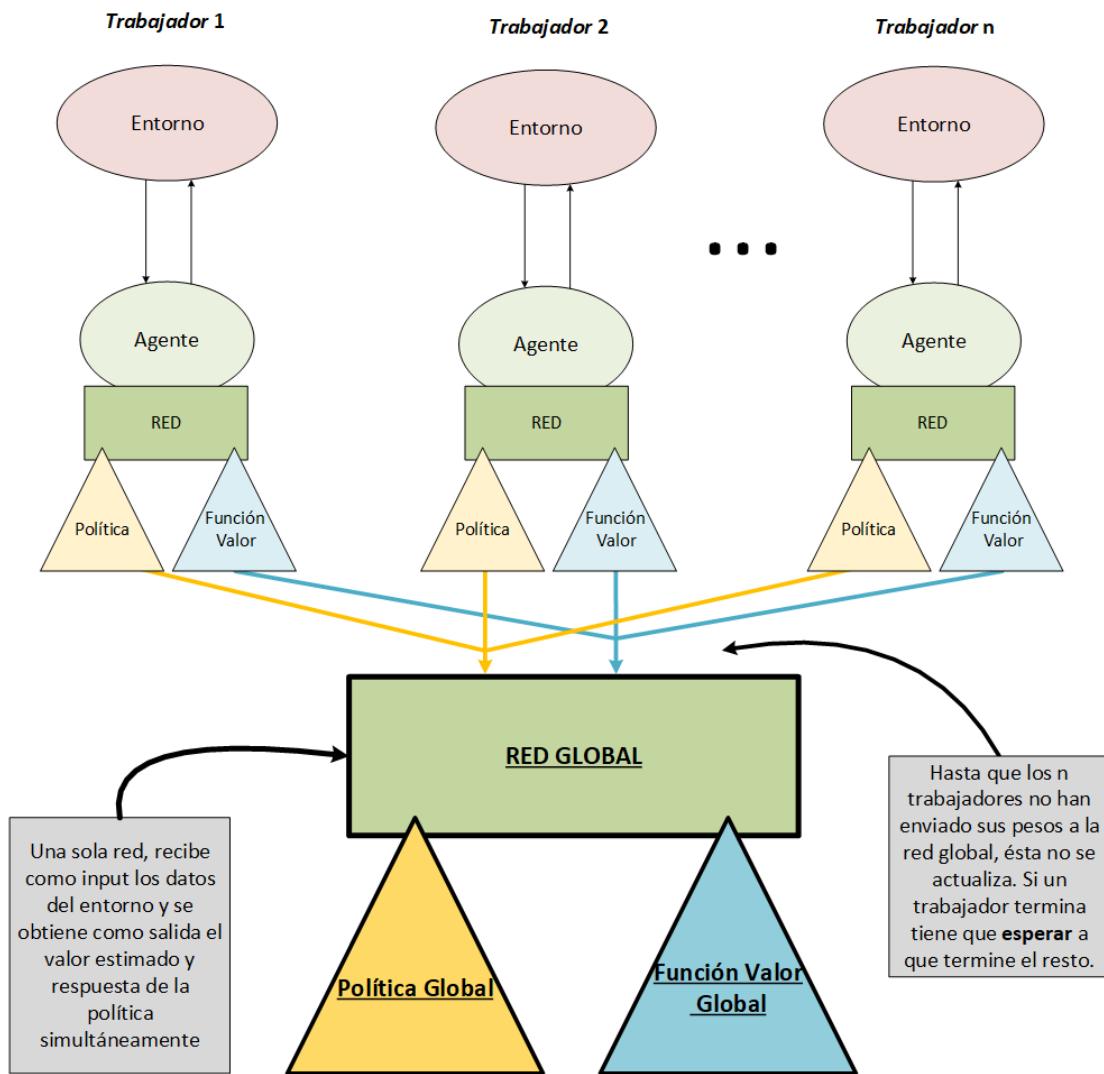


Figura 4.11: Esquema de algoritmo A2C.

Desarrollo y Experimentación

5	OpenAI baselines	73
6	Entornos Open AI Gym	75
6.1	MountainCar-v0	
6.2	MountainCarContinuous-v0	
6.3	Super Mario Bros	
7	Entorno de desarrollo	83
7.1	Metodología de experimentación	
8	Experimentación: Resultados Obtenidos	85
8.1	MountainCar-v0	
8.2	Super Mario Bros	

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. [38].

Podemos ver los avances que han obtenido en su página web [46], 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.

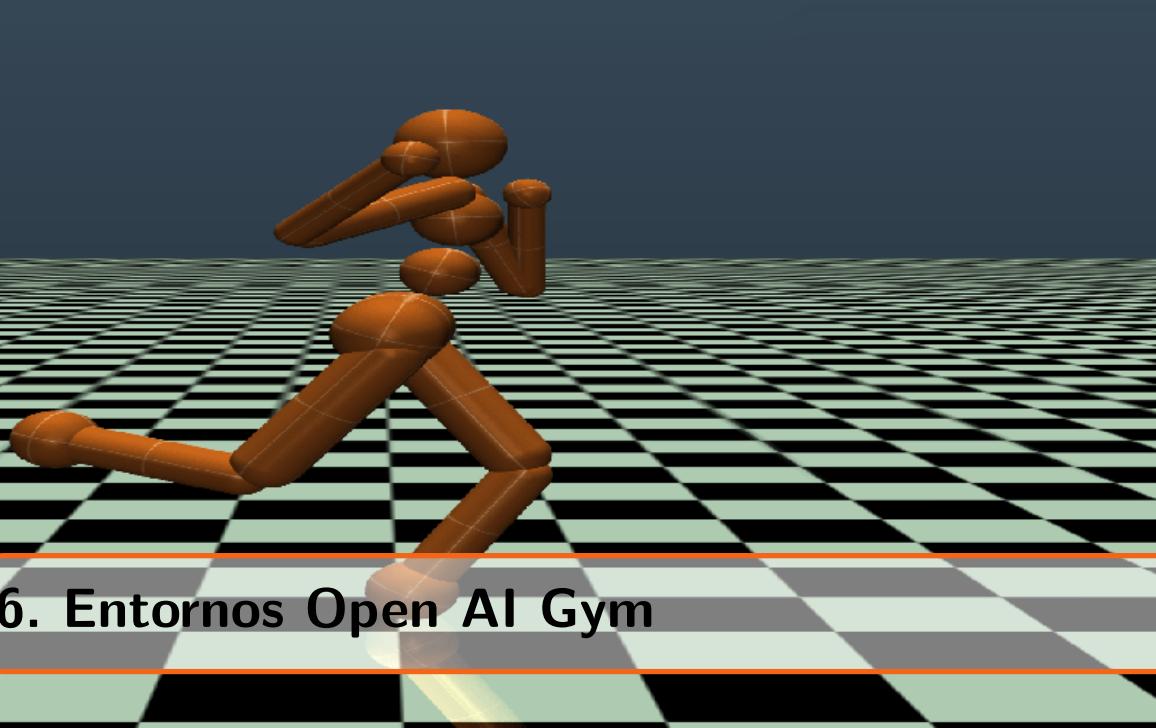
Uno de los principales motivos por los que se ha elegido a *OpenAI baselines* es su comunidad activa, la documentación y detalles de los algoritmos que implementa y la incorporación de ejemplos sencillos con la finalidad de facilitar el entendimiento de uso a los más nuevos.

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.

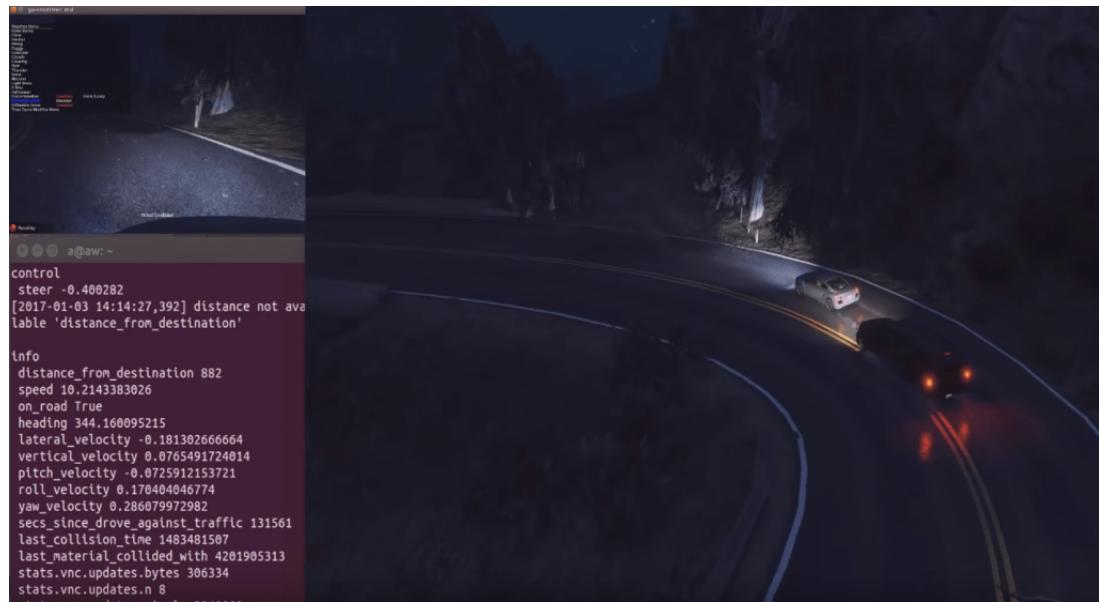


Figura 6.1: Agente entrenando conducción autónoma en entorno simulado de GTA V. Extraída de vídeo en Youtube. [51]

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. [40] [47]

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 [49]. 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 *paso* 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 *paso* 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. OpenAI Gym determina como **resultado satisfactorio** una recompensa total de -110 o mayor de media cada 100 episodios.[42]

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 *pasos* sucedidos hasta alcanzar la meta. Añadir un número máximo de *pasos* que pueden sucederse es altamente recomendable para evitar que suceda un episodio infinito en el que no logra alcanzar la meta.

Se considera como **resuelto** cuando se obtiene una media de recompensas por encima de 90 en los episodios. 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.[44].

6.3 Super Mario Bros

Es un famoso videojuego de plataformas, generalmente conocido. Fue diseñado por la compañía Nintendo en el año 1983 para la videoconsola Nintendo Entertainment System (NES). El objetivo es llegar al final de las fases que tiene prediseñadas sin “morir” antes, cada una de ellas con sus obstáculos y dificultades propias.

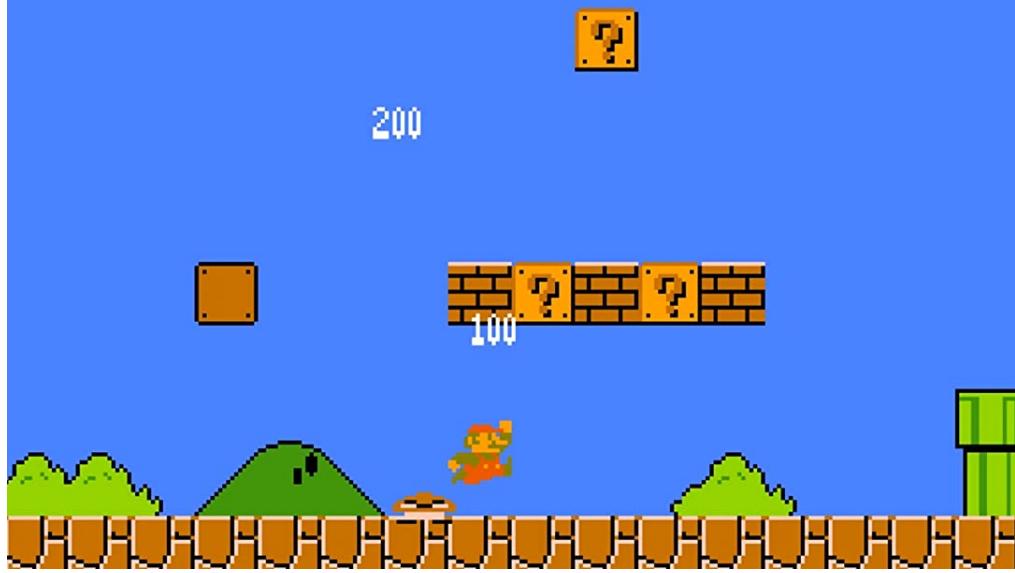


Figura 6.3: Entorno del videojuego Super Mario Bros.

Este consta de una mayor complejidad que el entorno anterior; obstáculos, enemigos de distinto tipo, agujeros que saltar, tuberías por las que meterse... Por defecto, *OpenAI baselines* no incorpora un entorno adaptado para su computación con técnicas DRL. Se ha hecho uso de la librería *gym-retro*[41] para adaptarlo a este framework y programarlo bajo las mismas librerías que usamos para el problema anterior. Esta librería lo importa a partir de la ROM original de NES, la cual se puede descargar previamente.

Se han encontrado algunas dificultades para acceder a las especificaciones del entorno. Resulta que no hay mucha documentación sobre los entornos que se importan a través de *gym-retro*, al menos no de forma directa.

Se accedió a las librerías de OpenAI, concretamente a las del archivo *run.py* y fue modificado para que se obtuviera los datos de los estados por terminal:

```
[[0.39215687 0.5568628 0.5568628 0.5568628 ]
 [0.5647059 0.5568628 0.44313726 0.5568628 ]
 [0.43529412 0.44705883 0.5686275 0.38431373]
 ...
 [[0.5137255 0.39607844 0.5568628 0.48235294]
 [0.5568628 0.5568628 0.5568628 0.5372549 ]
 [0.5568628 0.4509804 0.39607844 0.47058824]]
 [[0.36862746 0.4117647 0.30588236 0.4117647 ]
 [0.3254902 0.4117647 0.3647059 0.4117647 ]
 [0.34117648 0.32941177 0.4509804 0.3372549 ]
 ...
 [[0.47843137 0.36862746 0.4117647 0.37254903]
 [0.4117647 0.32156864 0.4117647 0.34509805]
 [0.4117647 0.40784314 0.3882353 0.4392157 ]]
 [[0.35686275 0.4117647 0.45490196 0.4117647 ]
 [0.46666667 0.4117647 0.41960785 0.4117647 ]
 [0.4 0.32941177 0.4509804 0.3254902 ]
 ...
 [[0.4862745 0.3764706 0.4117647 0.4509804 ]
 [0.4117647 0.45490196 0.4117647 0.4392157 ]
 [0.4117647 0.45490196 0.36862746 0.45490196]]]
```

Figura 6.4: Entrada del agente en Super Mario Bros.

Por lo obtenido en la figura 6.4, da la sensación de que se trata una imagen con 3 canales

de color (RGB) normalizados a valores entre 0 y 1 (en lugar a 0 y 255 que corresponde a un byte).

Si observamos el código de gym-retro [39], existe una definición de las **observaciones** o estados del entorno. Pueden ser imágenes RGB o un volcado de RAM, parece que por defecto se utiliza imágenes RGB como se estaba comentando.

Si se hace memoria del apartado 2.5, sabremos que las mejores redes neuronales que se pueden utilizar por debajo para este problema son las CNNs. El tipo de red es seleccionable. Por defecto, es CNN para este entorno, aunque puede ser cambiado a otros tipos ([48] para ver las definiciones de todas las redes de *OpenAI baselines*).

Entonces, estamos hablando de que el agente interpreta el entorno de una forma muy similar a los humanos. En lugar de tener un conjunto de variables estructuradas que define las observaciones, con información concreta, tiene una matriz con los valores RGB, siendo equivalente a observar la pantalla de la videoconsola directamente. Los humanos interpretamos esos valores de frecuencia como colores y el agente lo hace con valores numéricos, pero la filosofía es la misma.

Las **acciones** que puede hacer el agente son referentes únicamente al movimiento de Mario en el nivel:

- Nada
- A
- B
- derecha
- derecha + A
- derecha + B
- derecha + A + B
- izquierda
- izquierda + A
- izquierda + B
- izquierda + A + B
- abajo
- arriba

Siendo A la acción de saltar y B la acción de esprintar/lanzar bola de fuego cuando Mario se encuentra en una situación que lo permite.

Las acciones son discretas, aunque algunas de ellas se obtienen combinando más de una acción. Por ejemplo, la acción derecha+A desencadena un salto al mismo tiempo que se desplaza a la derecha, dos acciones que se pueden realizar por separado y que el resultado no sería el mismo.

En este videojuego, los saltos pueden ser más prolongados o menos. Esto se consigue dejando pulsado el botón de salto (A) durante más tiempo. El tiempo se encuentra discretizado (t). Entonces, en este entorno tener la acción A establecida en más de un *paso* consecutivo, equivale a un salto prolongado.

El juego original funciona a 25 FPS's, lo cual significa que cada segundo de juego tenemos 25 frames o inputs. Estos frames van a ser muy similares entre ellos, tanto en estados, recompensas, o acción a realizar. Esto retrasa mucho el entrenamiento. Para simplificar este proceso, se realiza la omisión de frames estocásticos donde cada n-simo frame se utiliza como estado y las acciones son iguales a las anteriores con una probabilidad p predefinida, para favorecer esas prolongaciones en los saltos, por ejemplo. Es una forma de tener menos datos con los que trabajar al mismo tiempo que no se compromete la eficacia del proceso de entrenamiento.

En cuanto a las **recompensas**, la función de recompensa acumulada es:

$$R = V + T + D + S \quad (6.1)$$

- **V**: Es la diferencia de la posición del agente en la coordenada x entre estados. Cuanto más se desplace hacia la derecha, mejor.
- **T**: Es la diferencia del tiempo de juego entre frames.
- **D**: Es un valor que penaliza un agente cuando muere en un estado. Esto favorece a que el agente trate de no morir bajo ningún concepto.
- **S**: Es la diferencia de puntuación del juego entre frames.

Esta función asume que el objetivo del juego es moverse a la derecha, ya que la meta en todos los modelos de nivel se encuentra en esa dirección, tan rápido como sea posible, sin morir y obteniendo todos los puntos que pueda en el proceso. Es importante que cada uno de los valores que lo componen estén debidamente estudiados y escalados, ya que el agente puede encontrar una mejor opción en detenerse obteniendo puntos y tratar de llegar a la meta cuando apenas le queda tiempo restante. Es la importancia que ya se ha mencionado en más de una ocasión de definir el problema lo mejor posible para que el proceso de aprendizaje sea adecuado.

Es interesante que en el valor de recompensa acumulada no se especifique de forma implícita que debe llegar al final del nivel. Al tratar de maximizar todo lo posible esa función de recompensa, lo conseguirá de forma indirecta.

Existen otras variantes del entorno de Super Mario Bros en Gym que pueden interpretar el problema de una forma distinta. Por ejemplo, en lugar de usar la imagen directamente, considerar un diccionario de información que define un estado del entorno de forma estructurada. En lugar de un conjunto de pixeles, estamos hablando de un conjunto de variables con la posición de Mario, enemigos, monedas, tiempo restante, vidas, etc.



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 necesarios 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, 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.

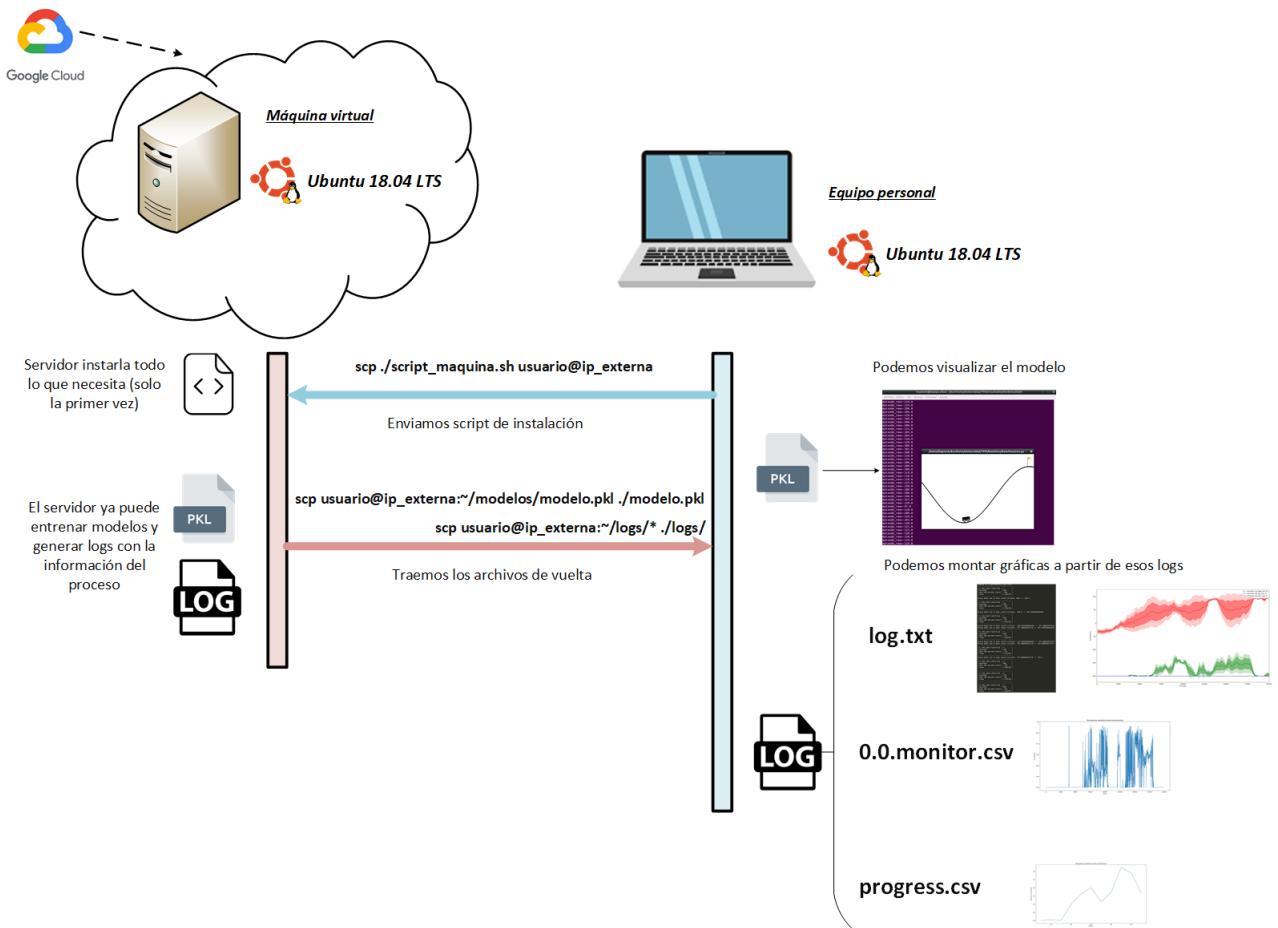
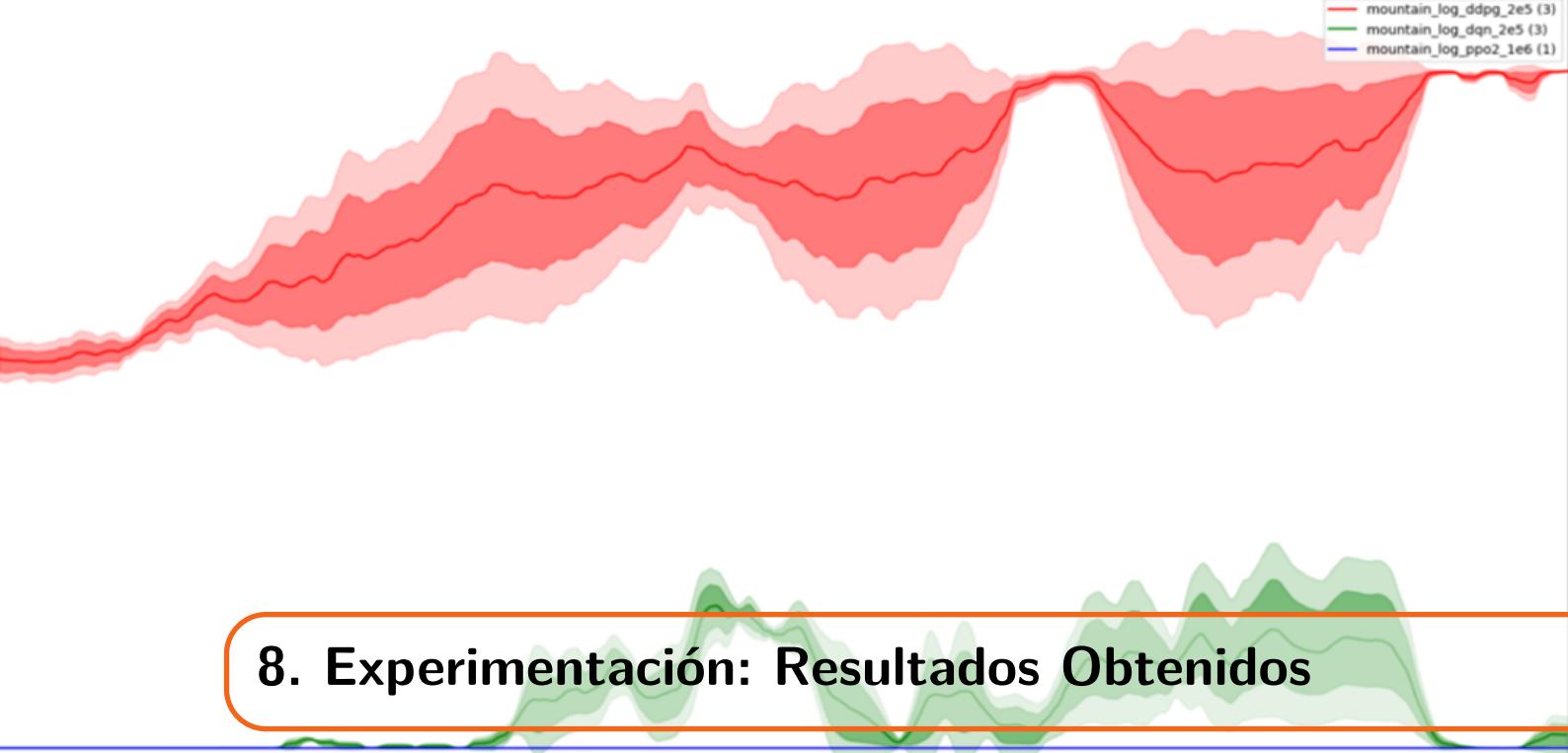


Figura 7.1: Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada



8. Experimentación: Resultados Obtenidos

En este capítulo, vamos a mostrar y discutir sobre 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

Se han obtenido una serie de modelos con las técnicas DQN, DDPG, PPO2 y A2C. Todas ellas han sido entrenadas en un total de 2,5 millones de *pasos*, el número de episodios depende de cada ejecución.

Al tratarse de un problema con información estructurada en el input, se ha decidido utilizar una arquitectura de red multicapa, es decir, parte densa vista en la sección [2](#). En baselines se define como tipo “mlp” y ya viene preconfigurada [\[48\]](#), por lo que no tenemos que preocuparnos por ello en principio.

A continuación se muestra en una tabla los resultados más relevantes obtenidos en estas pruebas.

	Media recompensa (cada 100 ep)	Semilla	Episodios Transcurridos	Tiempo (segundos)
DQN	-149.40	3	12600	6247
	-113.10	13	12800	6109
	-100.40	113	20600	5926
DDPG	94.30	3	23400	6569
	94.27	13	25700	6577
	94.03	113	25800	5248
PPO2	-200	113	12500	1315
A2C	-200	3	12500	5965

Recordemos que el algoritmo DDPG funciona en *MountainCarContinuous-v0* y que la función de recompensa y por tanto su rango de valores es diferente a DQN, PPO2 y A2C. En cuanto a PPO2 y A2C, solo se ha mostrado un resultado, porque en todos los experimentos ha ocurrido exactamente lo mismo; una recompensa de -200 (no alcanzar la meta nunca).

Los modelos más relevantes han sido grabados y subidos a *Youtube* (<https://www.youtube.com/channel/UChB02d0pWUenxhGThUB5IFQ>). A continuación, se muestran los enlaces de los vídeos para cada modelo:

Algoritmo	Pasos entrenamiento	URL Vídeo
DDPG	2,5 millones	https://www.youtube.com/watch?v=V8DdsyK0jKM
DQN (mejor caso)	2,5 millones	https://www.youtube.com/watch?v=1HL-060WiKY
DQN (caso ordinario)	2,5 millones	https://www.youtube.com/watch?v=73R_XU1f6Yg
PPO2	2,5 millones	https://www.youtube.com/watch?v=azSbhDxEno
A2C	2,5 millones	https://www.youtube.com/watch?v=ve4az81c_ek

8.1.1 DQN

En DQN, la mayoría de semillas alcanzan una buena media de recompensas muy pronto y después comienzan a empeorar (semilla 3 y 13). No obstante, en la semilla 113, ocurre algo poco común; converge pronto pero se mantiene, incluso mejora poco a poco los resultados de recompensa hasta alcanzar la mejor recompensa obtenida en entrenamientos para esta técnica (-100).

El vídeo del caso ordinario corresponde a la mayoría de semillas (en este caso semilla 3), vemos que alcanza el objetivo, pero no siempre. Por eso es importante alcanzar valores de recompensa medios de -110, para asegurarnos de conseguir un modelo que siempre acierte, independientemente del lugar en el que aparezca la carreta y de la ejecución en sí.

Para la semilla 113, el caso especial, observamos que siempre consigue resolver el problema, justo como estaba comentándose.

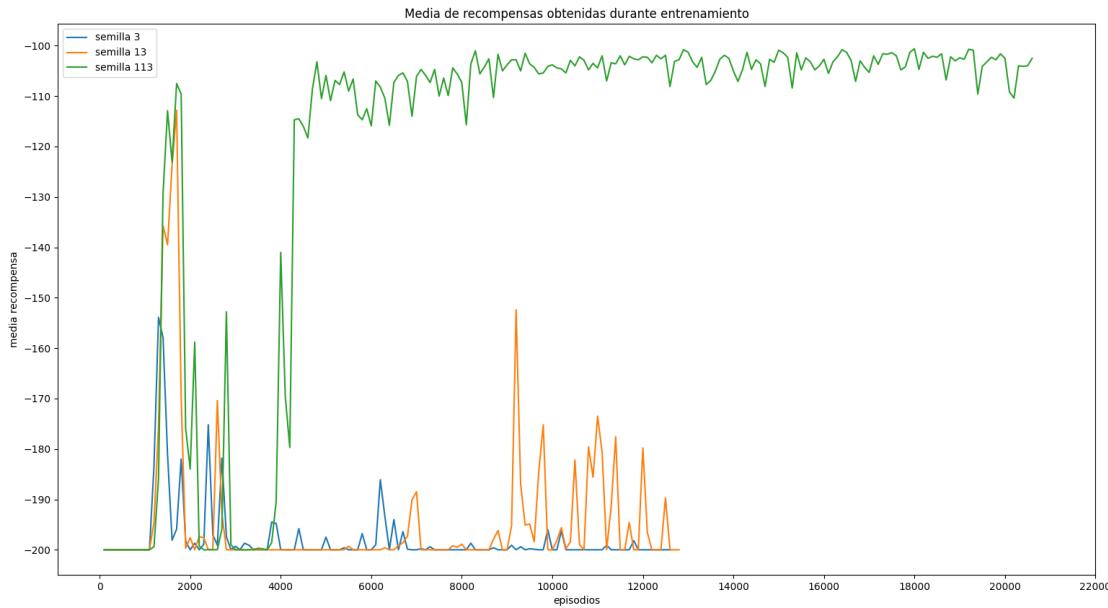


Figura 8.1: Media recompensas acumuladas cada 100 episodios, todas las semillas.

Por como está definido el problema, el número de episodios y la recompensa acumulada están relacionados. Cuanta mejor sea la recompensa de media, menos *pasos* por episodio. Al tener una duración de entrenamiento fija, esto se traduce en un mayor número de episodios.

La mayoría de modelos no alcanzan valores malos, siendo lo común encontrarnos recompensas comprendidas entre -140 y -110 (se han realizado entrenamientos con otras semillas diferentes de las mostradas), recordemos que una recompensa de -110 ya se considera un modelo totalmente eficaz. De forma extraordinaria, podemos dar con entrenamientos como en la semilla 113. No solo obteniendo una recompensa acumulada máxima más alta que el resto, sino capaz de mantenerse estable.

¿Por qué ocurre esto? A pesar de la apariencia, MountainCar es un problema complejo. La función de recompensa solo ofrece un valor positivo cuando alcanza la meta, siendo valores negativos en el resto de casos, esto es conocido como **sparse reward** (recompensa escasa o dispersa) y es un problema bastante frecuente en los problemas de aprendizaje por refuerzo. [19]

Esto hace que sea muy complicado para el algoritmo lograr conectar series de acciones que sean beneficiosas para una recompensa acumulada futura. Si esa escasez de recompensas positivas es intratable por el algoritmo, resultaría en un agente que nunca puede alcanzar el objetivo y aprender a representar una concatenación de acciones buena para el problema. Una solución posible a este inconveniente es entrenar con RL redes neuronales ya eficaces en cierta medida, en lugar de partir de cero. De hecho, es lo que hacen con AlphaGo [25], por ejemplo, entrenando a partir de modelos ya funcionales mediante aprendizaje supervisado de partidas de humanos.

Otra solución podría ser elaborar otra función de recompensa que no tuviera este problema, lo que se traduce en un **cambio de definición** de ese problema. Proponer una

función diferente puede influir y mucho en ese proceso de aprendizaje y cómo el agente consigue optimizarse.

Tener unas recompensas tan dispersas puede hacer que la exploración del agente se complique y no pueda llegar a converger. Por esa misma razón la aleatoriedad (semillas) influye tanto en los entrenamientos con esta técnica y la exploración, sobretodo al principio, es fundamental para tratar de encontrar esas recompensas que se necesitan.

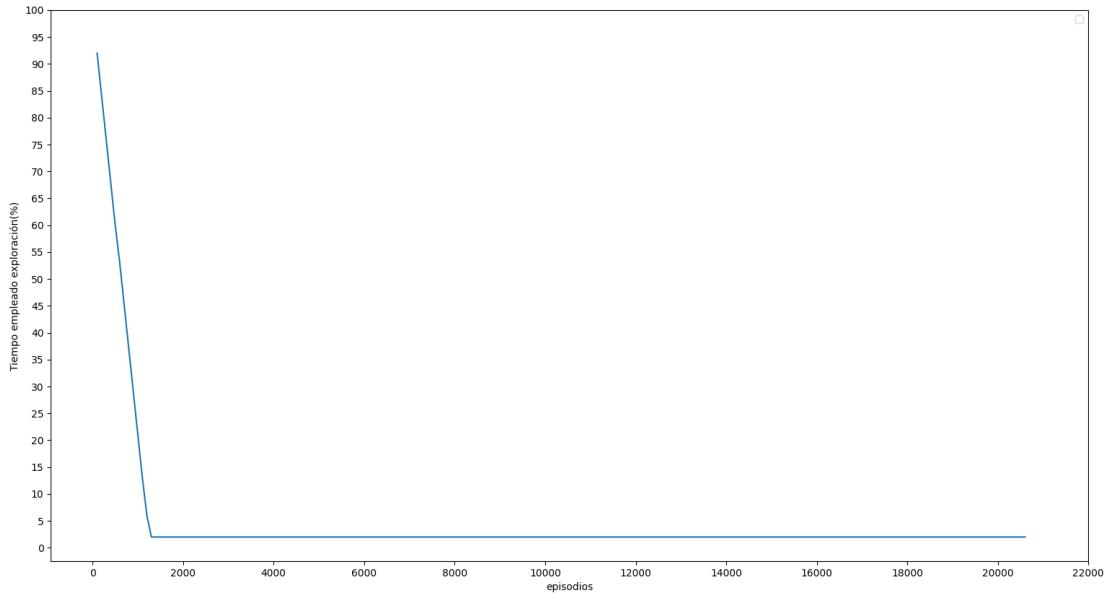


Figura 8.2: Tiempo empleado en explorar por *MountainCar* con semilla 113

La mayor parte de la exploración por parte del agente se produce en los primeros 1100 episodios del entrenamiento. Empezando con un 92% del tiempo empleado en explorar y decrece linealmente a un ritmo alto. A los 1300 episodios ya solo emplea un 2% del tiempo en explorar, hasta el final del entrenamiento se mantiene con esa dinámica. El resto de agentes entrenados siguen este patrón de exploración con DQN.

Esta es la principal razón por la que DQN es capaz de lidiar con *MountainCar-v0* a pesar del *sparse reward*. Es un problema en la que la exploración aleatoria en grandes cantidades se premia, y mucho, ya que alcanzar el objetivo es imprescindible para el aprendizaje. [19]

Como conclusión final, los agentes suelen alcanzar su mejor media de recompensas antes de los 2000 episodios con unos valores de entre -110 y -140, por lo que se puede concluir que esta técnica es capaz de resolver el problema de *MountainCar-v0* tal y como define el problema Gym, aunque no ofrece las mejores soluciones posibles ni logra su objetivo el cien por cien de las ocasiones (excepto en casos extraordinarios como el de la semilla 113).

Quizás con una reformulación de la función de recompensa los entrenamientos podrían ser más estables y predecibles, dependiendo menos de la exploración.

El entrenamiento de la semilla 113 ha sido un caso extraordinario, alcanzando su mejores validaciones y convergencia a partir de los 5000 episodios con una recompensa de -100 y siendo capaz de mantenerse a ese nivel. Sin embargo, no es lo habitual. Si mostramos un resumen de todos los entrenamientos en una misma gráfica podemos observar la influencia de la aleatoriedad en la exploración:

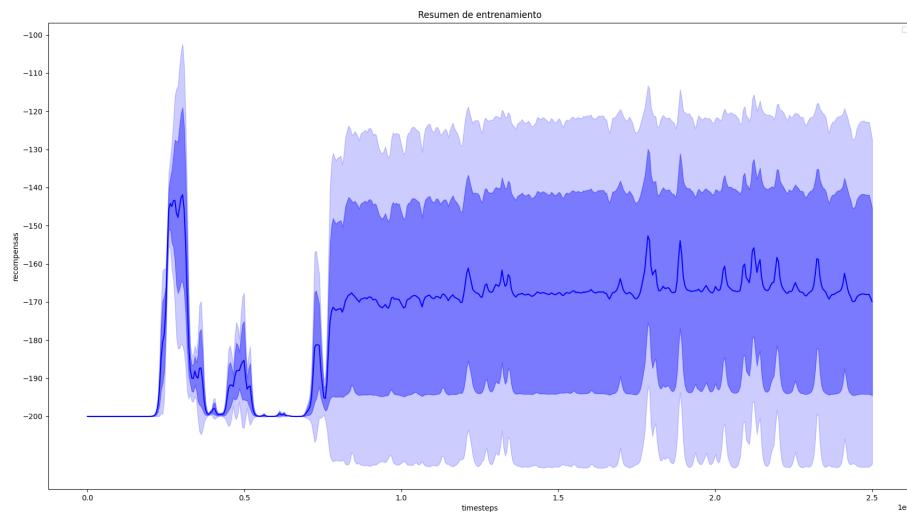


Figura 8.3: Resumen general de progreso de algoritmo DQN en *MountainCar-v0*.

Esta gráfica ha sido realizada a partir de los *pasos* dado que, como se mencionó anteriormente, si es el mismo valor en todos los modelos, cosa que no ocurre con los episodios.

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 mejor visible.

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

Vemos la enorme variabilidad que pueden tener DQN para *MountainCar-v0*, sobretodo pasados los 2000 episodios más o menos. Aún así puede lidiar con el problema y llegar a resolverlo. Además, utilizar una política de uso diferente de la función Q para generar experiencia y actualizarla puede influir positivamente en este sentido (figura 3.5).

8.1.2 DDPG

En DDPG, se han conseguido resultados mucho más **estables** que en DQN alcanzando siempre unos resultados muy semejantes independientemente de la semilla que se usa. Por lo que, a primera vista, parece ser que no es tan dependiente de la aleatoriedad durante el aprendizaje como en el caso de DQN para la versión discreta del problema.

Si comparamos el vídeo de DDPG con el vídeo del mejor modelo DQN, vemos que ambos consiguen siempre resolver el problema. No obstante, si nos fijamos en el comportamiento de ambos, vemos que DDPG tiene una forma de resolverlo más limpia que DQN.

DDPG ha aprendido mejor el concepto de momentum para resolver el problema. Si nos fijamos en DQN, hay veces que comienza el episodio moviéndose hacia la derecha, un primer movimiento totalmente innecesario, mientras que DDPG siempre hace el mismo movimiento y de forma prácticamente perfecta.

Los tiempos de ejecución para este problema son muy similares a los de DQN, quizás un poco más altos en general, lo cual tiene sentido dado que DDPG integra una parte crítica basada en DQN con adaptación a acciones continuas. Las recompensas obtenidas durante los entrenamientos han sido:

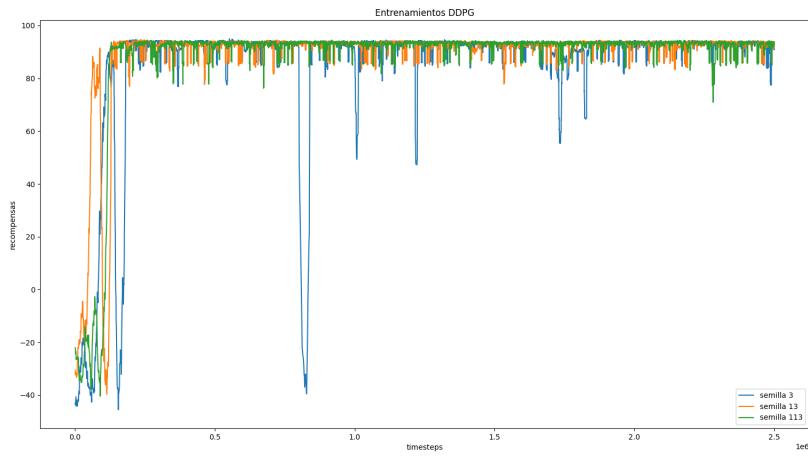


Figura 8.4: Agentes entrenados con DDPG en *MountainCar-v0*.

Como se puede apreciar en la figura 8.4, los modelos convergen bastante pronto, sobre los 200000 *pasos* de entrenamiento aproximadamente. Se mantienen muy estables, no mejoran más, pero tampoco empeoran. En la semilla 3 se produjeron algunas caídas repentinas, volviendo a sus valores habituales de recompensa rápidamente, por lo que no parece preocupante.

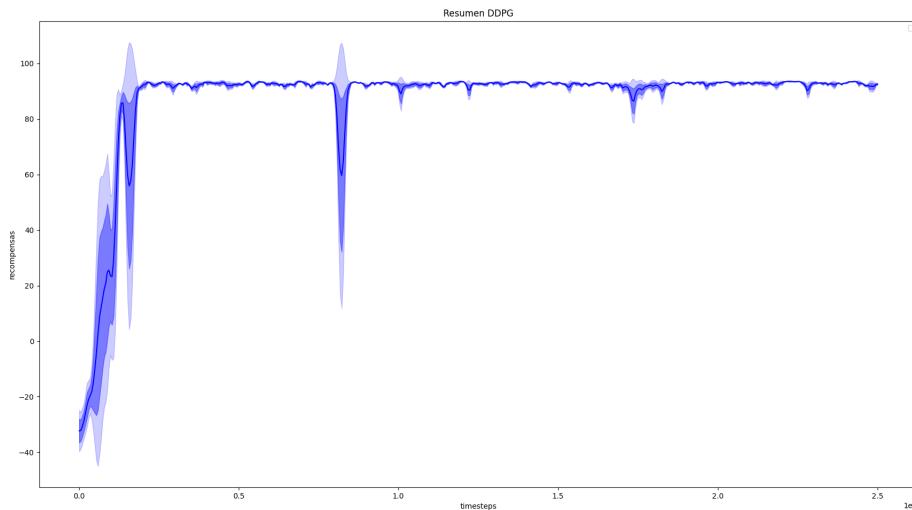


Figura 8.5: Agentes entrenados con DDPG en *MountainCar-v0*.

Siguen un patrón en el progreso de aprendizaje bastante claro independientemente de la semilla que se utilice. Como se puede observar en la figura 8.5, lidia mucho mejor con el *sparse reward* que DQN, ya que no depende tanto de la aleatoriedad realizada en cada una de las semillas, aunque la exploración sea muy importante.

DDPG tiene una buena estrategia de exploración (sección 4.3.2), introduciendo el ruido directamente en los pesos de la red neuronal que se está tratando de optimizar en lugar de las acciones (ϵ -greedy, por ejemplo), quizás esto favorezca esa menor dependencia de la aleatoriedad en sus entrenamientos. Esto reafirma que la **exploración** para problemas con escasez de recompensas es imprescindible para obtener buenos resultados.

Por último, mencionar que el problema de *MountainCar-v0* no necesita tantos pasos de entrenamiento para conseguir buenas soluciones siempre que utilicemos el algoritmo adecuado para ello. En DDPG con unos 300.000 pasos habría sido suficiente.

8.1.3 PPO2 y A2C

Lamentablemente, estos algoritmos no han conseguido ningún tipo de resultados positivos [19]. Simple y llanamente, no han logrado alcanzar la meta ni una sola vez durante sus entrenamientos, realizando episodios de duración máxima, 200 pasos.

¿Esto quiere decir que son malos algoritmos? En realidad no, es solo que no funciona para la definición de este problema. El *sparse reward* hace que en este caso sea inviable encontrar una solución del problema.

A2C presenta una **exploración pobre** en comparación a otras técnicas, utilizando su política óptima para generar experiencia en cada turno con la que aplicar gradientes y optimizarse, por lo que es muy probable que nunca alcance a resolver *MountainCar-v0*.

PPO2 , al utilizar épocas múltiples en *mini-lotes* debe garantizar que no se pierde las escasas recompensas en la actualización del gradiente. El problema es que si a partir de su política inicial (muy mala al principio) no consigue dichas recompensas, no podemos aprovecharlo.

Un dato bastante interesante es el tiempo de ejecución de PPO2, es realmente rápido, aunque no obtenga los resultados esperados. Como ya se mencionó en la sección ??, el algoritmo es mucho más sencillo que otros de su familia como TRPO.

Aunque no se hayan obtenido los resultados esperados, los logs contienen información que es interesante de cara a analizar el funcionamiento del algoritmo:

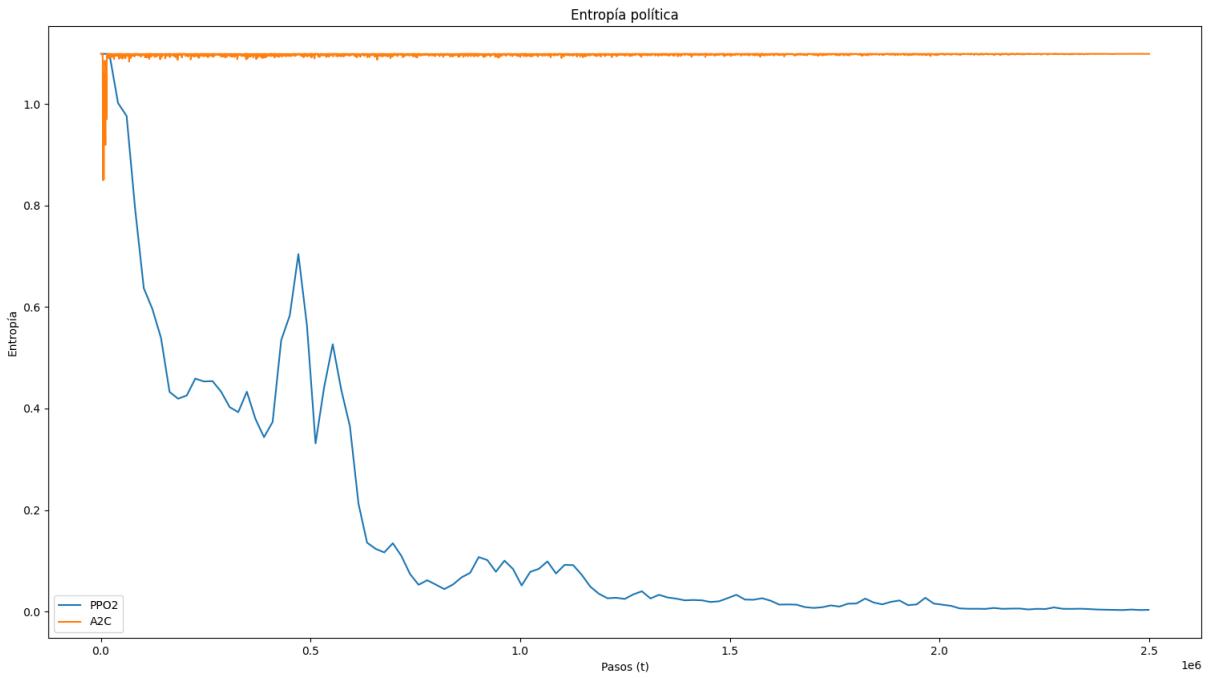


Figura 8.6: Entropía de política a lo largo de entrenamiento con técnicas A2C y PPO2 en *MountainCar-v0*.

La entropía es una medida que nos sirve para definir el grado de aleatoriedad de una política. Como ya sabemos, una política devuelve directamente una distribución de probabilidades de las acciones discretas o un valor real con cierta desviación en caso de acciones continuas. Entonces, dependiendo de esta entropía podemos deducir la exploración de estos agentes. En PPO2 comienza muy alta al principio y va decreciendo al final, mientras que A2C se mantiene alto durante todo el entrenamiento, quizás por el promedio de pesos que hace antes de actualizar la red global.

Si comparamos los vídeos mostrados de ambos algoritmos, a pesar de que ninguno obtiene resultados positivos, podemos notar que el comportamiento de ambos es diferente. A2C, al tener mayor entropía, vemos que muestra comportamientos diferentes dependiendo de cada ejecución. PPO2 tiende a realizar siempre el mismo comportamiento.

En cuanto al número de actualizaciones de política, en ambos casos tiene un aumento lineal de estas actualizaciones a medida que entrena, aunque PPO2 lo hace a un ritmo mucho más lento que A2C.

8.1.4 Comparativa de modelos

Si exponemos todos los resultados de modelos en una sola gráfica obtenemos lo siguiente:

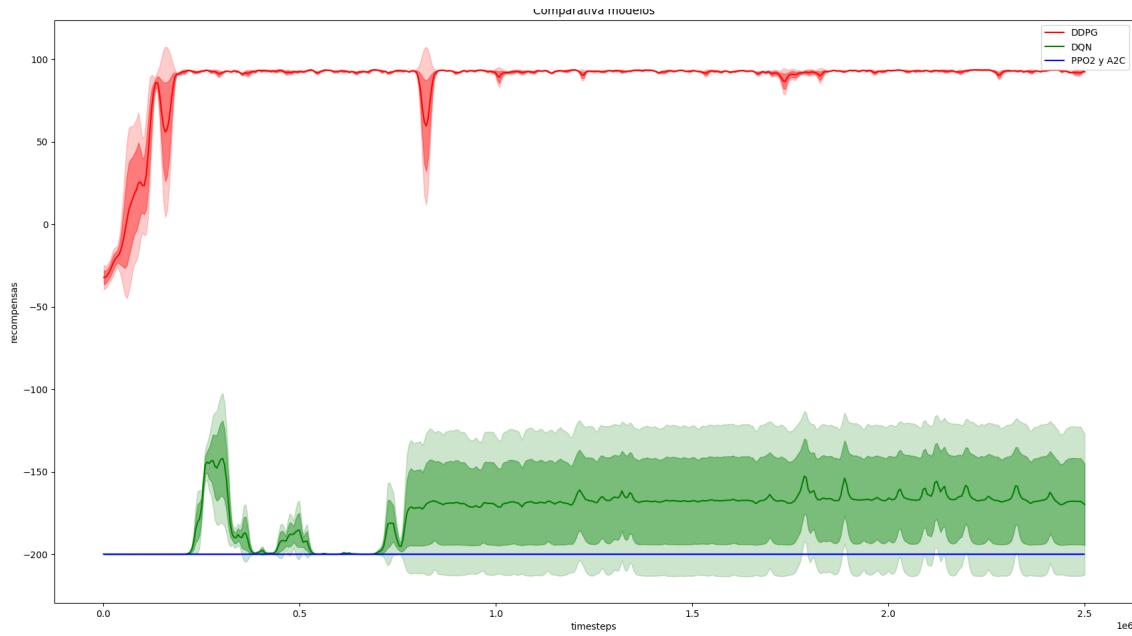


Figura 8.7: Comparativa de técnicas en *MountainCar*.

Hay que recordar que la escala de recompensas de *MountainCarContinuous-v0* (DDPG) es diferente a la de *MountainCar-v0* (DQN, PPO2 y A2C). En el primer caso se entiende por un buen modelo aquel capaz de alcanzar medias de recompensas superiores a 90, en el segundo casos superiores a -110.

Tanto PPO2 como A2C quedan descartados para la definición del problema actual dado que no consigue ningún resultado. Aunque DQN ha demostrado poder dar buenos resultados, es demasiado dependiente de la aleatoriedad en el aprendizaje, en gran parte por el *sparse reward*. Entonces, DDPG parece ser la mejor opción para las definiciones de *MountainCar* actuales, por su estabilidad y superar prácticamente siempre el umbral de recompensa a partir del cual se considera un buen modelo.

8.2 Super Mario Bros

Para los entrenamientos de Super Mario Bros, se han utilizado redes CNNs debido al tratamiento directo de frames del videojuego como input. Para ello usamos la red llamada “cnn” de los propios *baselines*. [48]

En cuanto a las técnicas utilizadas, se van a usar los algoritmos DQN, PPO2 y A2C. Se descarta DDPG por problemas al intentar utilizarlo en el entorno, ya que este funciona con *stable-baselines*.

Este es un problema mucho más complejo, como ya se ha mencionado en otras ocasiones. Por ello, es muy razonable esperar una mayor necesidad de interacciones con el entorno para que el aprendizaje en los algoritmos pueda converger en una buena solución del problema.

Primero se hicieron unas pruebas con unos dos millones y medio de *pasos*, esto no da ni para empezar con este problema. Sin embargo, sirvió para hacerse una idea de cuanto

iban a tardar los algoritmos y comprobar que todos los datos y modelo se guardaban correctamente tras finalizar la etapa de aprendizaje.

Tras estas primeras comprobaciones, se realizaron pruebas para cada uno de los algoritmos con **10 millones de interacciones** en cada uno. El resumen de los resultados obtenidos es el siguiente:

	Media recompensa	Semilla	Tiempo (segundos)	Nivel (mundo-fase)	URL
DQN	495.5	52	76260	1-1	No disponible
PPO2	2180	77	25820	1-1	https://www.youtube.com/watch?v=ytSx0ZolzvI
	1681	113	59693	3-1	https://www.youtube.com/watch?v=qtPmSKssJ1I
A2C	1941.55	84	23100	1-1	https://www.youtube.com/watch?v=VkrKye5PxjM&t=5s

Lamentablemente, no ha sido posible poder grabar el vídeo de ejecución del algoritmo DQN para *Super Mario Bros*.¹ Aunque, por suerte, es el algoritmo de menor interés dado que ofrece resultados muy pobres.

Si se observan los vídeos mostrados en la tabla anterior, vemos que en el caso de PPO2 para el nivel 1-1 consigue resolverlo una vez en la esquina superior izquierda. Para A2C, consigue resolver el nivel 2 veces, en la esquina superior izquierda y la esquina inferior derecha.

Lo más interesante, es que una vez superan el nivel, el entorno cambia al nivel siguiente (2-1), con el cual el agente no ha sido entrenado. Pese a fracasar, nos podemos dar cuenta de que se maneja bastante bien en ese nuevo nivel. Esto es señal de que ha conseguido entender ciertos conceptos, como los enemigos, obstáculos y demás. Está logrando en cierta medida **generalizar** en lugar de aprenderse una secuencia concreta de acciones para el nivel concreto con el que entrena, lo cual es muy deseable.

Para el caso de PPO2 en el nivel 3-1, lo más interesante es el tiempo de ejecución, el cual se dispara a más del doble pese a seguir siendo esos 10 millones de pasos. No obstante, centraremos el análisis en el nivel 1-1 a partir de ahora.

En este caso, no nos encontramos ante un problema de *sparse reward* como nos ocurría con *MountainCar-v0*, el entorno ofrece fuentes de recompensa a explotar desde el comienzo del episodio.

¹Hay un problema con gym retro y DQN a la hora de cargar los modelos entrenados que no se ha podido solucionar, existe un problema similar con atari reportado (<https://github.com/openai/baselines/issues/12>)

8.2.1 DQN

Como primera conclusión, DQN es un algoritmo lento relativamente a otras técnicas, tardando algo más de 9 horas en realizar esas 10 millones de interacciones. Además, una cosa muy interesante que se pudo observar es que aprovecha peor los recursos de la máquina virtual que se tiene configurada.

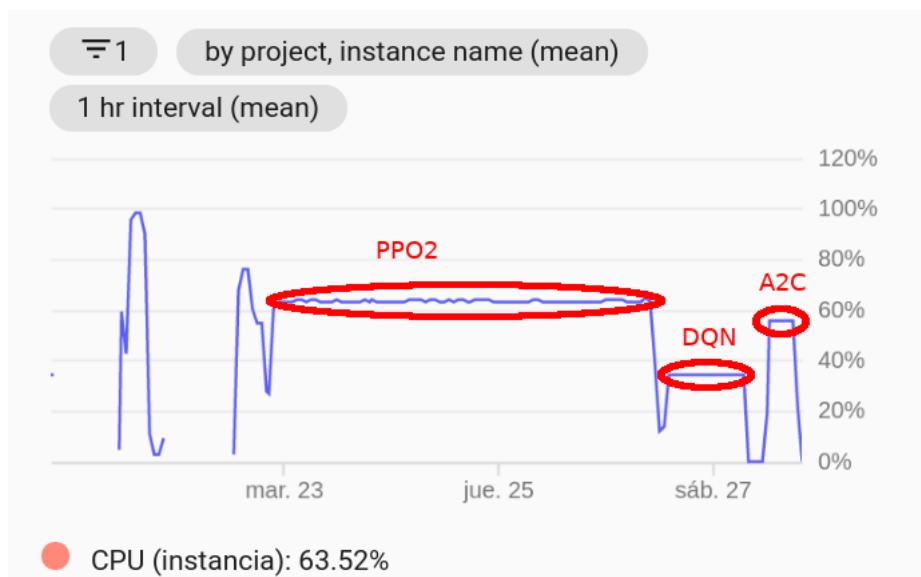


Figura 8.8: Porcentaje de uso CPU en máquina virtual en pruebas con Super Mario Bros (Google Cloud Monitoring).

Recordemos que A2C utiliza varios *trabajadores* y que PPO2 puede organizar mini-lotes por iteración independientes y ejecutarlos en paralelo, por lo que tiene sentido que puedan aprovechar los recursos disponibles de una forma más eficiente.

En cuanto a la GPU ocurre algo similar. Google Cloud no tiene por defecto una monitorización de este uso pero se ha revisado desde el comando `nvidia-smi` ejecutado en la máquina directamente. DQN no pasa del uso del 30% de la tarjeta gráfica, mientras que PPO2 y A2C suelen tener picos muy frecuentes de un 60% o 70%.

Además, para este entorno de Super Mario, no consigue una buena recompensa si lo comparamos a PPO2 y A2C, obteniendo apenas la cuarta parte de puntos que las otras dos técnicas para ese número de interacciones.

A continuación, se muestra el progreso del algoritmo durante su entrenamiento:

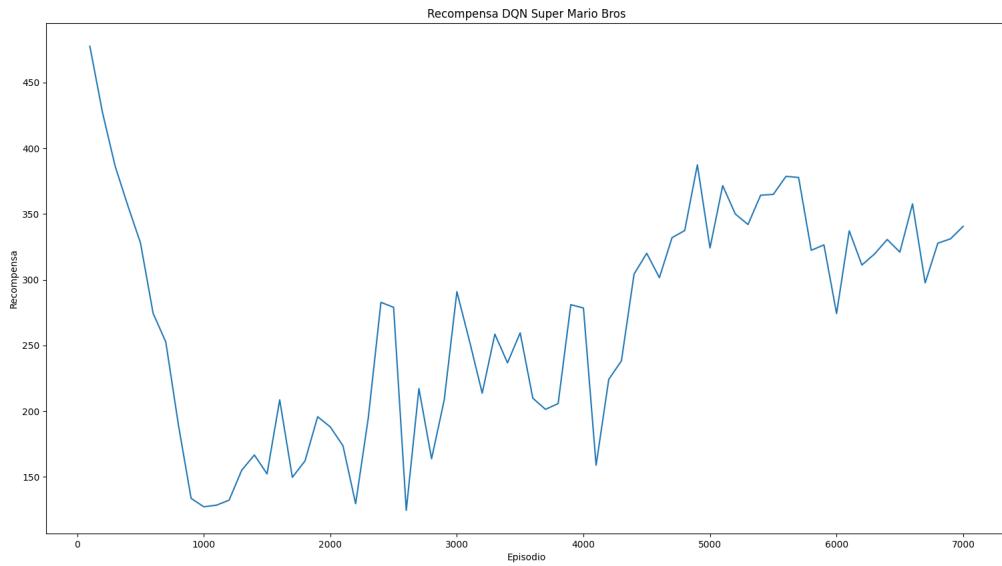


Figura 8.9: Recompensas obtenidas durante el entrenamiento DQN con 10 millones de pasos en *SuperMarioBros-Nes*.

Comienza a progresar poco a poco. El problema principalmente, como se estaba comentando, es el tiempo para empezar a tener estos resultados, es demasiado lento. En cuanto a la exploración:

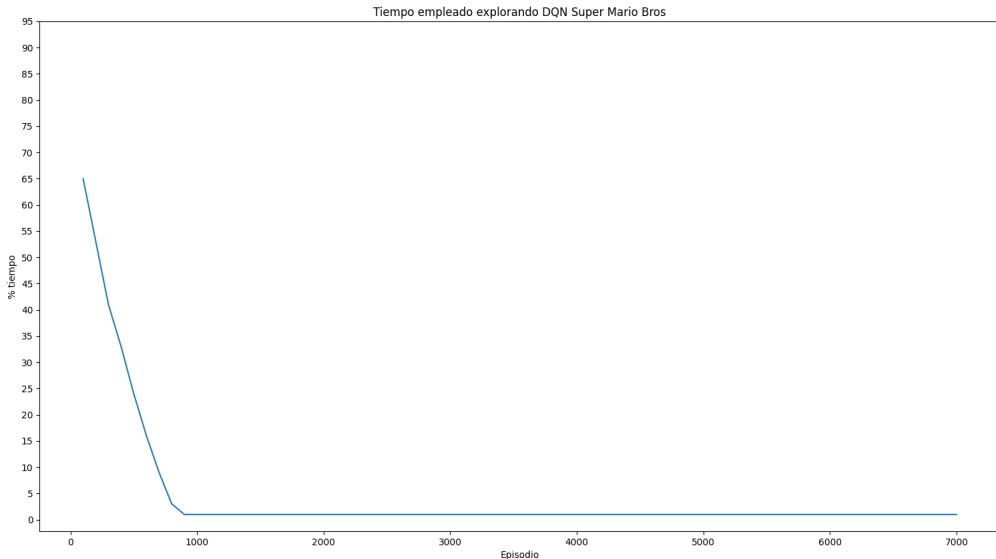


Figura 8.10: Porcentaje de tiempo explorado durante el entrenamiento DQN con 10 millones de pasos en *SuperMarioBros-Nes*.

Mantiene una exploración parecida a *MountainCar-v0* en la figura 8.2, muy alta al inicio y luego se estabiliza rápidamente al 2% durante el resto del entrenamiento, aunque es cierto que el porcentaje de tiempo nunca es superior al 65%, mientras que en *MountainCar-v0* era superior al 90% en el comienzo.

Hay que tener en consideración la existencia de una **limitación** de tiempo y recursos, no

es posible entrenar y probar todas las ideas. Por todas las razones expuestas anteriormente, se ha decidido no seguir realizando experimentos con esta técnica para este entorno. Nos centraremos más a partir de ahora en PPO2 y A2C que han conseguido mejores resultados en menos tiempo.

8.2.2 A2C y PPO2

Tanto A2C como PPO2 han mostrado mejores resultados que DQN y más rápido, alcanzando unas recompensas medias más altas (ver tabla anterior) y con unos tiempos de unas 6 horas y media para A2C y algo más de 7 horas para PPO2 (en el nivel 1-1).

Se muestran los resultados de ambas técnicas juntas por la similitud de los resultados obtenidos, los cuales incitan a una comparación directa entre ambos.

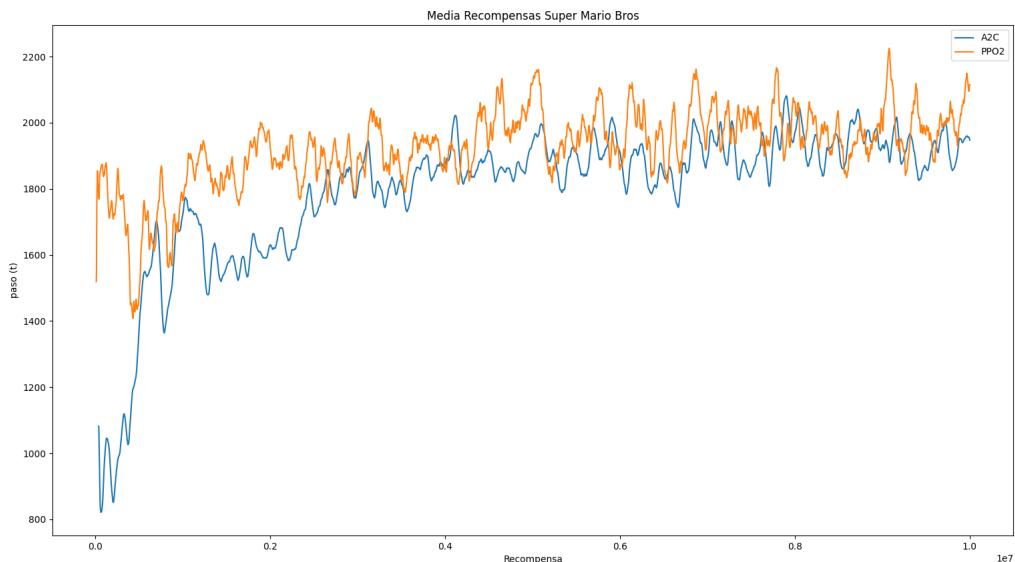


Figura 8.11: Media de recompensas durante el entrenamiento A2C y PPO2 con 10 millones de pasos en *SuperMarioBros-Nes*.

A2C empieza con recompensas más bajas que PPO2 aunque aumenta rápidamente hasta alcanzar valores entre 1800 y 2000 a partir de las 3 millones de interacciones aproximadamente.

PPO2 empieza con unas recompensas iniciales algo más elevadas, por lo que la diferencia no es tan pronunciada en esas primeras 3 millones de interacciones. No obstante, parece alcanzar resultados ligeramente mejores a A2C en general. Ambos mantienen un ritmo de mejora muy lento, por lo que cuesta llegar a saber qué algoritmo sería más eficiente si el tiempo de entrenamiento se alargara más. En otras palabras, no se detecta una convergencia clara de los algoritmos con solo 10 millones de pasos.

A continuación, se muestra la media de pasos por episodio durante el entrenamiento:

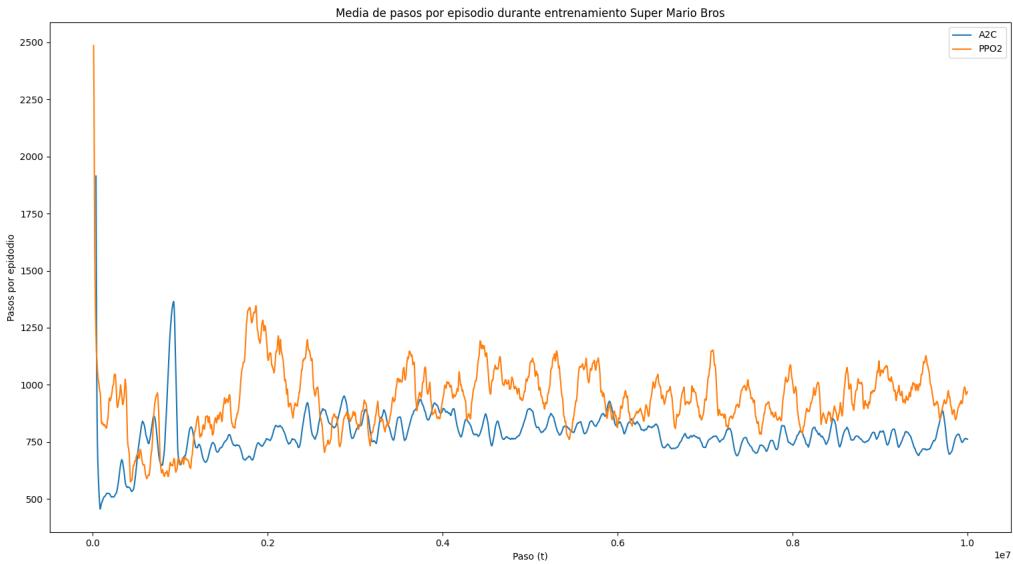


Figura 8.12: Media de pasos por episodio durante el entrenamiento A2C y PPO2 con 10 millones de pasos en *SuperMarioBros-Nes*.

PPO2 realiza unos episodios más prolongados que A2C durante prácticamente todo el entrenamiento. Por tanto, es inevitable pensar en la existencia de una relación entre la recompensa obtenida y la duración de los episodios.

Al comienzo del entrenamiento, ambos algoritmos presentan episodios muy largos, esto llama mucho la atención. El origen de esto, es que las políticas se encuentran tan iniciales, que aún no conocen la dinámica del juego. Esto provoca que no avancen y cueste mucho comenzar a dar resultados, generando una secuencia de acciones que no le permite al agente progresar. Probablemente al principio finalizaba los episodios cuando se agotaba el tiempo o, tras tanto tiempo realizando acciones prácticamente aleatorias, se topaba con un enemigo y moría.

Lo más interesante es que las interacciones por episodio tienden a acotarse a medida que la recompensa obtenida es mejor. a continuación se muestran dos figuras, la primera es solo A2C y la segunda A2C y PPO2, para que se pueda apreciar mejor:

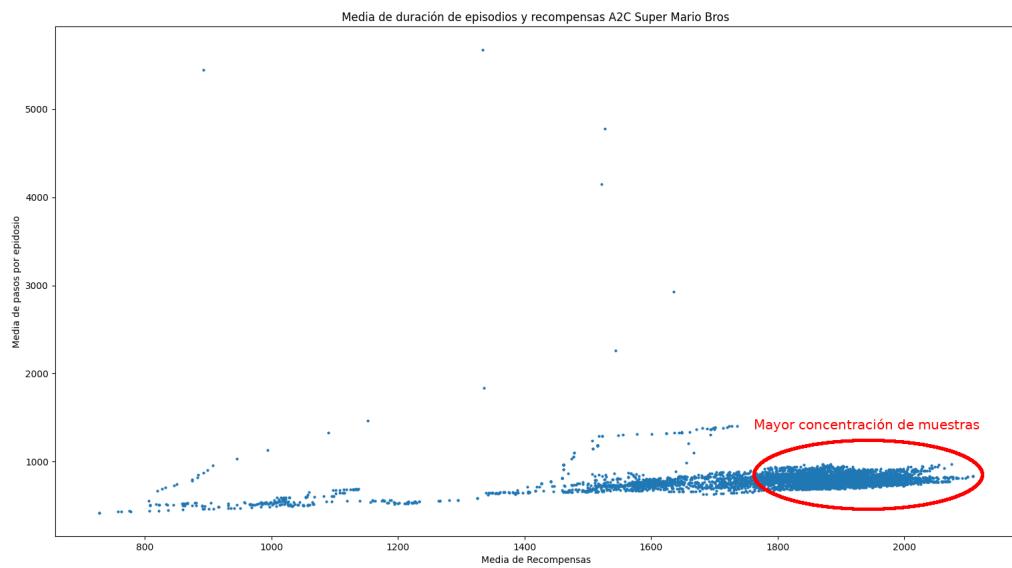


Figura 8.13: Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C en *SuperMarioBros-Nes*.

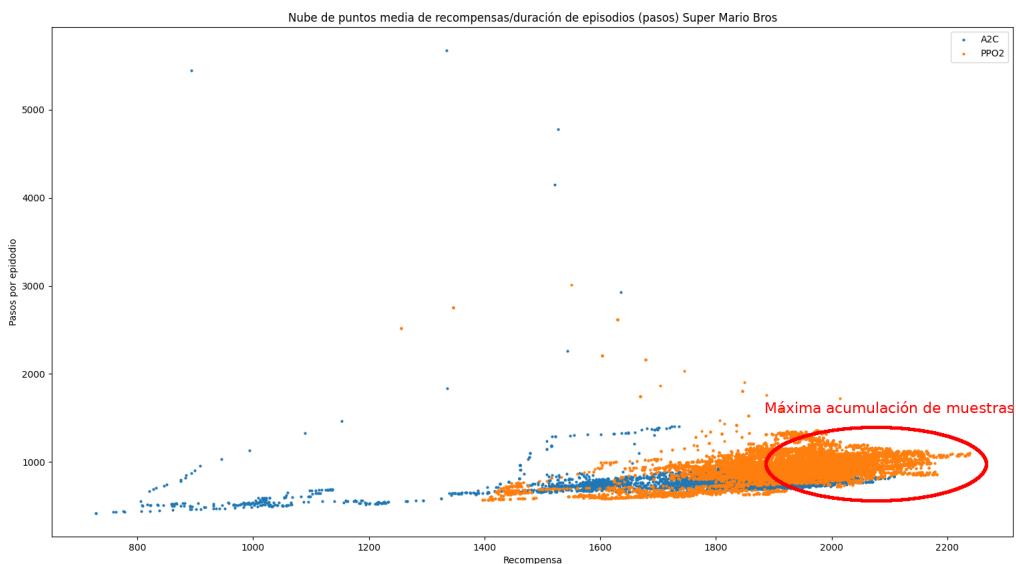


Figura 8.14: Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en *SuperMarioBros-Nes*.

Cuando se obtiene recompensas (eje x) pequeñas, es decir, los puntos situados a la izquierda, la duración de los episodios puede ser más variable, incluso hay casos aislados en los que la duración se dispara hasta los 5000 pasos (las muestras recogidas en los inicios del entrenamiento vistas en la figura 8.12).

A medida que se producen mejores recompensas (parte derecha de la nube de puntos), vemos que estos se concentran en la esquina inferior derecha, lo cual se traduce en tendencia a que los episodios duren poco y en un rango acotado, en PPO2 anda entre 800 y 1100 y

en A2C entre 700 y 950 aproximadamente.

Esto puede ser indicativo de que comienza a tener más claro el objetivo y se dirige hacia él tratando de perder el menor tiempo posible para optimizar las recompensas definidas por su función de recompensa.

Si observamos los valores de entropía de ambos modelos:

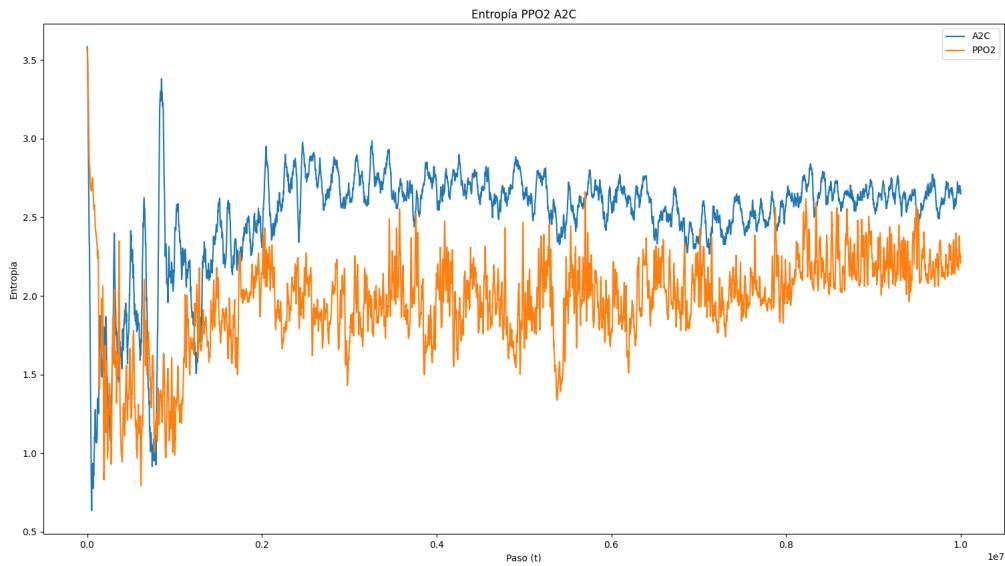


Figura 8.15: Valores de entropía de las políticas durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en *SuperMarioBros-Nes* nivel 1-1.

Vemos que, en general, A2C presenta unos valores de entropía mayores que PPO2. Si volvemos a fijarnos en los vídeos del comportamiento de ambos agentes para el nivel 1-1 (mirar enlaces de la tabla), nos damos cuenta de que los agentes pueden quedarse “atascados” a veces en un punto concreto. Ninguna función de recompensa es perfecta, si tiende a ir hacia la derecha siempre habrá veces que eso no le ayude (como en la parte que hay muchas monedas en pantalla). La exploración es muy importante incluso en fases más avanzadas del modelo para evitar que caiga en estos bucles de comportamiento o que, en caso de caer, puedan salir más fácilmente (moviéndose a la izquierda de vez en cuando, por ejemplo).

8.2.3 Desarrollo de PPO2

Según experimentos ya realizados con PPO2 y A2C (página 7 de [26]), en la mayoría de problemas mostrados tiende a ser mejor PPO2. Teniendo en cuenta la similitud de los entrenamientos mostrados hasta ahora, se tiene en consideración dichos experimentos para decidir realizar pruebas más ambiciosas con este algoritmo.

Entonces, se realizó un experimento de 100 millones de interacciones. Tardó un total de **85 horas, 25 minutos y 10 segundos** en realizar todo este entrenamiento. Empleando un total de pasos 10 veces superior al anterior experimento, se obtiene una recompensa de **4423 puntos** de máxima en los valores medios obtenidos por el agente.

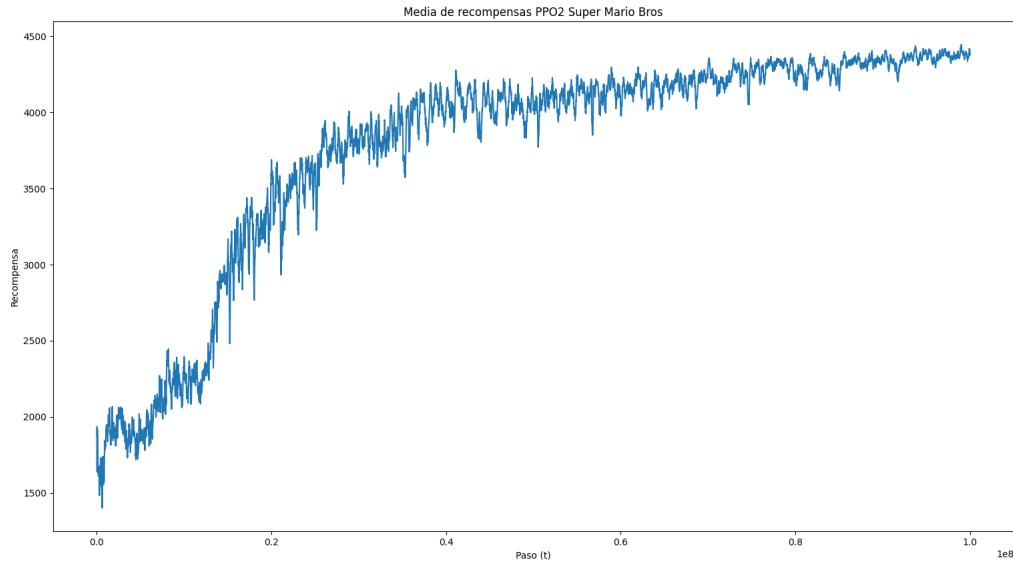


Figura 8.16: Media de recompensa durante entrenamiento de 100 millones de interacciones con PPO2 en *SuperMarioBros-Nes*.

Hay que tener en cuenta que el valor 0,1 del eje x equivale a toda la gráfica que veíamos en la figura 8.11. En esa figura parecía que los valores de recompensa se estaban comenzando a estabilizar. Es cuestión de relatividad con la escala, en realidad si se prolonga mucho en el tiempo (pasos), nos damos cuenta de que el algoritmo apenas estaba comenzando a dar sus primeros frutos.

Se produce una mejora bastante pronunciada sobre los 17 millones de pasos. En general, hay mejoras muy notables hasta los 40 millones, después parece estabilizarse más y tener mejoras más lentas y a un ritmo constante, aunque de nuevo podría depender de la escala.

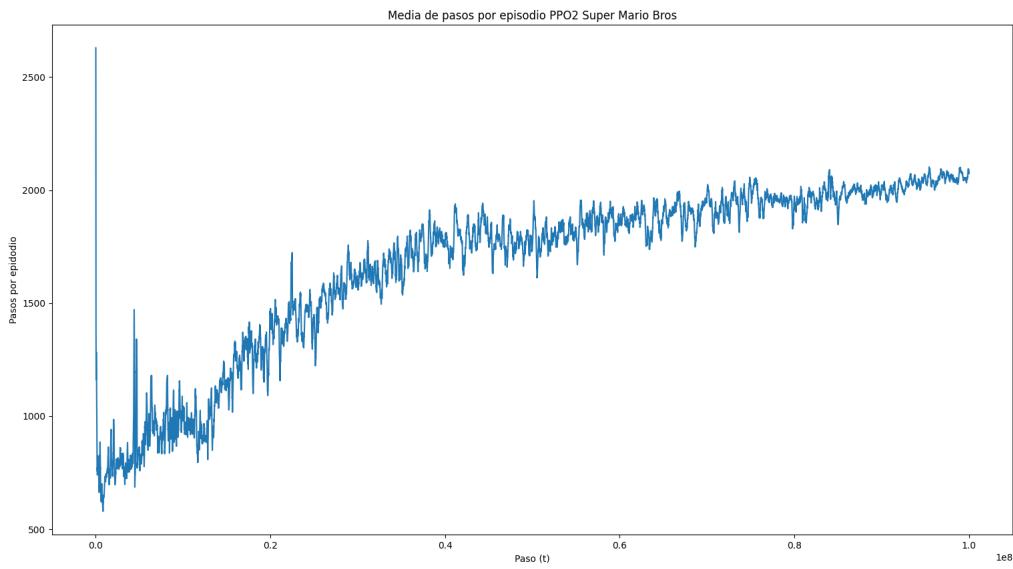


Figura 8.17: Media de duración de episodios (pasos) durante entrenamiento de 100 millones de interacciones con PPO2 en *SuperMarioBros-Nes*.

En cuanto a la duración de los episodios, vemos que existe esa tendencia de la que se estaba hablando. Cuanto más progresó el modelo existen menos variaciones en la duración de sus episodios. Además, existe una correlación directa con la recompensa que se obtiene por parte del agente. Todo esto nos da a entender que la función de recompensa está bien diseñada para el aprendizaje del modelo.

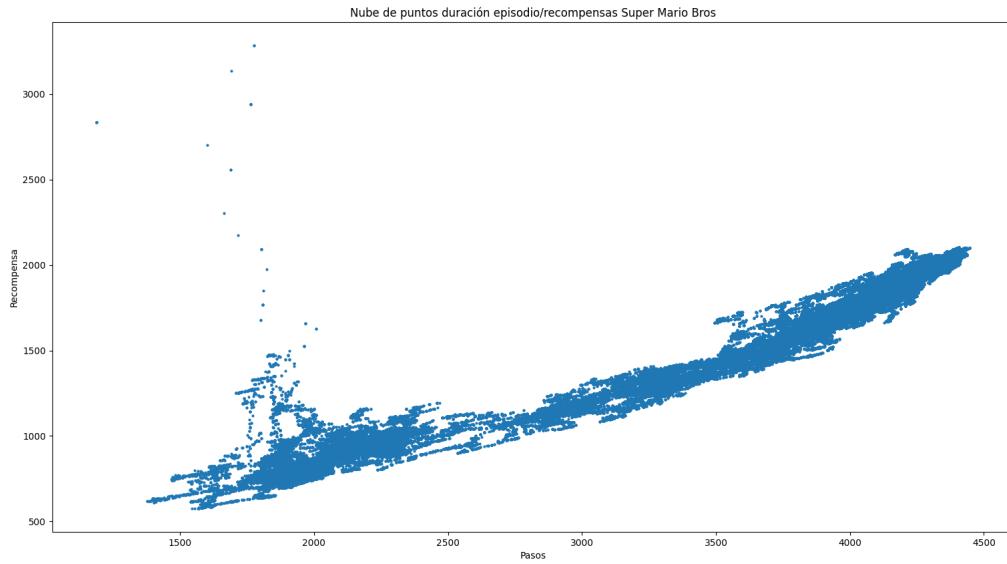


Figura 8.18: Nube de puntos entre media de recompensas y duración de episodios (pasos) durante entrenamiento de 10 millones de interacciones con A2C y PPO2 en *SuperMarioBros-Nes*.

Esa correlación tan clara que se puede observar indica que la obtención de recompensas

más altas favorece a que los episodios sean más largos y que, por tanto, progrese más en el nivel que antes.

Al observar el comportamiento del agente, tanto en los vídeos mostrados hasta ahora como en estos pequeños experimentos. se evidenció que el agente, además, aprende el nivel por partes, poco a poco. Entonces, en cada intento sabe llegar hasta cierto punto, y a partir de este el entorno le es desconocido y se equivoca.

IV

Posibles mejoras y conclusiones

9	Conclusiones finales	107
9.1	Posibles mejoras	
9.2	Conclusiones	
	Bibliografía	109
	Artículos	
	Libros	



9. Conclusiones finales

9.1 Posibles mejoras

A continuación, se plantea unas mejoras que se han ido pensado conforme se desarrollaba el presente proyecto. Me hubiera gustado incluir algunas de estas mejoras desde su inicio, pero lamentablemente los límites de tiempo y recursos de computación lo han hecho imposible:

- A pesar de haber leído documentación e indagado sobre ello, no ha sido posible llevar a cabo una investigación de algunos de los agentes más punteros en este sector tales como **AlphaGo** y **AlphaStar**, mencionados en varias ocasiones a lo largo de la documentación.
- Existen otras técnicas que no han sido probadas, algunas de ellas incluso mencionadas. En este trabajo se han descrito las técnicas más frecuentes, conocidas y utilizadas debido a su mejor rendimiento general frente al resto. Sin embargo, esto no significa que otras técnicas como TRPO, ACER, ACKTR, VPG y otros tantos más no puedan ser la mejor solución para ciertos problemas. Se podría estudiar estos algoritmos y experimentar con los mismos.
- *MountainCar-v0* es interesante de estudiar debido a su problema de *sparse reward*, mientras que *Super Mario Bros* tiene una complejidad de acciones y estados mucho mayor. Cada problema tiene sus características particulares, sería interesante plantear **más entornos** y problemas para ver como se defienden las técnicas expuestas.
- Desarrollar y modificar la implementación de *OpenAI gym* originales para *MountainCar-v0* con la finalidad de tratar de mejorar su definición de función de recompensa, tratando de mitigar el problema de escasez de recompensas positivas, al mismo tiempo que se estudia de una forma más detallada la *influencia directa* de esta función a la mejora del rendimiento en el aprendizaje de los algoritmos DRL.

- Realizar más experimentos en general y estudiar más aspectos del aprendizaje. Por falta de recursos y tiempo ha sido imposible estudiar cosas tales como modificar la tasa de descuento (γ) que viene predefinidos para los entornos y algoritmos de los *baselines* y estudiar de forma empírica hasta que punto estos hiperparámetros pueden modificar el comportamiento final del modelo. También se incluye aquí cambios en las capas de las redes neuronales utilizadas, ruido en los parámetros, etc.

9.2 Conclusiones

Cuando se comenzó este proyecto, no sabía lo que suponía el aprendizaje por refuerzo profundo más allá de una definición de dos frases. Ahora, me doy cuenta de que conozco menos de este campo de lo que realmente suponía al comienzo. Aún así, hemos podido aprender muchos conceptos, ideas, algoritmos y otras muchas cosas al mismo tiempo que hemos ido siendo conscientes de problemas y retos que estas técnicas presentan. El mundo del DRL es mucho mayor de lo que podía alcanzar a imaginar en un principio.

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 en sí mismo. Sin embargo, hay que entender que, la curiosidad es lo que nos ha permitido avanzar siempre. 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.

Los entornos *Gym* que hemos visto en este proyecto pueden entenderse como simplificaciones de problemas, en el mundo real tiene mayor complejidad que el que ofrece un nivel de Super Mario Bros, como la conducción automática de vehículos, por ejemplo.

No es viable tratar de resolver estos problemas directamente sin antes haber estudiado y mejorado técnicas que sean capaces de realizar acciones más simples. Esa es la importancia que tiene este tipo de investigaciones y es importante darles el valor que se merecen. Es más fácil investigar y desarrollar alrededor de estos entornos acotados y simplificados, antes de enfrentarnos a la incertidumbre de los problemas más realistas.

El DRL aún está en pleno desarrollo, al igual que la Inteligencia Artificial en general, todo esto está suponiendo la siguiente gran revolución, resolviendo problemas que antes se creían irresolubles o que apenas podíamos indagar en los mismos.



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 25).
- [2] Anaconda. “Who is Anaconda?” En: *Página oficial* (2020). <https://www.anaconda.com/about-us> (véase página 124).
- [4] OpenAI Baselines. “OpenAI Baselines: ACKTR & A2C”. En: *OpenAI Web* (2017). URL: <https://openai.com/blog/baselines-acktr-a2c/> (véanse páginas 64, 66).
- [5] Stable Baselines. “DDPG”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/modules/ddpg.html> (véase página 135).
- [6] Stable Baselines. “Examples”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/examples.html> (véase página 135).
- [7] Stable Baselines. “Welcome to Stable Baselines docs! - RL Baselines Made Easy”. En: *Github Docs* (2020). <https://stable-baselines.readthedocs.io/en/master/> (véase página 133).
- [9] Fernando Sancho Caparrini. “Ejemplo de gradiente descendente estocástico.” En: *Imagen* (consultado en 2020). <http://www.cs.us.es/~fsancho/?e=165> (véase página 30).
- [11] Jaime Crispí. “Introducción al deep learning parte 2: Redes Neuronales Convolucionales (imagen extraída)”. En: *Medium Web* (2019). URL: <https://medium.com/@jcrispis56/introducci%C3%B3n-al-deep-learning-parte-2-redes-neuronales-convolucionales-f743266d22a0> (véase página 33).
- [12] Irene Alvarado engineer y creative technologist. “redes Neuronales”. En: *Github* (Consultado en 2020). https://ml4a.github.io/ml4a/es/neural_networks/ (véanse páginas 26-28).

- [13] FileInfo. “.PKL File Extension”. En: *Página oficial* (2020). <https://fileinfo.com/extension/pkl> (véase página 128).
- [14] gcpping. “Measure your latency to GCP regions”. En: *gcpping.com* (2020). <http://www.gcping.com/> (véase página 120).
- [15] Docs Github. “Installation”. En: *Stable-Baselines Official* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/install.html> (véase página 124).
- [16] Google. “DeepMind”. En: *Página web oficial* (2020). <https://deepmind.com/> (véase página 18).
- [17] 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 122).
- [18] Google. “Instala controladores de GPU”. En: *Documentación* (2020). URL: <https://cloud.google.com/compute/docs/gpus/install-drivers-gpu#ubuntu-driver-steps> (véase página 125).
- [19] Joshua Hare. “Dealing with Sparse Rewards in Reinforcement Learning”. En: *University of Sheffield* (2019). URL: <https://arxiv.org/pdf/1910.09281.pdf> (véanse páginas 48, 87, 88, 91).
- [20] Hado van Hasselt. “Double Q-learning”. En: *Multi-agent and Adaptive Computation Group Centrum Wiskunde & Informatica* (2020). URL: <https://papers.nips.cc/paper/3964-double-q-learning.pdf> (véase página 52).
- [21] Red Hat. “Provisioning Ansible”. En: *Red Hat Ansible* (2020). <https://www.ansible.com/use-cases/provisioning> (véase página 123).
- [22] Iberdrola. “¿Qué es la Inteligencia Artificial?” En: <https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial> (2020) (véase página 15).
- [23] Yuchen Xie† & Zhuoran Yang Jianqing Fan Zhaoran Wang†. “A Theoretical Analysis of Deep Q-Learning”. En: *Cornell University* (2020). URL: <https://arxiv.org/pdf/1901.00137v3.pdf> (véanse páginas 49, 53).
- [24] Volodymyr Mnih Adria Puigdomenech Badia Mehdi Mirza Alex Graves Timothy P. Lillicrap Tim Harley David Silver & Koray Kavukcuoglu. “Asynchronous Methods for Deep Reinforcement Learning”. En: *Cornell University* (2016). URL: <https://arxiv.org/pdf/1602.01783.pdf> (véase página 64).
- [25] John Schulman & Filip Wolski & Prafulla Dhariwal & Alec Radford & Oleg Klimov. “Proximal Policy Optimization Algorithms”. En: *DeepMind* (2017). <https://arxiv.org/pdf/1707.06347.pdf> (véanse páginas 60, 62, 100).
- [26] 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 125).
- [27] Ludoteka. “Go”. En: *Ludoteka.com* (Consultado en 2020). <http://www.ludoteka.com/juego-go.html> (véase página 23).
- [28] 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 25).

- [30] 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> (véanse páginas 18, 37).
- [32] 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 17).
- [33] Netflix. “Alphago”. En: *Documental* (2017). <https://www.alphagomovie.com/> (véase página 18).
- [34] OpenAI. “Better Exploration with Parameter Noise”. En: *Página web oficial* (2017). URL: <https://blog.openai.com/better-exploration-with-parameter-noise/> (véanse páginas 62, 63).
- [35] OpenAI. “OpenAI Baselines: DQN”. En: *Página web oficial* (2017). <https://openai.com/blog/openai-baselines-dqn/> (véanse páginas 53, 56).
- [36] OpenAI. “Proximal Policy Optimization (PPO2)”. En: *Página web oficial* (2017). <https://openai.com/blog/openai-baselines-ppo/> (véase página 60).
- [37] OpenAI. “Vanilla Policy Gradient”. En: *OpenAI Spinning Up* (2018). URL: <https://spinningup.openai.com/en/latest/algorithms/vpg.html> (véase página 59).
- [38] OpenAI. “About OpenAI”. En: *Página web oficial* (2020). <https://openai.com/about/> (véase página 73).
- [39] OpenAI. “Gormato observaciones Gym Retro”. En: *Github* (2020). URL: <https://github.com/openai/retro/blob/fb44c13fa5bb59353afa1d5f752d73638f74cc7f/retro/enums.py> (véase página 80).
- [40] OpenAI. “Gym”. En: *Página web oficial* (2020). <https://gym.openai.com/> (véanse páginas 18, 76).
- [41] OpenAI. “Gym Retro”. En: *Github* (2020). URL: <https://github.com/openai/retro/tree/fb44c13fa5bb59353afa1d5f752d73638f74cc7f> (véase página 79).
- [42] OpenAI. “<https://gym.openai.com/envs/MountainCar-v0/>”. En: *Entornos Gym* (2020). <https://gym.openai.com/envs/MountainCar-v0/> (véase página 78).
- [43] OpenAI. “Loading and visualizing results (open in colab)”. En: *Github* (2020). <https://github.com/openai/baselines/blob/master/docs/viz/viz.ipynb> (véase página 132).
- [44] OpenAI. “MountainCarContinuous v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCarContinuous-v0> (véase página 78).
- [45] OpenAI. “OpenAI baselines”. En: *Repositorio Github oficial* (2020). <https://github.com/openai/baselines> (véase página 127).
- [46] OpenAI. “Progresos de OpenAI”. En: *Página web oficial* (2020). <https://openai.com/progress/> (véase página 73).
- [47] OpenAI. “Table of environments”. En: *Github* (2020). <https://github.com/openai/gym/wiki/Table-of-environments> (véase página 76).
- [48] OpenAI. “Tipos de redes en baselines”. En: *Github* (2020). URL: <https://github.com/openai/baselines/blob/master/baselines/common/models.py> (véanse páginas 80, 85, 93).

- [49] OpenAi. “Detalles técnicos del entorno MountainCar v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCar-v0> (véase página 77).
- [50] Trust Region Policy Optimization. “John Schulman, Sergey Levine, Philipp Moritz , Michael Jordan & Pieter Abbeel”. En: *Pieter AbbeelUniversity of California, Berkeley, Department of Electrical Engineering and Computer Sciences* (2017). URL: <https://arxiv.org/pdf/1502.05477.pdf> (véase página 61).
- [51] Craig Quiter. “DeepDrive Universe”. En: *Youtube* (2017). https://www.youtube.com/watch?time_continue=26&v=X4u2DC0LoIg&feature=emb_logo (véase página 76).
- [52] Satinder Singh & Yishay Mansour Richard S. Sutton David McAllester. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. En: *AT&T Labs* (2000). URL: <https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf> (véase página 58).
- [53] Vicente Rodríguez. “Redes neuronales convolucionales (imagen extraída)”. En: *Página web* (2018). URL: <https://vincentblog.xyz/posts/redes-neuronales-convolucionales> (véase página 33).
- [55] Juan Gómez Romero. “Curso sobre Aprendizaje Profundo por Refuerzo en ECI 2019”. En: *Github* (2019). <https://github.com/jgromero/eci2019-DRL> (véanse páginas 25, 35, 46-48, 50, 51, 54, 55, 57, 58, 60).
- [56] Juan Gómez Romero. “Sistemas Inteligentes para la Gestión de la Empresa. Tema 4: Modelos avanzados Deep Learning”. En: *Universidad de Granada* (2019) (véase página 25).
- [57] 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 18).
- [58] TensorFlow. “Configuraciones de compilación probadas (Linux)”. En: *Página web oficial* (2020). URL: https://www.tensorflow.org/install/source#tested_build_configurations (véase página 125).
- [59] Tensorflow. “Why TensorFlow”. En: *Página oficial* (2020). <https://www.tensorflow.org/about> (véase página 125).
- [60] 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 121).
- [61] Jordi Torres.AI. “DEEP LEARNING: INTRODUCCIÓN PRÁCTICA CON KERAS (PRIMERA PARTE) (imagen extraída)”. En: *Web Jordi Torres.AI* (2020). URL: <https://torres.ai/deep-learning-inteligencia-artificial-keras/> (véase página 32).
- [62] Oriol Vinyals. “AlphaStar: Mastering the Real Time Strategy Game StarCraft II - Oriol Vinyals”. En: *Youtube* (2019). <https://www.youtube.com/watch?v=3UdH3lPF7nE> (véanse páginas 18, 37).

- [63] Timothy P. Lillicrap & Jonathan J. Hunt & Alexander Pritzel & Nicolas Heess & Tom Erez Yuval Tassa & David Silver & Daan Wierstra. “Continuous control with deep reinforcement learning”. En: *DeepMind* (2019). <https://arxiv.org/pdf/1509.02971.pdf> (véase página 62).
- [64] Wikipedia. “Gráfica de aprendizaje por refuerzo.” En: *Imagen* (consultado en 2020). https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo (véase página 36).

Books

- [3] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learning*. Editado por . MIT Press, 2018 (véanse páginas 35, 57).
- [8] Fernando Berzal. *Redes Neuronales y Deep Learning*. Editado por . <https://sites.google.com/view/redes-neuronales/>. Amazon, 2019 (véase página 25).
- [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 133).
- [25] Max Pumperla y Kevin ferguson. *Deep Learning and The Game of Go*. Manning, 2019 (véanse páginas 15, 87).
- [31] Miguel Morales. *Deep Reinforcement Learning*. Editado por Manning. Grokking, 2018 (véanse páginas 35, 40, 46, 48, 58, 66, 67).
- [54] 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 25).

V

Apéndice

A	Configuración en la nube	117
A.1	Configuración de la Infraestructura	
A.2	Configuración de la Máquina	
B	Uso de la Máquina Virtual	127
B.1	Archivos generados en los logs	
B.2	Visualizar la información de los logs	
C	Stable Baselines	133
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*.

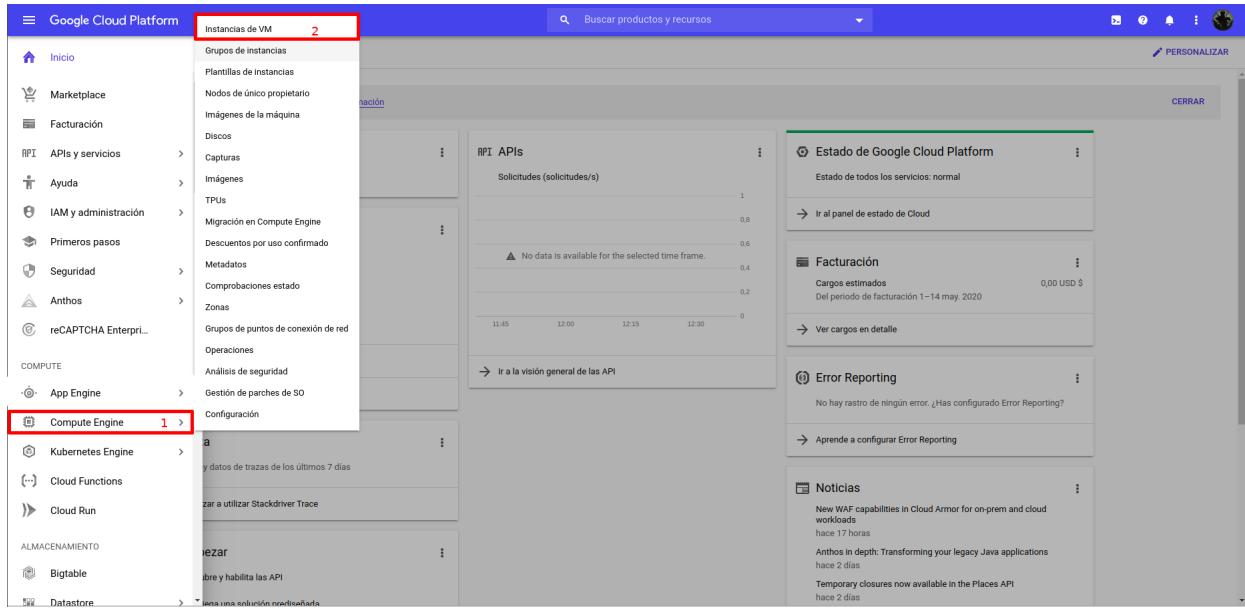


Figura A.1: Iniciando asistente para creación de máquina virtual.

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.

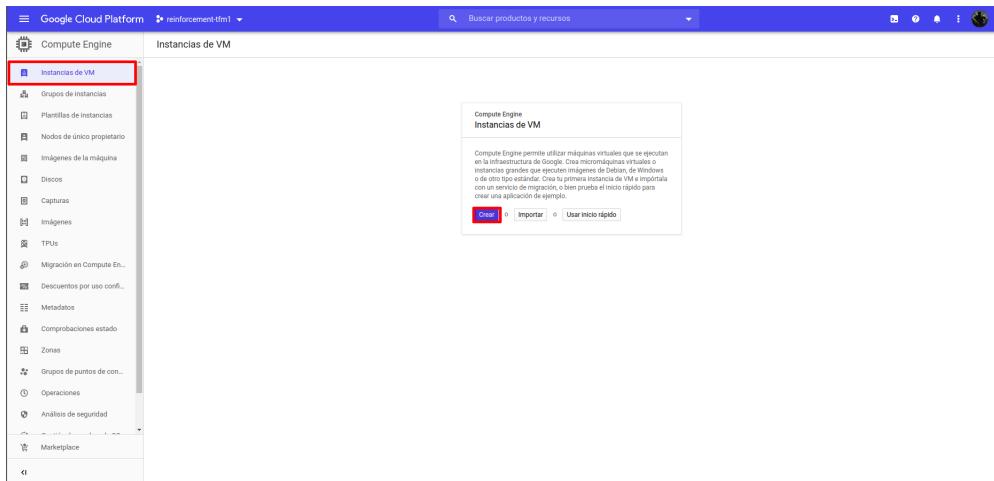


Figura A.2: Creando una máquina virtual.

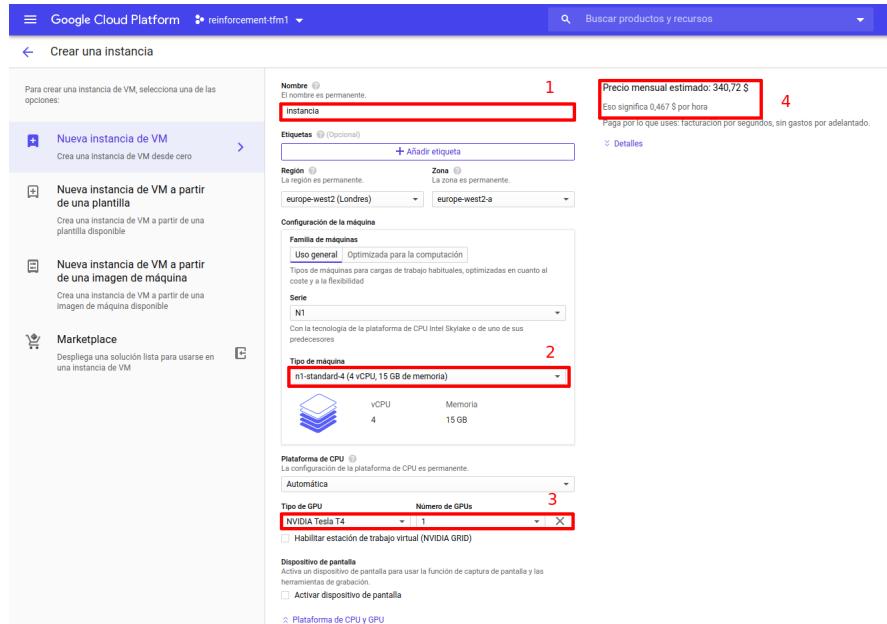


Figura A.3: Configurando la máquina virtual para el proyecto.

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:

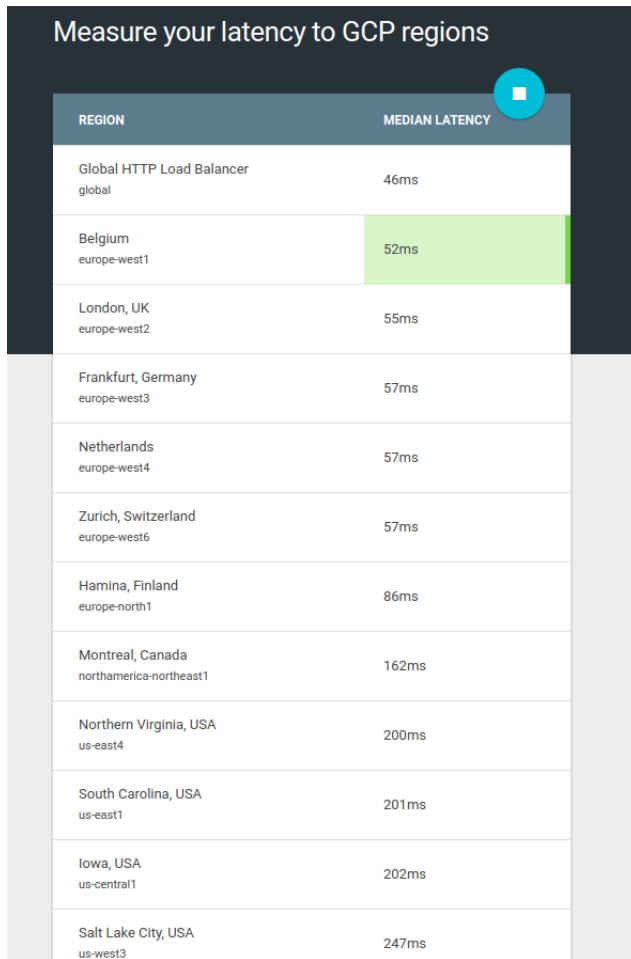


Figura A.4: Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [14]

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.[60]

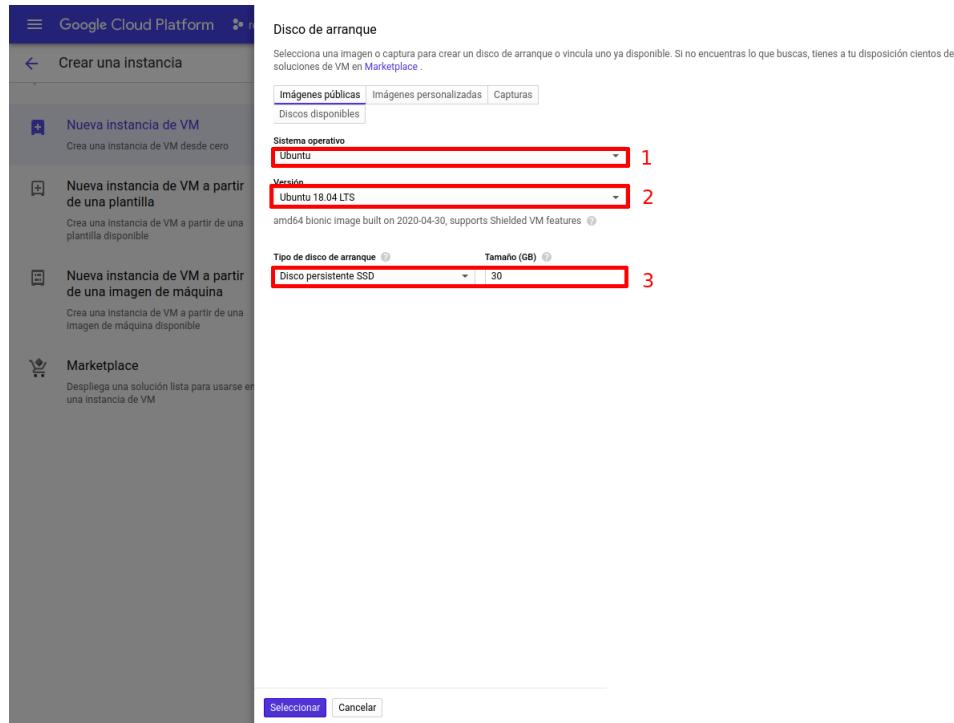


Figura A.5: Indicando Sistema operativo y memoria SSD a la máquina virtual.

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:

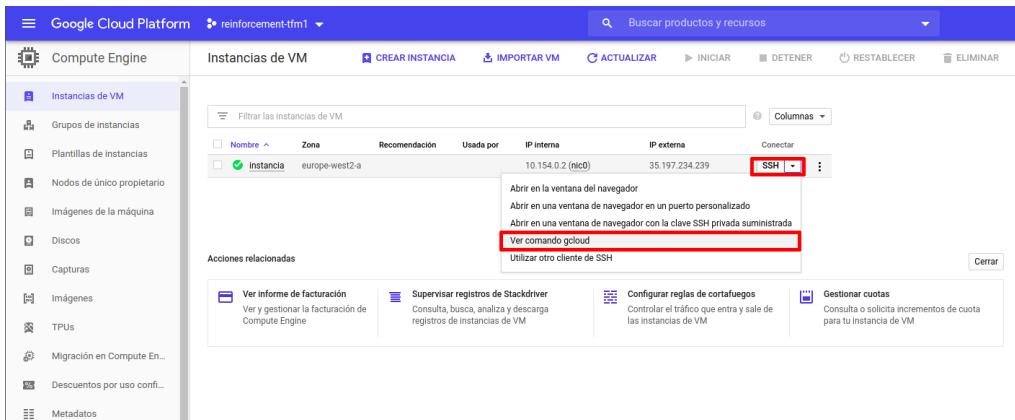


Figura A.6: Comprobando que la máquina virtual ha sido creada con éxito.

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 [17]. 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** [21], 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

#Para 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

#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

#Para correr la rom de Super Mario Bros con GYM
pip install gym-retro joblib atari-py opencv-python anyrl

cd ~/mario

python -m retro.import .
```

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 funcionase. 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.

Clonamos el repositorio Github de donde extraemos los *baselines* de *OpenAI*. 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*[15] 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.

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.

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**

res para esa tarjeta concreta, así *Tensorflow* puede utilizarla, es la librería que utiliza los baselines para las redes neuronales profundas.[59].

Estos controladores no son fáciles de configurar para que funcione con *tensorflow-gpu==1.14* que es la versión que mejor funciona en los baselines de OpenAI. No está incluido en el script y debe hacerse tras su ejecución, tanto la instalación de controladores como su posterior instalación de TensorFlow.

Tensorflow ofrece información de compatibilidades [58] sobre cada una de sus versiones. Para la versión 1.14 aconsejan una versión de Python entre la 3.3 y 3.7, cuDNN 7.4 y CUDA 10.0.

Sin embargo, esta combinación no sirvió para que los algoritmos reconociesen la tarjeta gráfica de la máquina. Tras realizar muchas pruebas con versiones, finalmente se consiguió que funcionase utilizando CUDA 10.2, Python 3.6, CuDNN 7.4 y controladores de video 440.100 (con el 410 que se recomienda tampoco funcionó).

Google Cloud ofrece una guía específica para Ubuntu 18.04 LTS, aunque en este caso no sirvió de ayuda. [18]

Es importante *reiniciar* la máquina virtual cuando finalicemos. Esto se hace concretamente por los controladores de la tarjeta gráfica. Si no lo hacemos, 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:

```
+--+
| NVIDIA-SMI 440.82      Driver Version: 440.82      CUDA Version: 10.2 |
+-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|=====+=====+=====+=====+=====+=====+=====+=====|
| 0   Tesla T4      Off  | 00000000:00:04.0 Off |          0 |
| N/A  77C   P0    51W /  70W |    848MiB / 15109MiB |  68%      Default |
+-----+-----+-----+-----+-----+-----+-----+-----+
+-----+
| Processes:
| GPU  PID  Type  Process name          GPU Memory |
|           Usage
|=====+=====+=====+=====+=====+=====+
| 0    6058  C    python                279MiB |
| 0    6172  C    python                279MiB |
| 0    7283  C    python                279MiB |
+-----+
```

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 [27]. Posteriormente será utilizado en sentido opuesto para **descargar los agentes** entrenados en la máquina al equipo local.

The image shows two terminal windows side-by-side. The left window is titled 'hapneck@Instancia: ~' and shows the output of a 'gcloud beta compute ssh' command. It includes system information, disk usage, memory usage, swap usage, and a note about security updates. The right window is titled 'hapneck@Equipo-Alex: ~/Escritorio/Universidad/TFM' and shows the execution of an 'scp' command to transfer a file named 'script_maquina.sh' to a remote host at '35.197.234.239'. A red arrow points from the left terminal's command area to the right terminal's command area.

```
(base) hapneck@Equipo-Alex:~$ gcloud beta compute ssh --zone "europe-west2-a" "instancia" --project "reinforcement-tfm"
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%          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:/
(base) hapneck@Equipo-Alex:~/Escritorio/Universidad/TFM$ The authenticity of host '35.197.234.239' (35.197.234.239) can't be established.
ECDSA key fingerprint is SHA256:198KwMPN+J98e09tYLZfEFLUeloyZXPXRQHLxBDX/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
(base) hapneck@Equipo-Alex:~/Escritorio/Universidad/TFM$ _
```

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[45], 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 [13]. 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 guardarlo, lo ejecutará para que podamos verlo.

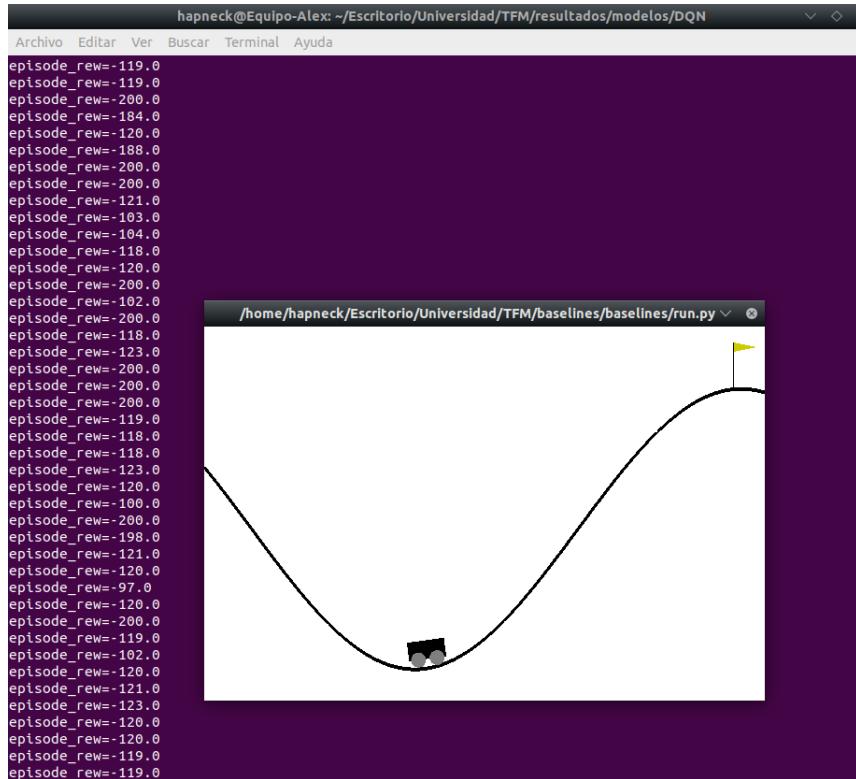


Figura B.1: Probando agente entrenado en entorno simulado *MountainCar-v0*.

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 *pasos* 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 ??) 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.[43]

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[7], 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**. Ha sido utilizado en *MountainCarContinuous-v0* para el algoritmo DDPG.

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
        guardara el modelo")

```

```

    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 [5] [6]. 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 actúa, 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.