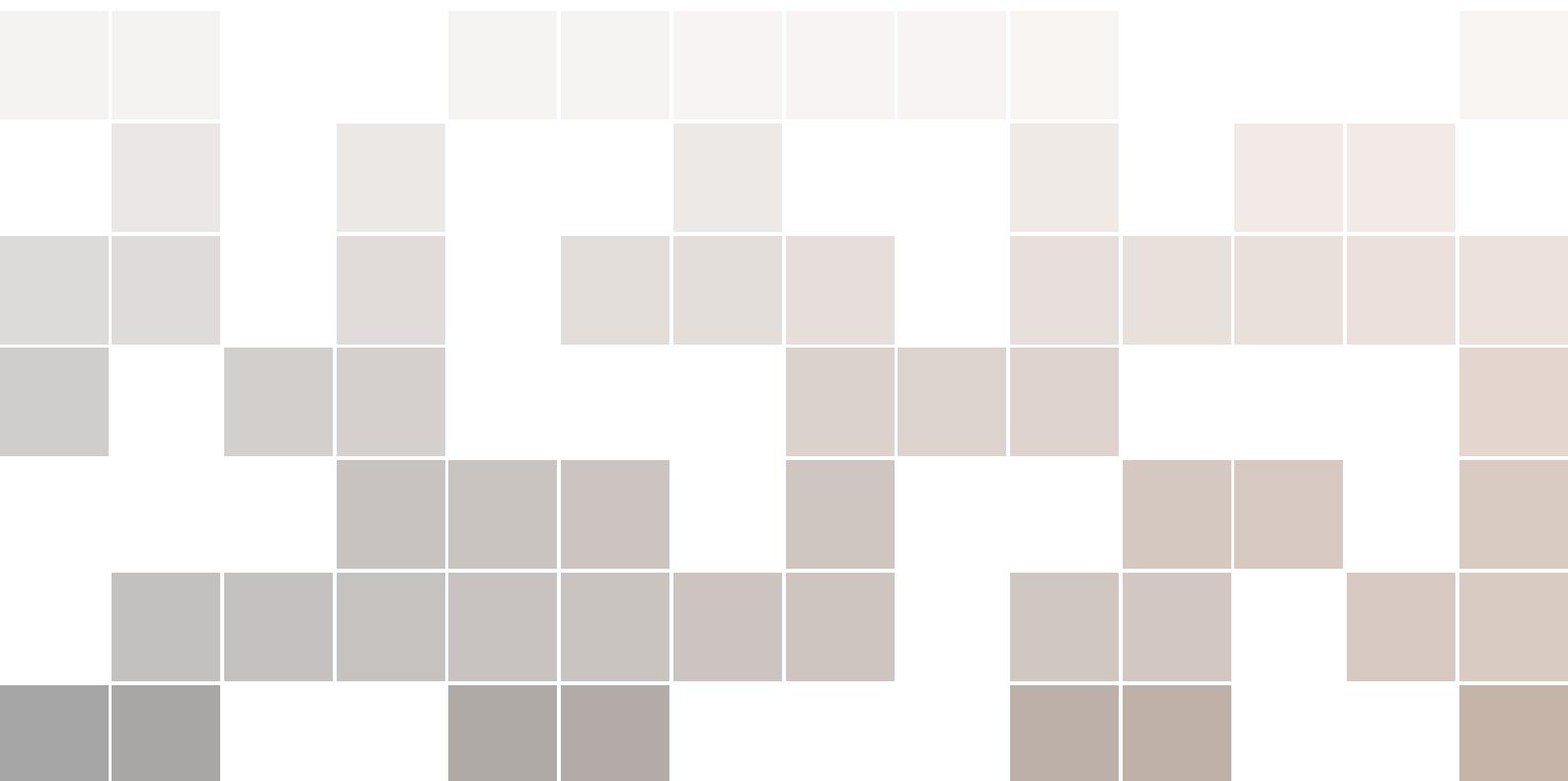


Aprendizaje profundo por refuerzo para la resolución de problemas.

Técnicas actuales

Alejandro Campoy Nieves

Tutor: Juan Gómez Romero



Copyright © 2020 Alejandro Campoy



Índice general

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

I	Introducción	
1	Introducción	13
1.1	Aprendizaje por refuerzo profundo en la actualidad	16
1.2	Objetivos	16
1.3	Planificación	17
II	Aprendizaje por Refuerzo Profundo: Conceptos Previos	
2	Redes Neuronales	23
2.1	Funciones de activación (h)	24
2.2	Funciones de salida (h')	26
2.3	Funciones de coste	26
2.4	Algoritmos de optimización	27
3	Aprendizaje por refuerzo	31
3.1	Episodios	32
3.2	Experiencia	34
3.2.1	Representación de la experiencia	34
3.3	Muestreo de Distribución de la Probabilidad	35

3.4	Técnicas	35
3.4.1	Política de gradientes	36
3.5	Q-Learning	37
3.6	Método actor-critic	38
3.6.1	Ventaja	38

III

Desarrollo y Experimentación

4	Aprendizaje por refuerzo profundo	45
4.1	Algoritmos	45
4.1.1	PPO2	46
4.1.2	DQN	47
4.1.3	DDPG	48
5	Entornos Open AI Gym	51
5.1	MountainCar-v0	52
5.2	MountainCarContinuous-v0	54
5.3	Entorno más complejo (aun sin decidir)	54
6	Entorno de desarrollo	55
6.1	Google Cloud Platform	55
6.2	Configuración de la Infraestructura	56
6.3	Configuración de la Máquina	60
6.4	Uso de la Máquina Virtual	64
6.5	Forma de trabajar	66
6.6	Archivos generados en los logs	67
7	Experimentación: Resultados Obtenidos	71
7.1	Visualizar la información de los logs	71
7.2	MountainCar-v0	72
7.2.1	DQN	72
7.2.2	PPO2	80
7.2.3	DDPG	82
7.3	Otro entorno más complejo (Aún sin decidir)	91
7.3.1	DQN	91
7.3.2	PPO2	91
7.3.3	DDPG	91

IV

Estado del arte

8	AlphaGO	97
9	AlphaStar	99

10	Conclusiones finales	103
10.1	Posibles mejoras	103
10.2	Conclusiones	103
	Bibliografía	105
	Artículos	105
	Libros	108

Índice de figuras

1.1	Categorización de tipos de aprendizajes en Machine Learning.	14
1.2	Estructura de conceptos dentro de la Inteligencia Artificial.	15
2.1	Esquema simplificado de una red neuronal [1]	23
2.2	Función sigmoide en plano bidimensional. Extraída de origen [9].	25
2.3	Función ReLU en plano bidimensional. Extraída de origen [9].	26
2.4	Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de origen [6].	28
3.1	Datos a manejar en aprendizaje por refuerzo por un agente. Imagen extraída y posteriormente modificada de origen [47].	32
3.2	Ciclos de aprendizaje por refuerzo (aunque se podría usar más de un episodio en cada vuelta).	33
3.3	Esquema de representación de la experiencia en aprendizaje por refuerzo.	35
3.4	La ventaja se incluye en la experiencia, pero es introducida al final del episodio, cuando ya se sabe todas las recompensas.	40
4.1	Ruido en el espacio de acciones (izquierda) frente a ruido en los parámetros de la red (derecha). Extraída de origen. [27]	48
5.1	Representación gráfica del problema MountainCar v0 de OpenAI Gym.	52
6.1	Iniciando asistente para creación de máquina virtual.	56
6.2	Creando una máquina virtual.	56
6.3	Configurando la máquina virtual para el proyecto.	57
6.4	Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [11].	58
6.5	Indicando Sistema operativo y memoria SSD a la máquina virtual.	59
6.6	Comprobando que la máquina virtual ha sido creada con éxito.	60
6.7	Entrando en la máquina virtual.	60
6.8	Muestra de uso de GPU al entrenar con los baselines.	64

6.9	Pasando el script a la máquina virtual para abastecerla.	64
6.10	Probando agente entrenado en entorno simulado <i>MountainCar-v0</i> .	66
6.11	Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada	
67		
7.1	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i>	72
7.2	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 7.1). Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i>	73
7.3	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 3 y en <i>MountainCar-v0</i> .	74
7.4	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i> .	75
7.5	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 7.4). Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i> .	75
7.6	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 13 y en <i>MountainCar-v0</i> .	76
7.7	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i> .	76
7.8	Recompensas obtenidas durante entrenamiento(versión suavizada de la figura 7.7). Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i> .	77
7.9	Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 46 y en <i>MountainCar-v0</i> .	77
7.10	Porcentaje de tiempo empleado en explorar para algoritmo DQN en <i>MountainCar-v0</i> .	79
7.11	Resumen general de progreso de algoritmo DQN en <i>MountainCar-v0</i> .	80
7.12	Recompensa obtenida con el algoritmo PPO2 en <i>MountainCar-v0</i> , independientemente de la semilla iteraciones y otros parámetros.	81
7.13	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 3 y en <i>MountainCar-v0</i> .	86
7.14	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3 y en <i>MountainCar-v0</i> .	87
7.15	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 13 y en <i>MountainCar-v0</i> .	87
7.16	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 13 y en <i>MountainCar-v0</i> .	88
7.17	Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 46 y en <i>MountainCar-v0</i> .	88
7.18	Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 46 y en <i>MountainCar-v0</i> .	89
7.19	Gráfica general obtenida durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3, 13 y 46 simultáneamente y en <i>MountainCar-v0</i> .	90
7.20	Gráfica comparativa de los tres algoritmos: DQN, PPO2 y DDPG en <i>MountainCar-v0</i> .	91

0.1 Resumen

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

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

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

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



Introducción

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



1. Introducción

La **Inteligencia Artificial** (IA) es una de las ramas o disciplinas de la informática relativamente más jóvenes que existen a día de hoy, nació en la década de 1960. Es el concepto más abstracto, genérico y amplio que podemos encontrar; se trata de una combinación de técnicas que nos permiten emular el comportamiento y capacidades de los seres humanos para resolver problemas de cualquier tipo. Incluyendo en este concepto cosas como planificación, reconocimiento de objetos, sonidos, hablar, traducir, etc.^[16]

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: ^[17]

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

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

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

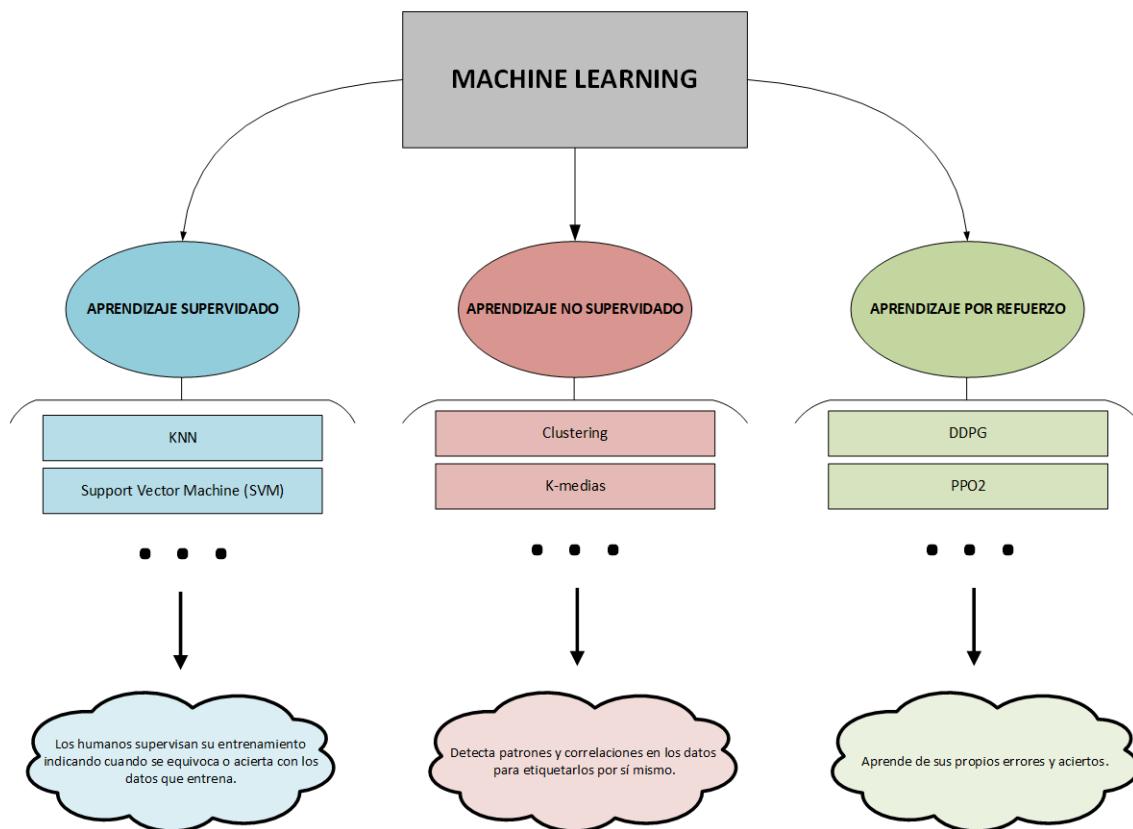


Figura 1.1: Categorización de tipos de aprendizajes en Machine Learning.

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

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

Estas redes neuronales son adecuadas en determinados problemas en los que no necesitamos saber **cómo piensa el agente**, simplemente queremos conseguir la **mejor respuesta** posible, sin más. Son usados cuando se dispone de mucha fuerza de computación y datos no estructurados (reconocimiento de imágenes, por ejemplo). Las redes neuronales pueden funcionar tanto en *aprendizaje supervisado* como en *aprendizaje por refuerzo*. Podemos indicarle los ejemplos con los que se quiere que aprenda y las salidas que debería de dar, o bien permitirle que explore el mismo y evalúe sus propias decisiones una vez obtiene recompensas negativas o positivas como fruto de las mismas, esto dependerá de la naturaleza del problema a resolver y de los datos que deba manejar.^[24]

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

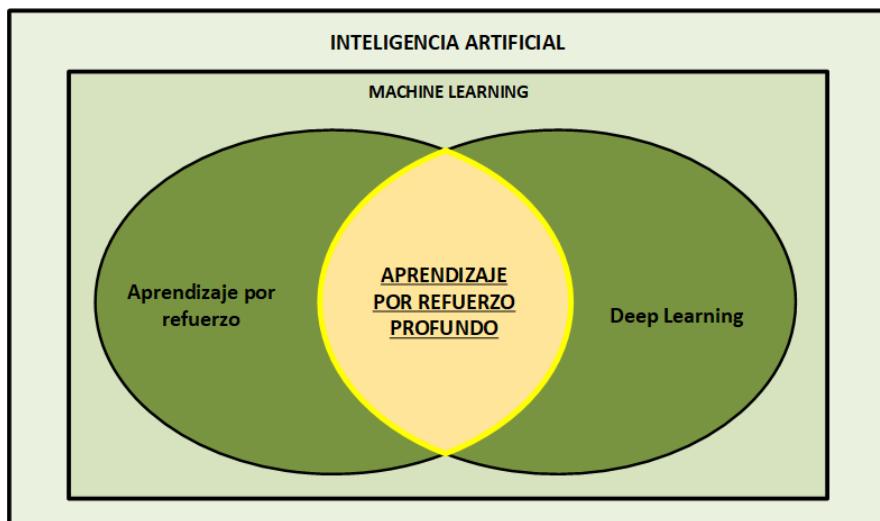


Figura 1.2: Estructura de conceptos dentro de la Inteligencia Artificial.

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

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

1.1 Aprendizaje por refuerzo profundo en la actualidad

Como se ha mencionado hace un momento, 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.^[13]

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.^{[40] [26]}

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.^{[45] [23]}

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

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

1.2 Objetivos

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

1. Entender los **conceptos principales** de la Inteligencia Artificial, concretamente todo lo relacionado con el aprendizaje por refuerzo profundo.
2. Entender, de una forma clara, el **funcionamiento y metodología** que siguen las **técnicas** más importantes de este campo.
3. Descubrir las posibilidades que nos da **gym** ^[31] para la construcción de **entornos o simulaciones** en los que entrenar a nuestros agentes sin la necesidad de espacios

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

físicos reales para ciertos problemas.

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

1.3 Planificación

Aprendizaje por Refuerzo Profundo: Conceptos Previos

2	Redes Neuronales	23
2.1	Funciones de activación (h)	
2.2	Funciones de salida (h')	
2.3	Funciones de coste	
2.4	Algoritmos de optimización	
3	Aprendizaje por refuerzo	31
3.1	Episodios	
3.2	Experiencia	
3.3	Muestreo de Distribución de la Probabilidad	
3.4	Técnicas	
3.5	Q-Learning	
3.6	Método actor-critic	

Introducción

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

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

En definitiva, esta tecnología para ciertos problemas puede quedarse algo corta o no conseguir aprender lo suficiente como para encontrarse satisfecho con los resultados, como sería el caso de jugar bien a **Go**. Aunque ya hablaremos de ese tema.[\[20\]](#)

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.

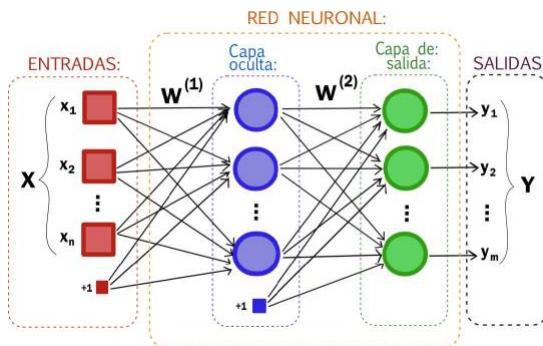


Figura 2.1: Esquema simplificado de una red neuronal [1]

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

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

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

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

Las capas ocultas, pudiendo ser una o más capas, no tienen una conexión directa con el entorno como hemos visto anteriormente. En su lugar, la entrada que recibe es la salida de la capa anterior(capa de entrada en este caso). La salida que devuelvan los nodos de esta capa serán utilizados como entrada en la capa siguiente. ¿Para que sirve esto? La respuesta resumida es que si solo tuviéramos la capa de entrada y salida, las entradas serían demasiado independientes entre sí a la hora de influir en la salida seleccionada entre todas las posibles. En general, los problemas del mundo real son mucho más complejos que eso y es necesario que la red sea capaz de detectar patrones e interdependencias entre los distintos valores de las entradas. Estas capas ocultas ayudan a dar esa **riqueza** al proceso.[9]

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)[9].

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

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

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

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

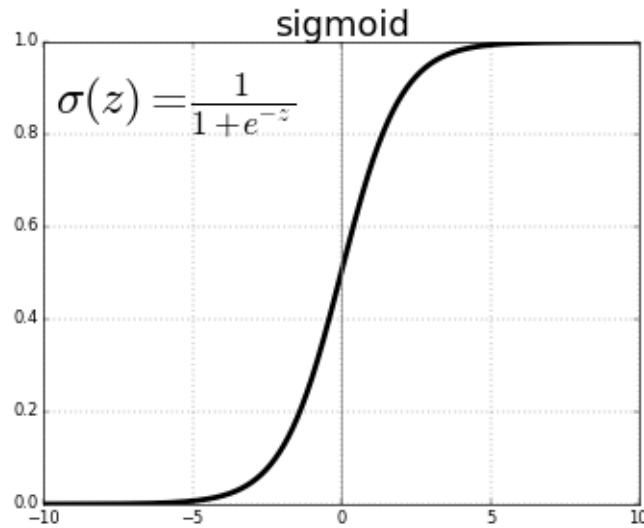


Figura 2.2: Función sigmoide en plano bidimensional. Extraída de origen [9].

En la figura 2.2, vemos como la salida se acota entre los valores 0 y 1 independientemente de las entradas que pueda recibir. El valor neutro de la entrada z (valor 0), corresponde con un valor intermedio de la salida(valor 0,5). A medida que la entrada es positiva y más alta, la salida se acerca al valor 1, mientras que cuando la entrada es negativa y menor, la salida se acerca al valor 0.

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

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

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

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

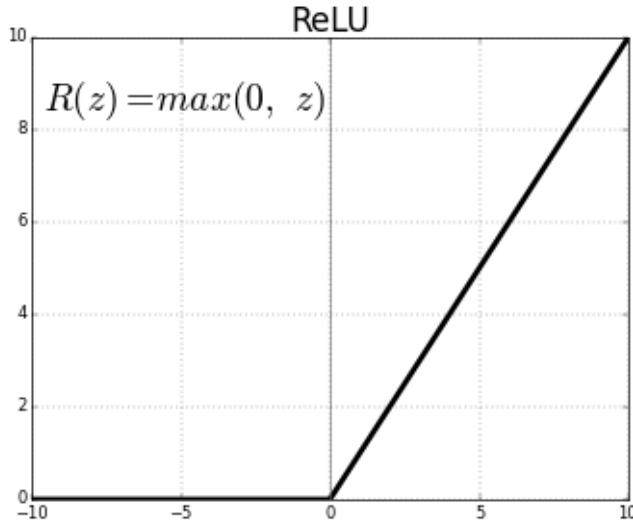


Figura 2.3: Función ReLU en plano bidimensional. Extraída de origen [9].

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

2.2 Funciones de salida (h')

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

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

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

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

2.3 Funciones de coste

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

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

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

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

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

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

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

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

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

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

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

2.4 Algoritmos de optimización

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

Los **algoritmos de optimización** buscan concretamente esto. Usando los valores de error que devuelve la función de perdida, determina como se van a actualizar los pesos con

la finalidad de que la red neuronal mejore en sus cálculos de salidas futuras. De nuevo, hay muchos algoritmos para realizar esta tarea, algunos de ellos son:

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

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

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

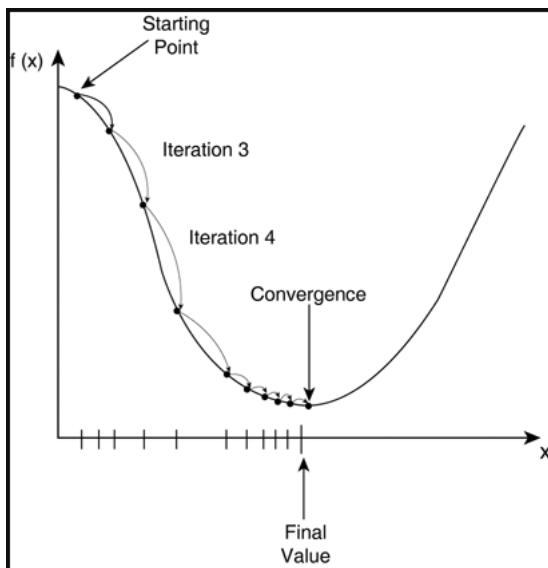


Figura 2.4: Ejemplo de proceso de gradiente descendente estocástico en una función f de un solo parámetro. Imagen extraída de origen [6].

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. [22] [39]



3. Aprendizaje por refuerzo

También es muy importante tener conocimientos con respecto a las técnicas de aprendizaje por refuerzo para desarrollar los algoritmos que veremos y utilizaremos más adelante. Comenzaremos por explicar algunos de sus conceptos básicos y metodología genérica que siguen, para luego centrarnos en algunas de las técnicas y estrategias más utilizadas y populares en la actualidad.

Una de las ventajas que tiene el paradigma del aprendizaje por refuerzo es que desarrollan algoritmos **genéricos**, con una abstracción alta de los problemas a resolver. Esto quiere decir que una misma técnica puede ser utilizada para entrenar dos agentes que resuelven dos problemas totalmente diferentes. Teniendo en cuenta y cambiando antes algunos parámetros para cada uno de ellos, como por ejemplo el formato del input que reciben del entorno, las acciones posibles que pueden realizar, el formato de la experiencia recolectada, el sistema de recompensas, etc.

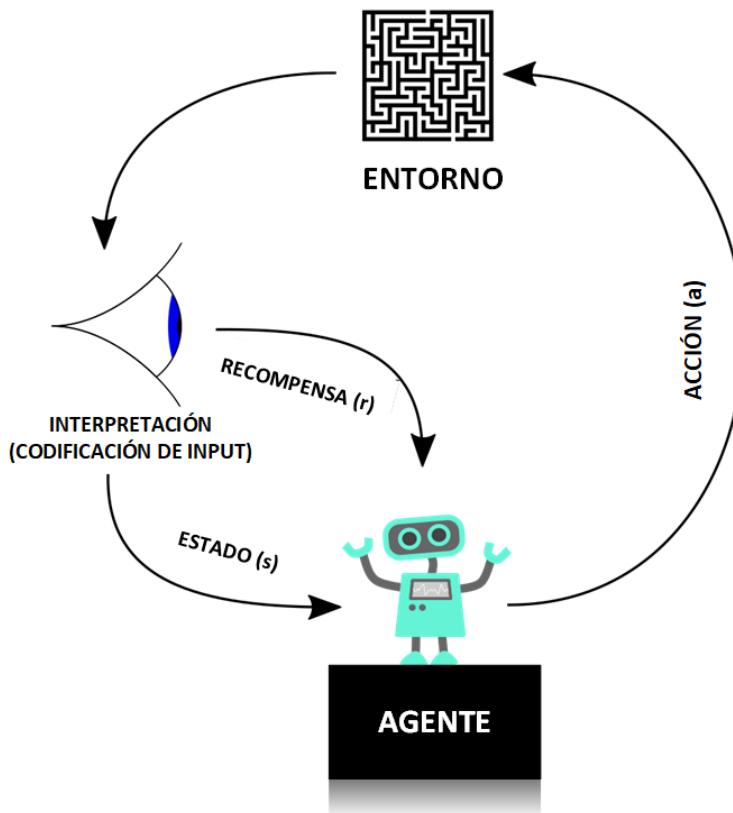


Figura 3.1: Datos a manejar en aprendizaje por refuerzo por un agente. Imagen extraída y posteriormente modificada de origen [47].

Veremos y entenderemos mejor esto cuando llevemos estas técnicas a la práctica, pero es muy importante mencionar esta gran ventaja. Una vez hemos definido una técnica de aprendizaje, puede ser utilizada casi directamente en cualquier problema para analizar los resultados que adquiere el agente. Obviamente, cada técnica podrá dar mejores o peores resultados dependiendo de los problemas en los que se apliquen claro.

Un factor que tenemos que tener muy claro es que el aprendizaje por refuerzo no está pensado para **crear una inteligencia artificial** del tipo que sea. Encaja mejor en la idea de **mejorar una inteligencia artificial** ya existente.

3.1 Episodios

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

Si lo definimos de una forma algo más formal, un episodio es una secuencia de estados del entorno, acciones realizadas y recompensas obtenidas que finalizan con un estado terminal del agente y entorno. Ya sea fracasando o teniendo éxito en su objetivo.

Estos episodios se repiten en la **fase de entrenamiento** del agente una y otra vez, tratando de recolectar experiencia de juego para mejorarse a sí mismo. La arquitectura de

este paradigma se describiría de la siguiente forma:

1. El agente realiza acciones con el entorno hasta llegar a un estado terminal, es decir, **realiza un episodio**.
2. Recolecta información(**experiencia**) durante el proceso.
3. **Entrena**. Modifica su comportamiento utilizando de algún modo concreto toda esa experiencia adquirida, dependiendo del **algoritmo de aprendizaje** utilizado. Se puede recolectar experiencia de más de un episodio.
4. **Evalúa** si ha conseguido mejorar realmente al actualizarse con esa experiencia.
5. Se **actualiza** en caso de que la evaluación determine que es mejor que antes.
6. **Repite** todo el proceso.

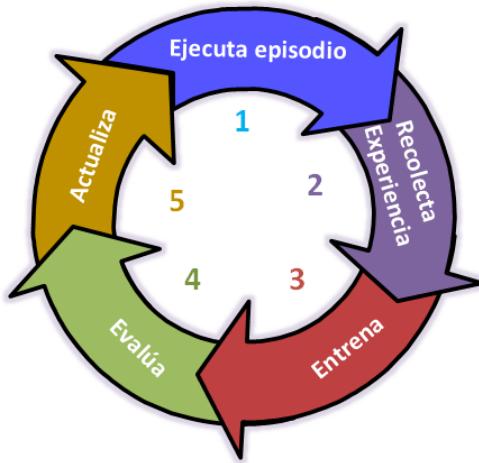


Figura 3.2: Ciclos de aprendizaje por refuerzo (aunque se podría usar más de un episodio en cada vuelta).

Una vez el agente realiza un episodio, esperamos que nuestro agente mejore. Sin embargo, nos iremos dando cuenta de que es muy fácil hacer que nuestro agente **empeore** si no tenemos cuidado.

En este campo es muy importante el paso número cuatro; **ir evaluando la fuerza del agente** para asegurarnos de que está progresando por el camino adecuado. Algunas formas de hacerlo pueden ser, dependiendo de la naturaleza del problema que tenga que solucionar:

- Cuando se trata de un entorno a resolver. Podemos definir una forma de **cuantificar el desempeño** del agente en el entorno. En caso de que consiga resolver el problema, cómo de bueno ha sido haciéndolo. En caso de que no consiga resolverlo, cómo de cerca ha estado de conseguirlo. En definitiva, un sistema que **evalúe las recompensas** que ha ido **consiguiendo** del entorno a medida que ha ido decidiendo qué acciones tomar en cada momento y que ayude a determinar cuando está mejorando. Veremos

las recompensas con mayor detalle más adelante.

- Cuando se trata de un juego competitivo, podemos **enfrentar el agente actualizado contra su anterior versión** para compararlo y ver si realmente esa actualización ha creado una versión superior de sí mismo. Esta técnica es conocida como **self-play**.

3.2 Experiencia

La experiencia que se obtiene de un episodio puede ser extraída o analizada de distintas formas. Principalmente, un agente debe decidir una acción de las posibles a realizar en el entorno en el que se encuentra, pasando a un estado nuevo tras realizarla. Por tanto, estas acciones también producirán cambios en el entorno que a su vez cambiarán los datos que percibe el agente de dicho entorno. Se necesita un *feedback* o retroalimentación que indique como de cerca está su objetivo. Entonces, debemos de ser capaces de calcular una **recompensa** del entorno en cada momento para tener una orientación y que el agente sepa estimar cuando una acción es positiva o negativa para alcanzar su objetivo.

La recompensa una vez alcanza un estado terminal está clara; positiva si ha logrado su objetivo, negativa si no lo ha conseguido. Sin embargo, no solo es conveniente saber cuando el agente consigue su objetivo o no. Si definimos en el problema algún **sistema de estimación de recompensas** por acción realizada, podremos construir una experiencia mucho más rica en el agente. Podríamos sacar información positiva o negativa de las decisiones tomadas en el transcurso de un episodio y no solo quedarnos con el desenlace del mismo. Es decir, todos los fracasos pueden tener acciones buenas y todos los aciertos pueden tener acciones malas. Si la detectamos, podemos aprender de una forma más eficiente y diversa, esa es la idea.

Hay muchas formas de estimar esa recompensa, hay que tener en cuenta que en problemas como el juego de GO, una acción puede no ser útil hasta 100 jugadas más adelante o, por el contrario, pasarnos factura en esas 100 jugadas a futuro.

Uno de los planteamientos más utilizados es la idea de **particionamiento** de la recompensa final del episodio durante todas las acciones o pasos del agente. ¿Cómo podemos particionar correctamente estas recompensas sin saber si esas influyen en el futuro como he mencionado antes? Una de las formas generales de hacerlo y que funcionan bastante bien en cualquier problema que pueda tener este tipo de dependencias a lo largo del tiempo es el **discounting**.

Esta técnica hace que la recompensa de una jugada dependa, no solo del estado actual que deja el entorno, sino de la suma de recompensas anteriores que se obtuvieron ponderadas. Se ponderan más las inmediatas a la actual y a medida que se encuentran más alejadas en términos de tiempo o número de acciones pasadas se va descontando esta ponderación para que sus influencias sean menos significativas.

3.2.1 Representación de la experiencia

La experiencia, visto de una forma más cercana a la programación, se suele almacenar en vectores o datasets. Por cada paso que el agente decide, se almacenan estos tres datos:

- Observaciones del entorno(*input*).
- Acción realizada(*output*).
- Recompensa obtenida(*reward*).

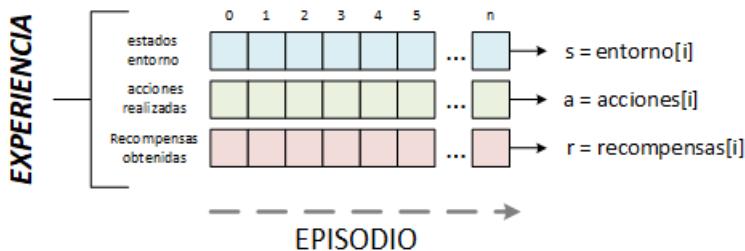


Figura 3.3: Esquema de representación de la experiencia en aprendizaje por refuerzo.

De esta manera el contenedor de experiencia irá aumentando a medida que avanza el episodio. Estos contenedores y la metodología que siguen para crearse cuando se ejecuta un agente tratan de ser **independientes** del problema que están resolviendo. En otras palabras, este sistema funciona igual independientemente del entorno o problema que se resuelve. Todos esos problemas tendrán su propia definición de inputs, outputs y recompensas y podrán ser utilizados en el momento que se decida entrenar y reforzar al agente con esa información adquirida.

3.3 Muestreo de Distribución de la Probabilidad

Es esencial que el agente **experimente** con la finalidad de que aprenda de situaciones en las que, por como funciona, nunca se vería implicado debido a las decisiones que está habituado a tomar y que sabe que normalmente funcionan.

Algunas de las soluciones para darle esa **exploración** en su árbol de decisiones puede ser que tome decisiones aleatorias en el 1% de las ocasiones, por ejemplo. Otra sería idear este **muestreo de distribución de probabilidad** de tal manera que cada acción tuviera un porcentaje de probabilidad de que el agente la ejecutara. Por tanto, esto haría que no siempre se seccionase la acción más apoyada por el agente. Sería una forma de introducir esa **aleatoriedad dirigida** al mismo tiempo que no se desestima el **criterio actual** que tiene.

Entonces, hay que procurar que el agente no sea muy **estricto** en el cálculo de estas distribuciones, ya que si una acción determinada adquiere el valor 1 y el resto de acciones el valor 0, no valdrá de nada la idea que acabamos de explicar. Esto puede ser un problema muy frecuente sobretodo en los primeros compases de aprendizaje con estas técnicas, por lo que debemos tener cuidado con esto.

3.4 Técnicas

A continuación, vamos a ver algunas de las técnicas de aprendizaje por refuerzo más utilizadas. Todo ello, desde un punto de vista teórico y conceptual.

3.4.1 Política de gradientes

Como se menciona en el apartado 3, normalmente estas técnicas se usarán para mejorar un agente ya funcional, y no para crear uno desde cero. Normalmente, estos agentes que se usan como puntos de partida suelen ser redes neuronales supervisadas que han sido entrenadas a partir de información de otros agentes resolviendo esos problemas en el mismo entorno o, aprendiendo de las decisiones que tomaron los propios humanos.

Llegados a este punto, nos encontramos ante un agente constituido por una red neuronal con unos **pesos o parámetros** explícitos en las conexiones entre capas, fruto del entrenamiento que ha tenido. Modificar estos pesos implica cambiar el comportamiento y naturaleza del agente ante el problema que resuelve, es como cambiar o modificar su cerebro e inteligencia.

Podríamos modificar estos pesos ligeramente de forma aleatoria, para ver si encontramos un agente mejor que el actual. Esto generalmente es inviable, ya que estas técnicas se suelen utilizar en problemas demasiado complejos y amplios como para depender de la suerte en encontrar una buena combinación de pesos.

En su lugar, podríamos analizar la experiencia que obtiene el agente al desenvolverse en el entorno con la finalidad de analizar y determinar cuales son los pesos que deberíamos modificar. Esta es la idea que trata de plasmar la **política de gradientes**. A rasgos generales, su funcionamiento es simple. Una vez el agente termina un episodio:

- Modifica los pesos de tal manera que **baja** la probabilidad de las jugadas que ha realizado a lo largo de la partida, si ha llegado a un estado terminal de **fracaso**.
- Modifica los pesos de tal manera que **sube** la probabilidad de las jugadas que ha realizado a lo largo de la partida, si ha llegado a un estado terminal de **éxito**.

Obviamente, una misma acción puede ser buena o mala dependiendo de la **situación** en la que nos encontremos. Cuando decimos que modificamos la probabilidad de que el agente realice una acción, nos referimos a que realice esa acción en situaciones iguales o parecidas en las que se encontraba en ese episodio.

Modificamos entonces, los pesos del agente para que tenga ese comportamiento a la hora de estimar las probabilidades, no las probabilidades en sí de forma manual, ya que no tendrá sentido al solo tener efecto cuando se encontrara en exactamente la misma situación.

La forma en la que el agente entiende una *situación similar* viene intrínseca en las redes neuronales y en como abstrae conocimientos y patrones del entorno, cosa que como ya mencionamos en el apartado 1, no podemos explicar de forma clara al tener un conjunto de pesos enormes.

No es una ciencia exacta, quiero decir, que haya tenido éxito en su objetivo final no quiere decir que todas las decisiones que ha tomado sean buenas. Ni que, en caso de no alcanzar el objetivo, todas las decisiones que ha tomado sean malas. Pero si es cierto que al hacer esto generalmente vamos a mejorar más de lo que empeoramos.

Igualmente, este es el concepto base. Organizaciones de investigación punteras como

DeepMind están constantemente trabajando sobre estas arquitecturas tratando de que sean más ricas y más acertadas. Por ejemplo introduciendo el concepto de discounting que mencioné en el apartado 3.2, entre muchas otras.

Una de las formas de mejorar más importantes que se han descubierto en este ámbito es la **asignación de crédito**. Consiste en detectar de algún modo las **decisiones más relevantes** de cada episodio y utilizar esas para optimizar los pesos del agente. Es uno de los problemas más relevantes del aprendizaje por refuerzo, ya que cuanto mejor sea la estrategia a seguir, mejor aprenderá el agente de su propia experiencia.

Para actualizar estos pesos utiliza los **gradientes descendentes** de una forma muy similar a lo que explicamos resumidamente en el apartado 2.4. Solo que en lugar de utilizar una etiqueta de aprendizaje supervisado para calcular la función de perdida, usamos la recompensa que se obtiene en la experiencia y optimizamos los pesos con esos valores.

3.5 Q-Learning

El **Q-Learning** o **aprendizaje por refuerzo con método de valores** toma un enfoque distinto al de política de gradientes explicado hace un momento. Imaginemos lo siguiente, cuando un comentarista de una partida de ajedrez dice la frase “las blancas van algo mejor que las negras”, no es un concepto preciso. El comentarista, a través de toda la experiencia que él mismo tiene sobre ese entorno, puede hacer ciertas predicciones de quién está mejor colocado en el tablero y, por tanto, estimar que las blancas tienen más probabilidades de ganar que las negras en ese momento(de cumplir su objetivo en el entorno).

Ese es el enfoque de esta técnica. El agente tiene la capacidad de **estimar** cuánta recompensa espera conseguir en el futuro a partir del estado actual del entorno. Ya no usaríamos una metodología de asignación de recompensas estática, como el método de **discounting** mencionado en el apartado 3.2, sino que en su lugar tendríamos un agente que utiliza esa misma experiencia para hacer su propia asignación de recompensas a sus propias acciones, y cada vez lo haría mejor.

La función que nos devuelve esta recompensa estimada se define matemáticamente como $Q(s, a)$, de ahí su nombre de *Q-learning*. La s representa el estado del entorno actual y la a representa la acción que se está considerando para realizar sobre s .

Para definir esta función Q se hace uso de las **redes neuronales (Deep Q-learning)**. El comienzo es el mismo que en la política de gradientes. El agente recolecta los estados, acciones y recompensa de los episodios(experiencia). La diferencia se produce llegado a este punto. La forma en la que esta metodología actualiza el comportamiento del agente a partir de la experiencia recolectada consiste en sustituir la política de gradientes por esta función y probar todas las acciones posibles (dominio de a) en el estado actual del entorno (s). Se realizará la acción que haya devuelto un mejor resultado en la función.

Pero recordemos que el aprendizaje por refuerzo premia mucho la **exploración**, por lo que no está mal que el agente de forma ocasional realice jugadas que el no haría normalmente, como mencionamos en el apartado 3.3 para que él mismo experimente qué pasaría si lo hace y aprenda de nuevas acciones buenas o acciones que son malas.

3.6 Método actor-critic

El aprendizaje por refuerzo actor-critic, o actor-crítico traducido al español, maneja una morfología de aprendizaje que los humanos utilizamos en muchas ocasiones a lo largo de nuestra vida sin darnos cuenta. Con ella, aprendemos y mejoramos en muchos aspectos.

Solemos supervisar, evaluar y, sobretodo, escuchar a individuos que son mejores que nosotros haciendo o desenvolviéndose en algo específico. Esta técnica trata de mezclar los puntos fuertes tanto de la **política de gradientes**(apartado 3.4.1) como del método **Q-Learning** (apartado 3.5):

- La política de gradientes hace de **actor**, ya que dado un estado del entorno decide la distribución de probabilidades de las distintas acciones que puede realizar para saber cuales de ellas son mejores y, por tanto, más seguras de que realice.
- La función de valor Q hace de **crítico**. Decide cuando el actor mencionado en el punto anterior está tomando buenas o malas decisiones y, por consiguiente, cuando está más cerca de cumplir su objetivo o cuando se está alejando del mismo respectivamente.

La información que aporta el crítico es fundamental para el proceso de entrenamiento del agente, del mismo modo que los consejos de un profesor podrían guiar nuestro propio estudio en algo y hacerlo más eficiente.

Digamos que, por un lado, el actor aprende a realizar aquellas acciones que el crítico indica que son más prometedoras. Por otro lado, el crítico va mejorando también a la hora de estimar esas recompensas de las acciones que realiza.

Un concepto esencial que debemos conocer cuando estemos hablando de esta técnica es la **ventaja**. Explicaremos a continuación cual es su significado y el por qué de su relevancia.

3.6.1 Ventaja

Ya vimos lo que era la **asignación de crédito** en el apartado 3.4.1, el cual era un problema a resolver y mejorar fundamental para que estas técnicas de aprendizaje por refuerzo fueran cada vez mejores y más eficientes utilizando la experiencia recolectada.

Cuando el agente consigue su objetivo. ¿Reforzamos todas las decisiones que ha tomado? Esto no es una buena idea. Si se tratase de un juego en el que nos enfrentamos a un rival, como GO, podría darse el caso de que nuestro contrincante no sea muy habilidoso y no haya sabido aprovechar las malas jugadas que ha realizado el agente. Mientras que nuestro entrenamiento está reforzando esas jugadas indiscriminadamente.

Entendemos la **ventaja** como una fórmula para estimar la relevancia de una decisión(acción) en el resultado final conseguido. Para entender cómo podría estimarse esa relevancia, una vez más, nos fijamos en como lo hacemos nosotros mismos, como si nos mirásemos en un espejo.

Imagina una partida de baloncesto. Un triple en el cuarto tiempo cuando el resultado es 78-80 supone un enorme cambio en el posible resultado final del partido. Sin embargo, si el resultado ya es 110-80, el triple carece de interés para los espectadores y muy difícilmente

les emocione a pesar de que la ejecución es la misma.

Este es el concepto principal en el cual gira esta técnica de estimación. Parece ser que las acciones más relevantes tienden a suceder en los momentos en los que el resultado final aún **permanece en duda**. La ventaja trata de desarrollar este concepto para el beneficio del propio agente que construimos.

Primero, necesitamos una función que defina el **valor de un estado**. Es decir, cómo de favorable o desfavorable es la situación del entorno para el cumplimiento del objetivo del agente. Se denominará con $V(s)$. Es parecido al Q-learning explicado en el apartado 3.5, pero se limita a evaluar el estado del tablero(s), no la relevancia de una acción en ese estado(no se tiene en cuenta a).

Entonces, introducimos este nuevo concepto junto con el Q-learning , de tal modo que la ventaja sería :

$$A = Q(s, a) - V(s)$$

El problema de esto es definir Q , ya que necesita de una red neuronal aparte y entrenarla para que estime el valor de las acciones en el entorno. Lo cual puede ser muy costoso y un problema añadido. Una posible solución podría sustituir la estimación Q por la recompensa final del entorno una vez finaliza el episodio:

$$A = R - V(s)$$

Lo que está haciendo el agente es calcular al mismo tiempo tanto su **función de estimación del valor** como la **función de su política**.

- Al principio de la partida, $V(s)$ normalmente valdrá 0. Esto es lógico, ya que un estado inicial del entorno no será más o menos favorable para el agente. Si al final cumple su objetivo, R tendrá un valor de 1(positivo), por lo que la ventaja de la primera acción será $1 - 0 = 1$.
- Imaginemos que el entorno está a punto de llegar a un estado terminal y que el agente las tiene todas consigo para cumplir su objetivo, por lo que $V(s)$ tiene un valor de 0,95. Si, en efecto, el agente gana la partida, la ventaja de este movimiento será $1 - 0,95 = 0,05$.
- Imagina la misma situación pero, en lugar de cumplir el objetivo, el agente acaba fracasando. Entonces, la recompensa final sería de -1(negativa). En ese caso la ventaja adquiere un valor de $-1 - 0,95 = -1,95$.

En los primeros movimientos al aún no haberse decidido nada, la ventaja tiende a mantenerse neutra. Sin embargo, cuando el agente tiene una situación ventajosa y mete la pata o, por el contrario, si el agente tiene ya muy pocas posibilidades de alcanzar su objetivo y consigue revertir la situación, la ventaja se dispara de forma negativa y positiva respectivamente.

En definitiva, hay que prestar atención en esos **cambios bruscos** del valor de la ventaja que se pueden producir en el transcurso de cada decisión que toma el agente. Son en esas decisiones en las que debemos tener mayor consideración a la hora de establecer una guía para la actualizaciones de los pesos de nuestro agente. Cuanto más **bruscos** sea el cambio de la ventaja, más relevantes serán esas decisiones, ya sea positiva o negativamente.



Figura 3.4: La ventaja se incluye en la experiencia, pero es introducida al final del episodio, cuando ya se sabe todas las recompensas.

Desarrollo y Experimentación

4	Aprendizaje por refuerzo profundo	45
4.1	Algoritmos	
5	Entornos Open AI Gym	51
5.1	MountainCar-v0	
5.2	MountainCarContinuous-v0	
5.3	Entorno más complejo (aún sin decidir)	
6	Entorno de desarrollo	55
6.1	Google Cloud Platform	
6.2	Configuración de la Infraestructura	
6.3	Configuración de la Máquina	
6.4	Uso de la Máquina Virtual	
6.5	Forma de trabajar	
6.6	Archivos generados en los logs	
7	Experimentación: Resultados Obtenidos	71
7.1	Visualizar la información de los logs	
7.2	MountainCar-v0	
7.3	Otro entorno más complejo (Aún sin decidir)	

Introducción

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

Las explicaciones serán realizadas a partir de entornos y técnicas **específicas** de tal modo que, probablemente, podremos entender detalles y puntos de vista concretos que, de otro modo, sería más complicado plasmar.

En primer lugar, explicaremos la fusión de las técnicas de aprendizaje por refuerzo y las redes neuronales vista anteriormente; el **aprendizaje por refuerzo profundo**. Utilizaremos para ello las librerías de **OpenAI** para aplicar esos algoritmos en **Python** y algunos entornos de **Gym** en los que le daremos la posibilidad a nuestros agentes de “ensayar” su tareas.

Por experiencia, sé que entrenar redes neuronales es una cosa computacionalmente costosa en términos generales, aunque obviamente dependa del problema que estamos tratando de resolver, sin contar con el plus añadido de las técnicas de aprendizaje por refuerzo que utilizaremos. Por ello, mi tutor Juan Y yo, decidimos que era mejor a largo plazo configurar 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.



4. Aprendizaje por refuerzo profundo

Ya explicamos lo que significa este concepto, recordad la figura 1.2. Sin embargo, no se mencionó nada de las técnicas más relevantes que utilizan. De las formas posibles que pueden ser usadas las redes neuronales (apartado 2) con aprendizaje por refuerzo (apartado 3) para conseguir estos resultados tan buenos y novedosos de los que se hablan.

Estas técnicas 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. [30].

Podemos ver los avances que han obtenido en su página web [36], siendo en general sobre inteligencia artificial, pero sobretodo relacionada con aprendizaje por refuerzo y redes neuronales.

4.1 Algoritmos

Los algoritmos que hemos utilizado han sido extraídos y analizados de **OpenAI baselines** en Github. Es un conjunto de implementaciones de alta calidad de algoritmos de aprendizaje por refuerzo y por refuerzo profundo. [35]

Estos algoritmos, a disposición del público, hacen que sea más fácil para la comunidad de investigadores replicar, refinar e identificar nuevas ideas, creando la base para trabajar con ellas. Incluso siendo utilizadas como método de comparación con otras técnicas de aprendizaje mas tradicionales. Tratan de situarse a la par con los trabajos más punteros en la actualidad. Sin embargo, tal y como se puede apreciar en su repositorio, tienen un constante mantenimiento tanto para mejorar como para solucionar ciertos errores que van surgiendo con su desarrollo. Por lo que, en ocasiones, nos podemos encontrar ante técnicas o

entornos de uso que no terminan de ser estables del todo, ya hablaremos de ello más adelante.

Son herramientas las cuales también se favorecen de las aportaciones de la comunidad, estando forkeada unas 3300 veces, por lo que a veces podemos encontrarnos ciertos problemas a la hora de utilizarlos y tendremos que solventarlos para conseguir el objetivo que buscamos. Conforme nos encontremos en el contexto adecuado explicaré mi experiencia personal de uso y como solventé los problemas que a mí personalmente me fueron surgiendo.

4.1.1 PPO2

Es el acrónimo de **Proximal Policy Optimization**, junto con el número que denota una reimplementación del mismo ante su versión original en baselines.

Proponen una nueva familia de métodos de **gradiente de políticas** 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 **ascenso de gradiente estocástico**.

Por consiguiente, sería un algoritmo desarrollado a partir de los apartados 3.4.1 y 2, junto con una de las optimizaciones vistas en 2.4.

Mientras que los métodos estándar de gradiente de políticas realizan una actualización de los pesos en la red neuronal por muestra de datos, se propone una **función objetivo más novedosa** que organizan esas actualizaciones y múltiples épocas en mini-batch o mini-lotes.

Como ya mencionamos, esta estrategia hace que tengamos un buen equilibrio entre calidad de las actualizaciones de sus parámetros con el tiempo de aprendizaje. Supera a otros métodos de gradiente de políticas en línea, 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 pasos o actualizaciones que debe realizar. Demasiado pequeño, y el aprendizaje es inconvenientemente lento. Demasiado grande, y la señal se ve demasiado afectada por el ruido(exploración).

Con estos problemas podríamos darnos de narices ante una caída catastrófica y repentina en la eficacia del agente. Como inconveniente general, necesitan millones o miles de millones de actualizaciones para aprender tareas simples, lo que supone recabar mucha experiencia; ensayos y errores. Por lo que estas técnicas no pueden ser utilizadas para cualquier tipo de problema, es dependiente de su complejidad.

Los investigadores han tratado de compensar estos defectos con algoritmos como *ACER* o *TRPO*(también disponibles en los baselines de Github) intentando restringir u optimizar de alguna manera el tamaño de las actualizaciones de política(pesos).

No obstante, *ACER* es mucho más complicado que *PPO2*, cuando solo ha sido demostrado funcionar mejor con *Atari*. Por otro lado, *TRPO* no es fácilmente compatible con

algoritmos que comparten parámetros entre una política y una función de valor o pérdidas auxiliares. Es decir, método *actor-critic* visto en el apartado 3.6.

Una diferencia importante entre PPO y PPO2 en los baselines de OpenAI, es que el segundo incorpora una implementación de **activación de GPU** que nos es muy útil dado nuestro entorno de trabajo que explicaremos en el apartado 6.2. Esto hace que el aprendizaje se produzca unas **tres veces más rápido** que en el PPO estándar, siempre que tengamos los recursos para poder aprovecharlo claro.

Además, esta versión incluye una variante de la función objetivo que es muy novedosa debido a que no se suele encontrar en otros algoritmos. Implementa una forma de actualizar la política compatible con el gradiente descendente estocástico, mientras que simplifica el algoritmo eliminando la perdida de KL, la cual tiene propagación hacia atrás y es costosa, y la necesidad de realizar actualizaciones adaptativas.

Esto hace que tenga un buen rendimiento y muy parecido a ACER en Atari, a pesar de ser mucho más sencillo que éste. Podemos ver algunos videos de ejemplo funcionando para problemas de control continuo, como hacer caminar un robot en un entorno simulado, en las referencias utilizadas. [29] [18]

4.1.2 DQN

Este algoritmo es el resultado de utilizar la técnica de **Q-Learning** explicada en el apartado 3.5 implementada en una **red neuronal profunda**, apartado 2.

Esto permite que esa técnica Q-Learning, perteneciente al aprendizaje por refuerzo, pueda ser viable y funcionar bien en entornos más complejos, de mayor dimensionalidad, como podrían ser los videojuegos o robótica.

OpenAI ha hecho un gran trabajo con este algoritmo, ya que tiene un **sistema de repetición**. Si no he entendido mal su funcionamiento, consiste en revisar la experiencia junto con las recompensas estimadas por DQN, de tal manera que puede comprobar donde la recompensa real difiere de la estimada de forma significativa, permitiendo al agente reajustarse en función de suposiciones realizadas de forma errónea por su parte. En otras palabras, es una manera de **reproducir sus recuerdos** para aprender de ellos una vez se han producido las consecuencias a posteriori.

Al combinar Q-learning con una red neuronal, siendo un trabajo concreto de ingeniería por parte de OpenAI, esta técnica presenta alguna diferencia de las vistas anteriormente en este trabajo por separado. Explicado de una forma esquematizada; divide la red neuronal en dos partes:

1. Una parte aprende a proporcionar una **estimación del valor** a cada paso del agente, define como de buena es su situación en ese momento.
2. La otra calcula las **ventajas potenciales de cada acción** realizada, sería la parte más familiar por nosotros comentada hasta el momento.

Ambas partes se **combinan** de una forma concreta, de tal manera que obtenemos un **único valor** como salida de la red. [28]

4.1.3 DDPG

Esta es la adaptación de la técnica de **actor-critic** vista en el apartado 3.6. Aquí el dominio de acciones a realizar por el agente debe ser **continuo** y no discreto, por la naturaleza del propio algoritmo. Esto nos dará problemas en el futuro que deberemos resolver, lo veremos en un momento más apropiado a lo largo de esta documentación.

Un concepto que debemos tener muy en cuenta para esta técnica será el **parámetro de ruido**. Esto hace referencia al concepto de introducir algo de **aleatoriedad** al agente como ya explicamos en el apartado 3.3

Cuando mencionamos esa distribución de probabilidades entre las distintas posibles decisiones o acciones del agente, en realidad mencionamos la forma “tradicional” de hacerlo, ya que introducimos el ruido directamente en el espacio de acciones del agente para conseguir esa **aleatoriedad dirigida** que mencionábamos.

Este algoritmo introduce el ruido de una forma **más novedosa**, ya que **modifica de forma directa** los parámetros de la red neuronal sobre la que se monta DDPG.

Esto altera de forma leve la política del agente, y por tanto, las decisiones que tomará, pero solo dependiendo de la situación actual en la que se encuentre (input en ese momento). Recordemos que esta técnica se combina con Q-Learning que estima como de prometedor es el futuro del agente, y ahí no estamos introduciendo ruido.

En resumen, introducimos **aleatoriedad** a los pesos de las conexiones que tiene la red neuronal. Los pesos están fijados por la técnica de entrenamiento a la que ha estado sujeta el agente hasta ese momento, pero esos pesos pueden ser modificados de forma leve a la hora de utilizar dicha red para tomar una decisión.

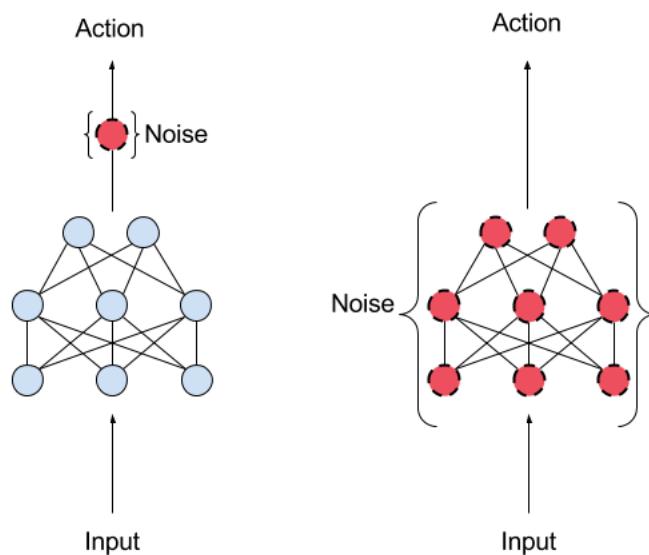


Figura 4.1: Ruido en el espacio de acciones (izquierda) frente a ruido en los parámetros de la red (derecha). Extraída de origen. [27]

Esta técnica ha demostrado mejorar la **exploración** de los agentes obteniendo final-

mente mejores resultados y comportamiento más elegantes, entendiendo por elegantes a decisiones o estrategias más parecidas al comportamiento humano.

Se cree que introducir ruido en los parámetros provoca que la exploración sea consistente a través del paso del tiempo, mientras que agregar el ruido en el espacio de acción conduce a una exploración **más impredecible** que no está correlacionada con los parámetros (inteligencia) del propio agente.

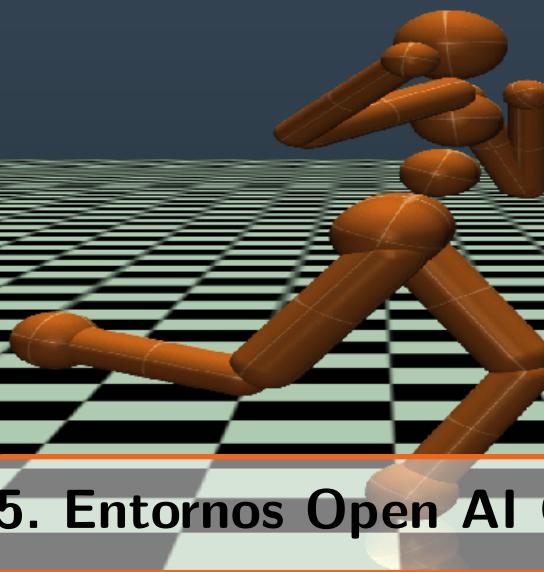
la idea está clara, ¿pero cuánto y cómo modificamos esos pesos de la red exactamente? Los investigadores tienen que enfrentarse a tres problemas principales:

1. Diferentes niveles de la red neuronal tienen **diferentes sensibilidades**.
2. La sensibilidad de los pesos de la política también puede variar dependiendo del **progreso del aprendizaje** por el que se encuentre el agente, siendo más difícil predecir las repercusiones que tendrán cambiar sus pesos.
3. Escoger la correcta escala de ruido es bastante complicado, ya que entender cómo influye en las decisiones del agente la modificación de esos parámetros es una tarea prácticamente imposible. Recordemos que una de las características de las redes neuronales es que **no podemos entender como toman sus decisiones** (apartado 1)

Se usa normalización en las capas para solventar el primer problema, situando a todos los parámetros en una escala unificada. En cuanto al segundo y tercer problema, se combate utilizando una **planificación adaptativa**.

Consiste en determinar el rango modificable de los parámetros en función de su espacio de valores. Funciona midiendo la perturbación en la toma de decisiones del agente dado los parámetros y por tanto, definiendo de una forma automática cómo pueden ser variados.

Como conclusiones extraídas sobre el *ruido*, considero que esta técnica **da mejores resultados** aunque es **más difícil** de afinar dado un problema. Cuando introducimos el ruido en el espacio de acciones, da peores resultados en general, pero es más simple de introducir y más interpretable dado que trabajamos directamente sobre las acciones que toma el agente. [46] [27]



5. Entornos Open AI Gym

Hemos explicado las técnicas que vamos a utilizar en el desarrollo de este proyecto. Lo que no hemos comentado aún son los **entornos** o problemas que vamos a resolver con este tipo de tecnología.

Podríamos utilizar estos algoritmos para resolver **problemas del mundo real**, del tipo que fuesen. Por ejemplo, hacer que un robot aprendiera a mantener el equilibrio a base de caerse. El problema principal de esto es el **presupuesto**. ¿Qué precio puede tener comprar o desarrollar un robot humanoide completo que permita todo tipo de movimientos, sensores para recoger información, etc?

Es algo que no es nada barato, menos para un estudiante normal y corriente. A esto tenemos que añadirle que el aprendizaje por refuerzo requeriría que el robot se cayese, por pura naturaleza del método de aprender. Esto deriva en que el robot probablemente se rompería y habría que repararlo, en el mejor de los casos, ya que si en vez de este ejemplo usáramos uno de aprender a conducir, que se rompiera el vehículo sería el menor de los problemas.

Por suerte, existe una solución para investigadores sin blanca y peatones inocentes en igual medida. La respuesta son los **entornos simulados**. Simular ese mismo entorno en un **mundo virtual** y entrenar esa inteligencia desde ahí. 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.

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. [31] [37]

5.1 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*).

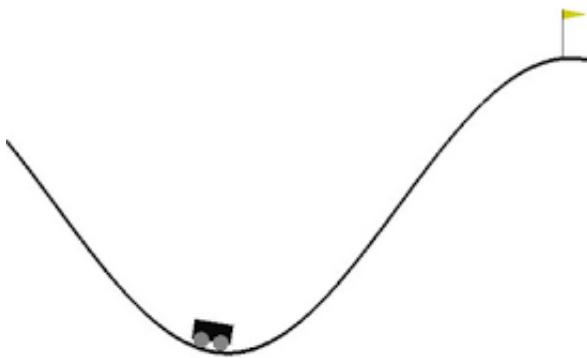


Figura 5.1: Representación gráfica del problema MountainCar v0 de OpenAI Gym.

Podemos ver los detalles técnicos en el repositorio oficial de OpenAi para Gym en Github [38]. 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 5.1 respectivamente.

- **velocity**: alcanza valores comprendidos entre $-0,07$ y $0,07$. Representa la velocidad a la que está yendo la carreta en ese instante de 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 instante 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 que el agente realice del entorno la carretilla 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 recompensa que el agente puede extraer del entorno podría ser pensada de diferentes formas. En este caso, Gym ha decidido que la recompensa para su entorno sea -1 en cada instante de tiempo (step) si no se encuentra en la meta y 1 en caso de alcanzarla.

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

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

La carreta puede tardar más o menos pasos en conseguir su objetivo ¿Cuando se considera un resultado razonable? La suma de recompensas desde el estado inicial hasta el estado final del entorno siempre será 1 menos el número de pasos realizados por la carreta. Ya que cada paso resta 1 y solo suma 1 una vez alcanza la meta.

A partir de ésto, es fácil llegar a la conclusión de que el entorno alcanza una recompensa total de -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 en 100 episodios.[32]

Creo que éste es un buen momento para destacar la **independencia** de los algoritmos de *baselines* con los entornos *Gym*. Nos damos cuenta de que siempre se va a trabajar sobre los mismos elementos: los datos del input, los posibles outputs y la recompensas que tenga el entorno en el que trabajemos. Los algoritmos de aprendizaje por refuerzo profundo siempre se centrarán en recopilar experiencia a partir de estos datos (que serán diferentes

dependiendo del entorno) y construir la **experiencia** del agente (ver figura 3.3) para su posterior procesamiento dependiendo del algoritmo que se trate.

El **uso genérico** de estos algoritmos para los entornos que cumplan con esta recopilación de experiencia es uno de los puntos fuertes de esta tecnología. Obviamente esos algoritmos luego pueden ajustarse y/o parametrizarse de una forma más específica para que funcione mejor en el entorno concreto que nos encontramos, pero no es necesario para que funcione como tal.

5.2 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 que son 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 las **recompensas**, siendo la recompensa final el valor 100 menos la suma cuadrática de los *timesteps* sucedidos hasta alcanzar la meta. Añadir un número máximo de *timesteps* 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(apartado 4.1.3), por lo que esta adaptación era necesaria.[34]

5.3 Entorno más complejo (aun sin decidir)



Google Cloud

6. Entorno de desarrollo

Como ya mencionamos en el apartado III, 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**, concretamente de **Google Cloud** para montar una máquina virtual que pudieramos utilizar para entrenar y guardar agentes en los entornos explicados.

6.1 Google Cloud Platform

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 a lo largo de este proyecto.

Antes de comenzar, comentar que me llevó bastante tiempo. 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 que me fui encontrando hasta ser capaz de realizar pruebas correctamente. Además, incluiré las referencias de aquellos artículos que me ayudaron a conseguir mi objetivo, por supuesto.

6.2 Configuración de la Infraestructura

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

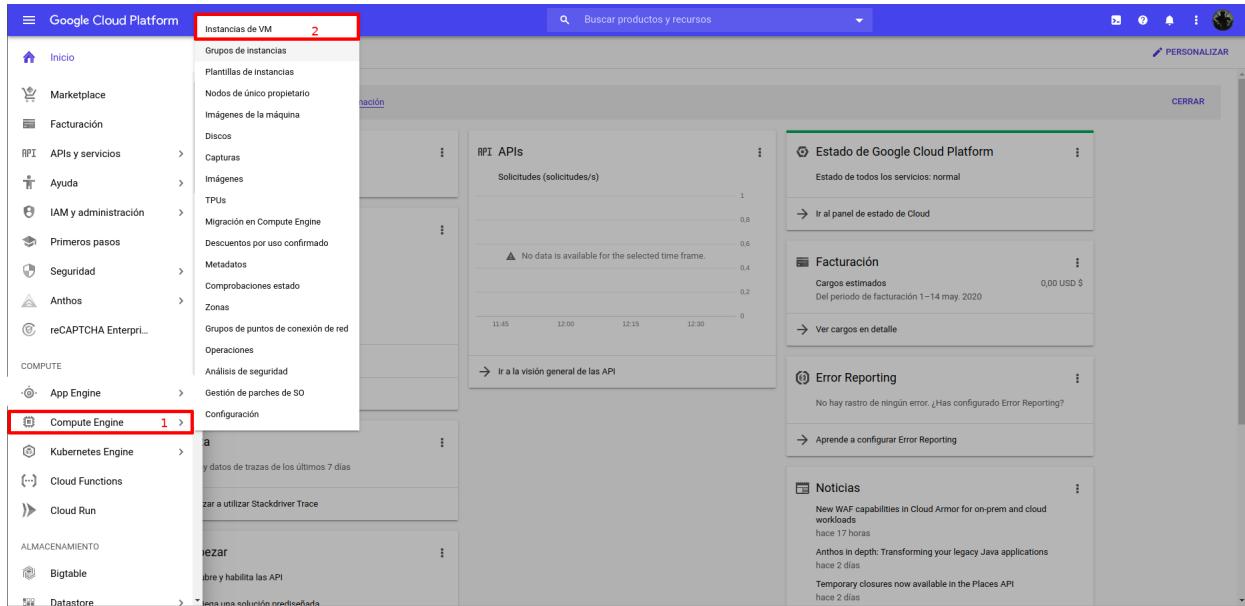


Figura 6.1: Iniciando asistente para creación de máquina virtual.

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.

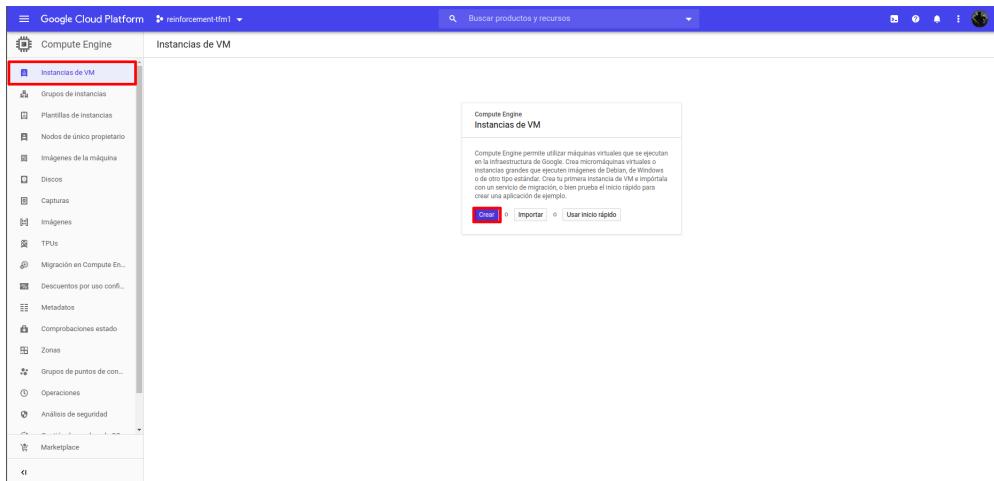


Figura 6.2: Creando una máquina virtual.

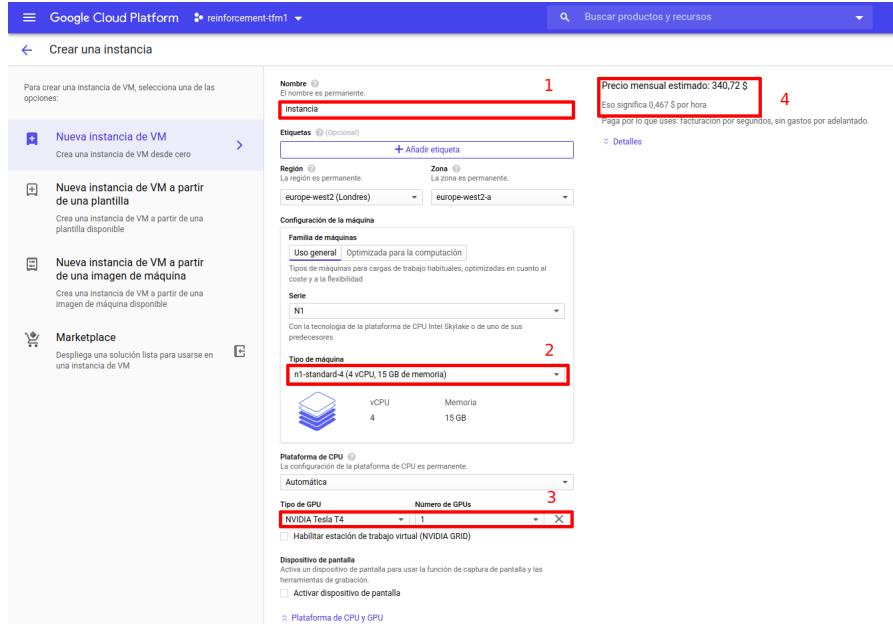


Figura 6.3: Configurando la máquina virtual para el proyecto.

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 que puede definir una mejor conexión con la máquina que vamos a crear ya que dependiendo de la zona podemos tener una latencia y catálogo de recursos a configurar diferentes.

Para tener en cuenta la latencia encontré una página web que indica, desde el lugar que estamos conectados, las distintas latencias que tenemos con cada una de las regiones existentes:

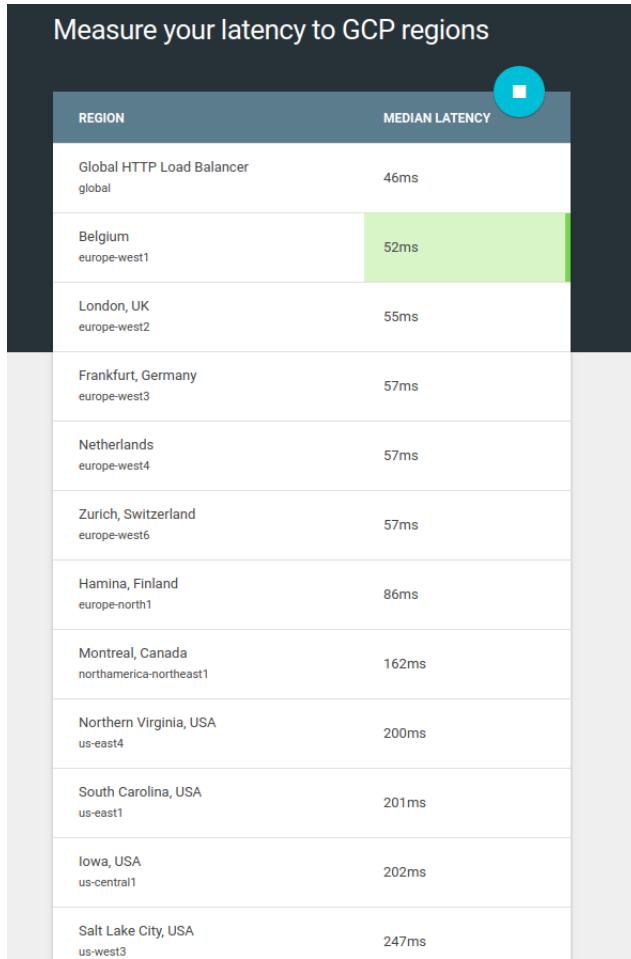


Figura 6.4: Consultando latencias de las distintas regiones de Google. Obtenidas de su página web [11]

Como podemos observar en la captura realizada en la figura 6.4, la región con mejor latencia es la de **Bélgica**. Sin embargo, finalmente decidí utilizar la región de **Londres** tal y como se puede ver en la figura 6.3. Esto se debe a que con la de Bélgica la mayoría de veces tenía problemas de disponibilidad para la GPU e incluso, en algunas ocasiones, me daba problemas una vez la máquina comenzaba a funcionar. Igual para cuando este proyecto esté terminado, ese inconveniente no existe. No conozco la administración de esos recursos internamente por parte de Google.

En definitiva, busqué un **equilibrio** entre buena conexión y buen abastecimiento de recursos. Luego, usé 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, considero que con esas 4 unidades tengo suficiente, 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 6.3, tenemos una **Nvidia Tesla T4**. En un principio esto también me daba error, 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.^[43]

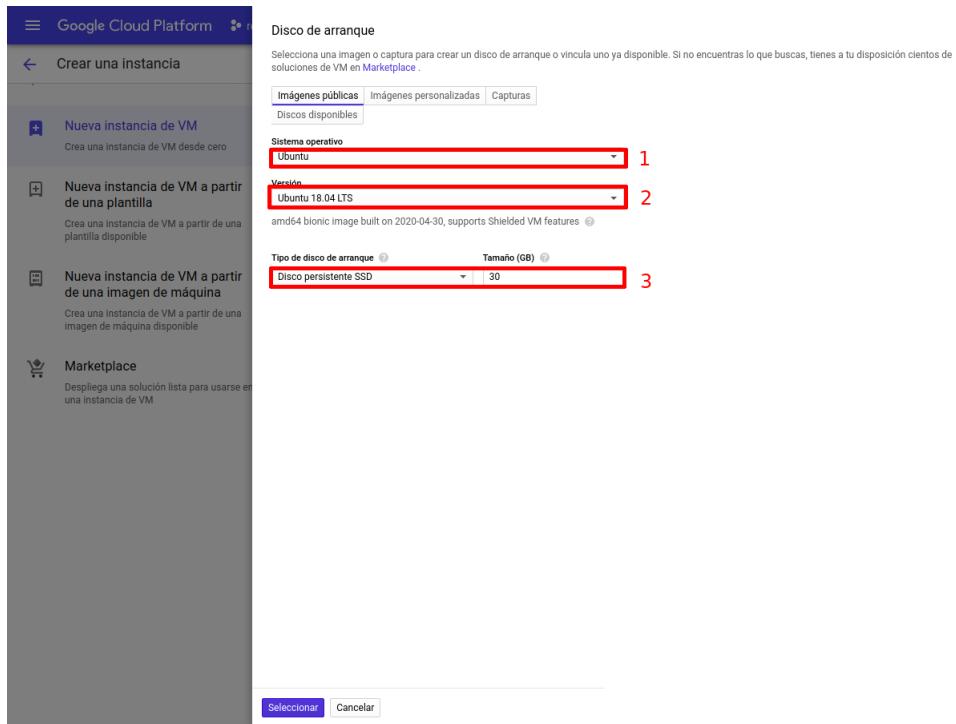


Figura 6.5: Indicando Sistema operativo y memoria SSD a la máquina virtual.

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 realicé la configuración en local, lo hice 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 decidí utilizar la misma imagen, además de añadirle **30 GB de SSD** para mejorar el rendimiento de la máquina. Intenté hacerlo con esa misma versión pero la instalación mínima, daba una gran cantidad de errores, sin saber muy bien el motivo.

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

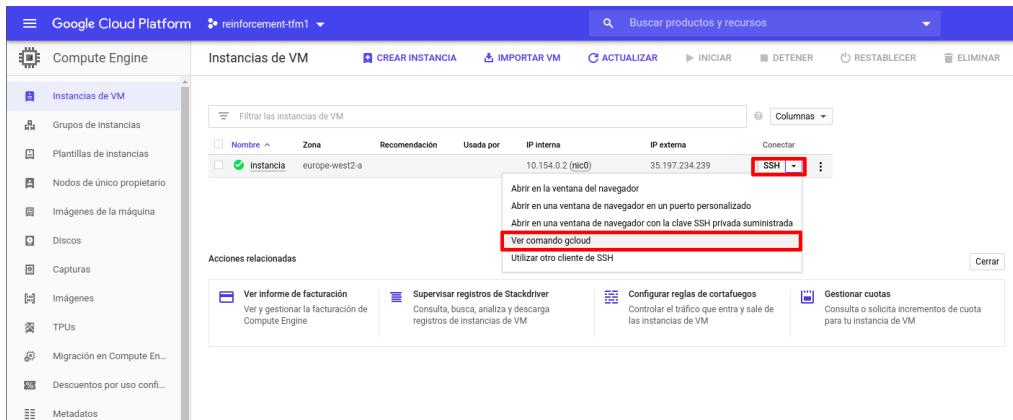


Figura 6.6: Comprobando que la máquina virtual ha sido creada con éxito.

6.3 Configuración de la Máquina

Hasta ahora, hemos estado detallando la configuración de infraestructura en la nube para la creación de la máquina virtual. A partir de ahora, vamos a explicar la **configuración** de esa máquina para poder hacer los experimentos con las distintas técnicas de aprendizaje por refuerzo profundo 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 [14]. Simplemente nos logeamos desde esos comandos y así podemos utilizar directamente el comando de conexión que aparece en la opción de la figura 6.6.

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 6.7: 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 **abastecimiento** 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** [15], 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, considero que no es necesario para nuestro caso, ya que vamos a contar con una única máquina y que, en cualquier caso, necesitaría un escalado vertical y no horizontal si llegase ese momento. Por ello, opté directamente por un método más “tradicional” aunque efectivo para nuestras necesidades; los scripts del bash de Ubuntu.

Simplemente concatené todos los comandos necesarios para instarlo todo en un script , el cual posteriormente enviaría al servidor y ejecutaría desde allí. Debo reconocer que he tenido muchos problemas a la hora de configurar la máquina, por lo que ese script fue actualizándose hasta que conseguía instalarlo todo de tal forma que me funcionase, este es el resultado final:

```
#!/bin/bash

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

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

sh Anaconda3-2020.02-Linux-x86_64.sh

source .bashrc

conda update -n base conda

conda update anaconda

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

#Entramos en dicho entorno
conda activate TFM

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

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

#instalamos pip
sudo apt install -y python3-pip
```

```
#instalamos git
sudo apt install -y git

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

cd baselines

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

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

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

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

pip install pytest

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

En el script anterior hacemos uso de un montón de cosas que vamos a explicar a continuación. 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 mi caso, he utilizado **Anaconda** [2]. Tuve que realizar un montón de intentos y pruebas hasta que conseguí crear una configuración que funcionase. No estaba seguro si en otros entornos futuros que pudiera probar iba a funcionar esa misma configuración. Por esa misma razón, decidí tener este **control de versiones** de Python y paquetes instalados. En caso de que sea necesario tener una configuración distinta para dos problemas distintos a resolver, o la interfaz del entorno simulado Gym necesite unas librerías diferentes a las de otra.

Busque el enlace de descarga en su página oficial y cree la petición del archivo con el comando *wget*, lo siguientes comandos son simplemente para actualizar conda y crear el entorno con una versión de Python 3.6, que fue la que me funcionó para mí.

Recordemos que configuramos una tarjeta gráfica bastante potente para la máquina (apartado 6.2). Por defecto, las librerías de *baselines* no van a ser capaces de reconocer esa GPU, aunque sean específicas para uso de la misma. Es necesario instalar los **controladores** para esa tarjeta concreta, cosa que me di cuenta tras hacer varios intentos y dar numerosas vueltas por Google, pensando que era un problema de los propios *baselines* y *Tensorflow*(que es la librería que utiliza los baselines para las redes neuronales profundas)[42].

Antes de instalar los *baselines* es importante instalar *TensorFlow*, como mencionamos hace un momento. Es necesario instalar la versión para GPU, de lo contrario no la utilizará aunque hayamos instalado los controladores. Solo he conseguido que funcione correctamente para la versión 1.14 (la versión que aparece en la guía de OpenAI[35]). Hay algunas guías con otras versiones, en mi caso no han funcionado, desconozco los motivos exactos.

Clonamos el repositorio Github de donde extraemos los *baselines* de *OpenAI* [35]. 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, creo que se trata de un error que tienen que solucionar.

Hacemos la instalación de *stable-baselines*[12] para poder usarlo concretamente en el algoritmo DDPG por un problema encontrado que explicaremos en la fase de experimentación (ver apartado 7.2.3).

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

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

Es importante *reiniciar* la máquina virtual al final del script. Esto se hace concretamente por los controladores de la tarjeta gráfica. Si no lo hacemos los controladores no funcionarán y la GPU no será utilizada cuando ejecutemos los algoritmos. A continuación se muestra un ejemplo de uso de GPU en la figura 6.8 de cuando se está entrenando y cuando no. La captura fue realizada en mi equipo local(por eso hay otros procesos como el *gnome*) y en el entorno *MountainCar-v0*. Motivo por el cual ocupa tan poca memoria rápida de la tarjeta, en este caso.[7][25][44]

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



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

Figura 6.8: 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 utilicé el comando **SCP**(secure copy), el cuál funciona también por encima de la tecnología SSH [19]. Posteriormente será utilizado en sentido opuesto para **descargar lo agentes** entrenados en la máquina a nuestro equipo local.

```
hapneck@Instancia:~$ 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:~$
```



```
hapneck@Equipo-Alex:~/Escritorio/Universidad/TFM$ scp ./script_maquina.sh hapneck@35.197.234.239:~
Warning: Permanently added '35.197.234.239' (RSA) to the list of known hosts.
The authenticity of host '35.197.234.239 (35.197.234.239)' can't be established.
RSA key fingerprint is SHA256:19bKwPwJ9869jVLE2fUkcdLV0ZKXPGRHLxbDX/uB.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '35.197.234.239' (RSA) to the list of known hosts.
script_maquina.sh                                         100% 1646    17.8KB/s   00:00
(base) hapneck@Equipo-Alex:~/Escritorio/Universidad/TFM$
```

Figura 6.9: Pasando el script a la máquina virtual para abastecerla.

6.4 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, debo explicar como utilizar estas librerías y herramientas de *OpenAi baselines y Gym*.

En la misma guía de Github[35], 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 y que vamos a ver a continuación:

```
> 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 4.1.2.

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, he puesto el entorno *MountainCar-v0*, el entorno mencionado en el apartado 5.1.

Por último, está el parámetro **--num_timesteps** el cual se utiliza para definir los pasos 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 guardar el formato PKL [10]. 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. Veremos estos datos en detalle durante la experimentación. Al igual que para guardar el modelo, especificamos una carpeta en la que almacenar los archivos referentes a estos datos.

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

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

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

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

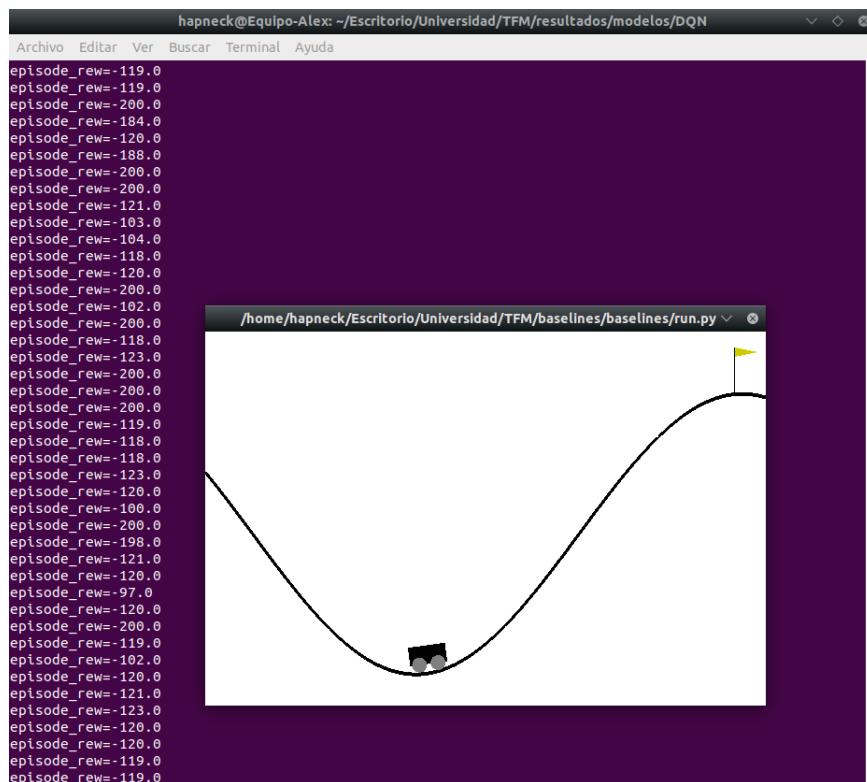


Figura 6.10: Probando agente entrenado en entorno simulado *MountainCar-v0*.

Podemos ver en la figura 6.10 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.

6.5 Forma de trabajar

La máquina cuenta con una mayor potencia, como es lógico, que mi ordenador personal. Sin embargo, la máquina virtual tiene una desventaja. Al comunicarme por SSH desde una terminal, no tengo a disposición la interfaz gráfica(gnome de linux) que *baselines* necesita para probar el agente de una manera visual. En otras palabras, la opción **`--play`** de la que hablé en el apartado anterior no funciona.

La solución que planteo es simple y sencilla. Entrenamos los modelos y obtenemos los logs en la máquina virtual y luego los traigo a mi equipo personal con el comando SCP para posteriormente visualizarlos desde la interfaz de mi equipo. 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.

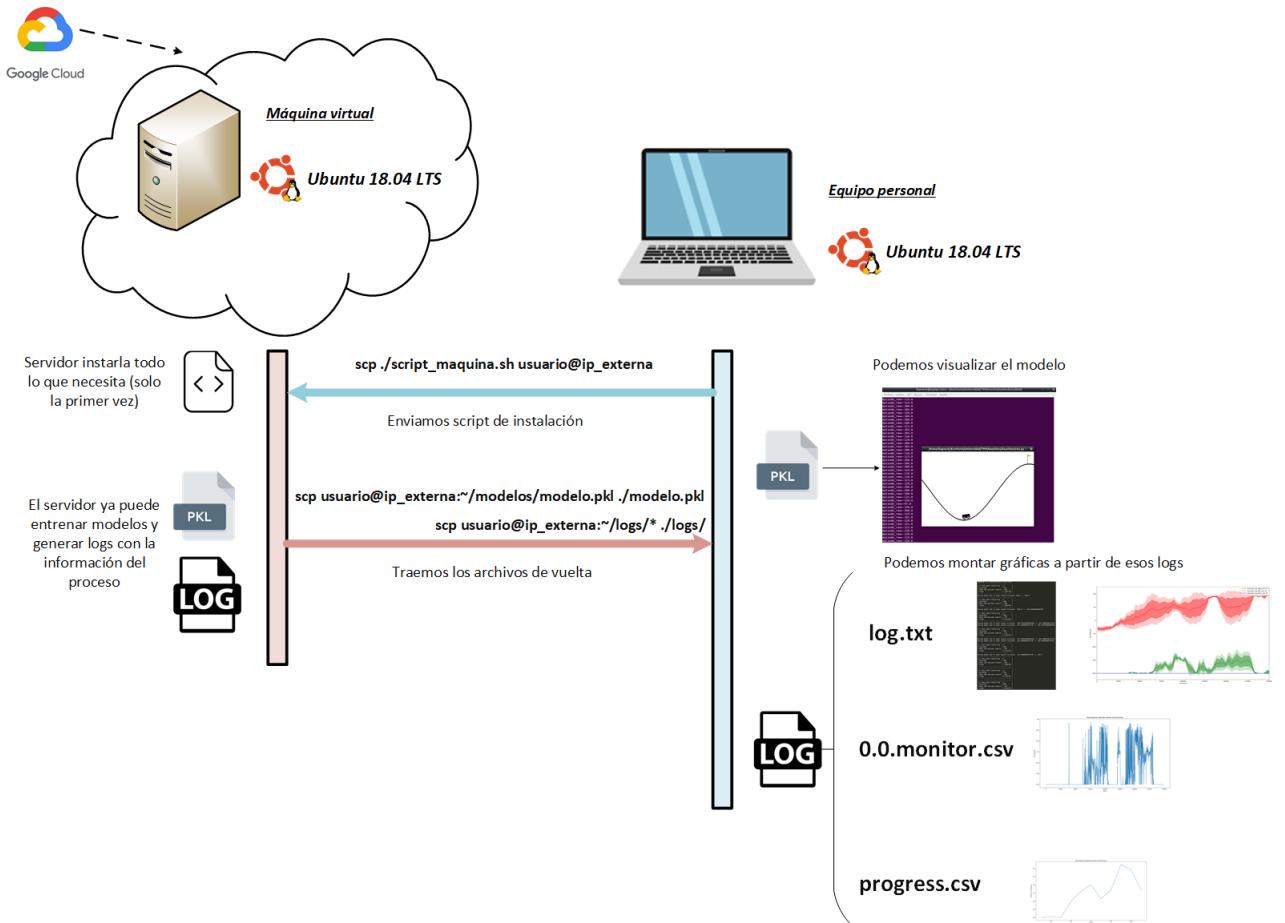


Figura 6.11: Esquema de metodología de trabajo entre mi equipo personal y la máquina virtual creada

6.6 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.

Conseguiremos 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 agente a partir de la experiencia, cuando la evaluación del progreso determina que ha mejorado(recordemos la figura 3.2).

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 sumatoria de recompensas que ha obtenido, el número de acciones o *timesteps* 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
...

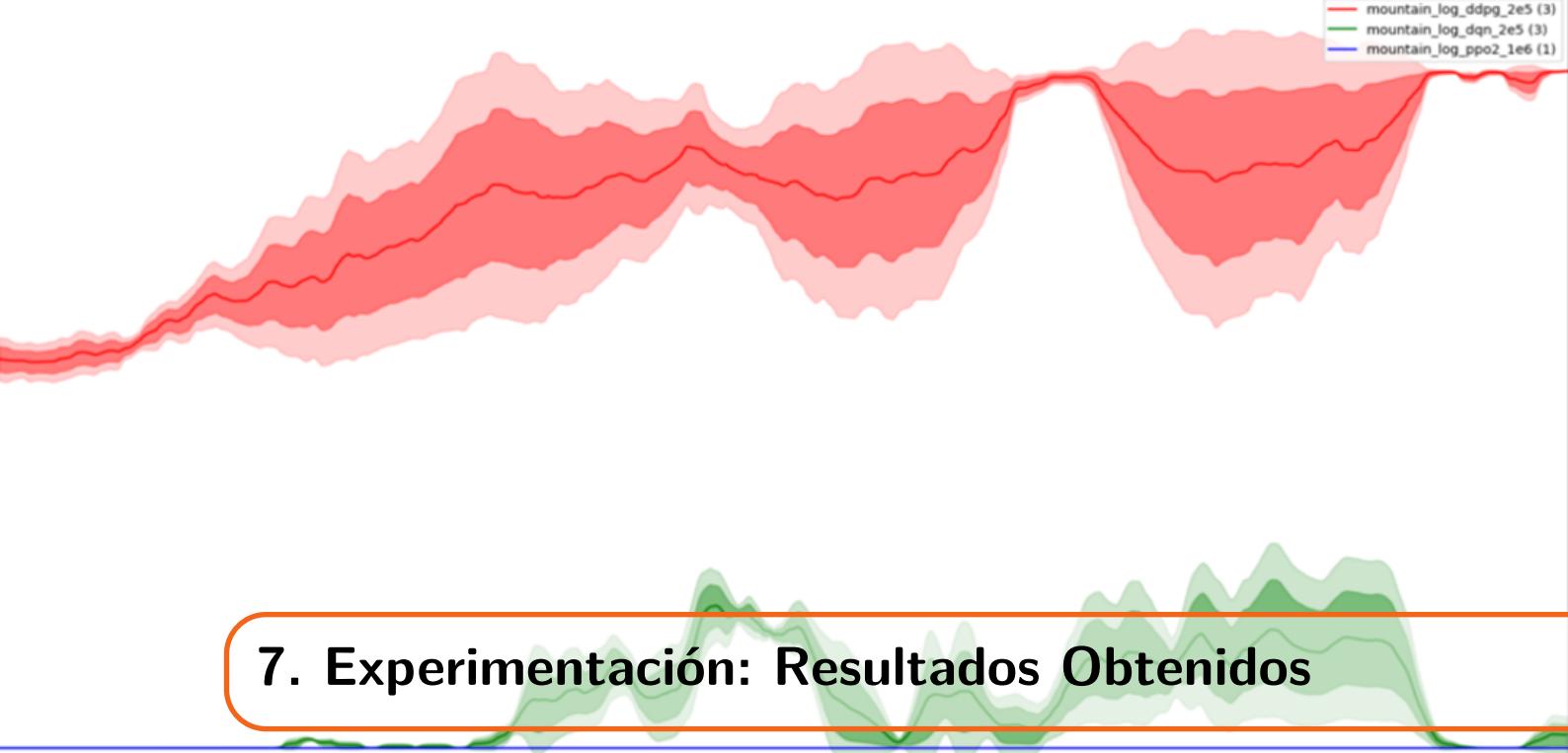
```

- **progress.csv:** Cada x 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*, veremos en la experimentación(apartado 7.2) como hacemos uso de estos datos de una forma más específica dependiendo de lo que haya en nuestras manos.



7. Experimentación: Resultados Obtenidos

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

Antes de empezar, voy a realizar una breve explicación de cómo he conseguido la **visualización** de los datos extraídos de los logs(apartado 6.6)

7.1 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 dejo referenciada al final de este apartado. Considero que está bien explicado su uso. Simplemente destacar que tuve que 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.[33]

Recomiendo usarla, es realmente útil. Principalmente porque no tienes que preocuparte de los diferentes archivos que hemos explicado en el apartado 6.6, 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.

7.2 MountainCar-v0

Comenzamos con el problema más sencillo que hemos utilizado (apartado 5.1). A pesar de su simpleza, podemos entender que dependiendo del problema, y de como lo interpretamos para el agente, esto puede hacer que algunas técnicas no sean efectivas o que den resultados mejores o peores. Lo veremos a continuación.

7.2.1 DQN

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

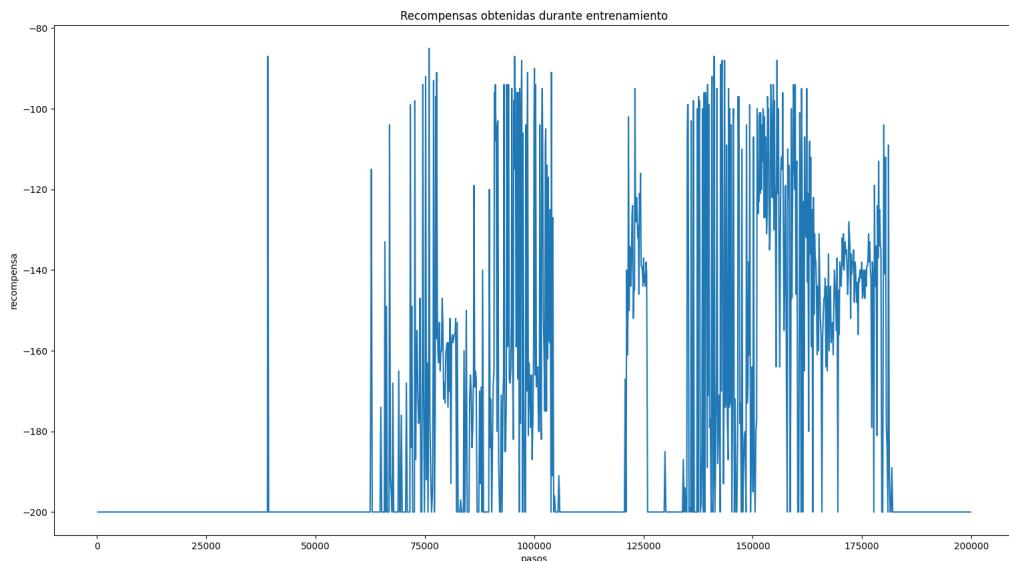


Figura 7.1: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 3 y en *MountainCar-v0*

Esta es la primera gráfica que aprendí a generar a partir de los *logs*. Con ella podemos observar las recompensas que fue obteniendo el agente en cada *timestep* que realizaba de experimentación. En el caso de la figura 7.1, vemos que consiguió resolverlo tempranamente en un caso aislado, además en pocos pasos, ya que tardó unos 90 *timesteps*(recompensa de -90). Lo cual quiere decir que aprovecho el momentum muy bien para conseguir subir.

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

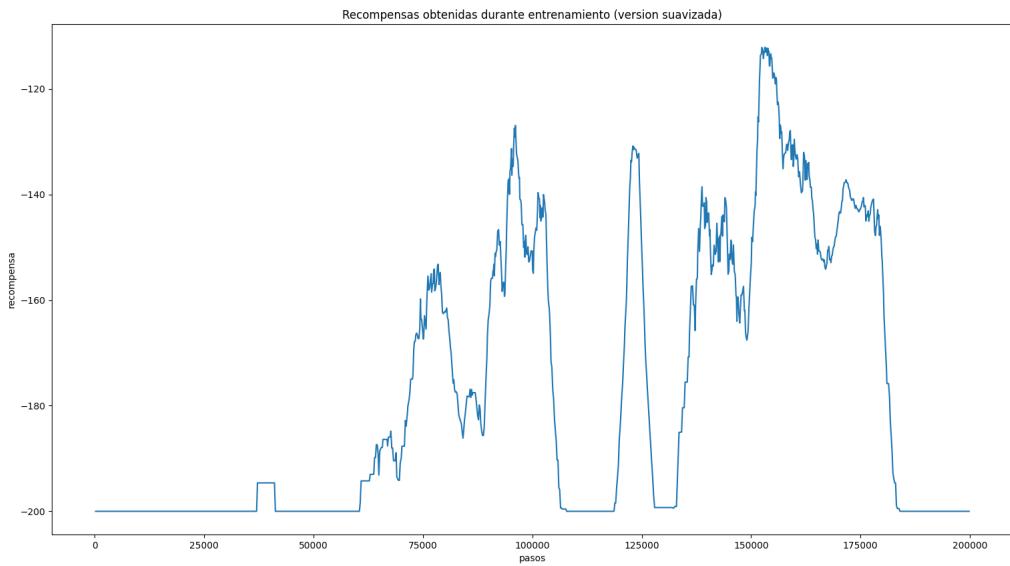


Figura 7.2: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 7.1). Algoritmo DQN, semilla 3 y en *MountainCar-v0*

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

El resultado es la figura 7.1. Fijémonos en el primer éxito que tiene. Ahora la subida en la recompensa es muy pequeña en comparación de las que hay más adelante, justo lo que sospechábamos.

Podemos ir más allá. En lugar de utilizar un suavizado de los datos de las recompensas, utilizar directamente las medias obtenidas por episodios, de los cuales hablamos en el apartado 6.6

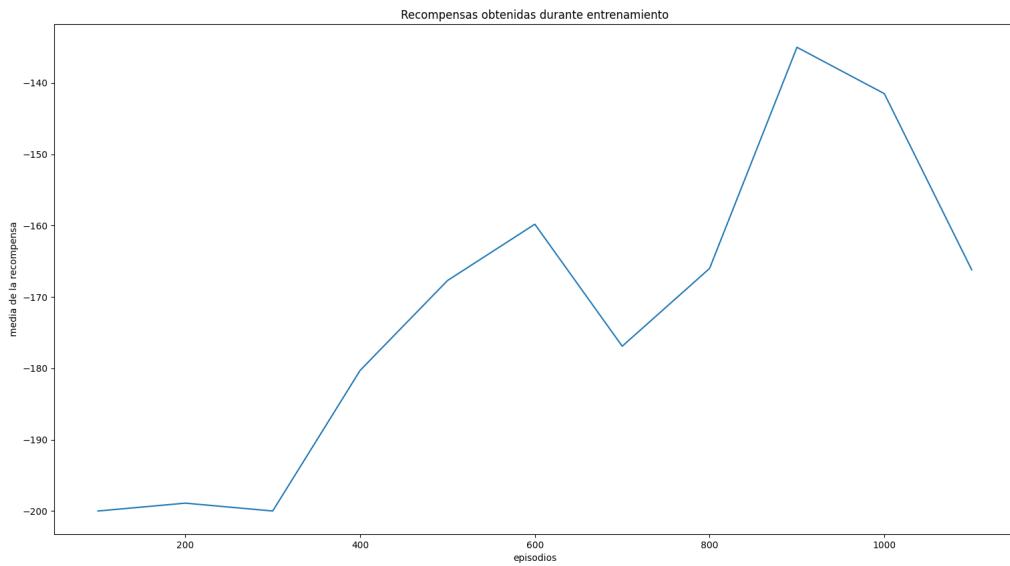


Figura 7.3: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 3 y en *MountainCar-v0*.

La figura 7.3, tiene el mismo objetivo que lo explicado anteriormente, dar más valor a aquellos éxitos que son resultado de un aprendizaje real y menos de la aleatoriedad, solo que aún más extremo.

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

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

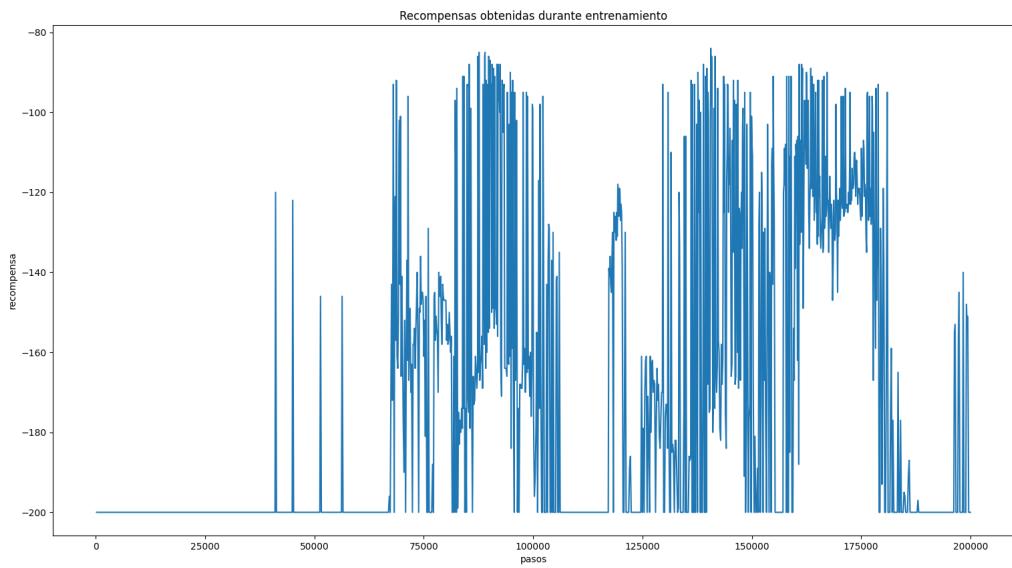


Figura 7.4: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

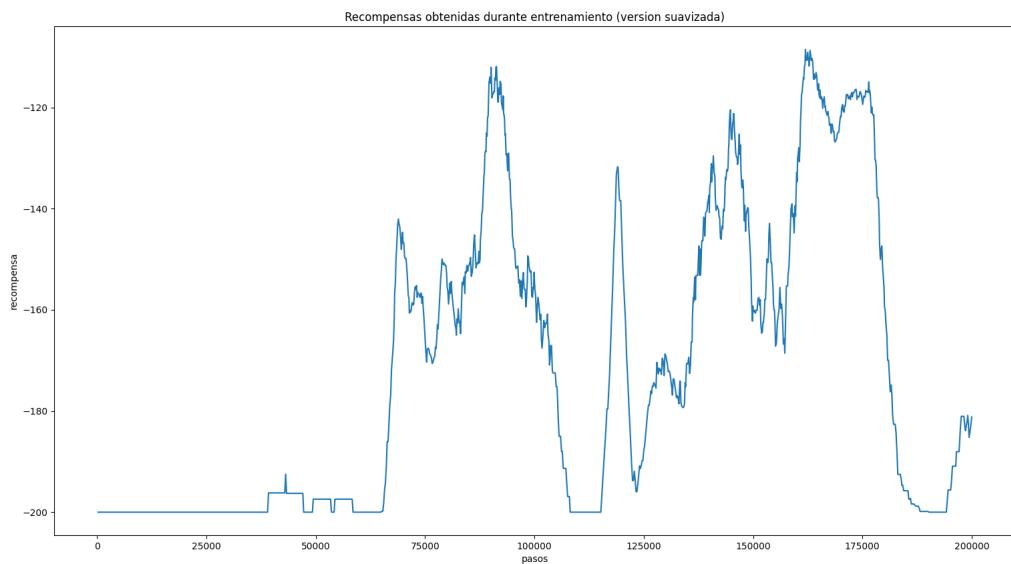


Figura 7.5: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 7.4). Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

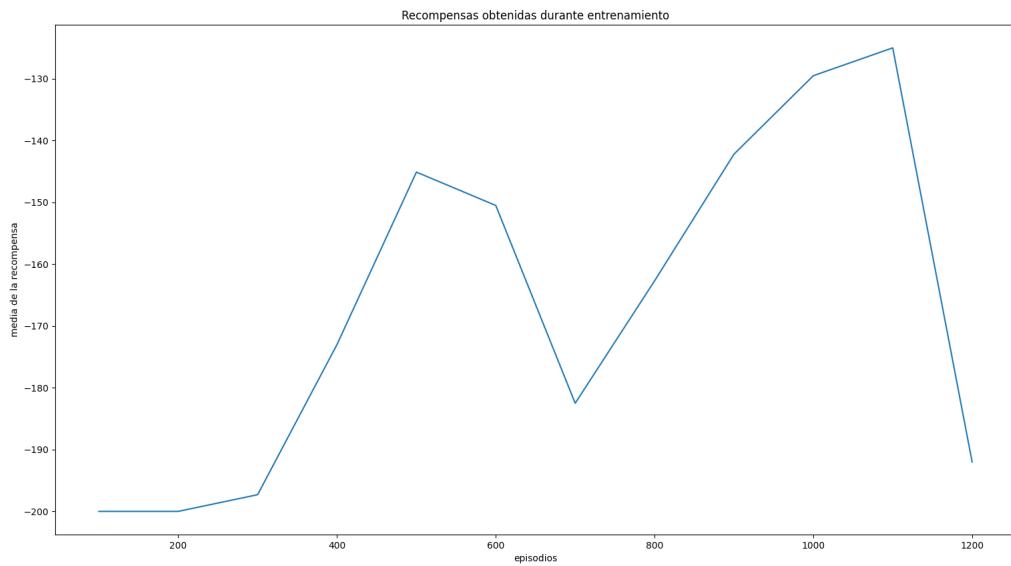


Figura 7.6: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 13 y en *MountainCar-v0*.

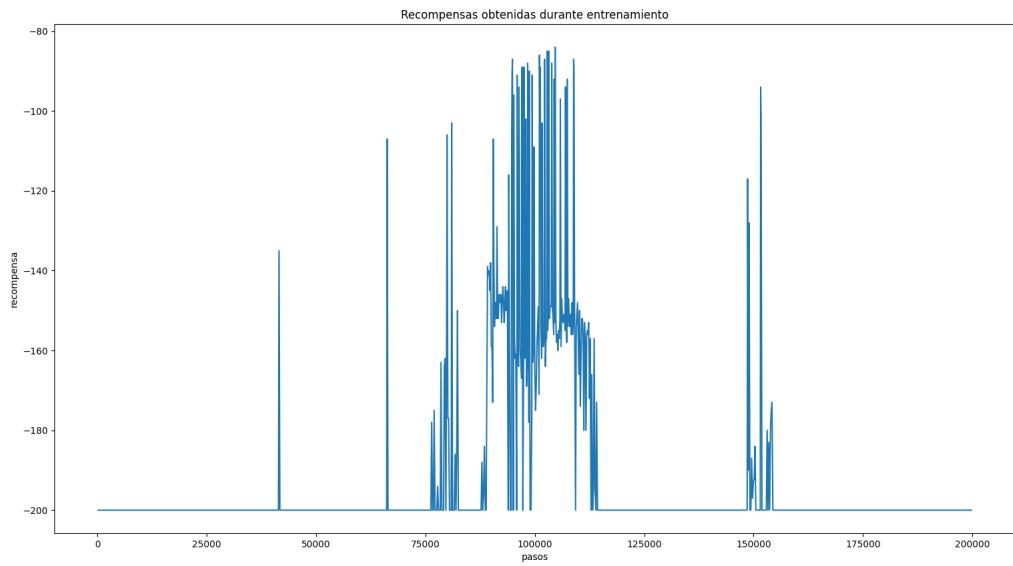


Figura 7.7: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

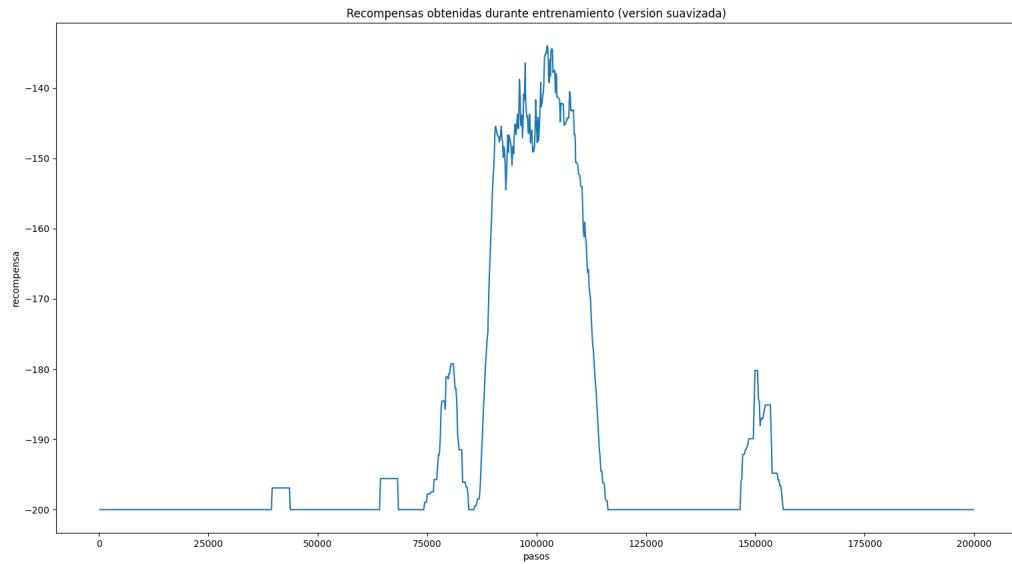


Figura 7.8: Recompensas obtenidas durante entrenamiento (versión suavizada de la figura 7.7). Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

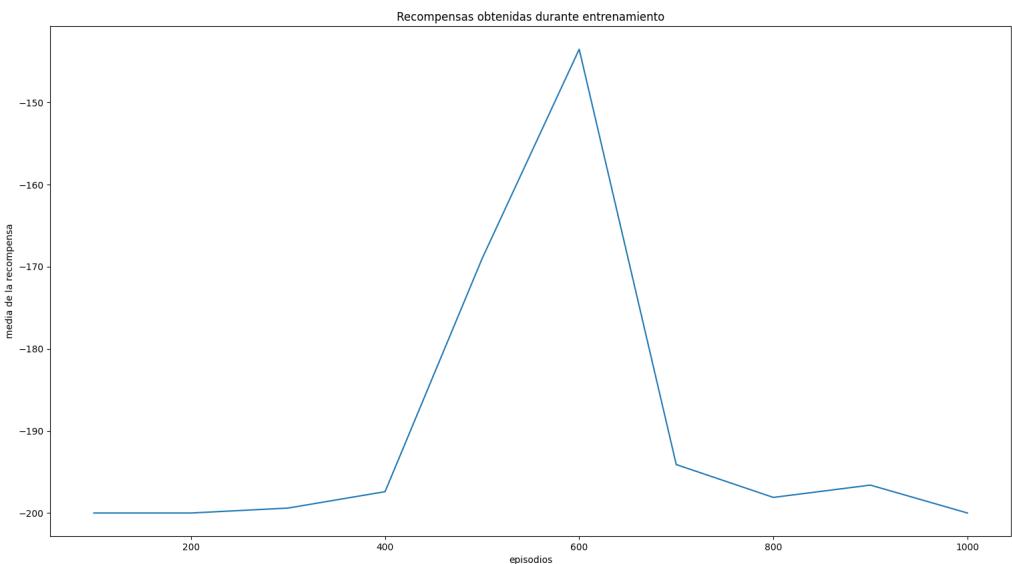


Figura 7.9: Media de recompensas obtenidas cada 100 episodios. Algoritmo DQN, semilla 46 y en *MountainCar-v0*.

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

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

solo va a dar resultados negativos.

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

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

En el apartado **5.1** mencionamos que la recompensa de este entorno es siempre -1 a no ser que alcance la meta, en ese paso el valor de la recompensa es 1. Creo que este sistema no ayuda al agente de ninguna manera a determinar si sus acciones son buenas o no y solo se puede dejar llevar por el resultado final para aprender. Entonces, el agente no puede diferenciar entre buenas y malas acciones. En su lugar, refuerza todas las acciones que ha realizado cuando tiene éxito, aunque dentro tenga decisiones malas y quita relevancia a todas las decisiones tomadas durante un fracaso, aunque en ese intento haya tomado algunas buenas decisiones en realidad.

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

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

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

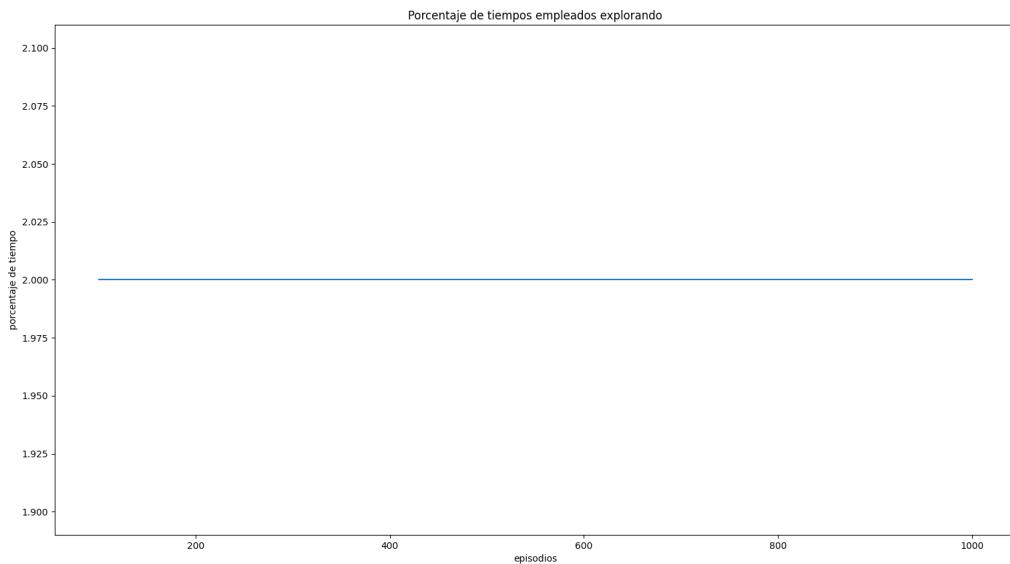


Figura 7.10: Porcentaje de tiempo empleado en explorar para algoritmo DQN en *MountainCar-v0*.

En la figura 7.10 vemos el porcentaje de tiempo empleado por el agente para explorar. Como ya hemos mencionado, no es mucho debido a la simplicidad del entorno y que explorar otras alternativas a las que el agente piensa no va a ser bueno. Sería interesante incluso implementar un sistema que redujera ese porcentaje a 0 cuando se superara una cierta recompensa. Estoy convencido de que esto mejoraría aún más los resultados o, al menos, estabilizaría más el aprendizaje. Por desgracia, no he tenido el suficiente tiempo para hacerlo.

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

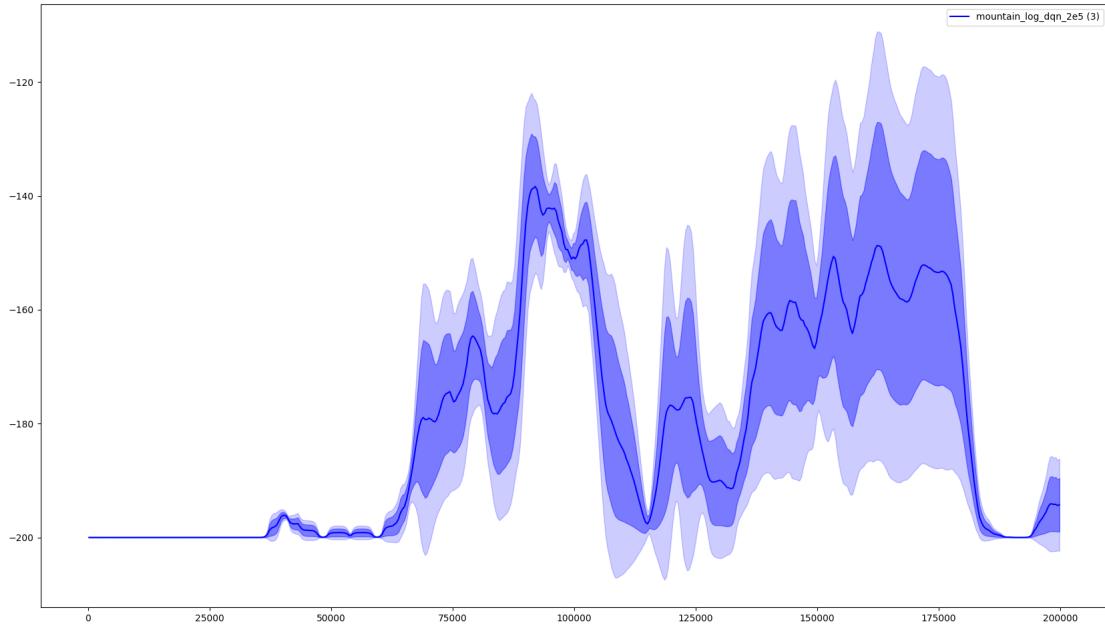


Figura 7.11: Resumen general de progreso de algoritmo DQN en *MountainCar-v0*.

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

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

Entonces, tenemos zonas más relevantes en la gráfica de la figura 7.11. Al principio coincide de forma perfecta que los tres modelos no son capaces de cumplir su objetivo, luego comienzan a tener algunos éxitos puntuales. La primera mejora sustancial de los datos es bastante común en el mismo punto del aprendizaje y después es un poco más variable de la aleatoriedad. Sin embargo, al final, hay una gran coincidencia de nuevo; todos empeoran hasta no ser capaces ni siquiera de cumplir el objetivo de alcanzar la meta y luego comienzan a remontar, pero aquí se acaba la gráfica. Esto me hace pensar que las conclusiones expresadas anteriormente son bastante firmes en cuanto a tiempo, sucediéndose en momentos similares en tres casos distintos.

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

7.2.2 PPO2

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

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

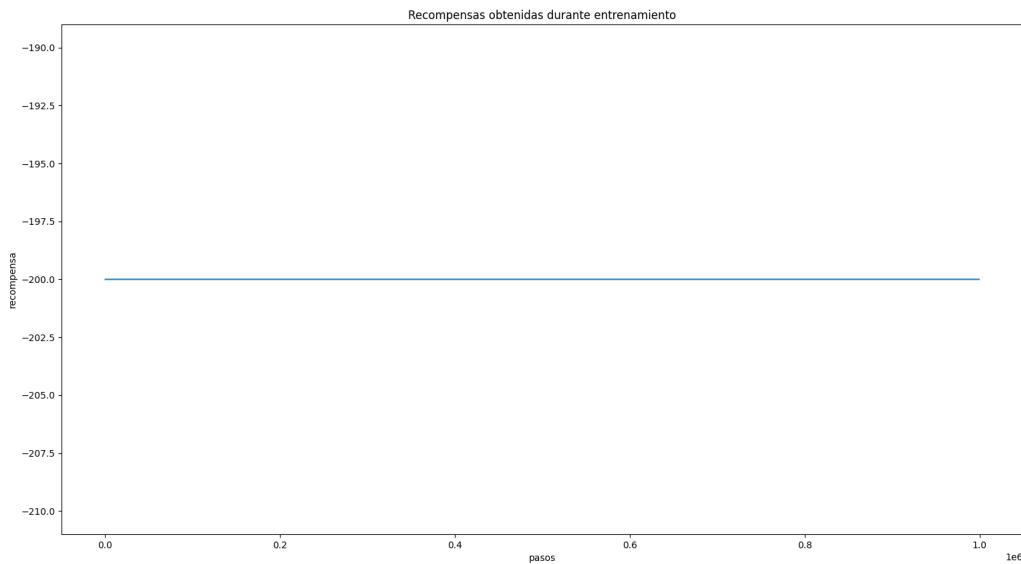


Figura 7.12: Recompensa obtenida con el algoritmo PPO2 en *MountainCar-v0*, independientemente de la semilla iteraciones y otros parámetros.

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

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

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

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

En definitiva, no cuenta con esa **memoria** que tiene DDPG, por ejemplo. Además, no valora lo prometedores que son las acciones en la solución como si hace DQN o DDPG, centrándose únicamente en los resultados una vez se consiguen, cosa que nunca llega (ver apartado 3 y 4.1).

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

técnica) o, al menos, que sea factible resolverlo a menudo con un agente totalmente aleatorio, no siendo este el caso.[21]

7.2.3 DDPG

Este algoritmo ha dado problemas desde el principio para poder ejecutarlo, aunque debo reconocer que ha merecido la pena dado que ha sido el que mejor resultados ha logrado. Como mencionamos en el apartado 3.6, este algoritmo trata de explotar los puntos fuertes de *Q-Learning* y de *Política de gradientes* al mismo tiempo. En este caso, se ha notado.

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

Estuve investigando mucho por Github, estudiando implementaciones forkeadas de otros usuarios que estaban tratando de solucionarlo. Incluso traté de modificar las implementaciones yo mismo, llegando a conseguir que guardase el modelo, aunque no guardaba parámetros importantes referentes al ruido, explicado en este trabajo. Haciendo que el modelo guardado no fuese el mismo que el entrenado, perdiendo parte de su esencia y, por tanto, calidad.[41]

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

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

Stable Baselines

En su documentación oficial[8], así como en su página web[5], encontramos las especificaciones para instalarlo, el cual está incluido en los comandos del script mostrado en la sección 6.3.

Se trata de unas implementaciones basadas en *OpenAI baselines*, al ser de código abierto ha surgido esta variante. Por lo que he podido ver, se esfuerzan más en buscar versiones estables de uso que en avanzar en todos los nuevos descubrimientos que surgen y nuevas técnicas. Podríamos decir que priorizan su **usabilidad** más que su **potencial**, por así decirlo. Creo que esa perspectiva es muy acertada para ir solucionando todos los errores que van surgiendo por parte de OpenAI, sin tener que preocuparse de tareas investigadoras.

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

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

```
import os
```

```

import argparse
import gym
import numpy as np

from stable_baselines.bench import Monitor

from stable_baselines.ddpg.policies import MlpPolicy
from stable_baselines.common.noise import NormalActionNoise, OrnsteinUhlenbeckActionNoise,
    AdaptiveParamNoiseSpec
from stable_baselines import DDPG
from stable_baselines.common.callbacks import BaseCallback
from stable_baselines.results_plotter import load_results, ts2xy

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

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

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

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

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

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

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

```

```

        self.model.save(self.save_path)

    return True

#Tratamiento de parametros del script
parser = argparse.ArgumentParser()
parser.add_argument("--num_timesteps", type=float, default=1e6, help="Indica el numero de
                    timesteps que va a tener el entrenamiento, por defecto 1e6")
parser.add_argument("--save_path", help="Indica la ruta y el nombre del archivo en el que se
                    va a almacenar el modelo.pkl")
parser.add_argument("--log_path", help="Indica la ruta en la que se almacena los logs del
                    proceso de aprendizaje")
parser.add_argument("--env", help="entorno gym que utiliza el modelo")
parser.add_argument("--seed", type=int, help="Semilla con la que se inicia el entrenamiento
                    del modelo")
args = parser.parse_args()

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

if args.save_path:
    save_dir=args.save_path

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

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

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

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

```

Tanto la implementación como la clase *SaveOnBestTrainingRewardCallback*, la cual sirve para guardar los modelos, han sido inspirados a partir de las implementaciones que ellos mismos ofrecen en su página web [3] [4]. Aún así, he leído que 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 que usaba para entrenar en la máquina virtual. Vemos que he incluido parámetros que se llaman de la misma forma que los *baselines* de OpenAI para que tenga una forma de uso muy similar.

La principal desventaja que observo con respecto a los *baselines* originales es la **monitORIZACIÓN** del progreso de aprendizaje. Encontré 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 6.6.

Para obtener la salida por pantalla en un archivo, equivalente al *log.txt*, simplemente usaba los **pipelines** de linux para guardarla en un archivo al mismo tiempo que sacaba la información por pantalla para poder visualizar su progreso y que todo estaba bien mientras entrenaba:

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

No obstante, no había una forma clara de poder conseguir la información equivalente del archivo *progress.csv*. Intenté incluso generar mi propio sistema para conseguir dicha información. Sin embargo, la librería no ofrece una forma clara de poder rescatar estos datos durante su aprendizaje, al menos que yo haya encontrado. Generar esa funcionalidad desde cero no me ha sido posible debido a todo el tiempo que habría que dedicarle.

Para cargar los modelos entrenados en el equipo personal y probarlos fabriqué 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, configuro parámetros para que tenga un uso similar a los *baselines* originales. Simplemente cargo el entorno y lo inicio, cargo el modelo y, dándole la observación del entorno, dejo que él mismo decida la siguiente acción, ejecutándola en el entorno.

Gym ofrece una variable que indica cuando se ha llegado a un estado terminal, en ese caso reinicio el entorno y vuelve a comenzar. Uno de los inconvenientes de probar los

modelos es que son **demasiado rápidos** al ejecutarse y es posible que cueste un poco ver como actua, es como si todo estuviera en cámara rápida.

Al poder generar yo mismo 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.

Resultados

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

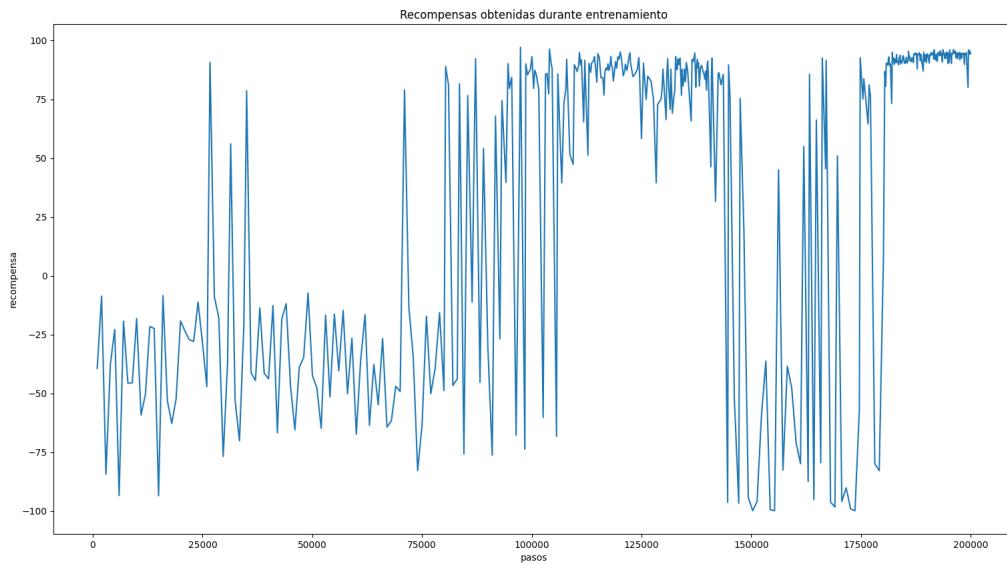


Figura 7.13: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 3 y en *MountainCar-v0*.

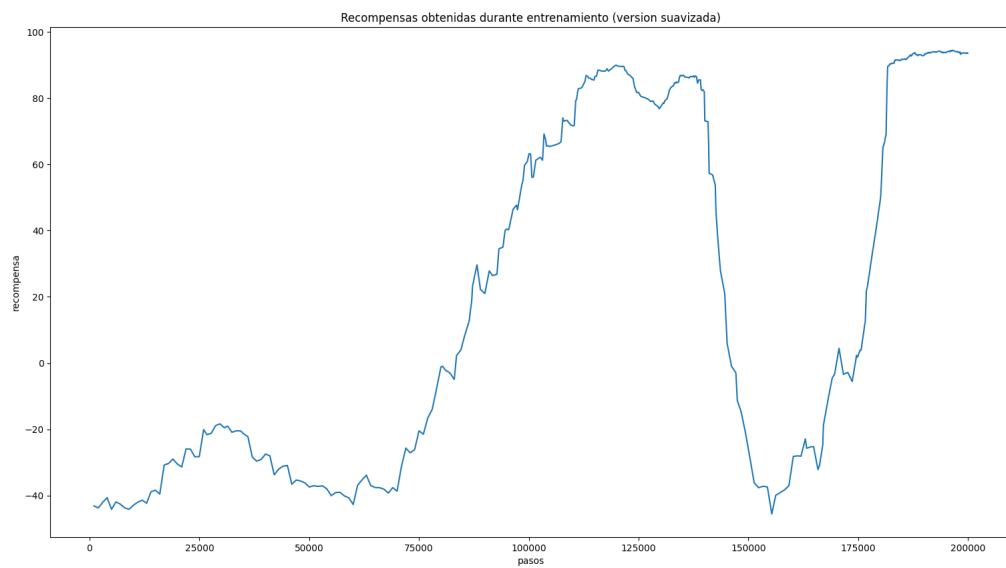


Figura 7.14: Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3 y en *MountainCar-v0*.

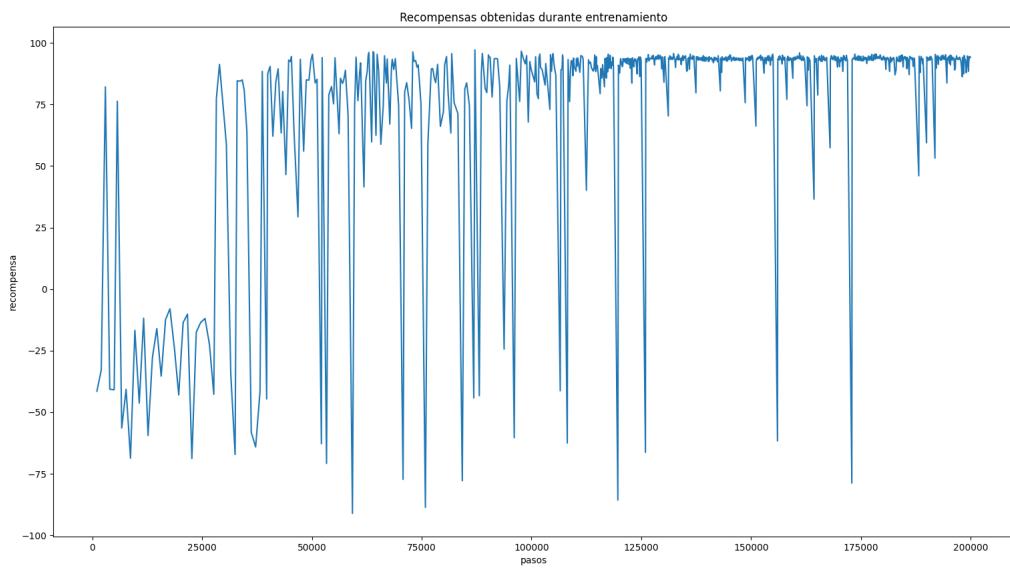


Figura 7.15: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 13 y en *MountainCar-v0*.

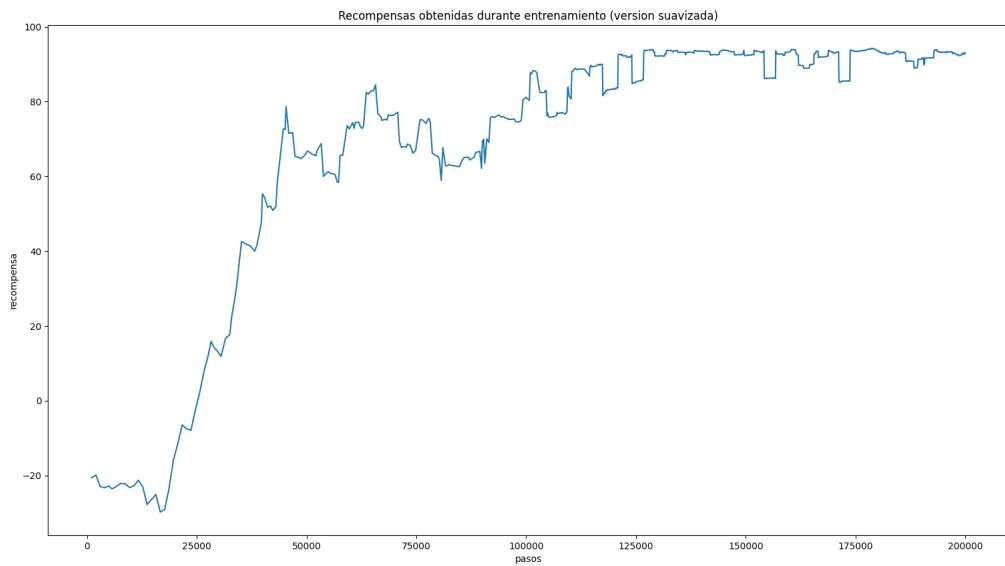


Figura 7.16: Gráfica de recompensas obtenidas durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 13 y en *MountainCar-v0*.

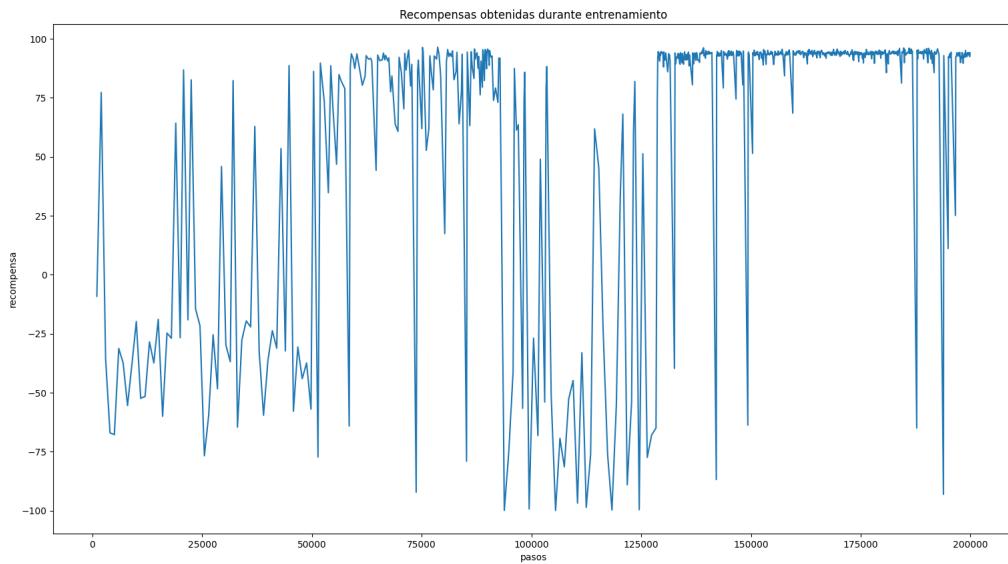


Figura 7.17: Gráfica de recompensas obtenidas durante entrenamiento. Algoritmo DDPG, semilla 46 y en *MountainCar-v0*.

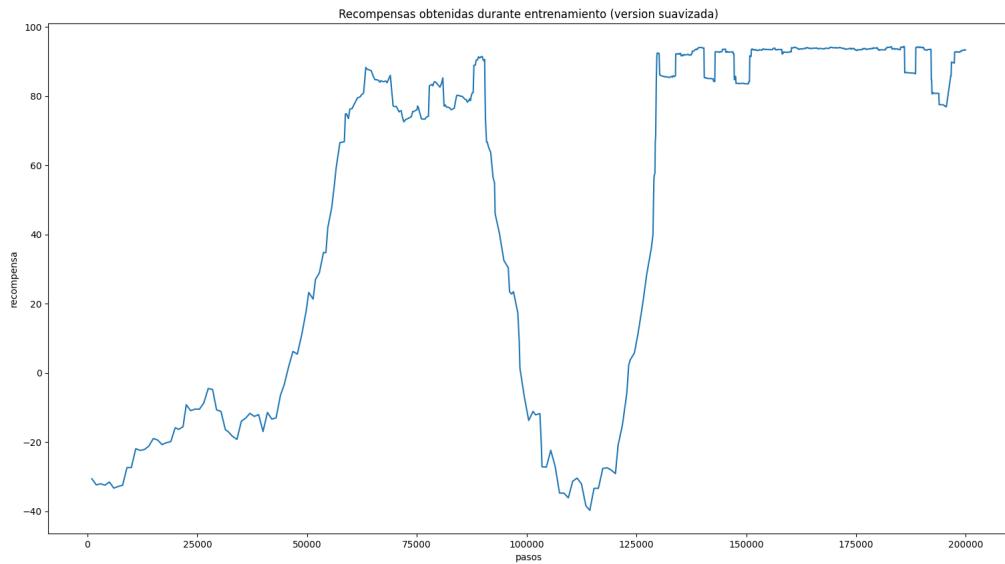


Figura 7.18: Gráfica de recompensas obtenidas durante entrenamiento (suavizadas). Algoritmo DDPG, semilla 46 y en *MountainCar-v0*.

Recordemos que este algoritmo se está ejecutado en la versión continua de MountainCar-v0 visto en el apartado 5.2. Por ello, el rango de recompensas no es el mismo al calcularse de diferente forma.

A simple vista, vemos que las gráficas son mucho más estables que en DQN, sobretodo para la semilla 13 en la figura 7.16. Recordemos que tiene dos partes, una política que decide cual es la siguiente acción y un Q-Learning en otra red neuronal que evalúa como de buena es esa decisión y supervisa su desempeño.

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

Si recordamos el apartado 4.1.3, el ruido es introducido directamente en los parámetros de la red neuronal y no en las acciones posibles. Esto quiere decir que ese ruido no tiene porqué modificar de forma directa las acciones que realiza, simplemente un poco su comportamiento.

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

Vamos a ver como afecta la aleatoriedad a este algoritmo:

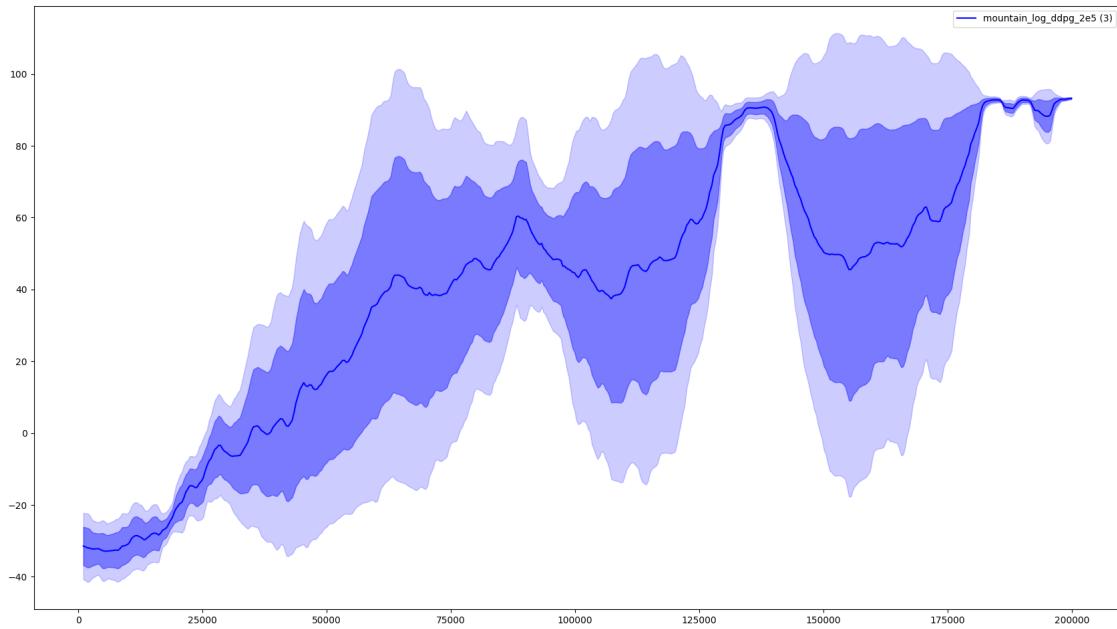


Figura 7.19: Gráfica general obtenida durante entrenamiento(suavizadas). Algoritmo DDPG, semilla 3, 13 y 46 simultáneamente y en *MountainCar-v0*.

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

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

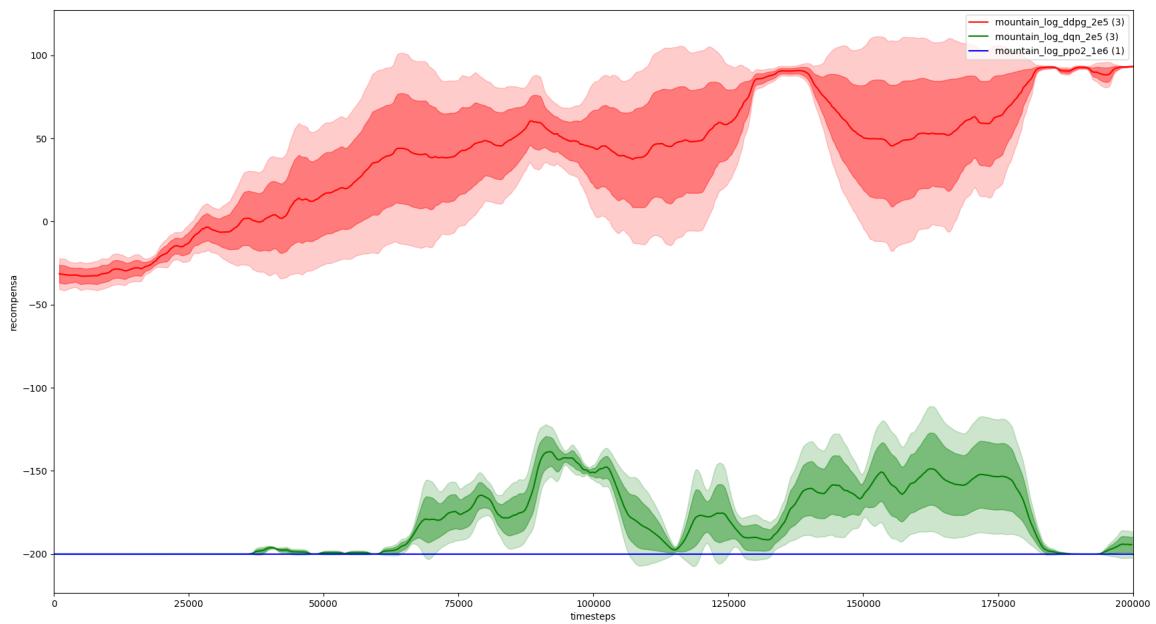


Figura 7.20: Gráfica comparativa de los tres algoritmos: DQN, PPO2 y DDPG en *MountainCar-v0*.

7.3 Otro entorno más complejo (Aún sin decidir)

7.3.1 DQN

7.3.2 PPO2

7.3.3 DDPG

IV

Estado del arte

8	AlphaGO	97
9	AlphaStar	99

Introducción

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

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

AlphaGo Zero

Discovering new knowledge

8. AlphaGO



9. AlphaStar

V

Posibles mejoras y conclusiones

10	Conclusiones finales	103
10.1	Posibles mejoras	
10.2	Conclusiones	
	Bibliografía	105
	Artículos	
	Libros	



10. Conclusiones finales

10.1 Posibles mejoras

10.2 Conclusiones



Bibliography

Articles

- [1] Claudio Amorim. “Imagen de red neuronal”. En: . (consultado en 2020). <https://artfromcode.wordpress.com/2017/04/18/red-neuronal-en-python-con-numpy-parte-1/> (véase página 23).
- [2] Anaconda. “Who is Anaconda?” En: *Página oficial* (2020). <https://www.anaconda.com/about-us> (véase página 62).
- [3] Stable Baselines. “DDPG”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/modules/ddpg.html> (véase página 84).
- [4] Stable Baselines. “Examples”. En: *Página oficial* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/examples.html> (véase página 84).
- [5] Stable Baselines. “Welcome to Stable Baselines docs! - RL Baselines Made Easy”. En: *Github Docs* (2020). <https://stable-baselines.readthedocs.io/en/master/> (véase página 82).
- [6] Fernando Sancho Caparrini. “Ejemplo de gradiente descendente estocástico.” En: *Imagen* (consultado en 2020). <http://www.cs.us.es/~fsancho/?e=165> (véase página 28).
- [7] Shiyu Chen. “Tutorial: Installation and Configuration of MuJoCo, Gym, Baselines”. En: *www.chenshiyu.top* (2019). <https://www.chenshiyu.top/blog/2019/06/19/Tutorial-Installation-and-Configuration-of-MuJoCo-Gym-Baselines/> (véase página 63).
- [8] Irene Alvarado engineer y creative technologist. “redes Neuronales”. En: *Github* (Consultado en 2020). https://ml4a.github.io/ml4a/es/neural_networks/ (véanse páginas 24-26).
- [9] FileInfo. “.PKL File Extension”. En: *Página oficial* (2020). <https://fileinfo.com/extension/pkl> (véase página 65).

- [11] gcpping. “Measure your latency to GCP regions”. En: *gcpping.com* (2020). <http://www.gcpping.com/> (véase página 58).
- [12] Docs Github. “Installation”. En: *Stable-Baselines Official* (2020). <https://stable-baselines.readthedocs.io/en/master/guide/install.html> (véase página 63).
- [13] Google. “DeepMind”. En: *Página web oficial* (2020). <https://deepmind.com/> (véase página 16).
- [14] 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 60).
- [15] Red Hat. “Provisioning Ansible”. En: *Red Hat Ansible* (2020). <https://www.ansible.com/use-cases/provisioning> (véase página 61).
- [16] Iberdrola. “¿Qué es la Inteligencia Artificial?” En: <https://www.iberdrola.com/innovacion/que-es-inteligencia-artificial> (2020) (véase página 13).
- [18] John Schulman & Filip Wolski & Prafulla Dhariwal & Alec Radford & Oleg Klimov. “Proximal Policy Optimization Algorithms”. En: *DeepMind* (2017). <https://arxiv.org/pdf/1707.06347.pdf> (véase página 47).
- [19] 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 64).
- [20] Ludoteka. “Go”. En: *Ludoteka.com* (Consultado en 2020). <http://www.ludoteka.com/juego-go.html> (véase página 21).
- [21] LupusPrudens. “PPO struggling at MountainCar whereas DDPG is solving it very easily. Any guesses as to why?” En: *Reddit* (2019). https://www.reddit.com/r/reinforcementlearning/comments/9o8ez0/ppo_struggling_at_mountaincar_whereas_ddpg_is/ (véase página 82).
- [22] 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 29).
- [23] Marcos Merino. “La inteligencia artificial AlphaStar se proclama ‘gran maestro’ de Starcraft II en igualdad de condiciones frente a los humanos.” En: *Xataka* (2019). <https://www.xataka.com/inteligencia-artificial/inteligencia-artificial-alphastar-se-proclama-gran-maestro-starcraft-ii-igualdad-condiciones-frente-a-humanos> (véase página 16).
- [24] Carlos García Moreno. “¿Qué es el Deep Learning y para qué sirve?” En: *Indra Blog Neo* (2020). <https://www.indracompany.com/es/blogneo/deep-learning-sirve> (véase página 15).
- [25] David Naranjo. “¿Cómo instalar los drivers de video Nvidia en Ubuntu 18.04?” En: *Ubunlog* (2020). <https://ubunlog.com/como-instalar-los-drivers-de-video-nvidia-en-ubuntu-18-04/> (véase página 63).
- [26] Netflix. “AlphaGo”. En: *Documental* (2017). <https://www.alphagomovie.com/> (véase página 16).
- [27] OpenAI. “Better Exploration with Parameter Noise”. En: *Página web oficial* (2017) (véanse páginas 48, 49).

- [28] OpenAI. “OpenAI Baselines: DQN”. En: *Página web oficial* (2017). <https://openai.com/blog/openai-baselines-dqn/> (véase página 47).
- [29] OpenAI. “Proximal Policy Optimization (PPO2)”. En: *Página web oficial* (2017). <https://openai.com/blog/openai-baselines-ppo/> (véase página 47).
- [30] OpenAI. “About OpenAI”. En: *Página web oficial* (2020). <https://openai.com/about/> (véase página 45).
- [31] OpenAI. “Gym”. En: *Página web oficial* (2020). <https://gym.openai.com/> (véanse páginas 16, 52).
- [32] OpenAI. “<https://gym.openai.com/envs/MountainCar-v0/>”. En: *Entornos Gym* (2020). <https://gym.openai.com/envs/MountainCar-v0/> (véase página 53).
- [33] OpenAI. “Loading and visualizing results (open in colab)”. En: *Github* (2020). <https://github.com/openai/baselines/blob/master/docs/viz/viz.ipynb> (véase página 71).
- [34] OpenAI. “MountainCarContinuous v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCarContinuous-v0> (véase página 54).
- [35] OpenAI. “OpenAI baselines”. En: *Repositorio Github oficial* (2020). <https://github.com/openai/baselines> (véanse páginas 45, 63, 65).
- [36] OpenAI. “Progresos de OpenAI”. En: *Página web oficial* (2020). <https://openai.com/progress/> (véase página 45).
- [37] OpenAI. “Table of environments”. En: *Github* (2020). <https://github.com/openai/gym/wiki/Table-of-environments> (véase página 52).
- [38] OpenAi. “Detalles técnicos del entorno MountainCar v0”. En: *Github* (2020). <https://github.com/openai/gym/wiki/MountainCar-v0> (véase página 52).
- [40] Crustian Rus. “’AlphaGo’ es el documental de Netflix que mejor explica lo que supuso la victoria de la IA de Google al campeón de Go.” En: *Xataka* (2018). <https://www.xataka.com/cine-y-tv/alphago-es-el-documental-de-netflix-que-mejor-explica-lo-que-supuso-la-victoria-de-la-ia-de-google-al-campeon-de-go> (véase página 16).
- [41] Sumsamkhan. “Saving and restoring DDPG agent”. En: *Github issue* (2017). <https://github.com/openai/baselines/issues/162> (véase página 82).
- [42] Tensorflow. “Why TensorFlow”. En: *Página oficial* (2020). <https://www.tensorflow.org/about> (véase página 62).
- [43] 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 59).
- [44] Kapil Varshney. “How to Setup Ubuntu 16.04 with CUDA, GPU, and other requirements for Deep Learning”. En: *Medium* (2018). <https://medium.com/@kapilvarshney/how-to-setup-ubuntu-16-04-with-cuda-gpu-and-other-requirements-for-deep-learning-f547db75f227> (véase página 63).
- [45] Oriol Vinyals. “AlphaStar: Mastering the Real Time Strategy Game StarCraft II - Oriol Vinyals”. En: *Youtube* (2019). <https://www.youtube.com/watch?v=3UdH3lPF7nE> (véase página 16).

-
- [46] 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 49).
 - [47] Wikipedia. “Gráfica de aprendizaje por refuerzo.” En: *Imagen* (consultado en 2020). https://es.wikipedia.org/wiki/Aprendizaje_por_refuerzo (véase página 32).

Books

- [8] 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 82).
- [17] Max Pumperla y Kevin ferguson. *Deep Learning and The Game of Go*. Manning, 2019 (véase página 13).
- [39] Ismael Pérez Roldán. *Clasificación de Obras de Arte Por Estilo Artístico Usando Redes Neuronales Convolucionales*. http://oa.upm.es/56163/1/TFG_ISMAEL_PEREZ_ROLDAN.pdf. Universidad Politécnica de Madrid, 2019 (véase página 29).