

Aprendizaje por refuerzo profundo de estrategias para juegos interactivos

por

Tomás Cánepa



Tesis de Licenciatura en Ciencias de la Computación
Facultad de Ciencias Físico Matemáticas y Naturales
Universidad Nacional de San Luis
San Luis, Argentina
Marzo de 2023

Director: Marcelo Errecalde
Co-Director: Sergio Burdisso

TABLA DE CONTENIDOS

LISTA DE FIGURAS	IV
----------------------------	----

CAPÍTULO

I. Introducción	1
1.1 Antecedentes y fundamentación	1
1.2 Objetivo e hipótesis de investigación	8
1.3 Organización del informe	9
II. Introducción al aprendizaje por refuerzo	11
2.1 Agentes y ambientes	12
2.2 Aprendizaje supervisado	13
2.3 Aprendizaje no supervisado	13
2.4 Aprendizaje por refuerzo	14
2.4.1 Componentes del aprendizaje por refuerzo	15
III. Procesos de decisión Markov	19
3.1 Procesos de decisión Markov	19
3.2 Funciones de valor	22
IV. Métodos para resolver problemas de aprendizaje por refuerzo	25
4.1 Clasificación de métodos	26
4.2 Value Iteration	27
4.3 Q-Learning	29
4.4 Deep Q-Learning	30
4.4.1 Redes neuronales	32
4.4.2 Deep Q-Learning	36
V. Experimentación	42

5.1	Introducción	43
5.2	Bibliotecas	44
5.2.1	OpenAI Gym	45
5.2.2	Stable-Baselines3	48
5.3	Experimentos	51
5.3.1	Batch Size	58
5.3.2	Buffer Size	65
5.3.3	Exploration Fraction	71
5.3.4	Framestack	77
5.3.5	Learning Rate	83
5.4	Análisis de resultados	89
VI. Conclusiones y Trabajo Futuro		95
6.1	Trabajo Futuro	96
BIBLIOGRAFÍA		99

LISTA DE FIGURAS

Figura

1.1	Interacción agente-ambiente.	2
1.2	Esquema de la representación neuronal de <i>Deep Q-Networks</i> para un mundo estocástico simple.	5
2.1	Esquema de la interacción de los componentes de <i>reinforcement learning</i>	17
3.1	Interacción agente-ambiente en un proceso de decisión Markov. . . .	20
4.1	Algoritmo Q-Learning.	30
4.2	Esquema de una red neuronal <i>feedforward</i> con cuatro neuronas en la capa de entrada, 2 neuronas en la capa de salida y dos capas ocultas.	32
4.3	Gráficas de las funciones Sigmoide, Tangente hiperbólica y ReLU. . .	33
4.4	Arquitectura de una red neuronal convolucional.	34
4.5	Convolución para una imagen representada por una matriz 4x4 y un kernel 2x2.	35
4.6	<i>Max pooling</i> para un mapa de activación 4x4, filtro 2x2 y stride 2. . .	36
4.7	Esquema de una arquitectura de red neuronal convolucional cuya entrada son imágenes y su salida acciones que se pueden realizar en juegos de Atari.	38
4.8	Algoritmo deep Q-Learning con <i>experience replay</i>	41
5.1	Ejemplo del uso de <i>OpenAI Gym</i>	48
5.2	Ejemplo del uso de <i>Stable-Baselines3</i>	50
5.3	<i>Breakout</i>	52
5.4	<i>Space Invaders</i>	53
5.5	<i>Boxing</i>	54
5.6	Curvas de entrenamiento de los modelos con <code>batch_size</code> modificado y del modelo base para el juego Breakout.	60
5.7	Curvas de entrenamiento de los modelos con <code>batch_size</code> modificado y del modelo base para el juego Space Invaders.	62
5.8	Curvas de entrenamiento de los modelos con <code>batch_size</code> modificado y del modelo base para el juego Boxing.	64
5.9	Curvas de entrenamiento de los modelos con <code>buffer_size</code> modificado y del modelo base para el juego Breakout.	66

5.10	Curvas de entrenamiento de los modelos con <code>buffer_size</code> modificado y del modelo base para el juego Space Invaders.	68
5.11	Curvas de entrenamiento de los modelos con <code>buffer_size</code> modificado y del modelo base para el juego Boxing.	70
5.12	Curvas de entrenamiento de los modelos con <code>exploration_fraction</code> modificado y del modelo base para el juego Breakout.	72
5.13	Curvas de entrenamiento de los modelos con <code>exploration_fraction</code> modificado y del modelo base para el juego Space Invaders.	74
5.14	Curvas de entrenamiento de los modelos con <code>exploration_fraction</code> modificado y del modelo base para el juego Boxing.	76
5.15	Curvas de entrenamiento de los modelos con <code>framestack</code> modificado y del modelo base para el juego Breakout.	78
5.16	Curvas de entrenamiento de los modelos con <code>framestack</code> modificado y del modelo base para el juego Space Invaders.	80
5.17	Curvas de entrenamiento de los modelos con <code>framestack</code> modificado y del modelo base para el juego Boxing.	82
5.18	Curvas de entrenamiento de los modelos con <code>learning_rate</code> modificado y del modelo base para el juego Breakout.	84
5.19	Curvas de entrenamiento de los modelos con <code>learning_rate</code> modificado y del modelo base para el juego Space Invaders.	86
5.20	Curvas de entrenamiento de los modelos con <code>learning_rate</code> modificado y del modelo base para el juego Boxing.	88

CAPÍTULO I

INTRODUCCIÓN

En este capítulo inicial, se realizará una introducción general al problema abordado en este trabajo, antecedentes de conceptos y publicaciones relevantes a la temática aquí discutida, se mostrarán los objetivos de este trabajo, así como también la estructura del mismo.

§1.1 Antecedentes y fundamentación

El concepto de *agente racional* en inteligencia artificial está directamente relacionado con el enfoque de considerar al comportamiento inteligente como un proceso de *selección de acciones*. Este proceso, en el que el agente decide *qué hacer*, se basa en la secuencia de observaciones del ambiente y de cualquier conocimiento previo que tuviera el agente, de manera tal de maximizar alguna medida de desempeño [35]. Si bien esta idea es sencilla de entender, no es tan sencillo diseñar políticas/estrategias efectivas para tomar estas decisiones, principalmente en problemas desafiantes del mundo real, como el desarrollo de *chatbots*, el control de autos autónomos y robots, o aprender a jugar juegos interactivos al nivel de como lo hace un humano.

Uno de los formalismos matemáticos más difundidos para modelizar este tipo de problemas es el conocido como Proceso de Decisión Markov, en inglés *Markov Decision*

Process y, abreviadamente, MDP. Los MDPs permiten abordar problemas de decisión secuencial, donde la utilidad del agente depende de una secuencia de decisiones y existe incertidumbre en el resultado de las acciones realizadas en cada paso. La idea intuitiva de este modelo es que el agente esté en contacto con el ambiente a través de percepciones y acciones. En cada paso de tiempo t , el agente recibe como entrada desde el ambiente una observación del estado actual del ambiente, s_t , y selecciona una acción a_t aplicable en el estado s_t . La acción generada por el agente modifica el estado del ambiente, el cual responde un paso de tiempo después con una observación s_{t+1} del nuevo estado y una señal de recompensa inmediata $r_{t+1} \in \mathbb{R}$ por la acción tomada previamente.

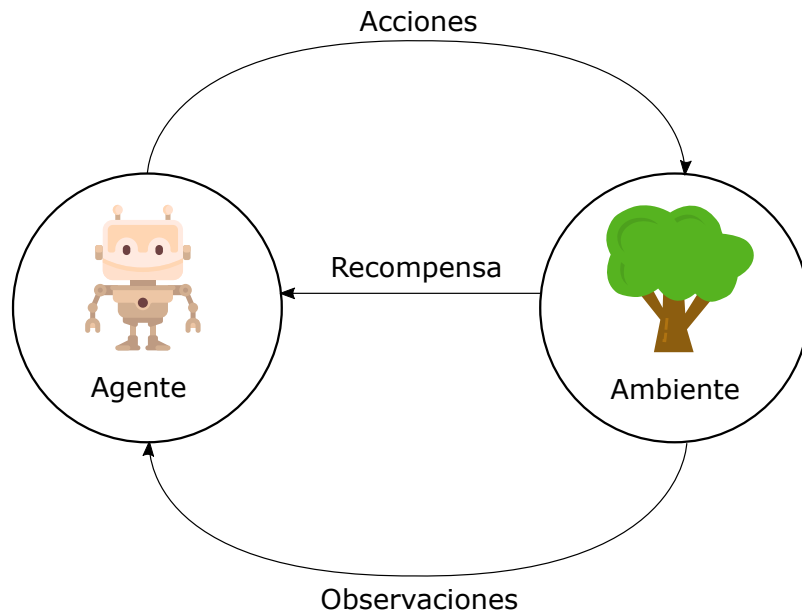


Figura 1.1: Interacción agente-ambiente.

Esta interacción se puede visualizar gráficamente en la Figura 1.1. Si los objetivos del agente están definidos en función de las recompensas inmediatas, la tarea del agente se reduce a encontrar un comportamiento que le permita decidir en cada estado

qué acción tomar para maximizar los valores de la señal de recompensa acumulados a largo plazo (o retorno). Entonces, la solución del Proceso de Decisión Markov se reduce a encontrar una política

$$\pi^*: S \rightarrow A \quad (1.1)$$

donde S denota el conjunto de estados del ambiente y A el conjunto de acciones, que en cada paso del tiempo t , en el estado s_t elige la acción a_t que maximiza el retorno esperado. Una formulación frecuente de retorno esperado es el retorno decrementado esperado, en el que se suma cada recompensa pero se decrementa geométricamente de acuerdo a un *factor de descuento* γ ($0 \leq \gamma < 1$) y el número de pasos transcurridos hasta que se recibió la recompensa:

$$\begin{aligned} E \{R_t\} &= E \{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots\} \\ &= E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \right\} \end{aligned} \quad (1.2)$$

Para obtener esta política óptima π^* , algunos métodos la derivan directamente mediante enfoques de *descenso del gradiente* (como *policy gradient*), mientras que otros la obtienen aprendiendo funciones de valor sobre los estados o las acciones (llamadas V^* y Q^*) y derivando a partir de ellas la política óptima. Una explicación detallada sobre las funciones de valor se dará en la sección 3.2.

Los MDPs no son un concepto nuevo, de hecho, se conocen desde por lo menos los años 50 [5]. Han sido la base de los métodos de programación dinámica¹ que resuelven relaciones de recurrencia llamadas *ecuaciones de Bellman*. También son el fundamento para los métodos de aprendizaje por refuerzo² [42], como por ejemplo

¹La programación dinámica es una técnica que consiste en dividir un problema en sub-problemas más pequeños y resolverlos recursivamente.

²Se utilizarán *aprendizaje por refuerzo*, *reinforcement learning* y *RL* de forma intercambiable.

Q-Learning, que han sido aplicados en innumerables problemas de control en el área de robótica [22], autos y vehículos voladores autónomos [49], sistemas de diálogo y *chatbots* [11][24],[4] y sistemas de recomendación [1][20]. Particularmente, uno de los dominios donde se ha mostrado la alta efectividad del aprendizaje por refuerzo es en el de desarrollo de agentes que aprenden a jugar juegos, que es el tipo de tarea en que se centrará este trabajo.

Se podría pensar que entrenar agentes que jueguen juegos con un alto desempeño no tiene ningún tipo de aplicación fuera de este dominio, sin embargo, en una publicación realizada recientemente por *DeepMind* [17], se muestra que es posible plantear problemas en forma de juego para que un agente, entrenado utilizando técnicas similares a aquellas utilizadas para entrenar agentes que juegan juegos, pueda resolverlos. Particularmente, se plantea el problema de encontrar algoritmos más rápidos de multiplicación de matrices que los ya existentes, es decir, se muestra la posibilidad de descubrir nuevos algoritmos utilizando aprendizaje por refuerzo profundo.

Ahora bien, tratar de lograr que las computadoras alcancen un desempeño equiparable o superior al de un humano en juegos no es un concepto para nada nuevo. Trabajos como el programa para jugar a las damas de Arthur Samuel cuentan con publicaciones realizadas en los años 50 [36][37]. Más recientemente, se encuentra *TD-Gammon* [43], hecho para jugar backgammon y la computadora *Deep Blue* [8] desarrollada por IBM que, en el año 1997, se convirtió en la primera computadora en ganarle un partida de ajedrez a un campeón mundial. *TD-Gammon* utilizó redes neuronales entrenadas con diferencia temporal (*TD* en su nombre hace referencia a *temporal difference*) y obtuvo un desempeño similar al de los mejores jugadores de backgammon del momento, por lo que se considera como uno de los éxitos más conocidos del aprendizaje por refuerzo.

TD-Gammon sirvió como base para un algoritmo que marcó un resurgimiento en el *reinforcement learning* en los últimos años: *Deep Q-Learning* (DQN) [26]. A diferencia

de los enfoques tradicionales como *Q-Learning*, que aprenden la función de valor Q^* asumiendo una representación tabular de la misma, DQN asume una representación neuronal profunda de varias capas convolucionales, que aprende no sólo los valores de la función Q^* sino también la representación de los estados en varios niveles de abstracción, como se muestra en la Figura 1.2. Detalles sobre la función Q^* serán dados más adelante, en la Sección 3.2, por el momento alcanza con conocer que es una función, por ende, puede ser aproximada mediante diversos métodos y que, a partir de ella, se puede derivar la política óptima π^* que se mencionó anteriormente como solución a un MDP.

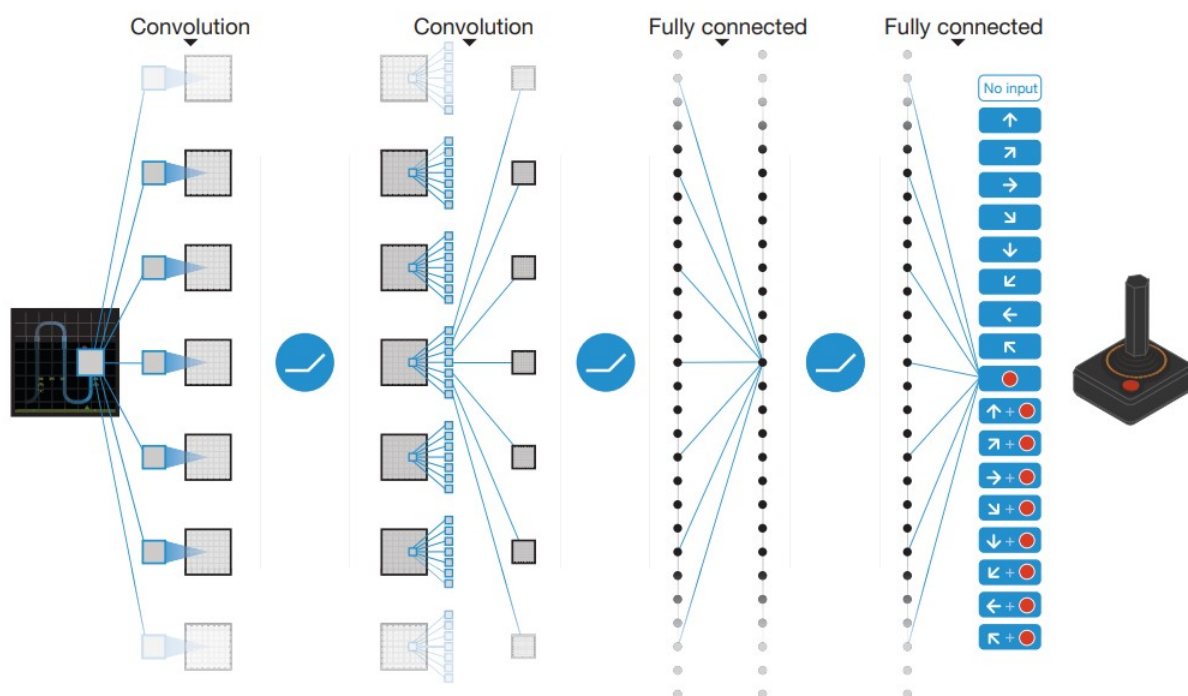


Figura 1.2: Esquema de la representación neuronal de *Deep Q-Networks* para un mundo estocástico simple.

Luego de su publicación en 2013, se presentaron diversas variantes de DQN [44][38][46], así como también nuevos algoritmos de aprendizaje por refuerzo [25][39]. Además se

produjo un surgimiento de sistemas que lograron derrotar a los mejores jugadores en diversos juegos [28][40][45].

Un aspecto a remarcar de los métodos como *Deep Q-Learning*, al ser aplicados a videojuegos, es que la forma de aprendizaje del agente es muy similar a la de los humanos: se conocen los controles, pero las reglas sobre cómo funciona el mundo presentado por el juego y el objetivo a cumplir deben ser descubiertos jugando repetidas veces (siempre bajo el supuesto de que no se conocen estos aspectos de antemano, cosa que es bastante factible en los juegos de los tiempos del *arcade*). Esto implica que el agente no tiene acceso al estado interno del juego, por lo que lograr un desempeño superior o igual al de un jugador humano en una variedad de escenarios, con reglas y objetivos distintos, resulta un aspecto desafiante.

Es interesante observar que *DeepMind*, en su discusión acerca de DQN [27], menciona el hecho de que no se realizó una optimización de hiperparámetros ya que computacionalmente sería muy demandante, sino que se llevó a cabo una búsqueda informal. Los hiperparámetros son valores que se utilizan para controlar el proceso de aprendizaje del modelo³ y pueden afectar el desempeño del mismo a la hora de realizar la tarea para la cual fue entrenado, así como también la velocidad con la que se produce el entrenamiento. En términos simples, no se realizó una búsqueda de los hiperparámetros que produzcan un modelo que obtenga los mejores resultados posibles, sino que se hicieron pruebas de distintos valores hasta encontrar un conjunto de hiperparámetros “robustos” que garanticen buenos resultados. Esta decisión, si bien es válida dentro del punto de vista ingenieril, no permitió hacer un análisis más sólido desde un punto de vista científico y que permitiera apreciar cuál es el verdadero impacto que tienen las distintas decisiones a la hora de elegir los hiperparámetros.

En la carrera de Licenciatura en Ciencias de la Computación, si bien se dictan algunos aspectos básicos de los agentes basados en utilidad y del aprendizaje por

³Generalmente, cuando se habla de *deep reinforcement learning*, se puede utilizar *modelo* y *agente* de forma intercambiable.

refuerzo, los mismos son introductorios y restringidos a “problemas de juguete”, que pueden ser abordados mediante representaciones tabulares de las funciones de valor con espacios de estados y acciones sencillos. Sin embargo, los problemas del mundo real, como los abordados en este trabajo final, presentan varios aspectos desafiantes e interesantes para su análisis.

Por un lado, ya no se puede asumir que el agente recibe un estado “Markov perfecto” (perfectamente observable⁴), sino que la efectividad de las decisiones estará dada por la posibilidad de considerar también 3 o más observaciones previas para, por ejemplo, identificar el movimiento de una pelota o proyectil. Algunas bibliotecas, como *StableBaselines3* [32], permiten especificar este aspecto mediante hiperparámetros como el *framestack*, que establece el número de *frames* (imágenes del juego) a apilar para proporcionar como entrada a la red neuronal.

Por otra parte, al usar redes neuronales profundas en lugar de representaciones tabulares de las funciones de valor, las garantías teóricas de convergencia ya no se mantienen y existe una mayor inestabilidad durante el aprendizaje. Dos métodos propuestos por *DeepMind* para superar estos problemas son *experience replay* y *target network* (o red objetivo). La idea básica de *experience replay* es actualizar la *Q-Network* que aproxima a Q^* utilizando un subconjunto aleatorio de experiencias almacenadas en una memoria de replay, en lugar de solamente utilizar la experiencia más reciente. Esto produce un aprendizaje más suave y evita la divergencia, mejorando la estabilidad. Con *target network*, por su parte, se mantienen 2 *Q-Networks*: la que aproxima a Q^* y una *target network*. Cada C pasos de actualización de los pesos de la *Q-Network*, se insertan dichos valores de pesos en otra *Q-Network* (es decir, la *target network*) y se mantienen fijos durante las siguientes C actualizaciones de la *Q-Network* original. Al igual que para el caso de *framestack*, existen hiperparámetros que permiten ajustar el rol que estos conceptos juegan dentro del aprendizaje.

⁴Un ambiente se dice totalmente o perfectamente observable cuando el agente tiene acceso a su estado completo a través de las percepciones que recibe [35].

Como fue expresado previamente, el análisis original de la propuesta de DQN se limitó a usar un conjunto fijo de hiperparámetros aceptables para todos los juegos, sin realizar ningún tipo de consideración en cómo afecta la complejidad del juego en cada uno de ellos, por ejemplo, cómo afecta la velocidad de aprendizaje, la forma de exploración, la incidencia a largo plazo de las acciones (valor de γ), entre muchas otras cosas. Este trabajo final apunta a cubrir esta brecha de investigación.

§1.2 Objetivo e hipótesis de investigación

La hipótesis de investigación para este trabajo es que el análisis metódico y exhaustivo del rol de los distintos hiperparámetros que afectan el aprendizaje en modelos de aprendizaje por refuerzo profundo, como DQN, permitirá establecer relaciones más claras entre la complejidad de los juegos, la efectividad del modelo, la velocidad de aprendizaje, la observabilidad del ambiente y distintos aspectos relacionados a la estabilidad del aprendizaje de una red neuronal profunda.

Así, el objetivo general consistirá en estudiar, analizar y experimentar con bibliotecas de amplio uso en aplicaciones industriales del aprendizaje por refuerzo, como OpenAI Gym [7] y StableBaselines3 [32], para analizar el rol que juegan los principales hiperparámetros usados en las mismas a la hora de aprender las políticas de decisión para juegos interactivos de distinta complejidad.

Alcanzar este objetivo general involucrará el cumplimiento de algunos objetivos más específicos:

1. Estudiar, analizar y poner en funcionamiento sistemas que permitan especificar distintos ambientes para el aprendizaje por refuerzo y los correspondientes soportes que implementan los algoritmos de *deep reinforcement learning* del

estado del arte.

2. Seleccionar distintos juegos, con niveles de complejidad variados, y distintos hiperparámetros que puedan afectar el proceso de aprendizaje de los mismos, como el grado de observabilidad del ambiente, la velocidad de aprendizaje, la tasa de exploración y el uso de *experience replay*, entre otros.
3. Entrenar, analizar y comparar los resultados obtenidos en estos casos con los resultados reportados en la literatura.
4. Evaluar la capacidad del aprendizaje por refuerzo para aprender modelos para estas tareas y en tareas similares que se puedan abordar en el futuro.

§1.3 Organización del informe

El informe está compuesto de 6 capítulos, siendo el capítulo actual la introducción al problema elegido para abordar durante este trabajo.

El capítulo II brinda una introducción al aprendizaje por refuerzo, su definición, sus componentes, y una comparación con otros tipos de aprendizaje existentes en el campo de la inteligencia artificial.

El capítulo III detalla el formalismo utilizado para modelar problemas de *reinforcement learning*, los procesos de decisión Markov.

El capítulo IV se centra en la teoría de distintos enfoques, al menos los más reconocidos y relevantes para el tipo de ambiente con el que se trabajará, que pueden ser utilizados para aprender la política (es decir, la solución del MDP).

El capítulo V describe los principales aspectos del trabajo experimental: detalles acerca de las bibliotecas utilizadas, los distintos modelos y sus resultados, así como también un análisis de dichos resultados.

En el capítulo final, el número VI, se detallaran las conclusiones del desarrollo de este trabajo final, la relación de los conceptos utilizados en él con los vistos durante la carrera de Licenciatura en Ciencias de la Computación, y el trabajo futuro que se podría realizar en esta temática, a modo de continuación.

□

CAPÍTULO II

INTRODUCCIÓN AL APRENDIZAJE POR REFUERZO

La idea de aprendizaje mediante la interacción con el ambiente es probablemente la primera en la que se piensa cuando se habla de la naturaleza del aprendizaje humano. Estas interacciones proveen una gran cantidad de información acerca de causa y efecto, las consecuencias de las acciones y qué hacer para alcanzar objetivos. Sin importar lo que se esté aprendiendo, se está extremadamente consciente de cómo el ambiente responde a lo que se hace y se busca influenciar lo que sucede mediante el comportamiento.

En este capítulo, se realizará una introducción al aprendizaje por refuerzo, el equivalente computacional al aprendizaje mediante la interacción. Se incluirán conceptos fundamentales del campo de la inteligencia artificial, como lo son los de agente y ambiente. Además, se mostrarán las diferencias y similitudes del aprendizaje por refuerzo respecto a otros enfoques dentro del aprendizaje automático¹, particularmente el aprendizaje supervisado y el aprendizaje no supervisado.

¹Se utilizarán su equivalente en inglés y la abreviatura del mismo (*machine learning* y *ML*) además de *aprendizaje automático* de forma intercambiable.

§2.1 Agentes y ambientes

En su forma más básica, una tarea dentro del *machine learning* siempre constará de al menos un *agente* que interactúa con un *ambiente*, es decir, lo percibe y actúa sobre él.

Definiremos formalmente a un *agente* como un sistema (o entidad) físico o virtual, *situado* en algún ambiente, que es capaz de actuar de manera *autónoma* y *flexible* en este ambiente a los fines de lograr los *objetivos* que le han sido delegados.

En cuanto al ambiente, es todo lo que se encuentra fuera del alcance directo del agente. El comportamiento del agente es afectado fuertemente por el ambiente en el que se encuentra: el agente percibe el ambiente a través de sensores y actúa mediante sus efectores, modificando el ambiente y sus percepciones futuras.

Dos aspectos importantes de la definición de agentes son la *autonomía* y la *flexibilidad*. La autonomía implica que el agente debe ser capaz de actuar sin la intervención directa de humanos y otros agentes, que tiene control de su propio estado interno, sobre sus acciones y puede, si es necesario, modificar su comportamiento en base a la experiencia (es decir, aprender). La flexibilidad implica que el agente es *reactivo* (es decir, percibe el ambiente y responde en un tiempo adecuado a los cambios que se producen en él), pero también es *proactivo* (puede tomar la iniciativa, no solo actuar en respuesta al ambiente); además, puede *interactuar* con otros agentes artificiales o humanos, de ser necesario.

Definidos estos conceptos presentes en prácticamente todas las tareas de aplicación de ML, se pasará a dar una noción de tres enfoques dentro de él: el aprendizaje *supervisado*, el aprendizaje *no supervisado* y el aprendizaje *por refuerzo*; que es el enfoque en el cual se basa este trabajo.

§2.2 Aprendizaje supervisado

En el aprendizaje supervisado, se responde la pregunta de cómo construir, automáticamente, una función que relacione una entrada con una salida dado pares (*entrada, salida*) como ejemplos. Dentro del aprendizaje supervisado, existen dos grandes grupos de problemas: *clasificación* y *regresión*.

En los problemas de *clasificación*, el objetivo es predecir una *clase*, dentro de una lista de clases posibles predefinidas de antemano. Por ejemplo, se pueden clasificar imágenes de perros, gatos e iguanas; donde las clases serían “perro”, “gato” e “iguana”. Como segundo ejemplo, se puede considerar la clasificación de emails en “spam” o “no spam”. Este último caso, se conoce como *clasificación binaria*, ya que solo se cuenta con dos clases posibles; mientras que el ejemplo anterior corresponde a una *clasificación multiclase*.

En las tareas de *regresión*, el objetivo es predecir un número continuo (o número real, en términos matemáticos). Un ejemplo de este tipo de tarea es predecir la temperatura ambiente de mañana dada la información de sensores meteorológicos.

La idea, entonces, del aprendizaje supervisado es que se dispone de ejemplos de entrada con su respectiva salida deseada y se quiere aprender cómo generar la salida para una entrada futura que todavía no se ha observado.

§2.3 Aprendizaje no supervisado

En contraste con el aprendizaje supervisado, se encuentra el aprendizaje no supervisado, en el que no se dispone de “etiquetas” en los datos (es decir, no se cuenta con la salida deseada para la entrada). Aquí, el objetivo principal es aprender una

estructura latente en los datos con los que se está trabajando. Un ejemplo común es el *clustering*, donde el algoritmo agrupa los datos de entrada en *clusters*, que pueden verse como subconjuntos disjuntos de los datos de entrada, en los que existe algún tipo de relación entre ellos.

§2.4 Aprendizaje por refuerzo

El aprendizaje por refuerzo, en inglés *reinforcement learning (RL)*, es aprender *qué hacer* (cómo relacionar situaciones con acciones) para maximizar una señal de recompensa numérica. No se le dice al agente qué acción tomar, sino que debe descubrir qué acciones producen la mayor recompensa mediante *prueba y error*. En los casos más complejos, las acciones pueden afectar no solo la recompensa inmediata, sino también la situación siguiente y, a través de esto, todas las recompensas subsiguientes. Estas características, la búsqueda por prueba y error y la recompensa futura (en inglés *delayed reward*), son las más distintivas del *reinforcement learning*.

El aprendizaje por refuerzo es distinto del aprendizaje supervisado en que no se cuenta con datos etiquetados que le digan al agente qué acción debe tomar, sino que debe descubrirlas por prueba y error (es decir, interactuando con el ambiente en el que está situado). Además, se distingue del aprendizaje no supervisado en que no intenta encontrar una estructura latente en datos de entrada dados, sino que se busca, como se mencionó anteriormente, maximizar una señal de recompensa. Por estos motivos, se considera al *reinforcement learning* como un tercer paradigma dentro del *machine learning*.

Uno de los desafíos que se presentan en RL, y no en otros tipos de aprendizaje, es el *trade-off* entre *exploración* y *explotación*. Esto es, para obtener una gran cantidad de recompensa, el agente debe tener una preferencia por las acciones que, al haberlas

tomado en el pasado, fueron efectivas en obtener recompensa. Sin embargo, para descubrir estas acciones, debe probar acciones que no han sido tomadas anteriormente. En otras palabras, el agente tiene que *explotar* el conocimiento obtenido para obtener una recompensa, pero también debe *explorar* para poder tomar mejores decisiones acerca de qué acción tomar en el futuro. Se produce un dilema, ya que debe producirse un balance entre explotación y exploración: el agente debe probar una variedad de acciones y favorecer aquellas que parecen ser las mejores.

Los problemas de aprendizaje por refuerzo se formalizan mediante procesos de decisión Markov², de los que se habló brevemente en la Sección 1.1, y serán discutidos en detalle en el Capítulo III. Por el momento, es suficiente con saber que son utilizados porque permiten capturar los aspectos más importantes del problema real, haciendo que un *agente* interactúe a lo largo del tiempo con un *ambiente* para conseguir un *objetivo*. El agente debe poder percibir el *estado* del ambiente y debe poder tomar *acciones* que afecten ese estado. Además, el agente debe tener uno o más *objetivos* relacionados con el estado del ambiente. Entonces, los MDPs permiten modelar estos tres aspectos del problema (percepción, acción y objetivo) de forma simple. [42]

§2.4.1 Componentes del aprendizaje por refuerzo

Al hablar de *reinforcement learning* hasta el momento, se han mencionado algunos de sus componentes, aquí se pasará a detallar cada uno de ellos. Dichos componentes son: el *agente*, el *ambiente*, la *política*, la señal de *recompensa*, la *función de valor* y, opcionalmente, un *modelo* del ambiente.

Como se mencionó en la Sección 2.1, el ***agente*** es una entidad física o virtual que interactúa con el ambiente ejecutando ciertas acciones, haciendo observaciones y recibiendo recompensas del ambiente por sus acciones. Por ejemplo, un agente puede

²Se recuerda que se referirá a este concepto, además, como MDPs.

ser un *software* que juega al ajedrez, un robot que debe atravesar un laberinto o, incluso, una persona.

El ***ambiente*** es el “resto del universo”, es decir, todo lo que se encuentra fuera del agente en la tarea considerada. Puede ser físico, como una calle y autos, o virtual, como un juego de computadora.

Una ***política*** define la forma en la que se comporta el agente. Como se mostró en la Sección 1.1 del capítulo anterior, esta política establece una relación que indica qué acción tomar en cada estado del ambiente.

Una señal de ***recompensa*** define el objetivo de un problema de aprendizaje por refuerzo. En cada paso de tiempo, el ambiente le envía al agente un número llamado *recompensa*³. El objetivo del agente es maximizar la recompensa total que recibe a largo plazo. Así, la señal de recompensa define cuáles son los eventos buenos y malos para el agente. Además, es la base para la modificación de la política: si una acción elegida por la política obtiene una recompensa muy baja (o incluso, negativa), entonces dicha política puede ser modificada para seleccionar otra acción cuando se presente esa situación en el futuro.

Mientras que la señal de recompensa indica qué es bueno en el sentido inmediato, una ***función de valor*** especifica qué es bueno a largo plazo. El *valor* de un estado es la cantidad total de recompensa que el agente puede esperar acumular en el futuro comenzando desde ese estado. En otras palabras, mientras que las recompensas indican la deseabilidad inmediata de estados del ambiente, el valor indica la deseabilidad a largo plazo de los estados, teniendo en cuenta los posibles estados siguientes y la recompensa disponible en ellos. El agente busca acciones que produzcan estados de mayor *valor*, no mayor *recompensa*, ya que dichas acciones llevan a obtener una mayor cantidad de recompensa a largo plazo. Estos valores deben ser estimados (y re-estimados) a partir de las secuencias de observaciones que el agente realiza, a

³Cabe destacar que este número puede ser positivo o negativo. En este último caso, correspondería a una penalización.

diferencia de las recompensas, que son provistas directamente por el ambiente.

El último componente de algunos sistemas de RL es un **modelo** del ambiente. Esto es, una representación que imita el comportamiento del ambiente y que permite inferir de qué forma se comportará. Por ejemplo, dado un estado y una acción, el modelo puede predecir el estado siguiente y la recompensa obtenida. Los modelos son utilizados para *planificar*, que se refiere a cualquier forma de decidir un curso de acción considerando situaciones futuras posibles antes de que ocurran. Los métodos de aprendizaje por refuerzo que utilizan modelos y planificación son llamados métodos *basados en modelo* y, aquellos que sólo utilizan prueba y error, se conocen como *libres de modelo*.

En la Figura 2.1 se muestra una representación esquemática del funcionamiento de la interacción del agente con el ambiente en el aprendizaje por refuerzo: el agente percibe el estado del ambiente a través de observaciones, selecciona una acción de acuerdo a la política aprendida, se modifica el estado del ambiente y recibe el valor de recompensa correspondiente por haber tomado la acción.

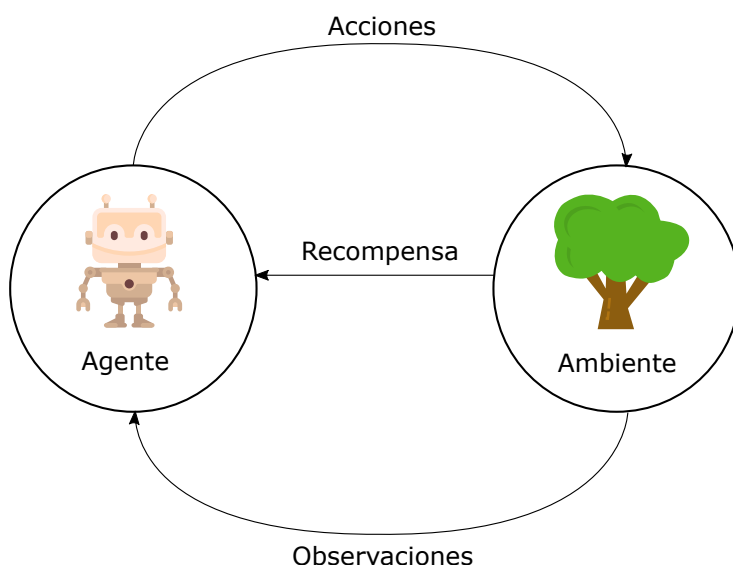


Figura 2.1: Esquema de la interacción de los componentes de *reinforcement learning*.

En el capítulo siguiente, se discutirá el modelo utilizado para representar esta interacción de manera formal.

□

CAPÍTULO III

PROCESOS DE DECISIÓN MARKOV

Los procesos de decisión Markov son una formalización clásica de *problemas de decisión secuencial*, donde las acciones no solo influyen en la recompensa inmediata, sino también en los estados subsiguientes y, a través de ellos, las recompensas futuras. Así, los MDPs implican un balance entre la recompensa inmediata y la recompensa futura. Una propiedad importante que deben satisfacer los problemas a modelar es la *propiedad Markov*, que establece que un estado en tiempo $t + 1$ sólo depende del estado en tiempo t , es decir, es independiente de los estados anteriores o, en otras palabras, el futuro sólo depende del presente y no del pasado. [42]

§3.1 Procesos de decisión Markov

En los MDPs, el agente y el ambiente interactúan continuamente de la misma forma en la que se ha discutido hasta aquí, de todas formas, se procede a especificar la interacción de manera formal. Más formalmente, en cada paso de tiempo¹ t , el agente recibe una representación del *estado* del ambiente $s_t \in S$, y selecciona una acción $a_t \in A$ aplicable en el estado s_t . Un paso de tiempo después, el agente recibe una

¹El tiempo es representado por una secuencia *abstracta* y discreta de pasos $t = 0, 1, \dots$. Estos pasos no representan necesariamente intervalos fijos del tiempo real.

recompensa numérica $r_{t+1} \in \mathbb{R}$ por la acción tomada previamente y el ambiente pasa a estar en un nuevo estado s_{t+1} .

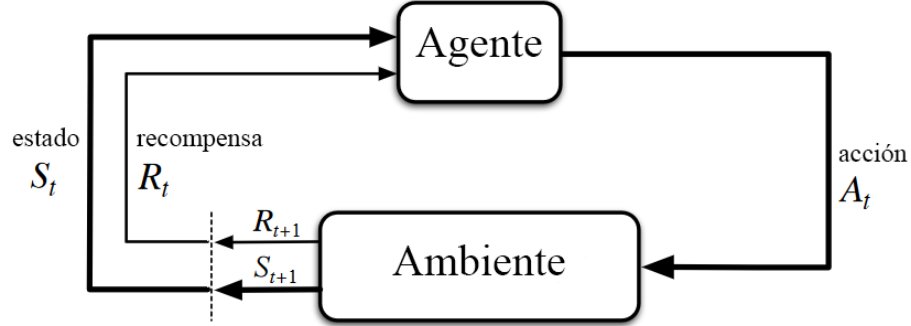


Figura 3.1: Interacción agente-ambiente en un proceso de decisión Markov.

Esta interacción se puede visualizar gráficamente en la Figura 3.1

Los procesos de decisión Markov consisten de:

- S , un conjunto de *estados* posibles del ambiente.
- A , un conjunto de *acciones* posibles que el agente puede tomar.
- $P(s, a, s') = \Pr \{s_{t+1} = s' | s_t = s, a_t = a\}$, las *probabilidades de transición*. Es decir, la probabilidad de que el agente se encuentre en el estado s' en el tiempo $t + 1$, si en el tiempo t ejecuta la acción a en el estado s .
- $R(s, a, s') = E \{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$, las *recompensas esperadas*. Es decir, la recompensa *esperada* en el tiempo $t + 1$ si en el tiempo t el agente realiza la acción a en el estado s y se produce la transición al estado s' en el tiempo $t + 1$.

Tanto las probabilidades de transición como las recompensas esperadas están definidas para todo $s, s' \in S$ y $a \in A(s)$, donde $A(s)$ es el conjunto de acciones aplicables en s .

Un aspecto importante a remarcar sobre los MDPs es que, al utilizar probabilidades de transición, permiten modelar problemas en los que el resultado de las acciones puede no ser el esperado, es decir, problemas en los que existe incertidumbre².

Debido a que las acciones podrían no tener el resultado esperado, simplemente dar una secuencia de acciones para que el agente ejecute no servirá de mucho, ya que podrían no conducir siempre a un estado terminal (es decir, a alcanzar el objetivo). Esto da la idea de que la solución debe decirle al agente qué debería hacer en cada estado en el que podría encontrarse. Una solución de este tipo se conoce como *política*. Entonces, la solución de un proceso de decisión Markov se reduce a encontrar una política

$$\pi^*: S \rightarrow A \quad (3.1)$$

que, en cada paso de tiempo t , en el estado s_t elige la acción a_t que maximiza el *retorno esperado*. El retorno, que se denotará como R_t , se define en función de la secuencia de recompensas $r_{t+1}, r_{t+2}, r_{t+3}, \dots$ acumulándolas en base a algún criterio particular. Una formulación frecuente de retorno esperado es el *retorno decrementado esperado*, en el que se suma cada recompensa pero se decrementa geométricamente de acuerdo a un *factor de descuento* γ ($0 \leq \gamma < 1$) y el número de pasos transcurridos hasta que se recibió la recompensa. En este caso, se dice que R_t es el *retorno decrementado* y que el agente maximiza el *retorno decrementado esperado*:

$$\begin{aligned} E\{R_t\} &= E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots\} \\ &= E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right\} \end{aligned} \quad (3.2)$$

²En términos de propiedades de ambientes: ambientes *estocásticos*.

§3.2 Funciones de valor

Como ya se estableció, la solución de un MDP se basa en encontrar una política que maximice el retorno esperado. Una de las formas de encontrar esta política se basa en estimar *funciones de valor*, de las que se habló brevemente en la Subsección 2.4.1 como un componente del aprendizaje por refuerzo. Las funciones de valor están definidas sobre el conjunto de estados (o sobre los de estados y acciones) y estiman qué tan bueno es para el agente estar en un estado (o qué tan bueno es ejecutar una acción en dicho estado). La noción de “cuán bueno” está dada en función del retorno esperado. Claramente, las recompensas que obtendrá el agente en el futuro dependerán de qué acciones realice, y las acciones que realiza, a su vez, dependen de la política que esté siguiendo. Entonces, esto da la idea de que las funciones de valor se definen con respecto a políticas particulares. Formalmente, una política es una función

$$\pi : S \rightarrow PD(A) \quad (3.3)$$

donde $PD(A)$ denota el conjunto de distribuciones de probabilidades sobre A . En otras palabras, dado un estado, indica las probabilidades de seleccionar cada acción posible.

El *valor de un estado s bajo una política π* , que se denotará como $V^\pi(s)$, es el retorno esperado cuando se comienza en el estado s y se sigue la política π . Formalmente

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \quad (3.4)$$

donde $E_\pi \{\cdot\}$ denota el valor esperado dado que el agente siga la política π . Se llamará a la función V^π la *función de valor-estado para la política π* .

Similarmente, el *valor de la acción a en el estado s* , que se denotará como $Q^\pi(s, a)$, es el retorno esperado cuando se comienza en el estado s , se toma la acción a y luego se sigue la política π . Formalmente

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a \right\} \quad (3.5)$$

se llamará a la función Q^π la *función de valor-acción para la política π* .

Hasta aquí, se han definido funciones de valor para políticas π arbitrarias. Sin embargo, en MDPs, el objetivo es encontrar una política óptima π^* . La política óptima π^* será aquella cuyo retorno esperado para todos los estados sea mayor o igual que el retorno esperado de cualquier política posible. En otras palabras, π^* es óptima si y sólo si $V^{\pi^*}(s) \geq V^\pi(s)$ para todo $s \in S$ y $\pi \in \Pi$, donde Π es el espacio de políticas posibles. Si bien puede haber más de una política óptima posible, se denotará a todas ellas como π^* , y todas comparten la misma *función de valor-estado óptima*, que se denotará como V^* , y se define como:

$$V^*(s) = \max_{\pi} [V^\pi(s)] \quad (3.6)$$

para todo $s \in S$.

Las políticas óptimas también comparten la misma *función de valor-acción óptima*, que se denotará como Q^* y se define como:

$$Q^*(s, a) = \max_{\pi} [Q^\pi(s, a)] \quad (3.7)$$

para todo $s \in S$ y $a \in A(s)$.

Entonces, V^* indica para cada estado cuál es el retorno esperado máximo que se puede obtener desde él, es decir, cuál es el retorno esperado que se obtiene siguiendo una política óptima. Por su parte, Q^* , indica el retorno esperado de tomar la acción

a en el estado s y luego seleccionar acciones de acuerdo a la política óptima.[42]

Establecidos estos conceptos sobre procesos de decisión Markov, se está en condiciones de hablar de las distintas técnicas que se pueden utilizar para encontrar una solución a los mismos. Esto es lo que concierne al capítulo siguiente.

□

CAPÍTULO IV

MÉTODOS PARA RESOLVER PROBLEMAS DE APRENDIZAJE POR REFUERZO

En este capítulo, se introducirá brevemente una clasificación de métodos utilizados para resolver problemas de *reinforcement learning*¹ y se especificará aquellos enfoques más relevantes para este trabajo. En particular, se considerarán 3 clases de métodos: *programación dinámica*, métodos *Monte Carlo* y *diferencia temporal*. Si bien dentro de estas clases existe una gran cantidad de métodos, aquí se introducirán 3 de ellos: *value iteration*², perteneciente a la clase de programación dinámica; *Q-Learning* y *Deep Q-Learning*³, ambos pertenecientes a la clase de métodos de diferencia temporal.

Como se mencionó hasta aquí, encontrar una solución para un problema de RL implica encontrar una política óptima que le indique al agente qué acción realizar en cada estado en el que se encuentre para maximizar el retorno esperado. Si bien estimar las funciones de valor, que se introdujeron en la Sección 3.2, no es la única forma de encontrar dicha política, todos los métodos que se mostrarán en este capítulo la obtienen de esta manera.

¹Cabe aclarar que, al utilizar procesos de decisión Markov para modelar problemas de RL, hablar de resolver MDPs y resolver problemas de aprendizaje por refuerzo es equivalente, por lo que se utilizan de forma intercambiable.

²Si bien este método no es directamente relevante para la experimentación realizada en este trabajo, se introduce de todas formas a modo de contraste con el resto de enfoques presentados.

³Se utilizará, además de su nombre completo en inglés, la abreviatura “DQN” para referirse a él.

§4.1 Clasificación de métodos

Se recuerda que, en la Sección 2.4.1, al hablar de las componentes del aprendizaje por refuerzo, se mencionó que uno de ellos podía ser opcional: el *modelo*. De acuerdo a su presencia o ausencia, se puede clasificar a los métodos de RL en aquellos basados en modelo y aquellos libres de modelo, respectivamente. Esta noción será útil a la hora de caracterizar las clases de métodos que se introducirán aquí.

Como se estableció al inicio, se considerarán 3 clases de métodos para resolver problemas de *reinforcement learning*: programación dinámica, métodos Monte Carlo y métodos de diferencia temporal.

El término ***programación dinámica***⁴ se refiere a una colección de algoritmos que pueden ser usados para computar políticas óptimas dado un modelo perfecto del ambiente como un proceso de decisión Markov, es decir, corresponden a métodos basados en modelo. Este *modelo perfecto* quiere decir que se deben conocer lo que se llamará las *dinámicas de paso* del ambiente, es decir, las probabilidades de transición de un estado a otro luego de tomar una acción y las recompensas esperadas en cada estado por ejecutar una acción que produzca una transición hacia otro (o $P(s, a, s')$ y $R(s, a, s')$, como se definieron en la Sección 3.1 al introducir el concepto de MDPs). Además, al contar con un modelo del ambiente, no requieren que el agente interactúe con él, por lo que se conocen como métodos *off-line*. Debido a su necesidad de un modelo perfecto y a que son muy computacionalmente costosos, su utilidad es limitada.

Los métodos ***Monte Carlo***, a diferencia de los de programación dinámica, no requieren un modelo del ambiente, sino que requieren solamente experiencia, que corresponde a *muestras* de estados, acciones y recompensas obtenidas a través de la

⁴La programación dinámica implica dividir un problema en subproblemas, resolverlos, almacenar sus resultados y utilizarlos para formular la solución al problema original.

interacción con el ambiente (o, en otras palabras, de manera *on-line*). Al no requerir un modelo, no se necesitan conocer las dinámicas de paso del ambiente, por lo que corresponden a métodos libres de modelo. Los métodos Monte Carlo se definen solamente para tareas episódicas, en las que la experiencia se divide en *episodios* que eventualmente finalizan sin importar qué acciones se seleccionen. Una vez finalizado el episodio, se realiza una actualización de la estimación de la función de valor y la política.

El aprendizaje por ***diferencia temporal*** es una combinación entre las ideas de los métodos Monte Carlo y programación dinámica. Al igual que los métodos Monte Carlo, los métodos de diferencia temporal aprenden directamente a través de la experiencia, sin necesidad de un modelo de las dinámicas de paso del ambiente (es decir, son libres de modelo); y, al igual que los métodos de programación dinámica, actualizan sus estimaciones basándose en parte en otras estimaciones aprendidas, sin esperar por un resultado final. [42]

Establecidas estas 3 grandes clases de métodos para resolver problemas de *reinforcement learning*, se pasará a tratar cada uno de los métodos específicos que se presentaron al inicio de este capítulo de manera individual.

§4.2 Value Iteration

Value Iteration es un método que corresponde a la clase de programación dinámica, por lo que requiere un modelo perfecto del ambiente, con sus dinámicas de paso⁵. En

⁵Se recuerda que las dinámicas de paso del ambiente hacen referencia a las probabilidades de transición y a las recompensas esperadas.

este método, la computación de V^* se basa en actualizar las estimaciones de funciones de valor de los estados utilizando las estimaciones de funciones de valor de estados sucesores.

La regla de actualización que utiliza Value Iteration para aproximar la función de valor-estado óptima V^* tiene la siguiente forma:

$$V_{k+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s' \in S} P(s, a, s') [R(s, a, s') + \gamma V_k(s')] \quad (4.1)$$

El algoritmo trabaja en forma iterativa, partiendo de una inicialización arbitraria, V_0 , de los valores de V de todos los estados, que es utilizada en la parte derecha de la regla de actualización para producir las nuevas estimaciones V_1 sobre la parte izquierda, que son utilizadas para obtener las estimaciones V_2 y así sucesivamente, obteniendo V_{k+1} en función de V_k hasta que una política óptima es eventualmente obtenida. Formalmente, Value Iteration requiere un número infinito de iteraciones para converger exactamente a V^* (es decir, cuando $k \rightarrow \infty$), pero en la práctica se finaliza cuando la función de valor V cambia muy poco respecto a la obtenida en la iteración anterior. [41]

En cuanto a la complejidad del algoritmo, si el problema tiene n estados y m es el mayor número de acciones disponibles en cualquier estado, una sola iteración de Value Iteration requiere a lo más $O(mn^2)$ operaciones. Esto lo hace inapropiado para problemas con espacios de estado muy grandes, como aquellos que poseen muchos juegos (por ejemplo, el *Backgammon*).

§4.3 Q-Learning

Q-Learning [48][47] corresponde a los métodos de diferencia temporal, por lo que aprende directamente de la experiencia sin necesidad de un modelo del ambiente (es decir, es libre de modelo) y, además, actualiza sus estimaciones a partir de otras previamente aprendidas. A diferencia de Value Iteration, aproxima la política óptima de forma *on-line* a partir de la experiencia obtenida por el agente en su interacción con el ambiente. Esto permite enfocarse sobre el conjunto de estados que se sabe que son más relevantes que otros con respecto al comportamiento óptimo, ya que el conjunto de estados seleccionados para actualización está dado por aquellos que el agente efectivamente visita en su interacción con el ambiente. Como su nombre lo indica, Q-Learning aprende la función Q^* , utilizando para ello la siguiente regla de actualización:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (4.2)$$

Como se puede observar, no se necesitan las dinámicas de paso del ambiente. Además, la función de valor-acción Q aprendida aproxima directamente a Q^* , la función de valor-acción óptima, independientemente de la política que se siga. La política afecta qué pares de estado-acción son visitados y actualizados. Sin embargo, todo lo que se requiere para una convergencia correcta es que todos los pares continúen siendo actualizados (es decir, que todas las acciones sean ejecutadas en cada estado un número infinito de veces). Bajo esta condición y que el factor de aprendizaje α sea decrementado apropiadamente, Q-Learning ha demostrado que sus estimaciones de Q convergen con probabilidad 1 a Q^* [47]. El algoritmo de Q-Learning completo se muestra en la Figura 4.1 [41]

1. Inicializar: Tabla $Q(s, a)$ (con 0's o valores arbitrarios) para todo $s \in S$ y para todo $a \in A(s)$.
2. Repetir (por cada episodio)
3. Inicializar s .
4. Repetir (por cada paso del episodio):
5. Elegir a desde s usando una politica derivada desde Q .
6. Ejecutar accion a , observar estado resultante s' y recompensa r .
7. $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
8. $s \leftarrow s'$
9. hasta que s es terminal

Figura 4.1: Algoritmo Q-Learning.

Si bien en la teoría el ciclo de Q-Learning se realiza infinitamente, en la práctica el aprendizaje se realiza por episodios, donde cada episodio comienza en un estado inicial hasta alcanzar alguna condición definida previamente (por ejemplo: llegar a un estado objetivo, alcanzar un estado absorbente, superar un número máximo de iteraciones, etc.).

Cuando en el algoritmo se habla de la elección de una acción a partir de una política derivada de Q , se debe tener en cuenta que se requiere una política que realice un mínimo de exploración de acciones alternativas. Un ejemplo de una política de este tipo es ϵ -*greedy*, que selecciona acciones de forma *greedy*⁶ pero, con probabilidad ϵ , elige una acción de forma aleatoria.

§4.4 Deep Q-Learning

Si bien los métodos presentados hasta aquí mostraron obtener muy buenos resultados en diversas aplicaciones [43][33][16], suelen presentar problemas cuando tratan con

⁶Una política *greedy* se basa en tomar la mejor elección en el momento, sin considerar las consecuencias futuras.

espacios de estados de gran tamaño. A menos que el conjunto de estados del ambiente sea lo suficientemente pequeño como para poder ser representado a través de una tabla, se debe utilizar alguna forma de aproximación de funciones. Este proceso requiere de representaciones que capturen los aspectos más relevantes de la tarea, necesarios para obtener un buen desempeño (en inglés, *features*⁷).

Un aproximador de funciones muy utilizado en las últimas décadas son las *redes neuronales*. En 2013, un equipo de investigadores de *DeepMind* utilizaron una red neuronal profunda para aprender representaciones de estados del ambiente y se llamó a este enfoque *deep Q-Learning* [26][27]. Si bien las redes neuronales fueron utilizadas con anterioridad en *reinforcement learning*, como en *TD-Gammon*[43], la diferencia radica en que en ellos la entrada de la red neuronal estaba dada por una representación de estados especializada, o *hecha a mano*, para cada problema al que se aplicaba. En este nuevo método, se combina Q-Learning con *redes neuronales convolucionales*⁸ *profundas* para automatizar el proceso de obtención de representaciones de estados y se llamó a dicha red *deep Q-network* (abreviadamente, DQN). En la publicación, se mostró cómo DQN puede obtener un gran desempeño en distintos problemas sin tener que utilizar representaciones específicas en cada uno de ellos. Para realizar esta demostración, se hizo que DQN aprenda a jugar a 49 juegos diferentes de *Atari 2600*⁹ y, por ende, aprenda 49 políticas distintas, pero manteniendo la misma entrada (es decir, imágenes del juego) e hiperparámetros para cada una de estas tareas.

Antes de proceder en más detalle sobre *deep Q-Learning*, se introducirá un concepto clave en él y que ya fue mencionado: las redes neuronales.

⁷Además, se referirá a ellos como “representaciones de estados” del ambiente.

⁸Las redes neuronales convolucionales son un tipo de red neuronal que toma como entrada arreglos espaciales de datos, como lo son las imágenes.

⁹Atari 2600 es una consola de juegos hogareña introducida en 1977 por Atari, Inc.

§4.4.1 Redes neuronales

Una *red neuronal* es una red de unidades interconectadas, a las que se les llama *neuronas*. Estas neuronas se organizan en capas y cada una de estas capas está conectada a la siguiente. En la Figura 4.2 se muestra una red neuronal *feedforward*, lo que quiere decir que no hay ciclos en la red. La red de la figura cuenta con una *capa de salida* que consiste de dos neuronas, una *capa de entrada* con consta de cuatro neuronas, y dos *capas ocultas*: capas que no son de entrada ni de salida. Cuando se cuenta con una gran cantidad de capas ocultas, se dice que la red es una *red nueronal profunda* (en inglés, *deep neural network*) y corresponde a una familia de métodos de machine learning que se conoce como *deep learning* (o *aprendizaje profundo*).

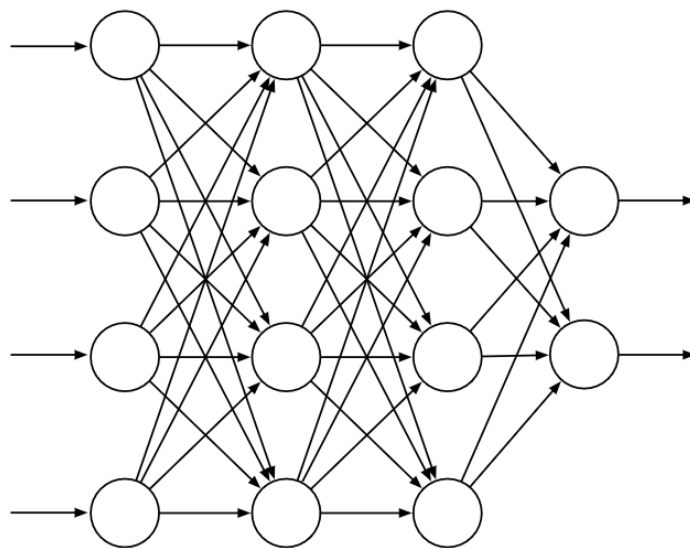


Figura 4.2: Esquema de una red neuronal *feedforward* con cuatro neuronas en la capa de entrada, 2 neuronas en la capa de salida y dos capas ocultas.

Se asocia un valor $w \in (R)$ a cada una de las conexiones entre neuronas (las flechas de la figura) al que se le llama *peso*. Un peso representa qué tan importante es esta

conexión para predecir el resultado final.

Las neuronas, por lo general, computan una suma ponderada de sus señales de entrada y luego, al resultado, le aplican una función no lineal, que se llama *función de activación*, para producir el resultado, o activación, de la neurona. Algunas de las funciones de activación más utilizadas son:

- **Sigmoid (o Sigmoide):** $f(x) = \frac{1}{1 + e^{-x}}$
- **Tangente hiperbólica (o tanh):** $f(x) = \tanh(x)$
- **ReLU (*rectified linear unit*):** $f(x) = \max(0, x)$

La forma de estas funciones se puede apreciar gráficamente en la Figura 4.3

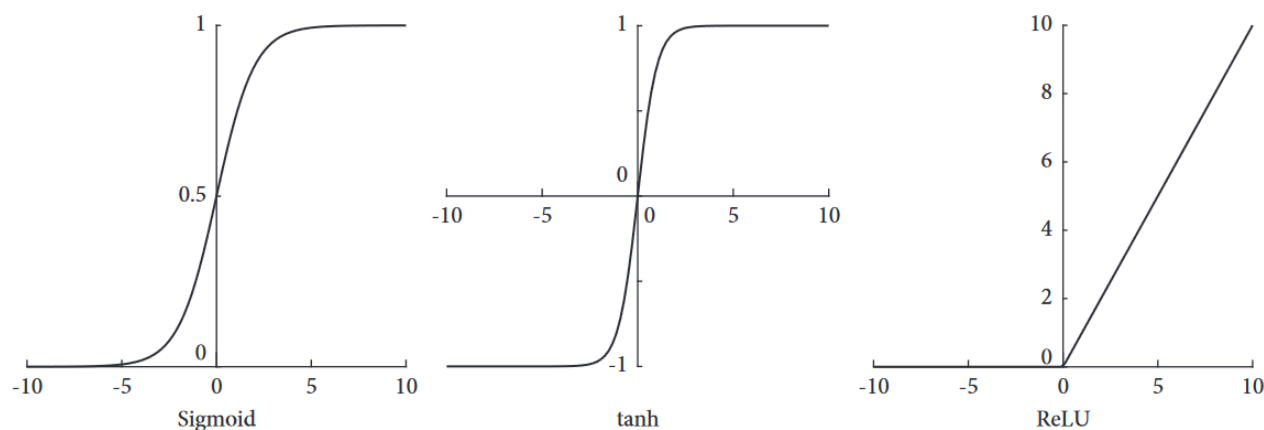


Figura 4.3: Gráficas de las funciones Sigmoide, Tangente hiperbólica y ReLU.

Es decir, cada neurona computa una suma de la forma $(x_1 * w_1 + x_2 * w_2 + \dots + x_n * w_n) + b$, donde cada x_i corresponde a una entrada de la neurona, w_i al peso de cada conexión y b es un valor que se llama sesgo (o *bias*) que se utiliza para ajustar la función de activación hacia la izquierda o la derecha. Luego, al resultado de dicha suma, le aplica alguna función de activación para producir la salida de dicha neurona.

El ajuste de estos valores de pesos w y de sesgo b se realiza utilizando un algoritmo conocido como *backpropagation* [34]. En términos simples, *backpropagation* consiste de alternar barridos hacia adelante y hacia atrás en la red neuronal ajustando tanto los valores de w como los de b para minimizar una medida de la diferencia entre el resultado obtenido por la red y el resultado esperado (es decir, el *error*).

Una clase de redes neuronales que resultó ser particularmente útil en el aprendizaje por refuerzo son las *redes neuronales convolucionales*, un tipo de redes cuya entrada es usualmente una imagen. Como puede verse en la Figura 4.4, su arquitectura consta de 2 grandes partes: la extracción de características y la clasificación, donde ésta última corresponde con la arquitectura de una red neuronal normal.

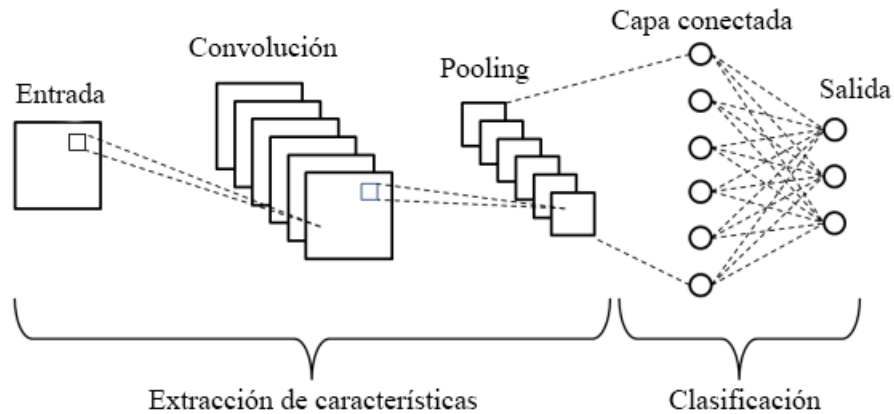


Figura 4.4: Arquitectura de una red neuronal convolucional.

Las nuevas capas distintivas que se introducen son las capas *convolucionales* y las de *pooling*. En las capas convolucionales, se realiza la operación de *convolución*, que consiste en calcular el producto punto entre la matriz que conforma a la imagen de entrada y una nueva matriz a la que se le llama *kernel* para obtener una representación más pequeña de la imagen, llamada *mapa de activación*. En la Figura 4.5 puede verse

un ejemplo de esta operación.

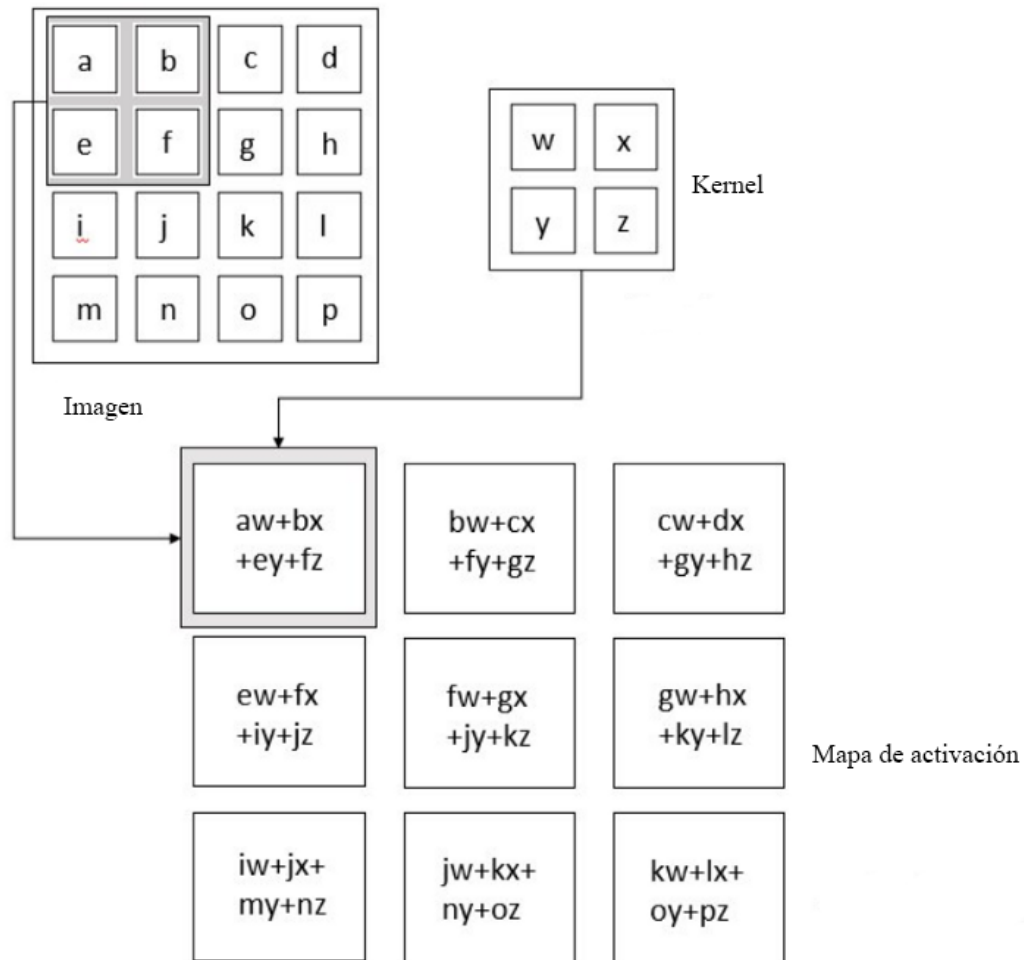


Figura 4.5: Convolución para una imagen representada por una matriz 4x4 y un kernel 2x2.

Al resultado de las capas convolucionales, al igual que en las redes neuronales normales, se le suele aplicar una función de activación. La diferencia es que, en este caso, se aplicaría sobre el mapa de activación.

Luego de la capa convolucional, se encuentra la capa de *pooling*, que cuenta con una matriz llamada *filtro* y un valor de *stride* (también llamado paso o zancada) que indica la cantidad de posiciones que debe moverse el filtro sobre el mapa de activación.

La operación más común en la capa de pooling es el *max pooling*, que puede verse gráficamente en la figura 4.6, y consiste en realizar “pasadas” del filtro sobre el mapa de activación, tomar el valor más alto de la submatriz que se forma y moverse a la siguiente submatriz una cantidad de pasos dada por *stride*.

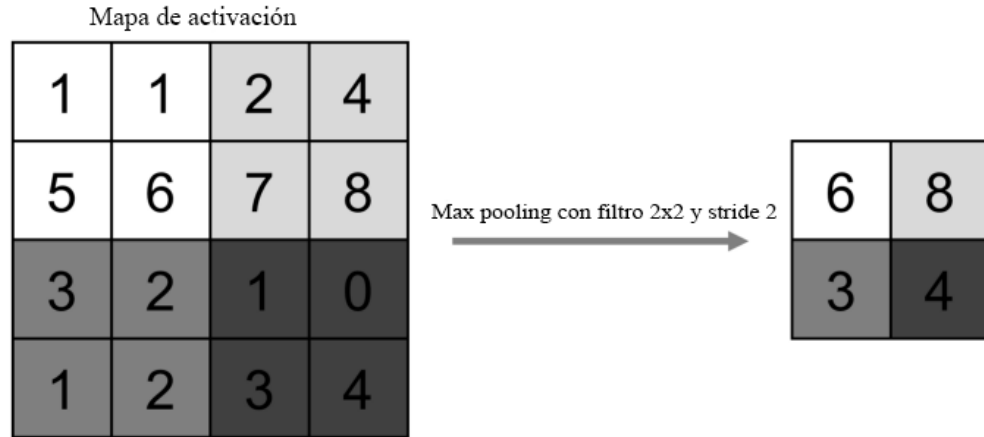


Figura 4.6: *Max pooling* para un mapa de activación 4x4, filtro 2x2 y stride 2.

Habiendo presentado esta introducción a algunos de los conceptos que hacen a las redes neuronales, se pasará a hablar más en detalle sobre *deep Q-Learning*.

§4.4.2 Deep Q-Learning

Como se mencionó en la introducción a esta sección, *deep Q-Learning* es un enfoque que consiste en utilizar *deep Q-Networks*, es decir, una combinación de redes neuronales convolucionales con Q-Learning, para que el agente aprenda representaciones de estados del ambiente y, de esta forma, no depender de representaciones hechas a mano específicamente para cada tarea. Esto permitió que DQN obtenga puntajes

iguales, o incluso más altos, que los de un jugador humano en diversos juegos de Atari 2600 sin introducir modificaciones específicas por juego.

Cabe destacar que, si bien se podrían haber tomado las 60 imágenes por segundo de 210×160 píxeles con 128 colores como entrada a DQN, para reducir la demanda computacional, se realizó un preprocesamiento de estas imágenes para reducirlas a imágenes de 84×84 en escala de grises. Como tomar una sola imagen de muchos juegos no es suficiente para determinar aspectos como, por ejemplo, la trayectoria de un proyectil o un enemigo, y, por ende, tener una representación de estados que cumpla la propiedad Markov¹⁰; se utilizaron *stacks* (o pilas) de los últimos 4 *frames* (o imágenes) del juego. Entonces, la entrada de la red neuronal convolucional utilizada tiene dimensión $84 \times 84 \times 4$.

La arquitectura de *deep Q-Network* consta de 3 capas convolucionales ocultas, una capa completamente conectada (en inglés, *fully connected layer*) oculta, y la capa de salida. Las 3 capas convolucionales sucesivas de DQN producen 32 mapas de activación de 20×20 , 64 mapas de activación de 9×9 , y 64 mapas de activación de 7×7 . La función de activación utilizada en ellas es ReLU ($\max(0, x)$). Las 3136 neuronas ($64 \times 7 \times 7$) en esta tercera capa convolucional se conectan a cada una de las 512 neuronas en la capa conectada oculta, donde, a su vez, cada una de ellas se conecta a las 18 neuronas de la capa de salida (una para cada acción posible en un juego de Atari)¹¹. Una representación esquemática de una arquitectura de este tipo se puede observar en la Figura 4.7.

¹⁰En este caso, que se cumpla la propiedad Markov indica proveerle al agente toda la información necesaria para tomar una decisión sobre qué acción tomar en cada estado, sin depender de los anteriores.

¹¹Cabe destacar que la cantidad de acciones posibles varía entre 4 y 18, dependiendo del juego, entonces no todas las neuronas de salida son funcionales en todos los juegos.

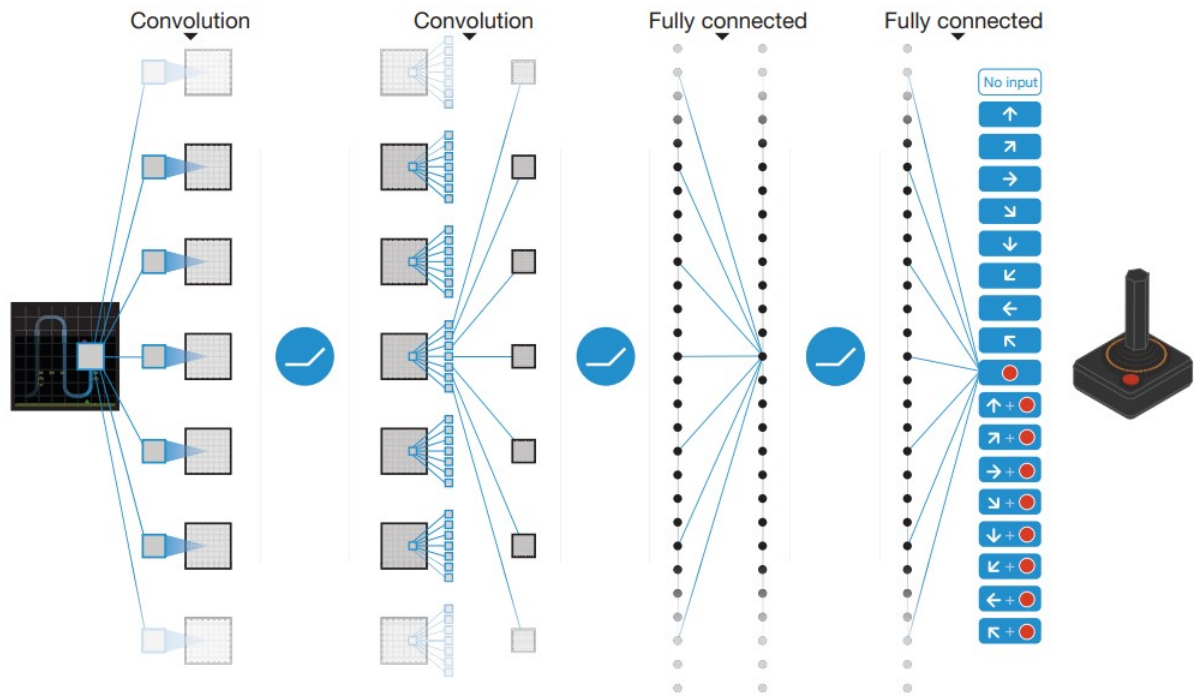


Figura 4.7: Esquema de una arquitectura de red neuronal convolucional cuya entrada son imágenes y su salida acciones que se pueden realizar en juegos de Atari.

La señal de recompensa que se utilizó dependía de cómo se modificase el puntaje dentro del juego de un paso de tiempo a otro: +1 cuando se incrementaba, -1 cuando se decrementaba y 0 en cualquier otro caso. Esto permitió estandarizar la recompensa en todos los juegos, sin importar de qué forma midieran el puntaje de forma particular.

Otro aspecto importante, es que se realizó una búsqueda informal para un conjunto de hiperparámetros que funcionaran bien dentro de varios juegos y se mantuvieron fijos para todos ellos.

DQN utilizó la siguiente forma de Q-Learning para realizar una actualización de los valores de los pesos de la red neuronal convolucional:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a, \mathbf{w}_t) - Q(s_t, A_t, \mathbf{w}_t) \right] \nabla Q(s_t, A_t, \mathbf{w}_t) \quad (4.3)$$

donde \mathbf{w}_t es el vector de los pesos de la red, A_t es la acción seleccionada en el paso de tiempo t , s_t y s_{t+1} son los *stacks* de imágenes preprocesadas del juego en el tiempo t y $t + 1$, respectivamente.

Al utilizar un aproximador de funciones no lineal, como lo son las redes neuronales convolucionales, para aproximar la función Q se suelen presentar problemas de inestabilidad o divergencia. Para superar estos problemas de inestabilidad, se realizaron dos modificaciones significativas al algoritmo de Q-Learning:

- *Experience replay*
- *Target Network (o red objetivo)*

En ***experience replay*** se almacenan *experiencias* del agente en cada paso de tiempo t (donde cada experiencia tiene la forma $e_t(s_t, a_t, r_{t+1}, s_{t+1})$) en una *memoria de replay* D ¹². Luego de que se ejecute una acción a_t en un estado representado por el *stack* de imágenes s_t y se devuelva la recompensa r_{t+1} y el estado s_{t+1} (representado por un *stack* de imágenes), se añade la tupla $(s_t, a_t, r_{t+1}, s_{t+1})$ a la memoria de replay. En cada paso de tiempo t , se aplican actualizaciones de Q-Learning a subconjuntos (o *minibatches*) de experiencias tomadas aleatoriamente de manera uniforme de la memoria de replay D . Luego, el agente selecciona y ejecuta una acción de acuerdo a una política ϵ -greedy¹³. Es decir, en lugar de que s_{t+1} pase a ser el estado actual s_t , se toma una experiencia e desconectada, que no corresponde al estado inmediatamente siguiente a s_t , para la próxima actualización de los valores Q .

¹²Esta memoria de replay tiene un tamaño finito N , por lo que se almacenan las últimas N experiencias del agente.

¹³Se recuerda que una política de este tipo selecciona acciones de forma *greedy* (la mejor opción que se presenta en el momento, sin considerar las consecuencias futuras) pero, con probabilidad ϵ , elige una acción de forma aleatoria.

Entonces, la idea básica de *experience replay* es actualizar la *Q-Network*¹⁴ que aproxima a Q^* utilizando un subconjunto aleatorio de experiencias almacenadas en una memoria de replay, en lugar de solamente utilizar la experiencia más reciente. Esto permite utilizar cada una de estas experiencias almacenadas para muchas actualizaciones, lo que hace que el agente aprenda de forma más eficiente de la experiencia recolectada en su interacción con el ambiente. Además, al aprender de experiencias aleatorias en lugar de consecutivas, se rompe la correlación que existe entre estados consecutivos muy similares. Por último, al usar *experience replay*, la distribución de comportamiento se promedia a lo largo de estados previos, lo que produce un aprendizaje más suave y evita divergencia, mejorando la estabilidad [26].

Se realizó una segunda modificación a Q-Learning estándar para mejorar la estabilidad, la adición de una ***target network***. El valor objetivo de una actualización de Q-Learning depende de la estimación actual de la función de valor-acción. Cuando se usa un método de aproximación parametrizado para representar los valores de función de valor-acción, el objetivo es una función de los mismos parámetros que están siendo actualizados. En este caso, el objetivo es $\gamma \max_a Q(s_{t+1}, a, \mathbf{w}_t)$, mostrado en la Ecuación 4.3. La dependencia de \mathbf{w}_t produce complicaciones en el proceso, ya que actualizaciones muy frecuentes pueden producir oscilaciones o divergencia.

Para evitar este problema, se mantienen 2 *Q-Networks*: la que aproxima a la función Q^* y una llamada *target network*. Cada cierto número, C , de pasos de actualización de los pesos \mathbf{w} de la *Q-Network*, se insertan dichos valores de pesos en otra *Q-Network* (conocida como *target network*) y se mantienen fijos en ella durante las siguientes C actualizaciones de los pesos \mathbf{w} de la red original. Las salidas de esta red duplicada se usan como valores objetivo de Q-Learning durante las siguientes actualizaciones de la *Q-Network*. Denotando como \hat{Q} a la salida de la *target network*, es posible modificar la regla de actualización mostrada en la Ecuación 4.3 de la siguiente

¹⁴Se utilizarán *Q-Network* y *deep Q-Network* de forma intercambiable.

forma:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[r_{t+1} + \gamma \max_a \hat{Q}(s_{t+1}, a, \mathbf{w}_t) - Q(s_t, A_t, \mathbf{w}_t) \right] \nabla Q(s_t, A_t, \mathbf{w}_t) \quad (4.4)$$

En la Figura 4.8, se muestra el algoritmo Deep Q-Learning que incorpora los conceptos detallados previamente [27].

1. Inicializar la memoria de replay D a la capacidad N
2. Inicializar la funcion de valor-accion Q con pesos aleatorios θ
3. Inicializar la funcion de valor-accion *target* \hat{Q} con pesos $\theta^- = \theta$
4. **Para** episodio = 1, M **hacer**
5. Inicializar la secuencia $s_1 = \{x_1\}$ y la secuencia preprocesada $\phi_1 = \phi(s_1)$
6. **Para** $t = 1, T$ **hacer**
7. Con probabilidad ϵ seleccionar una accion aleatoria a_t
 en otro caso, seleccionar $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$
8. Ejecutar la accion a_t en el emulador, observar la recompensa r_t y la imagen x_{t+1}
9. Hacer $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$
10. Almacenar la transicion $(\phi_t, a_t, r_t, \phi_{t+1})$ en D
11. Tomar un subconjunto aleatorio de transiciones $(\phi_j, a_j, r_j, \phi_{j+1})$ de D
12. Hacer $\gamma_j = \begin{cases} r_j & \text{si el episodio termina en el paso } j + 1 \\ r_j + \gamma \max'_a \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{en otro caso} \end{cases}$
13. Hacer un paso de descenso del gradiente en $(\gamma_j - Q(\phi_j, a_j; \theta))^2$ respecto a los parametros de red θ
14. Cada C pasos resetear $\hat{Q} = Q$
15. **Fin Para**
16. **Fin Para**

Figura 4.8: Algoritmo deep Q-Learning con *experience replay*

Con este contenido sobre los diversos algoritmos de aprendizaje por refuerzo, concluye la introducción a los conceptos teóricos que forman parte de este trabajo y se pasará a hablar de la experimentación realizada.

□

CAPÍTULO V

EXPERIMENTACIÓN

Hasta este punto, se discutieron los aspectos que hacen a la teoría del *reinforcement learning*, las partes que lo conforman y los algoritmos que pueden utilizarse para aplicarlo a distintas tareas. Ahora, una pregunta válida que surge es: ¿a qué tareas puede ser aplicado el aprendizaje por refuerzo de forma exitosa?

Primeramente, cabe destacar que aquellas tareas a las que se quiera aplicar aprendizaje por refuerzo deben poder ser modeladas como un *proceso de decisión Markov* (que fueron discutidos en el Capítulo III). Es decir, se deben poder establecer un conjunto de estados posibles del ambiente, un conjunto de acciones que el agente puede ejecutar, las probabilidades de transición de un estado a otro al ejecutar una acción y las recompensas esperadas al ejecutar cada acción en cada estado. Si bien, como ya se vio hasta aquí, no es estrictamente necesario contar con este modelo de antemano, la idea es que el problema cumpla con la propiedad Markov o, en otras palabras, sea *Markoviano* (o se pueda implementar alguna modificación que lo convierta en tal). Teniendo en cuenta esto, el *reinforcement learning* puede ser aplicado en varios tipos de problemas, algunos de los más populares son: robótica [22], autos y vehículos voladores autónomos [49], sistemas de diálogo y *chatbots* [11][24],[4] sistemas de recomendación [1][20] y, en el que nos enfocaremos de ahora en adelante, el desarrollo de agentes que aprendan a jugar juegos.

§5.1 Introducción

Observando las publicaciones realizadas por *DeepMind* acerca de DQN [26][27], se menciona el hecho de que no se realizó una optimización de *hiperparámetros* ya que sería computacionalmente muy intensivo, sino que se llevó a cabo una búsqueda informal. Los *hiperparámetros*, se recuerda, son valores que se utilizan para controlar el proceso de aprendizaje del modelo¹ y pueden afectar el desempeño del mismo a la hora de realizar la tarea para la cual fue entrenado, o la velocidad con la que se produce su entrenamiento. En términos simples, no se realizó una búsqueda de los hiperparámetros que produzcan un modelo que obtiene los mejores resultados posibles, sino que se hicieron pruebas de distintos valores hasta encontrar un conjunto de ellos que produzcan buenos resultados.

Algunas preguntas que surgen pueden ser: ¿qué tan importantes son los distintos hiperparámetros para el desempeño de un modelo entrenado con este algoritmo? ¿Se podría alcanzar un mejor desempeño si se hace foco en un juego en particular, en lugar de usar hiperparámetros que funcionen bien en una gran variedad de ellos? ¿Cómo afectan el desempeño los valores de los hiperparámetros elegidos? ¿Existen valores de hiperparámetros con los que se logre conseguir mejores resultados que utilizando aquellos publicados?

Teniendo en cuenta estas preguntas, se realizó el entrenamiento de varios modelos utilizando el algoritmo *deep Q-Learning* y 3 juegos de la consola *Atari 2600* para tratar de responderlas. El código utilizado para estos experimentos puede encontrarse en GitHub.

¹Se le llamará *modelo* al resultado obtenido después de ejecutar la etapa de entrenamiento de un algoritmo determinado. Representa qué fue aprendido por el algoritmo y contiene todo lo necesario para realizar predicciones. En este contexto, si bien *modelo* referencia a la red neuronal que se utiliza para aprender la política, por lo general se utiliza de forma intercambiable con *agente*.

§5.2 Bibliotecas

Antes de detallar la experimentación realizada, se introducirán las bibliotecas utilizadas para la misma.

En el Capítulo II, se mostraron las distintas componentes que hacen al aprendizaje por refuerzo y cómo interactúan: el *agente* que ejecuta cierta política, el *ambiente* con el que interactúa, las *acciones* que pueden ser realizadas por el agente, las *recompensas* que recibe del ambiente por sus acciones, y las *observaciones* que el agente realiza del estado actual del ambiente.

Al poder aplicarse *reinforcement learning* a una gran variedad de problemas, surge la necesidad de ciertos *benchmarks* que permitan evaluar qué tanto se ha progresado en este campo y, además, al ser tan importante para el resultado final la forma en la que se define el problema a abordar, se necesita un conjunto de ambientes estándar que haga más sencillo reproducir los resultados obtenidos en distintas publicaciones. *OpenAI Gym* [7][6] busca abordar estos problemas, brindando una gran variedad de ambientes para diversas aplicaciones y definiendo una interfaz que se ha vuelto estándar dentro del aprendizaje por refuerzo. La biblioteca permite la creación de ambientes que devuelven al agente observaciones del mismo, define las acciones que éste puede realizar, retorna las recompensas de la ejecución de dichas acciones e indica cuándo se ha alcanzado un estado final en la tarea.

Hasta aquí estarían cubiertos 4 de los 5 aspectos mencionados anteriormente, por lo que resta hablar del agente. A lo largo del resurgimiento del aprendizaje por refuerzo producido en la última década, han surgido diversas implementaciones de los algoritmos publicados que permitieron replicar los resultados obtenidos en la investigación. De forma similar a lo que sucede con la definición de ambientes, detalles pequeños de implementación pueden tener un efecto muy grande en el desempeño de los algorit-

mos, haciendo difícil realizar una comparación entre ellos. *Stable-Baselines3*²[32][31], un *fork* de *Baselines* de *OpenAI*[15], busca resolver estos problemas proporcionando una implementación de 7 algoritmos de *deep reinforcement learning*³ con los que es posible obtener los mismos resultados que en las publicaciones donde fueron introducidos. Además, posee una *API* muy legible y fácil de usar. Por estos motivos, fue la implementación elegida para realizar este trabajo.

Establecidas la importancia de las bibliotecas y las razones para su uso, se pasará a mostrar a ambas en más detalle.

§5.2.1 OpenAI Gym

OpenAI Gym [7][6] es una biblioteca desarrollada y mantenida por *OpenAI*. Su objetivo principal es proveer una gran cantidad de ambientes de diversos tipos para realizar experimentos de aprendizaje por refuerzo, utilizando una interfaz sencilla y unificada. *Gym* cuenta con más de 150 ambientes (cada uno de ellos con varias “versiones”), que pueden ser divididos en varios grupos:

- **Problemas de control clásicos:** son tareas sencillas utilizadas en *papers* como demostración o *benchmark*. Cuentan con espacios de acciones y de observaciones bastante pequeños.
- **Atari 2600:** juegos de la plataforma Atari de los años 70. Cuenta con 63 juegos diferentes.
- **Algorítmicos:** problemas cuyo objetivo es realizar una pequeña tarea computacional, como sumar números o copiar símbolos.

²Se utilizará, además, la forma abreviada *SB3* para referirse a esta biblioteca.

³Estos métodos consisten en el uso de redes neuronales profundas. Deep Q-Learning es un ejemplo de este tipo de métodos.

- **Box2D:** ambientes que utilizan el simulador de física *Box2D* [9][10] para tareas como aprender a caminar o controlar vehículos.
- **MuJoCo:** ambientes que utilizan el simulador de física *MuJoCo* [13][14] para hacer varias tareas de control, como hacer caminar a un robot cuadrúpedo o hacer caminar o levantarse del suelo a un robot humanoide.
- **Robótica:** ambientes en donde brazos o manos robóticas deben realizar distintas tareas, como levantar un objeto u orientarlo de cierta forma.
- **Toy text:** ambientes con tareas y juegos sencillos basados en texto.

Los ambientes en *Gym* son representados por la clase `Env`, que tiene las siguientes componentes:

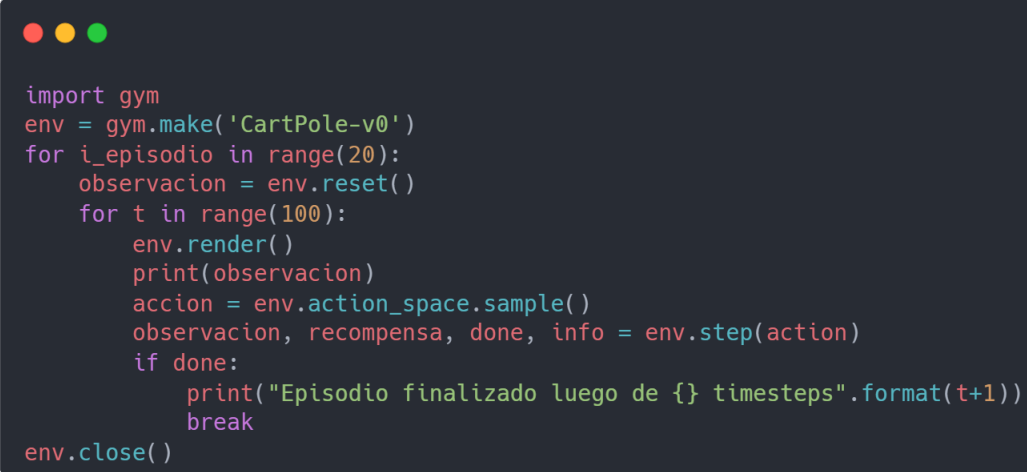
- **Un conjunto de acciones (`action_space`)** que pueden ser ejecutadas en el ambiente. Estas acciones pueden ser discretas, continuas o una combinación de ambas. Las acciones discretas son un conjunto fijo de cosas que el agente puede hacer, por ejemplo, moverse a la derecha, a la izquierda, hacia arriba o hacia abajo. Las acciones continuas tienen un valor asociado, por ejemplo, pisar un acelerador con diferentes niveles de fuerza o girar un volante cierto ángulo. No se limita solamente a ejecutar una acción a la vez, podrían, por ejemplo, presionarse múltiples botones a la vez o mover el volante y pisar el acelerador al mismo tiempo; *Gym* permite anidar varios espacios de acciones en una sola acción para permitir estos casos.
- **Un conjunto de observaciones (`observation_space`)**, que es la información que se le provee al agente acerca del ambiente, además de la recompensa. Pueden ser tan simples como algunos números o imágenes a color de múltiples cámaras.

- Un método **reset()**, que hace que el ambiente vuelva a su estado inicial y retorna un vector con la observación inicial.
- Un método **step()**, que tiene diversas tareas: notifica al ambiente qué acción va a ser realizada en el próximo paso (que es pasada como argumento), obtiene la nueva observación del ambiente luego de esta acción, obtiene la recompensa recibida por el agente en este paso y obtiene la indicación de si el episodio ha terminado. Esto tiene la siguiente forma:
 - **observation:** es un vector o matriz de *NumPy*⁴ [21].
 - **reward:** es el valor numérico de la recompensa.
 - **done:** es un indicador *booleano*, que es verdadero cuando termina el episodio.
 - **info:** puede ser cualquier información que sea específica de un ambiente en particular. Por lo general, en la práctica este valor es ignorado.
- Un método **render()**, que muestra en pantalla el ambiente (es decir, el juego, la simulación o lo que corresponda según la tarea elegida).

La clase `Env` cuenta con algunos métodos además de los mostrados, pero éstos son los más importantes a comprender para utilizar *OpenAI Gym* en la práctica.

Para finalizar, en la Figura 5.1, se muestra un ejemplo sencillo del uso de *Gym* en el que se crea el ambiente *CartPole*, se itera durante 20 episodios y, durante 100 *frames* (*timesteps*), el agente ejecuta una acción aleatoria.

⁴*NumPy* es una biblioteca para el lenguaje *Python* que añade soporte para arreglos y matrices multidimensionales de gran tamaño. Además, cuenta con funciones para operar con estas estructuras.



```

import gym
env = gym.make('CartPole-v0')
for i_episodio in range(20):
    observacion = env.reset()
    for t in range(100):
        env.render()
        print(observacion)
        accion = env.action_space.sample()
        observacion, recompensa, done, info = env.step(accion)
        if done:
            print("Episodio finalizado luego de {} timesteps".format(t+1))
            break
    env.close()

```

Figura 5.1: Ejemplo del uso de *OpenAI Gym*.

§5.2.2 Stable-Baselines3

Stable-Baselines3 [32][31] es un conjunto de implementaciones de algoritmos de *reinforcement learning* en *PyTorch*⁵[2]. Cuenta con implementaciones de los siguientes 7 algoritmos:

- **A2C:** una variante síncrona y determinística de *Asynchronous Advantage Actor Critic (A3C)* [25]
- **DDPG:** *Deep Deterministic Policy Gradient* [23]
- **DQN:** *Deep Q-Network* [26]
- **HER:** *Hindsight Experience Replay* [3]
- **PPO:** *Proximal Policy Optimization* [39]

⁵*PyTorch* es una biblioteca de *machine learning* de código abierto, desarrollada principalmente por Meta (previamente conocida como Facebook).


- **SAC:** *Soft Actor Critic* [19]
- **TD3:** *Twin Delayed Deep Deterministic Policy Gradient* [18]

Algunos de los métodos más importantes con los que cuentan las implementaciones de los algoritmos son:

- **learn ()**: se utiliza para entrenar un modelo con el algoritmo correspondiente y devuelve el modelo entrenado. Permite hacer un *log* del entrenamiento y uno de sus parámetros más importantes es la duración del mismo.
- **predict ()**: toma como parámetro una observación y retorna la acción a realizar seleccionada por el agente.
- **save ()**: permite guardar un archivo con el modelo entrenado.
- **set_env ()**: toma como parámetro un ambiente de la forma de *OpenAI Gym*, verifica que sea correcto y lo asigna como ambiente a utilizar.

Además, las clases de los algoritmos cuentan con atributos que representan a los hiperparámetros del algoritmo en particular, la arquitectura de la red neuronal a utilizar (que, en SB3 se le llama *política*, pero no debe confundirse con la política, a nivel teórico, introducida previamente), el ambiente a utilizar, entre otros valores.

A continuación, en la Figura 5.2, se provee un ejemplo sencillo que utiliza el algoritmo *Soft Actor Critic* y el ambiente *Pendulum* para entrenar un modelo, guardarlo, cargarlo y obtener una acción a partir de la política aprendida.



```

import gym
from stable_baselines3 import SAC
# Entrenar un agente usando el algoritmo Soft Actor-Critic en el ambiente Pendulum-v0
env = gym.make("Pendulum-v0")
modelo = SAC("MlpPolicy", env).learn(total_timesteps=20000)
# Guardar el modelo
modelo.save("sac_pendulum")
# Cargar el modelo entrenado
modelo = SAC.load("sac_pendulum")
# Empezar un nuevo episodio
obs = env.reset()
# ¿Qué acción tomar en el estado 'obs'?
accion, _ = modelo.predict(obs, deterministic=True)

```

Figura 5.2: Ejemplo del uso de *Stable-Baselines3*.

Una ventaja de *Stable-Baselines3* por sobre otras bibliotecas de *reinforcement learning* es que cuenta con una muy buena documentación, donde se muestran ejemplos, se detallan tanto los atributos como los métodos de cada clase correspondiente a los algoritmos implementados, y se muestra para qué tipo de ambiente es apto cada algoritmo.

Además de las implementaciones de algoritmos, la biblioteca provee funcionalidades extra, como una implementación del preprocesamiento de imágenes de juegos de *Atari* usado en publicaciones como la de *deep Q-Learning* [26] que se discutió en la Sección 4.4.2, funciones para verificar que un ambiente siga la forma de la *API* de *OpenAI Gym* (que resulta de utilidad cuando se definen ambientes personalizados), entre otras. Esto hace que SB3 sea una biblioteca muy completa y útil para realizar experimentos en el campo del aprendizaje por refuerzo.

§5.3 Experimentos

Como se mencionó en la introducción de este capítulo, surgen algunas preguntas a partir de las publicaciones sobre *Deep Q-Networks*, en particular acerca de la importancia de los valores de los hiperparámetros en el desempeño de los modelos que pueden ser generados con este algoritmo. Para tratar de responder estas inquietudes, se entrenaron diversos modelos, variando en cada uno de ellos el valor de un hiperparámetro en particular, sobre 3 juegos de *Atari 2600*: *Breakout*, *Space Invaders* y *Boxing*. Se introducirán brevemente tanto los ambientes como las acciones que el agente puede realizar en cada uno de ellos.

Breakout cuenta con una capa de ladrillos en la parte superior de la pantalla y el objetivo del jugador es destruirlos todos haciendo rebotar una pelota en una paleta antes de agotar las vidas con las que se cuenta. Una imagen del juego se muestra en la Figura 5.3. Las acciones disponibles son:

1. Moverse a la derecha.
2. Moverse a la izquierda.
3. Comenzar (utilizada únicamente para iniciar cada partida).
4. No hacer nada.

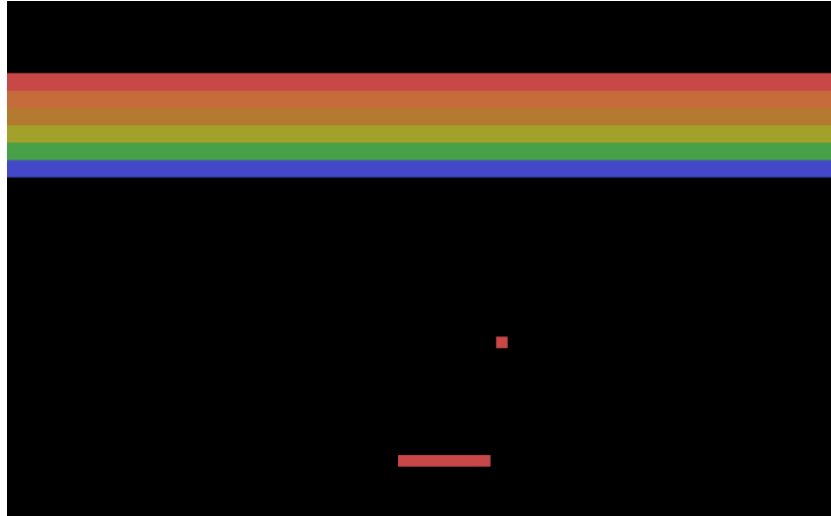


Figura 5.3: *Breakout*.

Space Invaders consta de rondas de *aliens* que descienden en forma de grilla. El objetivo del jugador es obtener la puntuación más alta destruyendo la mayor cantidad de ellos, sin que los *aliens* lo destruyan a él. La Figura 5.4 muestra una imagen del juego. Las acciones que se pueden realizar en él son:

1. Moverse a la derecha.
2. Moverse a la izquierda.
3. Disparar.
4. Disparar moviéndose a la derecha.
5. Disparar moviéndose a la izquierda.
6. No hacer nada.



Figura 5.4: *Space Invaders*.

Boxing es un juego de boxeo, en el que el jugador que consiga primero conectar 100 golpes o que, pasados 2 minutos, tenga la mayor cantidad de golpes conectados, gana. Se muestra una imagen del juego en la Figura 5.5. Las acciones disponibles en este juego son:

1. Moverse hacia arriba.
2. Moverse hacia abajo.
3. Moverse hacia la derecha.
4. Moverse hacia la izquierda.
5. Moverse hacia arriba a la derecha.
6. Moverse hacia arriba a la izquierda.
7. Moverse hacia abajo a la derecha.
8. Moverse hacia abajo a la izquierda.

9. Golpear.
10. Golpear moviéndose hacia arriba.
11. Golpear moviéndose hacia abajo.
12. Golpear moviéndose hacia la derecha.
13. Golpear moviéndose hacia la izquierda.
14. Golpear moviéndose hacia arriba a la derecha.
15. Golpear moviéndose hacia arriba a la izquierda.
16. Golpear moviéndose hacia abajo a la derecha.
17. Golpear moviéndose hacia abajo a la izquierda.
18. No hacer nada.



Figura 5.5: *Boxing*.

La elección de estos juegos no es trivial, ya que 2 de ellos pertenecen a la categoría de aquellos en los que DQN obtuvo resultados iguales o superiores a los de un jugador humano [26] y en uno de ellos, *Space Invaders*, los humanos siguen obteniendo puntuaciones más altas. Además, su complejidad aumenta, tanto en la cantidad de acciones disponibles para ser realizadas por el agente como en los elementos que debe tener en cuenta para alcanzar las puntuaciones más altas posibles (la ubicación de los enemigos o las coberturas, por ejemplo) en cada uno de ellos.

A continuación, se mostrarán los hiperparámetros que se pueden modificar en *deep Q-Learning*, así como también las técnicas de preprocesamiento de imágenes que se utilizan en su publicación original para reducir el costo computacional de la ejecución del algoritmo.

El preprocesamiento incluye:

- Obtener el estado inicial del ambiente tomando un número aleatorio de acciones “*no-op*” cada vez que se hace un reinicio del mismo.
- Hacer que el agente experimente el juego y, por ende, decida qué acción realizar cada 4 *frames*⁶. A esto se le llama *frame-skipping*. La acción elegida se repite en los *frames* omitidos.
- Hacer *Max-Pooling*, como se introdujo en la Sección 4.4.1, con las últimas imágenes observadas.
- Enviar la señal de que terminó el juego cuando el agente pierde una vida, en lugar de dejar que utilice todas las vidas que tenga disponibles.
- Redimensionar las imágenes de 210x160 píxeles a 84x84 píxeles.
- Transformar la observación a escala de grises.

⁶Se le llama *frame* a cada imagen individual que compone un video. O en este caso, un juego.

- Hacer que las recompensas positivas sean 1, las negativas -1 y mantener sin cambios aquellas que sean 0. Esto se llama *clipping* de recompensa, se utiliza para hacer que los distintos ambientes sean más parecidos entre sí y abstraer del sistema de puntuación de cada juego.

Otro aspecto importante a la hora de utilizar DQN es la idea de *frame stacking*, que consiste en que la entrada de la red neuronal convolucional utilizada sea un conjunto de los últimos 4 *frames* del juego. Si se tomara como entrada una sola imagen, el agente no podría determinar aspectos importantes para jugar correctamente a la mayoría de los juegos, como el movimiento de objetos en la pantalla, ya que solo poseería la información de un solo punto de su trayectoria. En otras palabras, *frame stacking* hace a la tarea más *Markoviana*.

En cuanto a los hiperparámetros que se pueden modificar en la implementación del algoritmo provista por SB3, se cuenta con los siguientes:

- **policy**: en términos de *Stable-Baselines3*, se le llama política a la arquitectura de la red neuronal convolucional a utilizar en el algoritmo.
- **n_timesteps**: es el número de pasos en total que el agente hará en el ambiente. Cada paso corresponde a un *frame* del juego.
- **buffer_size**: el tamaño de la memoria de replay.
- **learning_rate**: determina cómo se ajustan los pesos de la red neuronal respecto a la función de pérdida.
- **batch_size**: es el tamaño del subconjunto de experiencias tomadas de la memoria de replay.
- **learning_starts**: durante cuántos pasos se recolectan transiciones antes de que el agente comience a aprender.

- **target_update_interval**: cada cuántos pasos se actualiza la *target network*.
- **train_freq**: cada cuántos pasos se actualiza el modelo.
- **gradient_steps**: cuántos pasos de gradiente hacer luego de cada actualización.
- **exploration_fraction**: la fracción del período de entrenamiento en la cual se va reduciendo la tasa de exploración (que comienza en 1, o 100 %)
- **exploration_final_eps**: valor final de la probabilidad de elegir una acción aleatoria, es decir, se reduce la tasa de exploración hasta que alcance este valor.
- **gamma**: el factor de descuento.
- **framestack**: la cantidad de *frames* a apilar para proporcionar como entrada a la red neuronal.

Teniendo en cuenta estos hiperparámetros, se realizó el entrenamiento de 30 modelos variando los valores de 5 hiperparámetros para observar su impacto en los resultados obtenidos en *Breakout*, *Space Invaders* y *Boxing*, comparados con los modelos para cada juego entrenados con los hiperparámetros propuestos por la biblioteca *Stable-Baselines3* basados en las publicaciones [30] y con los resultados obtenidos por *DeepMind* en la publicación original [26]. A los modelos entrenados con estos hiperparámetros se les llamará *modelos base* y constan de los siguientes valores:

- **policy**: “CnnPolicy”, que es el nombre que se le da en la biblioteca utilizada a la arquitectura de red neuronal convolucional discutida en la Sección 4.4.2.
- **n_timesteps**: 10.000.000
- **buffer_size**: 100.000

- **learning_rate:** 0,0001
- **batch_size:** 32
- **learning_starts:** 100.000
- **target_update_interval:** 1000
- **train_freq:** 4
- **gradient_steps:** 1
- **exploration_fraction:** 0,1
- **exploration_final_eps:** 0,01
- **gamma:** 0,99
- **framestack:** 4

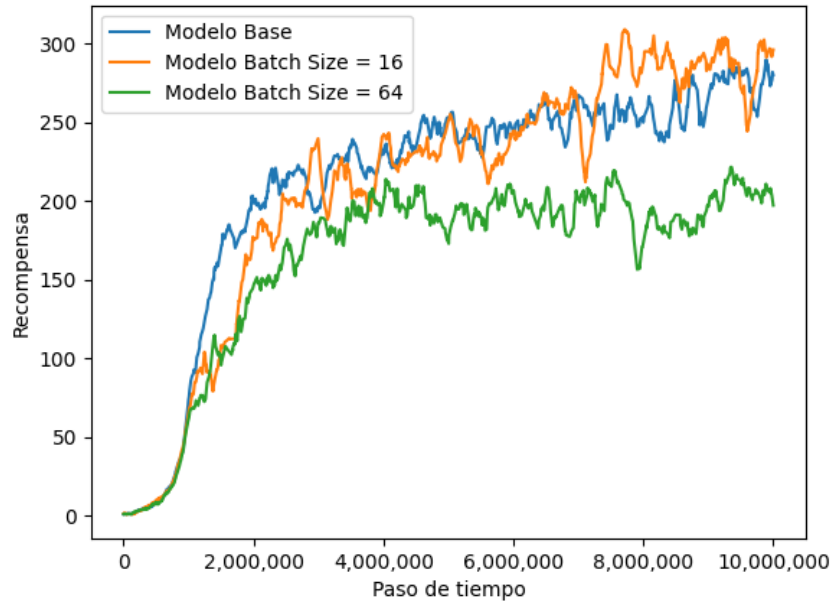
Los hiperparámetros sobre los que se trabajó fueron: `framestack`, `learning_rate`, `batch_size`, `exploration_fraction` y `buffer_size`. A continuación, se verá el análisis del proceso de entrenamiento de los modelos para cada hiperparámetro modificado.

§5.3.1 Batch Size

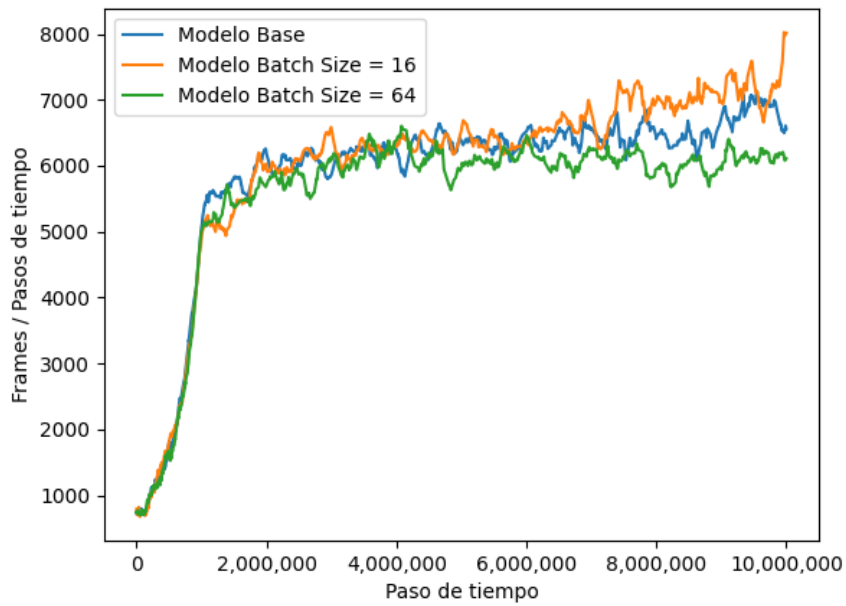
En estos modelos, se realizaron 2 modificaciones del valor de `batch_size` respecto a los hiperparámetros propuestos: se redujo a 16 y se aumentó a 64, esto implica que se toman la mitad o el doble de experiencias de la memoria de replay, respectivamente, que en el modelo base.

En la Figura 5.6, se muestran las curvas de entrenamiento para el juego *Breakout*. En ella, se observa que aumentar el valor de `batch_size` hace que el modelo converja

en una recompensa menor respecto al modelo base y su contraparte con menor valor del mismo hiperparámetro, sin un aumento significativo a partir de los 4 millones de *timesteps*. En cuanto a la longitud de episodio, el comportamiento es el esperado: aumenta a lo largo del tiempo, lo que equivale a que el agente juega más tiempo a medida que avanza el entrenamiento. Nuevamente, alrededor de los 4 millones de *timesteps*, el modelo con un mayor `batch_size` parece converger y no obtener una mejora significativa a partir de ese punto.



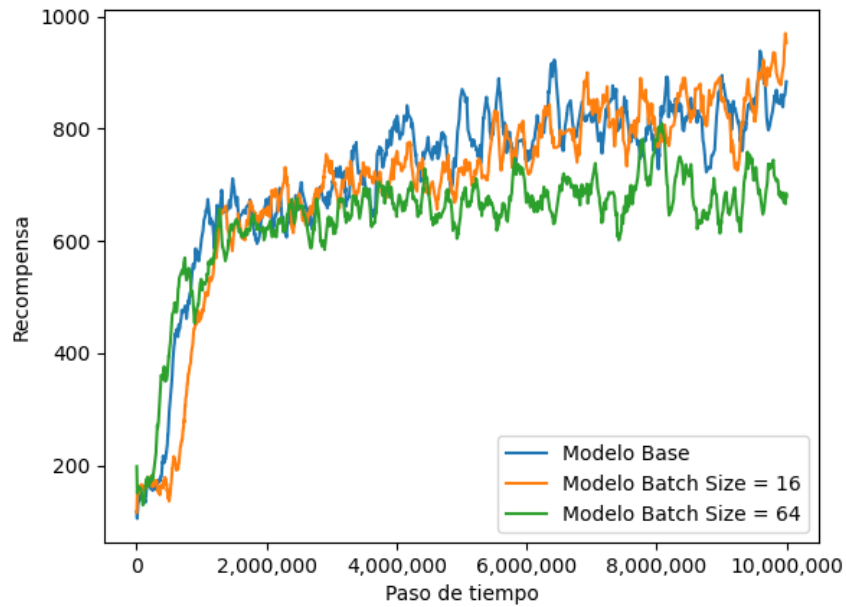
(a) Curva de recompensas de los modelos con **batch_size** modificado.



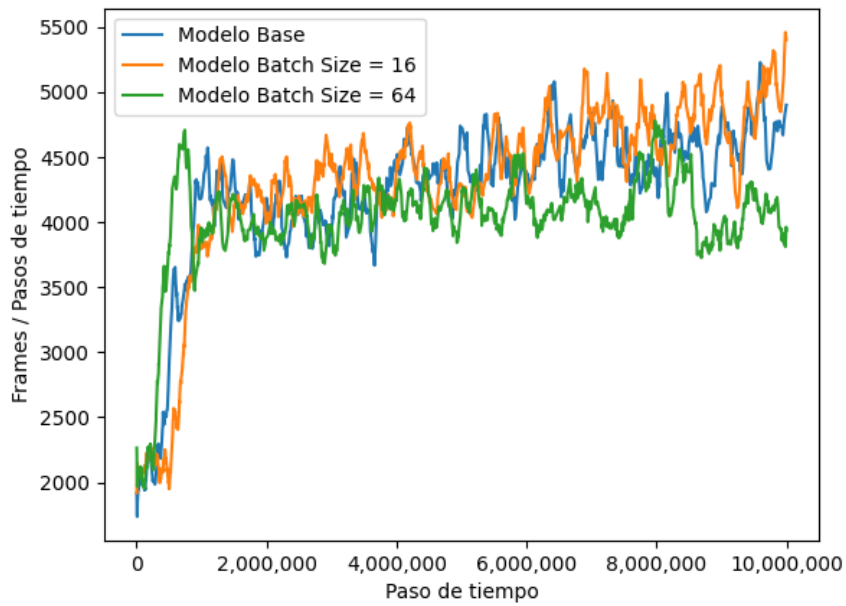
(b) Curva de longitud de episodio de los modelos con **batch_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.6: Curvas de entrenamiento de los modelos con **batch_size** modificado y del modelo base para el juego Breakout.

En lo que respecta a *Space Invaders*, en la Figura 5.7, se observa un comportamiento similar: el modelo con mayor `batch_size` parece converger antes que el resto y con valores menores, tanto en recompensas obtenidas como en la longitud del episodio.



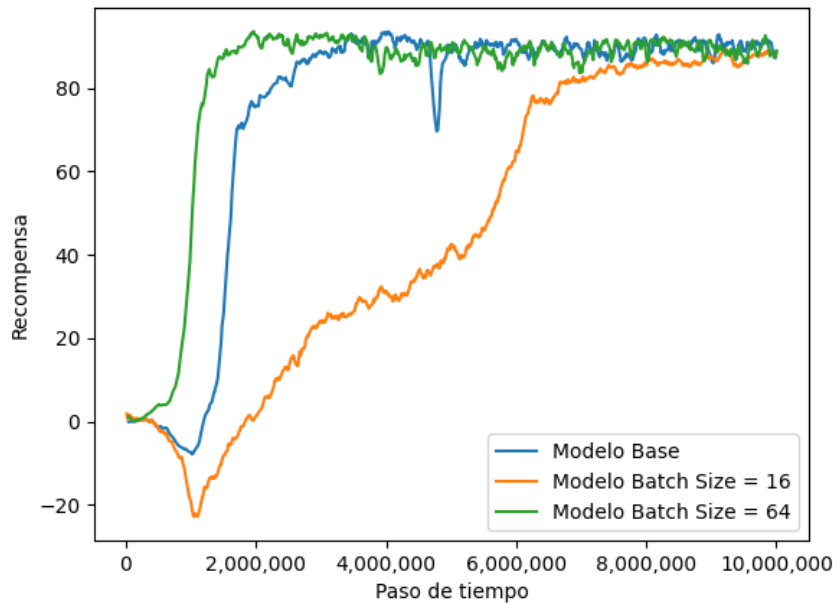
(a) Curva de recompensas de los modelos con **batch_size** modificado.



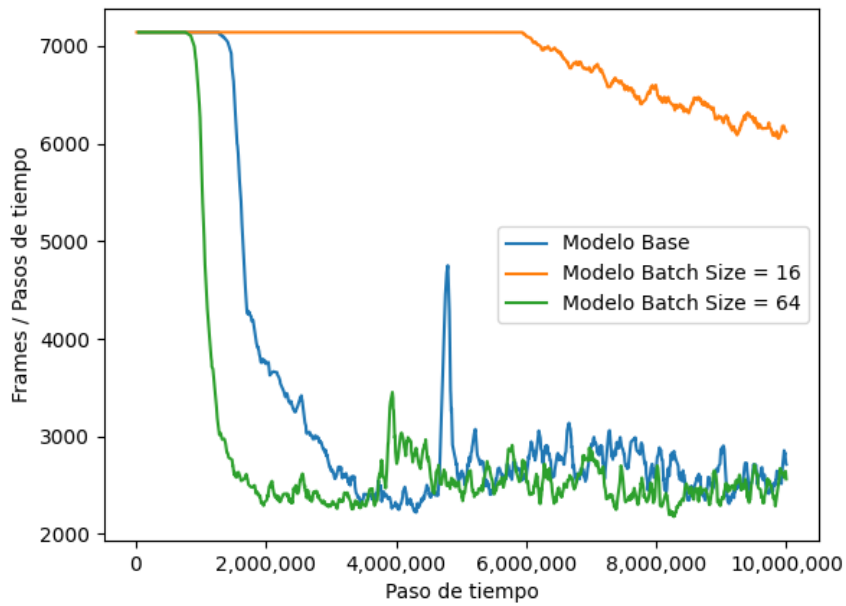
(b) Curva de longitud de episodio de los modelos con **batch_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.7: Curvas de entrenamiento de los modelos con **batch_size** modificado y del modelo base para el juego Space Invaders.

En cuanto a *Boxing*, se recuerda que la puntuación tiene un máximo de 100 puntos y un tope de 2 minutos de tiempo de juego, por lo que el objetivo del agente es llegar a la puntuación máxima en el menor tiempo posible (es decir, la longitud de episodio debería decrecer en lugar de aumentar). En este caso, en la Figura 5.8, se observa, primeramente, que tanto el modelo base como el de mayor `batch_size` convergen mucho antes de la cantidad de *timesteps* sugerida, siendo este último el primero en converger, lo que indica que aquí dicha cantidad de pasos de tiempo es excesiva. En cuanto al modelo con menor `batch_size`, si bien parecería que logra alcanzar los mismos valores de recompensa que el resto, utiliza muchos más *timesteps* y su longitud de episodio no decrece significativamente, lo que podría sugerir que, a pesar de que el agente alcanza la puntuación máxima (es decir, gana la partida), consume bastante más tiempo de juego para hacerlo que los otros 2 agentes aquí mostrados.



(a) Curva de recompensas de los modelos con **batch_size** modificado.



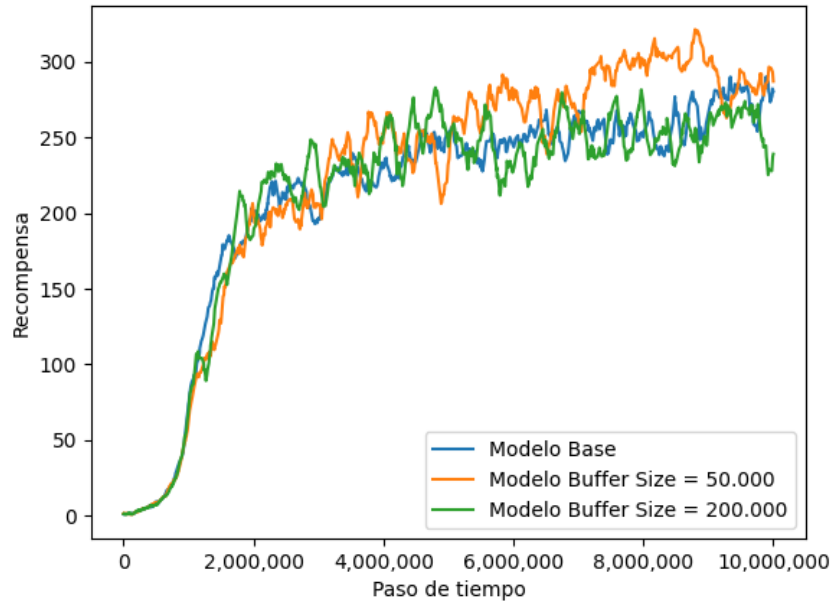
(b) Curva de longitud de episodio de los modelos con **batch_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.8: Curvas de entrenamiento de los modelos con **batch_size** modificado y del modelo base para el juego Boxing.

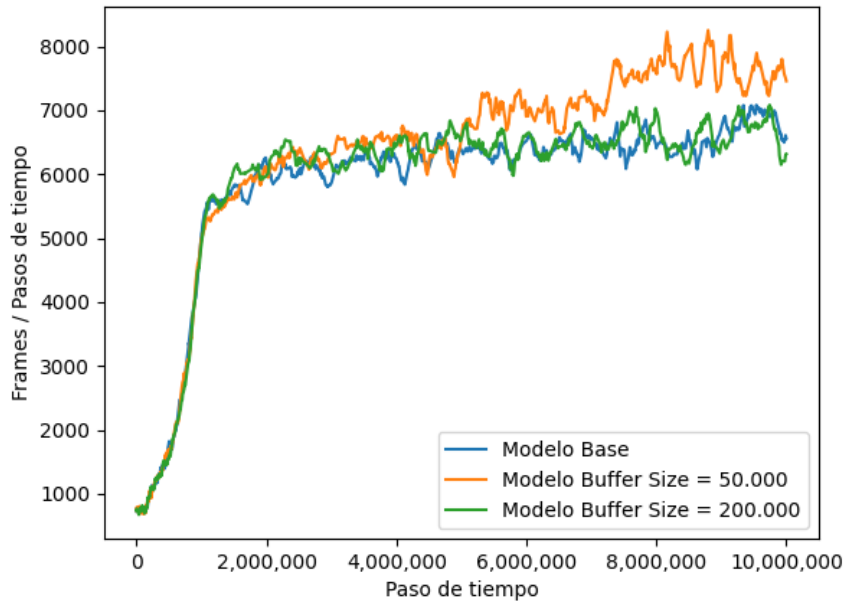
§5.3.2 Buffer Size

En estos modelos, se realizó una reducción del tamaño de la memoria de replay a la mitad y un aumento del tamaño de la memoria de replay al doble de su capacidad respecto a la utilizada en el modelo base.

En la Figura 5.9 se muestran las curvas de recompensa y de longitud de episodio del entrenamiento de estos modelos comparadas con las del modelo base. En ellas, se puede observar que el modelo con un menor tamaño de memoria de replay pareciera obtener recompensas más altas que el resto y, además, lograr un mayor tiempo de juego, indicado por una longitud de episodio más alta. Esto podría ser un indicio de mejor desempeño, pero este análisis se realizará más adelante.



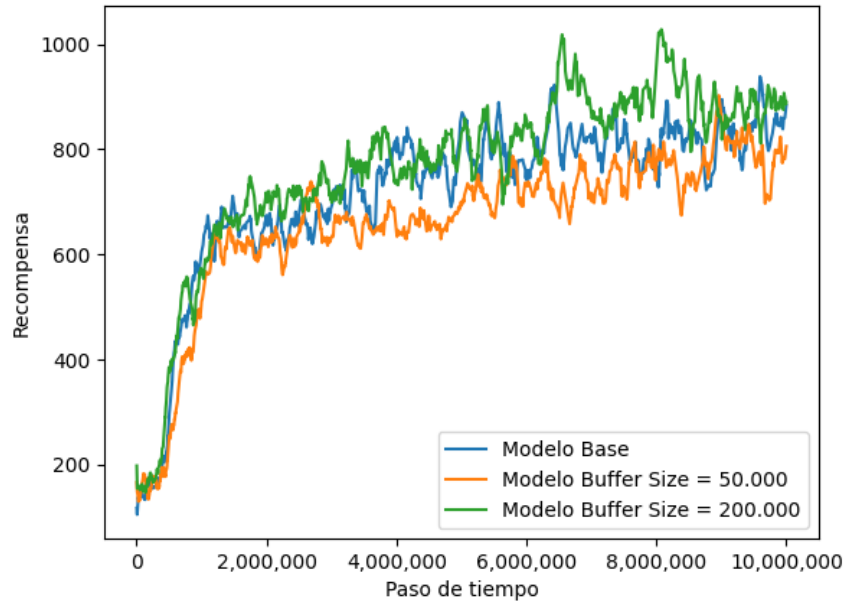
(a) Curva de recompensas de los modelos con **buffer_size** modificado.



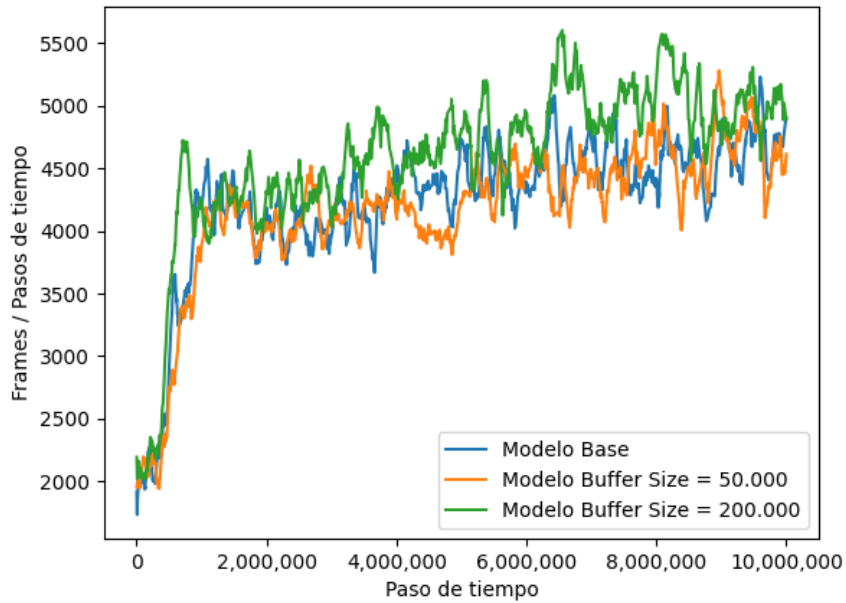
(b) Curva de longitud de episodio de los modelos con **buffer_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.9: Curvas de entrenamiento de los modelos con **buffer_size** modificado y del modelo base para el juego Breakout.

En *Space Invaders*, como se observa en la Figura 5.10, parece no haber distinciones demasiado significativas entre los modelos, fuera de algunos momentos en los que el modelo con mayor tamaño de memoria de replay consigue mayores recompensas y longitudes de episodio respecto al resto de modelos presentados en la gráfica.



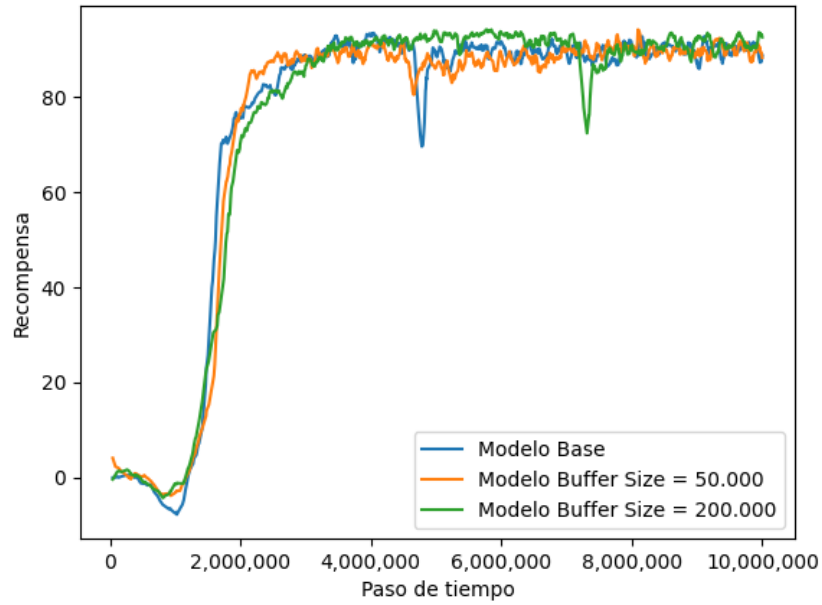
(a) Curva de recompensas de los modelos con **buffer_size** modificado.



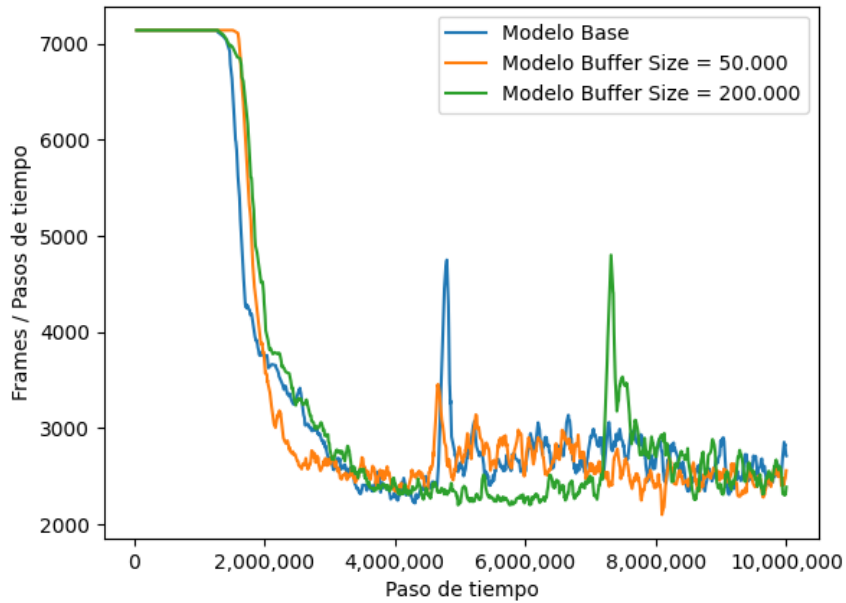
(b) Curva de longitud de episodio de los modelos con **buffer_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.10: Curvas de entrenamiento de los modelos con **buffer_size** modificado y del modelo base para el juego Space Invaders.

En el caso del entrenamiento de los modelos para el juego *Boxing*, mostrado en la Figura 5.11, nuevamente se observa que converge varios pasos de tiempo antes que los sugeridos en todos los modelos presentados, y, fuera de algunas fluctuaciones en la longitud de episodio, el comportamiento es el esperado para todos los modelos en este juego: un aumento en la recompensa y una reducción en la longitud de episodio a medida que transcurre el entrenamiento.



(a) Curva de recompensas de los modelos con **buffer_size** modificado.



(b) Curva de longitud de episodio de los modelos con **buffer_size** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

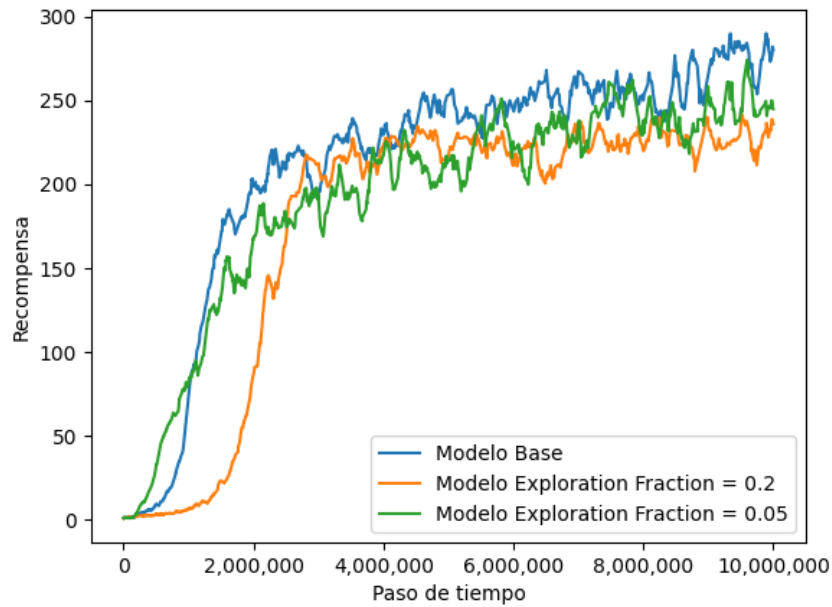
Figura 5.11: Curvas de entrenamiento de los modelos con **buffer_size** modificado y del modelo base para el juego Boxing.

§5.3.3 Exploration Fraction

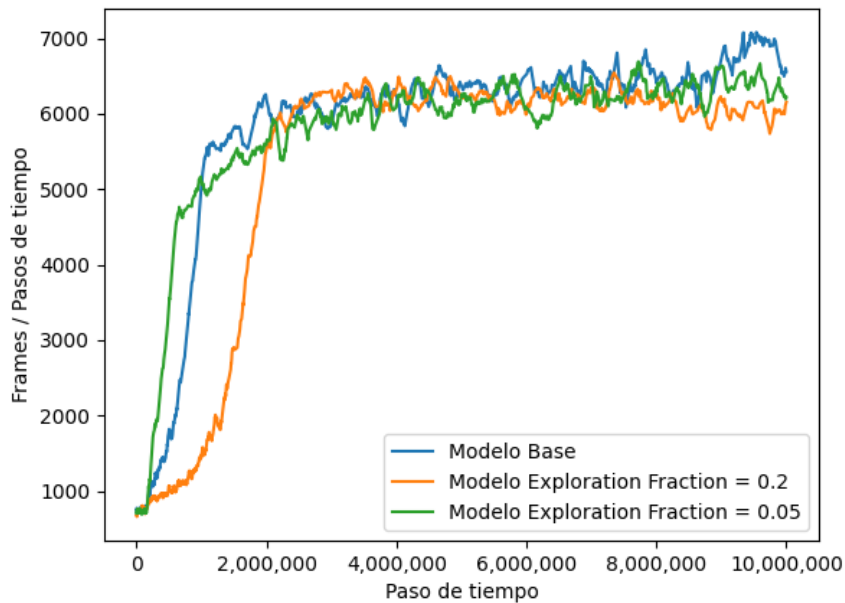
Para estos modelos, se realizó una reducción en la `exploration_fraction`⁷, lo que indica que se realizan más acciones exploratorias debido a una reducción menos agresiva en la tasa de exploración, y un aumento en ella, lo que indica que se realizan menos acciones exploratorias por un decremento más agresivo en la tasa de exploración, respecto a los valores sugeridos utilizados en el modelo base.

En la Figura 5.12 se muestran las curvas de entrenamiento para estos modelos en *Breakout*. En ellas, se puede observar que el modelo que realiza menos exploración (es decir, aquel que tiene una `exploration_fraction = 0.2`) toma más *timesteps* en alcanzar altas puntuaciones y mayores longitudes de episodio, pero fuera de esta observación, los 3 modelos presentados parecen converger de manera similar.

⁷Se recuerda que este valor indica en cuánto porcentaje se reduce la tasa exploración a medida que transcurre el entrenamiento. Es decir, controla cuánto explora el agente durante el entrenamiento.



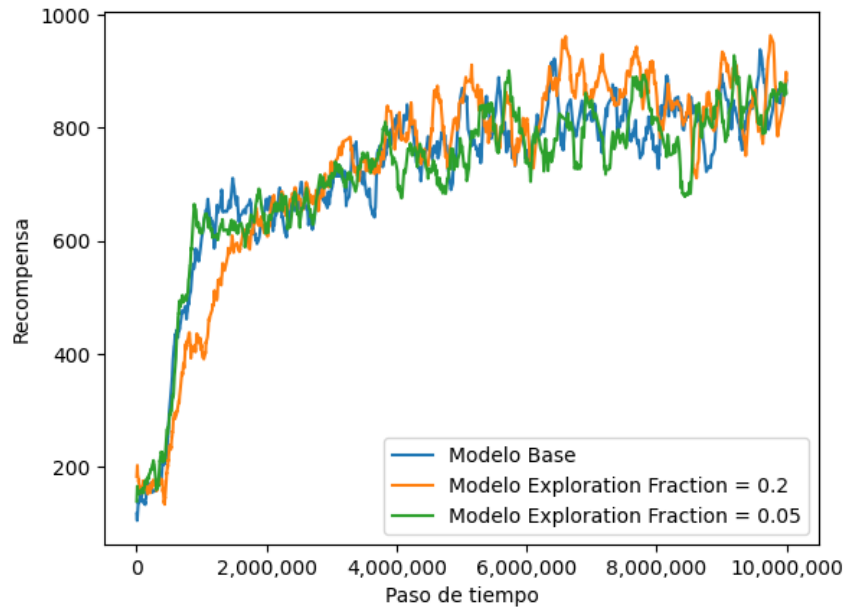
(a) Curva de recompensas de los modelos con `exploration_fraction` modificado.



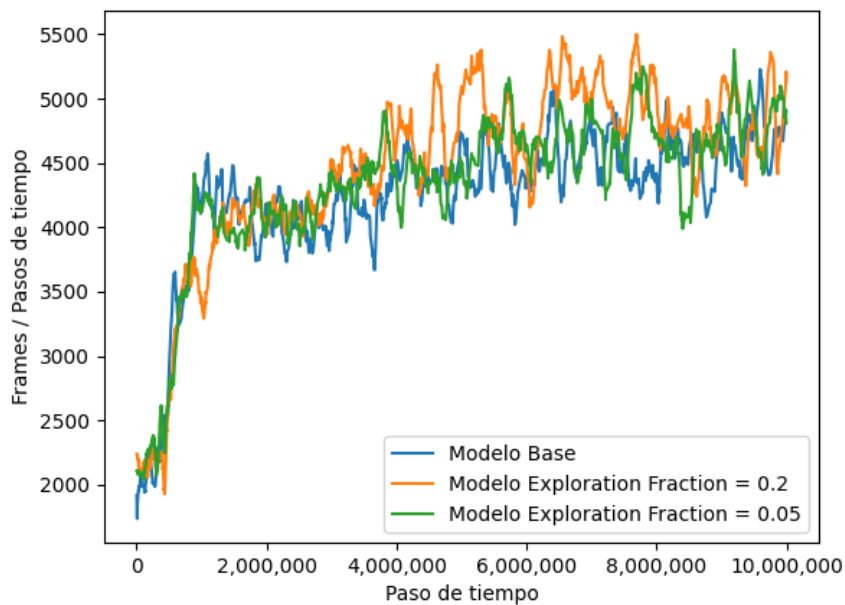
(b) Curva de longitud de episodio de los modelos con `exploration_fraction` modificado. La longitud de cada episodio se mide en `frames` o pasos de tiempo.

Figura 5.12: Curvas de entrenamiento de los modelos con `exploration_fraction` modificado y del modelo base para el juego Breakout.

Este comportamiento, aunque de forma algo menos distintiva, parece repetirse en las curvas de entrenamiento de *Space Invaders*, como puede verse en la Figura 5.13.



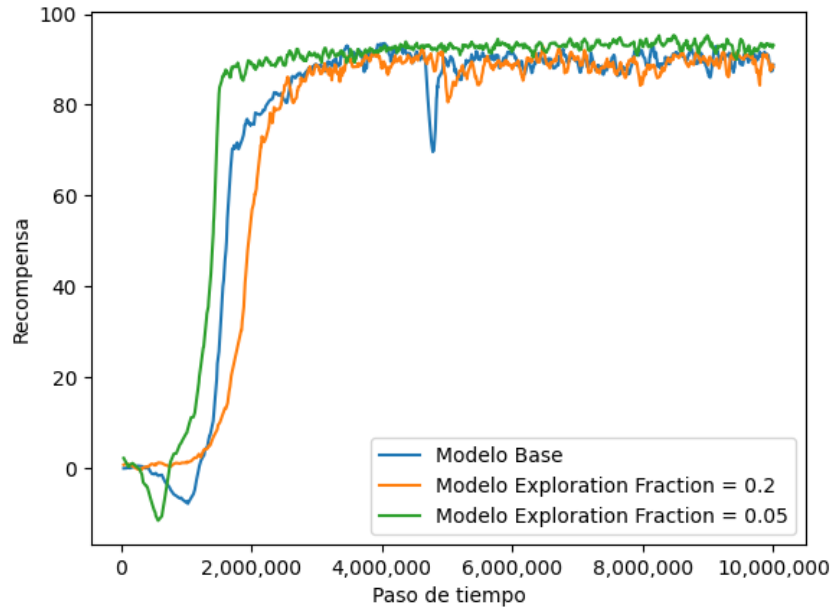
(a) Curva de recompensas de los modelos con `exploration_fraction` modificado.



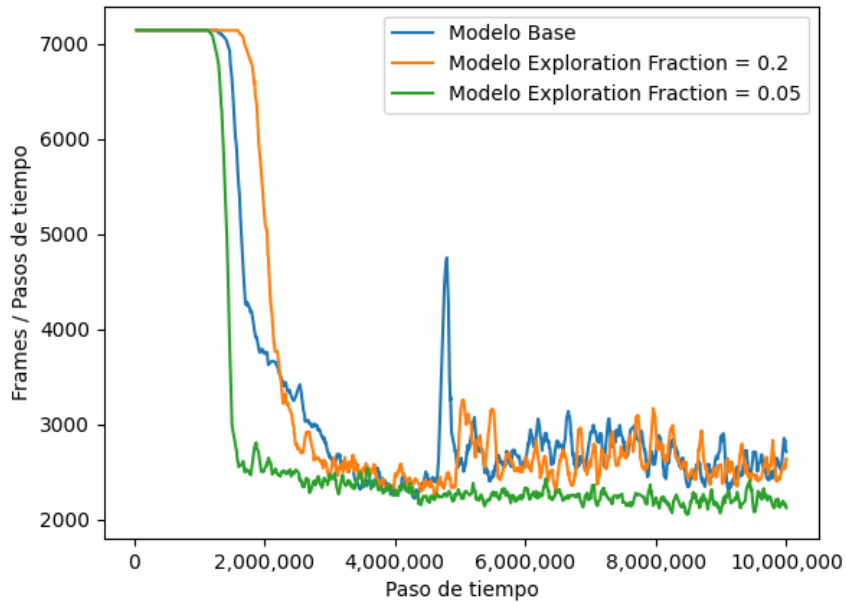
(b) Curva de longitud de episodio de los modelos con `exploration_fraction` modificado. La longitud de cada episodio se mide en `frames` o pasos de tiempo.

Figura 5.13: Curvas de entrenamiento de los modelos con `exploration_fraction` modificado y del modelo base para el juego Space Invaders.

En el caso del juego *Boxing*, cuyas curvas de entrenamiento se muestran en la Figura 5.14, el modelo que realiza una mayor cantidad de acciones exploratorias a lo largo del entrenamiento (es decir, aquel con una `exploration_fraction = 0.05`) logra converger a mayores valores de recompensa y reducir la longitud de episodio en una menor cantidad de *timesteps* que el resto de modelos presentados en la gráfica. Esto parece indicar que *Boxing* se beneficia de la exploración mucho más que *Breakout* y *Space Invaders*.



(a) Curva de recompensas de los modelos con `exploration_fraction` modificado.



(b) Curva de longitud de episodio de los modelos con `exploration_fraction` modificado. La longitud de cada episodio se mide en `frames` o pasos de tiempo.

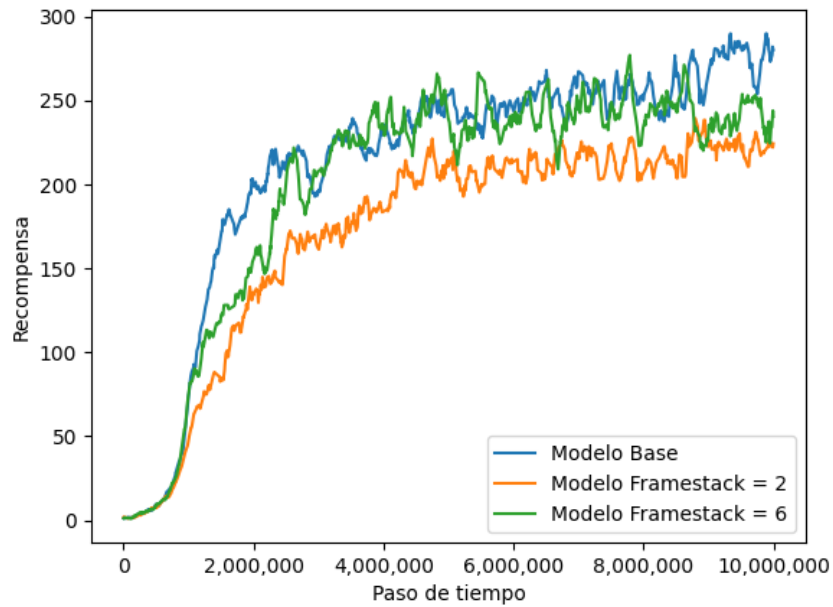
Figura 5.14: Curvas de entrenamiento de los modelos con `exploration_fraction` modificado y del modelo base para el juego Bo-xing.

§5.3.4 Framestack

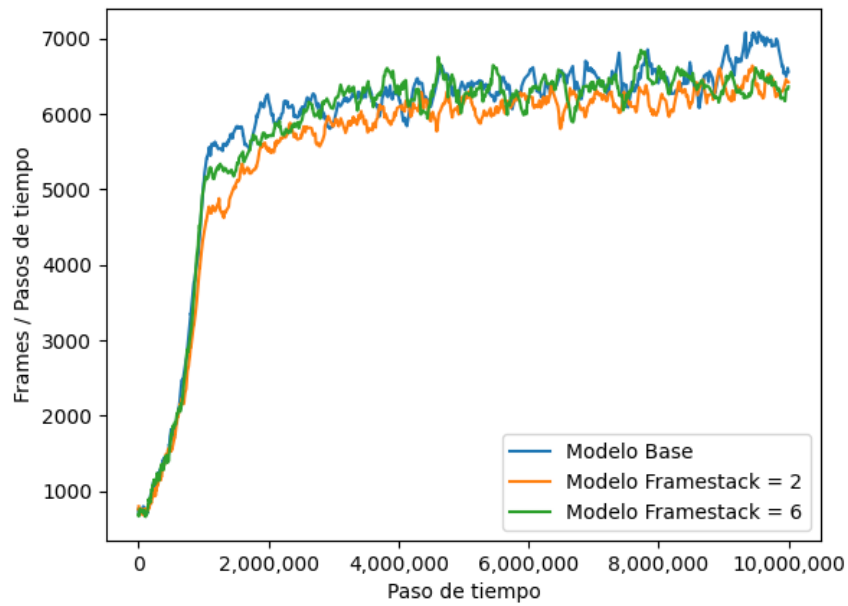
Para el entrenamiento de estos modelos, se realizó una reducción de la cantidad de *frames* que se proveen como entrada a la red neuronal convolucional, así como también un aumento de la misma, respecto a la cantidad utilizada en los hiperparámetros propuestos que, se recuerda, es de 4 *frames*⁸.

En la Figura 5.15 se muestran las curvas de entrenamiento de los modelos con `framestack` modificado y el modelo base para *Breakout*. En ellas, puede verse que los modelos cuyo valor de `framestack` fue modificado convergen a valores de recompensa menores que el modelo base. En cuanto a la longitud de episodio, no parecen mostrar diferencias demasiado significativas.

⁸Cabe destacar que, independientemente del valor del *framestack*, el agente sigue seleccionando acciones cada 4 *frames*, debido al *frame-skipping* utilizado en el preprocesamiento.



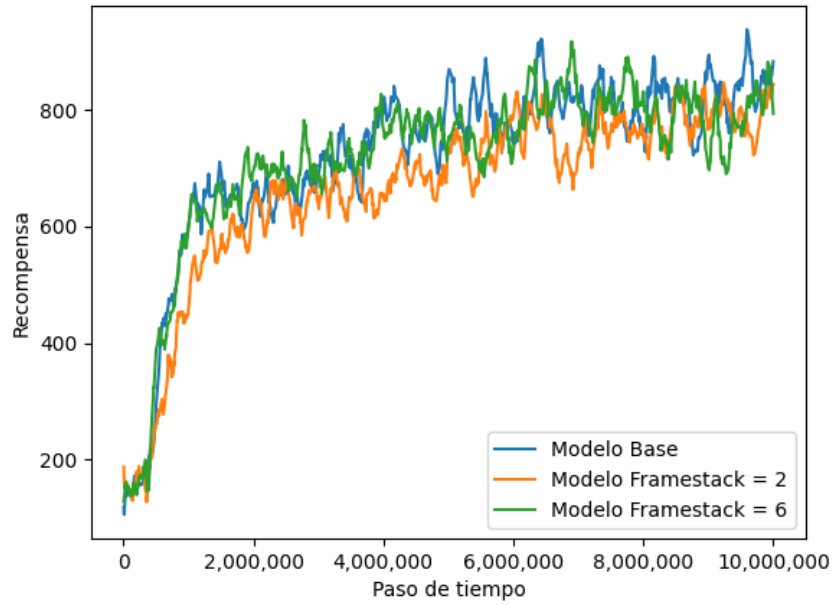
(a) Curva de recompensas de los modelos con **framestack** modificado.



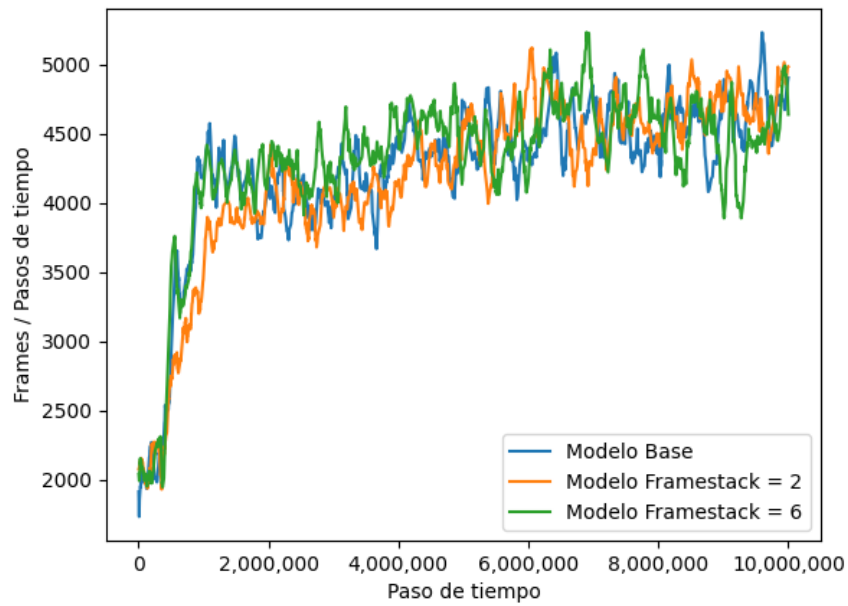
(b) Curva de longitud de episodio de los modelos con **framestack** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.15: Curvas de entrenamiento de los modelos con **framestack** modificado y del modelo base para el juego Breakout.

En cuanto al entrenamiento de los modelos en *Space Invaders*, mostrado en la Figura 5.16, no parece observarse ningún aspecto particularmente distintivo entre los modelos presentados.



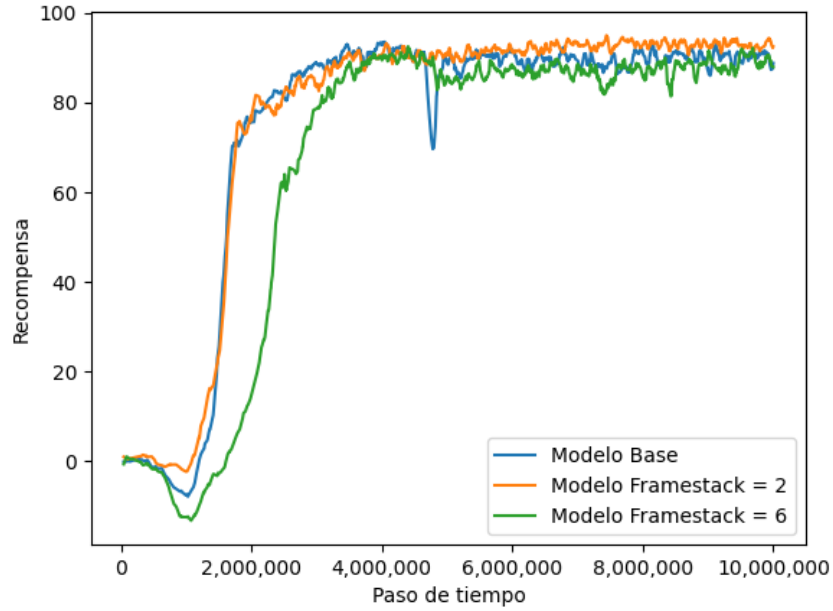
(a) Curva de recompensas de los modelos con **framestack** modificado.



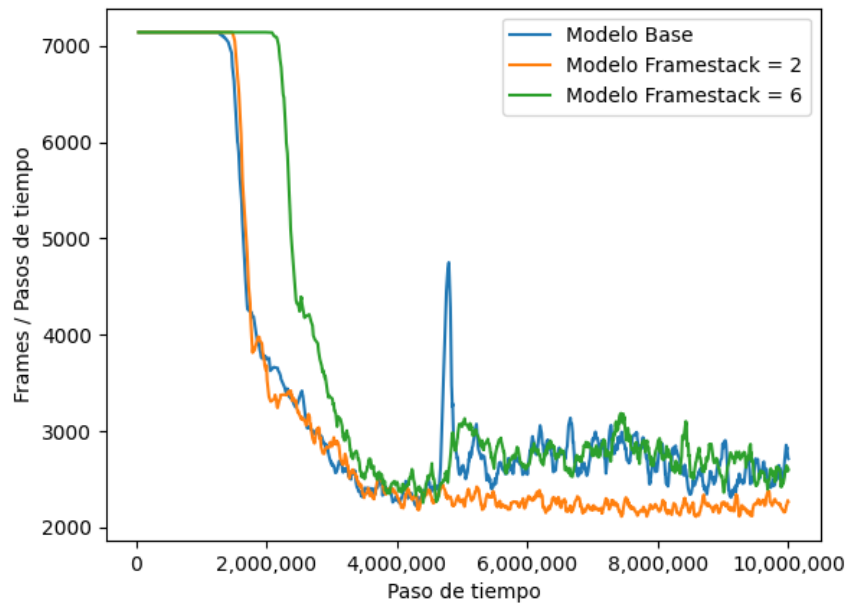
(b) Curva de longitud de episodio de los modelos con **framestack** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.16: Curvas de entrenamiento de los modelos con **framestack** modificado y del modelo base para el juego Space Invaders.

En el juego *Boxing*, el entrenamiento mostrado en la Figura 5.17, indica que reducir el `framestack` produce valores de recompensa similares al modelo base y una reducción en la longitud de episodio. Un aumento del mismo, sin embargo, causa que el modelo demore una mayor cantidad de *timesteps* en alcanzar recompensas altas. Nuevamente, y como ha sido el caso de todos los modelos presentados hasta aquí para este juego, el entrenamiento de los modelos converge en una cantidad de pasos de tiempo menor a la propuesta.



(a) Curva de recompensas de los modelos con **framestack** modificado.



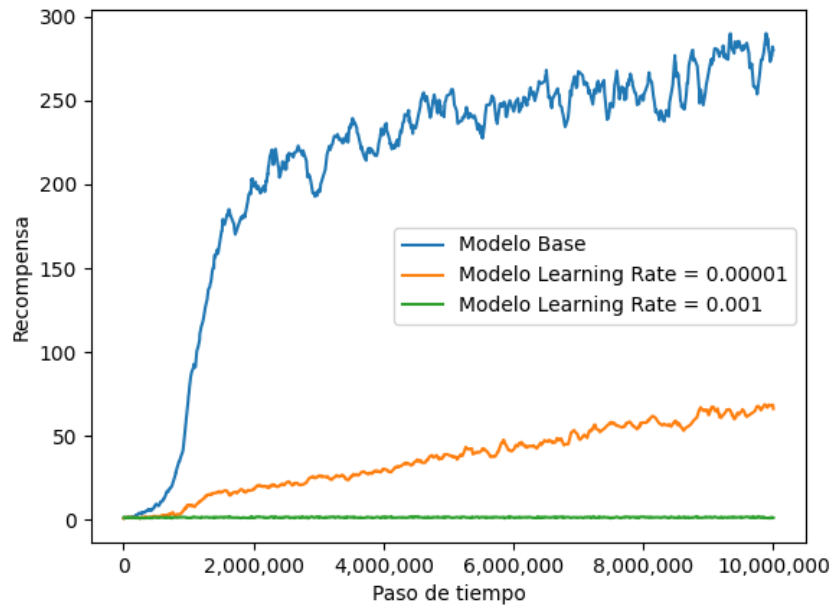
(b) Curva de longitud de episodio de los modelos con **framestack** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.17: Curvas de entrenamiento de los modelos con **framestack** modificado y del modelo base para el juego Boxing.

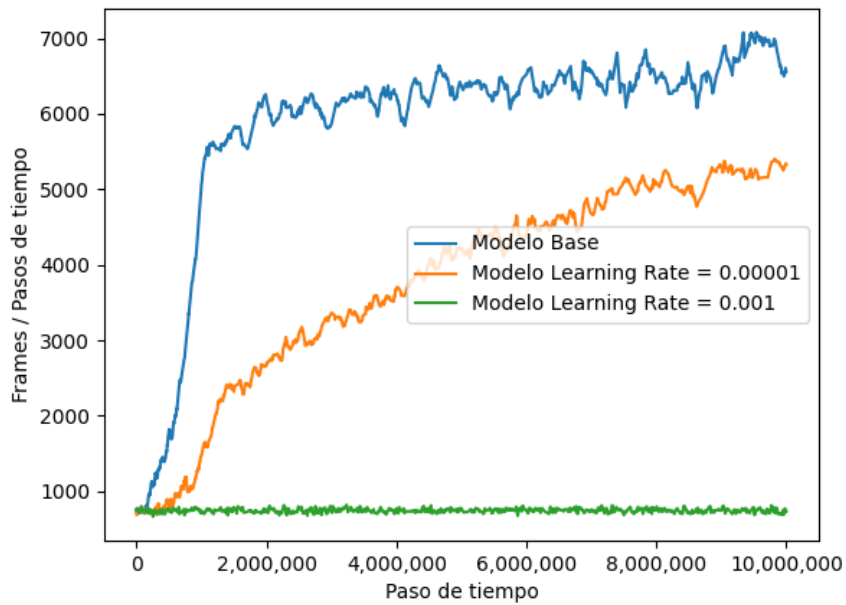
§5.3.5 Learning Rate

En estos modelos, se realizó una reducción y un aumento en el valor del hiperparámetro `learning_rate`, que controla el ajuste de los pesos de la red neuronal respecto a la función de pérdida.

Como puede observarse en la Figura 5.18, modificar el valor de este hiperparámetro produjo un entrenamiento con un desempeño significativamente peor que el del modelo base para *Breakout*.



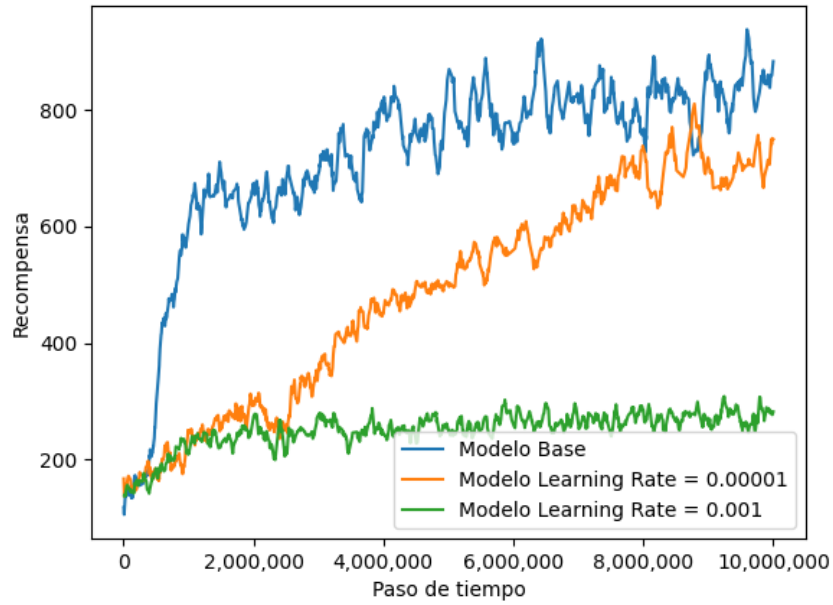
(a) Curva de recompensas de los modelos con **learning_rate** modificado.



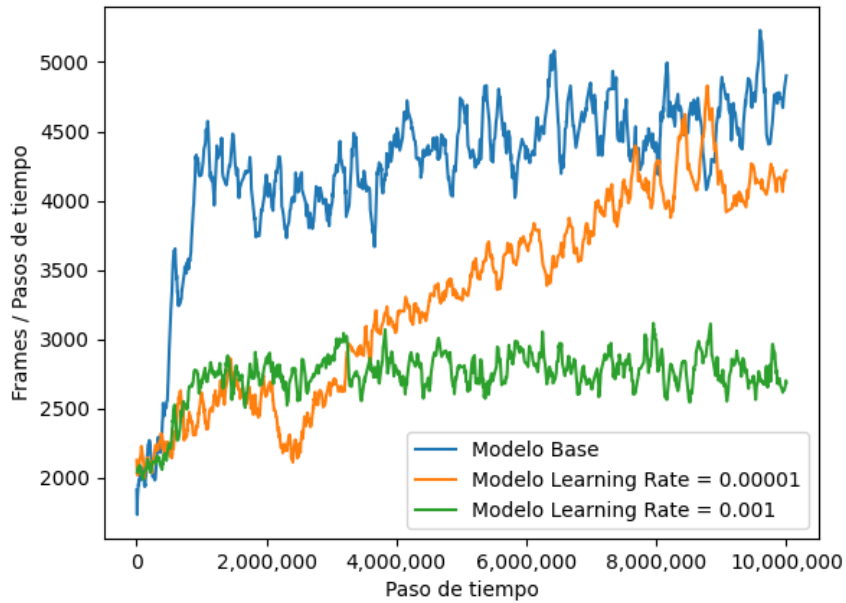
(b) Curva de longitud de episodio de los modelos con **learning_rate** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.18: Curvas de entrenamiento de los modelos con **learning_rate** modificado y del modelo base para el juego Breakout.

En *Space Invaders*, sin embargo, una reducción del valor de `learning_rate` parecería obtener valores de recompensa y longitudes de episodio algo más en línea con los valores esperados, como puede observarse en la Figura 5.19. Esto podría resaltar la diferencia del efecto que tienen los distintos hiperparámetros en cada juego particular.



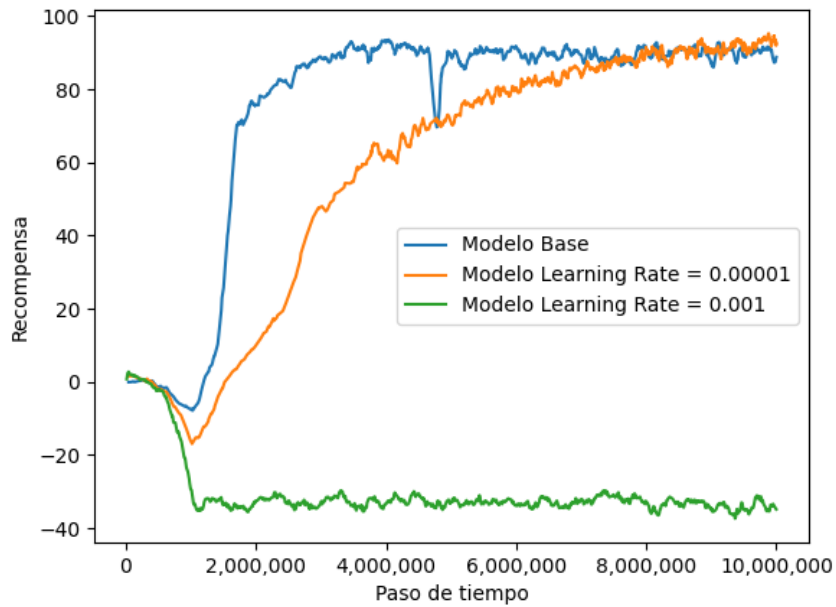
(a) Curva de recompensas de los modelos con **learning_rate** modificado.



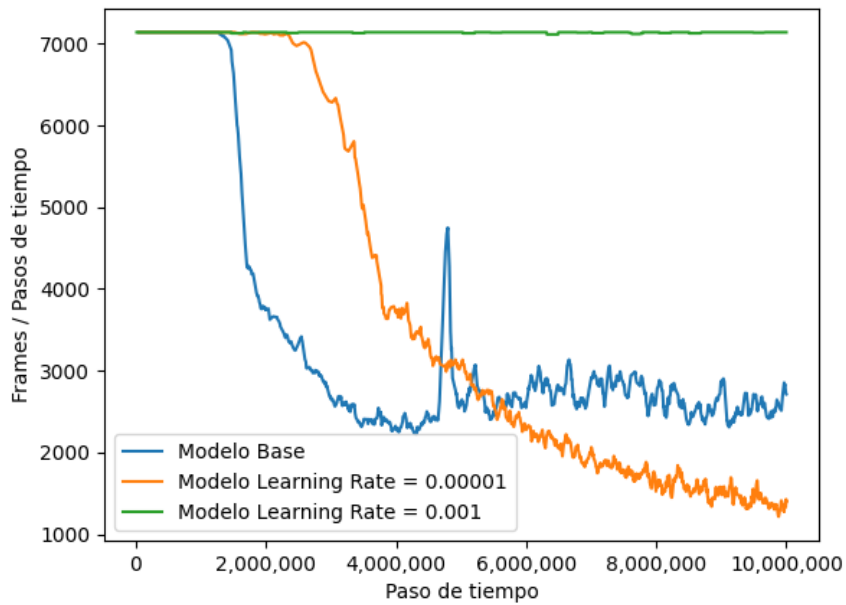
(b) Curva de longitud de episodio de los modelos con **learning_rate** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.19: Curvas de entrenamiento de los modelos con **learning_rate** modificado y del modelo base para el juego Space Invaders.

Finalmente, en *Boxing*, el entrenamiento presentó la forma mostrada en la Figura 5.20. Puede observarse que una reducción en el `learning_rate`, si bien en una mayor cantidad de *timesteps*, logra que el entrenamiento converja a los mismos valores de recompensa que el modelo base e, incluso, logra una longitud de episodio menor que éste. En cuanto al modelo restante, no logra un buen desempeño para nada, con recompensas negativas, que indican que el agente pierde contra su oponente, y una longitud de episodio constante de 7000 *frames* que, considerando que el juego se ejecuta a 60 *frames* por segundo, corresponden a los 2 minutos de tiempo máximo que puede durar una partida de *Boxing*.



(a) Curva de recompensas de los modelos con **learning_rate** modificado.



(b) Curva de longitud de episodio de los modelos con **learning_rate** modificado. La longitud de cada episodio se mide en **frames** o pasos de tiempo.

Figura 5.20: Curvas de entrenamiento de los modelos con **learning_rate** modificado y del modelo base para el juego Boxing.

Mostrado el entrenamiento de los distintos modelos entrenados, se pasará a realizar el análisis del desempeño obtenido por ellos en cada juego propuesto y, además, realizar una comparación con los resultados de las publicaciones originales de *deep Q-Learning*.

§5.4 Análisis de resultados

Como se mencionó previamente, el objetivo de llevar a cabo la experimentación aquí planteada fue el de tratar de responder algunas de las preguntas que surgen a partir de la lectura de la presentación del algoritmo *Deep Q-Learning*. Estas fueron: ¿qué tan importantes son los distintos hiperparámetros para el desempeño de un modelo entrenado con este algoritmo? ¿Se podría alcanzar un mejor desempeño si se hace foco en un juego en particular, en lugar de usar hiperparámetros que funcionen bien en una gran variedad de ellos? ¿Cómo afectan al desempeño los valores de los hiperparámetros elegidos? ¿Existen valores de hiperparámetros con los que se logre conseguir mejores resultados que utilizando aquellos publicados?

En la Tabla 5.1 se muestra la recompensa obtenida por cada uno de los modelos en *Breakout*, *Space Invaders* y *Boxing*, así como también los resultados reportados por el modelo entrenado por *DeepMind* y el desempeño de un jugador humano, también tomado de la publicación original de DQN [26]. Cabe destacar que la recompensa mostrada es la media obtenida por el agente correspondiente luego de completar 100 partidas del juego, en el caso de los modelos entrenados en esta experimentación; la media obtenida luego de 2 horas de juego, en el caso de los puntajes obtenidos por jugadores humanos; y el mejor episodio obtenido luego de jugar 135.000 *frames*, en el caso de los modelos originales propuestos por *DeepMind*.

Recompensa			
Modelo	Breakout	Space Invaders	Boxing
Modelos Base	287,02	736,5	92,4
Framestack = 2	328,8	880,1	97,94
Framestack = 6	328,24*	795,45	89,55
Learning Rate = 0,001	2	285	-25
Learning Rate = 0,00001	54,88	1057,3	97,48
Batch Size = 16	262,63	823,75	86,83
Batch Size = 64	221,44	730,1	94,81
Exploration Fraction = 0,05	317,85	705,25	90,9
Exploration Fraction = 0,2	255,52	791,7	92,8
Buffer Size = 50.000	274,15*	950,05	93,35
Buffer Size = 200.000	287,28	904,95	91,11
DeepMind	225	1075	71,8
Jugador Humano	31	3690	4,3

Tabla 5.1: Tabla que muestra las recompensas obtenidas por los distintos modelos entrenados, los originales propuestos por *DeepMind* y un jugador humano en cada uno de los juegos considerados. Los asteriscos indican puntajes en los que el modelo quedó atrapado en un loop infinito al jugar. Se resaltan los puntajes más altos, obtenidos por agentes de RL, para cada juego.

Aquellos modelos que, en su puntaje, poseen un asterisco (`Framestack = 6` y `Buffer Size = 50.000`), fueron propensos a quedar atrapados en un loop infinito en el juego *Breakout*, donde no pierden ninguna vida, pero tampoco rompen más ladrillos. Si bien, al no dejar caer la pelota, el agente no pierde puntuación (es decir, no se lo penaliza) tampoco hace que aumente, por lo que estos agentes aprendieron a, en cierto punto de la partida, repetir la misma acción una y otra vez, sin importar que su recompensa siempre sea 0. Esto puede deberse a que los resultados dependen del entrenamiento particular, que no es exactamente igual a otro, por más que posean los mismos parámetros y se entrene por la misma cantidad de tiempo. En otras palabras, los resultados dependen de la experiencia particular que el agente recolecte del ambiente, que puede no ser siempre la misma, dada la naturaleza del mismo, y de cómo se actualice la *Q-Network*. Esto hace que cada entrenamiento produzca un modelo distinto, pero que puede producir resultados similares dentro de cierto margen

de valores.

Como se puede observar en la Tabla 5.1, el agente que obtuvo una mayor recompensa en *Breakout* fue aquel que utiliza un *framestack* de 2 imágenes, con una media de 328,8 puntos. En *Space Invaders*, el modelo original de *DeepMind* obtuvo los mejores resultados, pero, de aquellos aquí entrenados, el agente que utiliza un *learning rate* de 0,00001 obtuvo la puntuación más alta, con una media de 1057,3 puntos. Sin embargo, el desempeño obtenido por los agentes de aprendizaje por refuerzo sigue siendo peor que el de un jugador humano, ya que aquí las estrategias para obtener puntajes altos se extienden a lo largo de mucho tiempo y, por ende, se debe considerar más que solamente el estado actual del ambiente [26]. En cuanto a *Boxing*, nuevamente el agente con un *framestack* de 2 imágenes vuelve a ser aquel que obtiene el mayor puntaje medio, con 97,94 puntos.

Resulta interesante destacar que, en aquellos modelos que tienen un buen desempeño, el agente aprende a desarrollar una estrategia que utiliza para maximizar su puntaje. En *Breakout*, el agente genera “canales” rompiendo los ladrillos ubicados uno arriba de otro para hacer que la pelota rebote en los ladrillos superiores y, en *Boxing*, el agente pone al oponente contra las cuerdas casi instantáneamente para poder golpearlo en repetidas ocasiones hasta conectar 100 golpes, sin que éste sea capaz de defenderse.

Otro aspecto importante es que en todos los casos existe más de un modelo que obtuvo mejores resultados que los modelos base. Esto demuestra que, si bien los hiperparámetros propuestos presentan un buen punto de partida, con una modificación pequeña se pueden obtener mejores resultados. Además, puede verse que se presentan juegos específicos que se benefician de ciertas modificaciones mientras que otros no. En este caso, reducir el *learning rate* produjo una mejora considerable en los puntajes obtenidos por el agente en *Space Invaders* y *Boxing*, mientras que en *Breakout* fue uno de los modelos con peor desempeño. Aquí se ve que concentrarse en modificar

los hiperparámetros respecto a un solo juego produce mejores resultados. Particularmente, los puntajes medios obtenidos por el agente en cada juego se benefician de distintos factores. En *Breakout*, un *framestack* más grande y más pequeño, realizar más acciones exploratorias (es decir, reducir la tasa de exploración más lentamente mediante la *exploration fraction*) e incrementar el tamaño de la memoria de replay, produjeron mejores resultados que el modelo base y los reportados por *DeepMind*. En *Space Invaders*, un *framestack* más grande y más pequeño, un menor *learning rate*, tomar un subconjunto más pequeño de experiencias de la memoria de replay (es decir, reducir el *batch size*), realizar menos acciones exploratorias (es decir, reducir la tasa de exploración más agresivamente mediante la *exploration fraction*) y utilizar tamaños de memoria de replay más pequeños y más grandes, produjeron mejores resultados que aquellos obtenidos por el modelo base. Finalmente, en *Boxing*, un *framestack* más pequeño, un *learning rate* más pequeño, un tamaño mayor del subconjunto de experiencias que se toman de la memoria de replay, realizar menos acciones exploratorias y reducir el tamaño de la memoria de replay, produjeron un beneficio en los puntajes obtenidos por el agente respecto a aquellos obtenidos en el modelo base y por *DeepMind*. Además, en contraste, se puede visualizar como, al incrementar el *learning rate* a 0,001 el agente es incapaz de aprender lo suficiente como para alcanzar un desempeño aceptable en el tiempo de entrenamiento estipulado.

El hecho de que el agente que utiliza un *framestack* de 2 imágenes haya obtenido el mejor desempeño en 2 de los 3 juegos presentados, puede llevar a concluir que, por más que debido al preprocesamiento aplicado la acción elegida por el agente se repita durante 4 *frames*, un *stack* de 2 imágenes como observación del estado del ambiente parecería ser suficiente para resolver la observabilidad parcial que se presenta en ellos, permitiendo que el agente obtenga toda la información necesaria para tomar una decisión sobre qué acción tomar, pudiendo determinar la posición de objetos que pueden moverse en la pantalla y hacia dónde se dirigen.

En cuanto a los tiempos de entrenamiento de los modelos planteados, en la Tabla 5.2 se observa que, si bien aquellos modelos que tuvieron mejor desempeño tomaron más tiempo en completar su entrenamiento, la diferencia en *performance* hace que este tiempo valga la pena. Además, puede observarse que ciertos hiperparámetros, como el *batch size* o un *framestack* con más imágenes, tienen efectos significativos en el tiempo de entrenamiento.

Modelo	Tiempo de entrenamiento		
	Breakout	Space Invaders	Boxing
Modelos Base	12h 39m	15h 25m	18h 11m
Framestack = 2	15h 10m	15h 12m	18h 4m
Framestack = 6	17h 10m	16h 48m	19h 31m
Learning Rate = 0,001	17h 1m	16h 18m	18h 10m
Learning Rate = 0,00001	16h 1m	15h 52m	18h 22m
Batch Size = 16	15h 18m	15h 7m	17h 11m
Batch Size = 64	17h 23m	17h 14m	20h 10m
Exploration Fraction = 0,05	15h 30m	15h 30m	18h 12m
Exploration Fraction = 0,2	15h 30m	15h 18m	18h 12m
Buffer Size = 50.000	15h 35m	15h 32m	18h 22m
Buffer Size = 200.000	15h 36m	15h 41m	18h 32m

Tabla 5.2: Tabla que muestra el tiempo que tomó el entrenamiento de cada uno de los modelos, medido en horas y minutos.

Es importante recordar que el tiempo de entrenamiento se mide en *timesteps*, que corresponden a *frames* o imágenes del juego, y que el tamaño propuesto fue de 10.000.000. Este valor pareció ser adecuado para la mayoría de situaciones, exceptuando aquellos modelos que debido a un *learning rate* muy pequeño incrementaban sus puntuaciones muy lentamente y, se podría especular, requerirían un valor mayor para alcanzar un desempeño similar al del resto de los modelos. Otra excepción se da en el caso de *Boxing*, donde el entrenamiento parece converger alrededor de los 4 millones de *timesteps* en la mayoría de los modelos, por lo que utilizar 10.000.000 en

este juego en particular es excesivo y produce que se prolongue el tiempo de entrenamiento innecesariamente, lo que resalta nuevamente la importancia de adaptar los hiperparámetros a cada juego con el que se esté trabajando.

En síntesis, se observó que, si bien los hiperparámetros propuestos por las publicaciones originales de *Deep Q-Learning* pueden ser utilizados como un buen punto de partida y producen un desempeño muy bueno en la mayoría de los juegos, modificarlos en función del juego en particular que se esté utilizando produce un incremento en el desempeño del agente, ya que distintos juegos pueden beneficiarse de distintos valores de hiperparámetros. Además, el uso de estos hiperparámetros no garantiza los mismos resultados exactos que aquellos publicados, debido a las variaciones que pueden producirse en el entrenamiento individual de cada modelo. Como aspecto final, se destacan las limitaciones de los enfoques de aprendizaje por refuerzo respecto a ambientes en los que, si bien se pueden realizar modificaciones que los hagan más *markovianos*, la estrategia (o política) que se debe seguir para obtener un alto desempeño en ellos requiere considerar distintos aspectos además del estado actual, como eventos del pasado o aquellos que ocurrirán en el futuro distante.

□

CAPÍTULO VI

CONCLUSIONES Y TRABAJO FUTURO

El resurgimiento del aprendizaje por refuerzo en la última década trajo consigo muchos enfoques novedosos para producir resultados realmente nunca antes vistos en el área, sobre todo en el aprendizaje de estrategias que permiten a un agente obtener un desempeño superior al de los mejores jugadores humanos, tanto en juegos de mesa como en videojuegos. Además, recientemente, se mostró que los métodos utilizados para entrenar a este tipo de agentes, pueden ser aplicados efectivamente para resolver problemas de otro tipo, como la obtención de nuevos algoritmos, planteando la tarea como un juego. Tal vez el mas resonante en los últimos meses es ChatGPT [4], un modelo de lenguaje generativo sucesor de InstructGPT [29] basado en un sistema de aprendizaje por refuerzo con retroalimentación humana (RLHF) propuesto en [12]. ChatGPT ha generado una verdadera conmoción en distintos ámbitos científicos, académicos y educativos realizando tareas que, hasta el presente, sólo estaban reservadas para el ser humano.

Este desarrollo de nuevos enfoques requiere poder entender y analizar cómo funcionan aquellos ya existentes: qué los hace tan efectivos, dónde se presentan sus limitaciones, si es posible hacer algo respecto a ellas, entre otros aspectos. En este trabajo, se mostró que llevar a cabo un análisis de este tipo y, además, en línea con la tarea que se busca resolver, a la hora de utilizar estos métodos del *estado del arte* resulta beneficioso respecto a los resultados que se obtienen.

En cuanto a la aplicación de conceptos adquiridos a los largo del cursado de la Licenciatura en Ciencias de la Computación durante el desarrollo de este trabajo, se destacan: los procesos de decisión Markov, 2 de los 3 algoritmos aquí presentados para resolverlos (Value Iteration y Q-Learning) y los conceptos de agentes y ambientes, vistos en el curso “Inteligencia Artificial”; y tanto las redes neuronales como las redes neuronales convolucionales, con las que se trabajó en la optativa “Aprendizaje Automático y Minería de Datos”.

Además de la puesta en práctica de conceptos adquiridos durante el transcurso de la carrera, este trabajo requirió del aprendizaje de nuevas herramientas, lenguajes de programación y enfoques para poder ser completado. En particular, el lenguaje *Python*, las bibliotecas de *reinforcement learning* *OpenAI Gym* y *Stable-Baselines3*, la visualización de datos con *Matplotlib*, y el algoritmo *deep Q-Learning*. Si bien el lenguaje *Python* en particular no es utilizado durante el cursado de las asignaturas del plan de estudio, conceptos que permiten la adquisición y utilización de nuevos lenguajes son ideas claves de cursos como “Introducción a la Computación”, “Programación I”, “Programación II” y “Análisis Comparativo de Lenguajes”.

Cabe destacar, adicionalmente, la capacidad de análisis y el deseo de entendimiento que se adquiere a través del estudio de la Licenciatura en Ciencias de la Computación. Un aspecto fundamental no sólo para el desarrollo de la investigación introducida aquí, sino también para el aprendizaje de nuevos conceptos, que es una habilidad mandataria debido a lo rápidamente cambiante de esta disciplina.

§6.1 Trabajo Futuro

Si bien en la experimentación realizada en este trabajo se ajustaron los valores de algunos hiperparámetros, un aspecto que resultaría interesante observar es de qué

forma afectan los resultados aquellos que no se han modificado hasta ahora. Además, basado en estos resultados, realizar el entrenamiento de modelos que combinen distintos valores de estos hiperparámetros, en lugar de modificarlos de uno en uno, y analizar los resultados que producen. De forma adicional, se podría modificar el pre-procesamiento de las imágenes del juego que se proveen como entrada a la *Q-Network*, haciéndolas más grandes, más pequeñas o modificando la cantidad de *frame-skipping*.

Otro posible aspecto a analizar es el de la arquitectura de la red neuronal convolucional utilizada en *deep Q-Learning*, modificándola y observando los resultados que se obtienen, trabajando bajo el supuesto de que pueda existir una arquitectura que produzca resultados similares a los mostrados siendo más sencilla o que pueda existir otra que produzca resultados incluso mejores.

Es importante destacar que el mismo análisis que fue realizado en este trabajo, e incluso los mencionados más arriba, podría ser ampliado a un subconjunto más grande de juegos de *Atari*, dada la cantidad de juegos de los que se cuenta con resultados provistos en las publicaciones. Por otro lado, se podría proceder de la misma manera con el resto de los algoritmos provistos por la biblioteca *Stable-Baselines3* e, inclusive, realizar una comparación entre cada uno de ellos de acuerdo a su forma de trabajo y resultados obtenidos.

Como consideración final, debido a la gran cantidad de ambientes provistos por *OpenAI Gym*, existe la posibilidad de explorar otros dominios de aplicación del aprendizaje por refuerzo, o hasta definir un ambiente personalizado, lo que abre la puerta a otra gran cantidad de aspectos a analizar.

□

BIBLIOGRAFÍA

BIBLIOGRAFÍA

- [1] M Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey. January 2021.
- [2] Meta AI. Pytorch. <https://pytorch.org/>.
- [3] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. July 2017.
- [4] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity, 2023.
- [5] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.
- [6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. <https://www.gymnasium.dev/>.
- [7] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI gym. June 2016.
- [8] Murray Campbell, A. Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.
- [9] Erin Catto. Box2d. <https://box2d.org/>.
- [10] Erin Catto. Box2d. <https://github.com/erincatto/box2d>.
- [11] Tai-Liang Chou and Yu-Ling Hsueh. A task-oriented chatbot based on LSTM and reinforcement learning. In *Proceedings of the 2019 3rd International Conference on Natural Language Processing and Information Retrieval*, New York, NY, USA, June 2019. ACM.
- [12] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In I. Guyon,

- U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.
- [13] DeepMind. Mujoco. <https://mujoco.org/>.
 - [14] DeepMind. Mujoco github. <https://github.com/deepmind/mujoco>.
 - [15] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. <https://github.com/openai/baselines>, 2017.
 - [16] Carlos Diuk, Andre Cohen, and Michael L Littman. An object-oriented representation for efficient reinforcement learning. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, New York, New York, USA, 2008. ACM Press.
 - [17] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, October 2022.
 - [18] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. February 2018.
 - [19] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. January 2018.
 - [20] Liwei Huang, Mingsheng Fu, Fan Li, Hong Qu, Yangjun Liu, and Wenyu Chen. A deep reinforcement learning based long-term recommender system. *Knowledge-Based Systems*, 213:106706, 2021.
 - [21] Jim Hugunin. Numpy. <https://numpy.org/>.
 - [22] Jens Kober, J Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *Int. J. Rob. Res.*, 32(11):1238–1274, September 2013.
 - [23] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. September 2015.
 - [24] Shrikant Malviya, Piyush Kumar, Suyel Namasudra, and Uma Shanker Tiwary. Experience replay-based deep reinforcement learning for dialogue management optimisation. *ACM trans. Asian low-resour. lang. inf. process.*, May 2022.

- [25] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. February 2016.
- [26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. December 2013.
- [27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dhharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.
- [28] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. 2019.
- [29] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.
- [30] Antonin Raffin. Hiperparámetros propuestos por stable-baselines3. <https://github.com/DLR-RM/rl-baselines3-zoo/blob/master/hyperparams/dqn.yml>.
- [31] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3 github. <https://github.com/DLR-RM/stable-baselines3>.
- [32] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.
- [33] Martin Riedmiller, Thomas Gabel, Roland Hafner, and Sascha Lange. Reinforcement learning for robot soccer. *Auton. Robots*, 27(1):55–73, July 2009.
- [34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

- [35] Stuart Russell and Peter Norvig. *Artificial intelligence*. Pearson, Upper Saddle River, NJ, 3 edition, December 2009.
- [36] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.
- [37] A. L. Samuel. Some studies in machine learning using the game of checkers. ii—recent progress. *IBM Journal of Research and Development*, 11(6):601–617, 1967.
- [38] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. November 2015.
- [39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. July 2017.
- [40] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.
- [41] Richard S Sutton and Andrew G Barto. *Reinforcement Learning*. Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, February 1998.
- [42] Richard S Sutton and Andrew G Barto. *Reinforcement Learning*. Adaptive Computation and Machine Learning series. Bradford Books, Cambridge, MA, 2 edition, November 2018.
- [43] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, March 1995.
- [44] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. September 2015.
- [45] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P Agapiou, Max Jaderberg, Alexander S Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, November 2019.

- [46] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. November 2015.
- [47] Christopher J C H Watkins and Peter Dayan. Q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.
- [48] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge, 1989.
- [49] Changxi You, Jianbo Lu, Dimitar Filev, and Panagiotis Tsiotras. Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning. *Robotics and Autonomous Systems*, 114:1–18, 2019.