Universitat Rovira i Virgili (URV) y Universitat Oberta de Catalunya (UOC)

Máster Universitario en Ingeniería Computacional y Matemática

TRABAJO FINAL DE MÁSTER

Área: Inteligencia Artificial

Aprendiendo de la memoria RAM de la NES

Autor: Daniel Barajas Higuera

Tutor: Samir Kanaan Izquierdo

Profesor: Carles Ventura Royo

Barcelona, 15 de septiembre de 2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	Aprendiendo de la Memoria RAM de la NES
Nombre del autor:	Daniel Barajas Higuera
Nombre del colaborador/a docente:	Samir Kanaan Izquierdo
Nombre del PRA:	Carles Ventura Royo
Fecha de entrega (mm/aaaa):	09/2020
Titulación o programa:	Máster en Ingeniería Computacional y Matemática
Área del Trabajo Final:	Inteligencia Artificial
Idioma del trabajo:	Español
Palabras clave	Deep Q-Learning, NES, RAM

Dedicatoria

A la mujer que inspiró que esto fuera posible, Samus Aran.

Abstract

In this project, reinforcement learning algorithms are explored to play the games Donkey Kong, Ice Climber, Kung Fu, Super Mario Bros, and Metroid from the NES console. The Deep-Q learning algorithm is used to experiment with the RAM representation of the state. DQN algorithm extensions as Double DQN and Dueling Network Architectures are explored too. Some simple strategies to reduce the state-space and action-space are proposed in addition to reward functions to create an easy to train agent with low computational resources. Two ways to reduce the RAM representation of the state were tested, RAM map method worked well just in the training phase meanwhile the activated bytes method get better results also in gameplay.

Keywords: DQN, RAM, NES.

Resumen

En este proyecto se experimentó con algoritmos de aprendizaje por refuerzo jugando Donkey Kong, Ice Climber, Kung Fu, Super Mario Bros y Metroid de la consola NES. El algoritmo DQN y sus variantes como Doble DQN y Dueling DQN fueron usados para experimentar con la representación de estados mediante la RAM. Se proponen algunas estrategias para reducir la dimensionalidad de los estados y las acciones. También se proponen funciones de recompensa para crear un agente fácil de entrenar con pocos recursos computacionales. Se probaron dos maneras de reducir la dimensión de la RAM, el mapa de la RAM funcionó bien sólo en fase de entrenamiento mientras que el método de los bytes activados consiguió mejores resultados.

Palabras clave: DQN, RAM, NES.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto y justificación

Los videojuegos se han convertido en un entorno conveniente para experimentar en la construcción de agentes que simulan la inteligencia humana. En años recientes un grupo de investigación logró que por primera vez un agente fuese capaz de mostrar habilidades sobrehumanas en juegos de la Atari 2600.

Metroid es un juego creado para la consola NES en 1985 que combina plataformas 2D y exploración en el cual no hay una recompensa delimitada diferente a terminar el juego. Esta consola al ser una sucesora de la Atari presenta juegos con mayor nivel de complejidad que Atari.

1.2. Objetivos

El propósito de este proyecto es crear un agente capaz de jugar Metroid. En este juego se le proporciona muy poca información visual al jugador sobre su avance, no hay mapas ni puntajes. Este exceso de dificultad puede compensarse usando información de la memoria RAM del juego en lugar de los pixeles de la pantalla. Los objetivos generales del proyecto se listan a continuación.

- Crear un agente que juegue Metroid de NES (Nintendo Enterteiment System).
- Comparar la eficiencia entre algunos algoritmos de aprendizaje por refuerzo.
- Probar el desempeño del agente con otros juegos de la consola.

1.3. Visión general de la memoria

Esta memoria está compuesta por el estudio del algoritmo DQN y los elementos necesarios para su implementación en cinco juegos de la consola NES, el capítulo 2 cubre el estado del arte que se consultó para planear el proyecto desde el punto de vista teórico, mientras que el capítulo 3 resume los algoritmos y las técnicas usadas en la implementación. El capítulo 4 describe los juegos en los cuales fueron probados los algoritmos para los cuales se propuso la metodología para abordarlos en el capítulo 5. Los resultados de la implementación son presentados en el capítulo 6. El capítulo 7 presenta las conclusiones obtenidas y el trabajo futuro propuesto.

Capítulo 2

Estado del Arte

2.1. Juegos y aprendizaje por refuerzo

Cuando se habla de inteligencia artificial, machine learning o ciencia computacional es inevitable mencionar a Alan Turing; él planteaba que podemos enseñarle a las máquinas de una manera análoga de la misma manera que se le podría enseñar a un cerebro humano. En 1950 establece una diferencia entre lo que llama "comportamiento completamente disciplinado" y "comportamiento inteligente" [28]. En ambos, se tiene una idea del comportamiento de la máquina. Por un lado, en el *comportamiento completamente disciplinado* se tiene una completa certeza, "una clara fotografía del estado de la máquina en cada momento", se conocería qué está programado en su totalidad. Por otro lado, en el *comportamiento inteligente*, la fotografía no es completa, el conocimiento de la máquina estaría limitado, si bien "hasta cierto punto el profesor puede predecir el comportamiento del alumno, no necesariamente tiene una idea de cómo se está educando la máquina"[21].

En 1997 Deep Blue, un computador creado por IBM para jugar ajedrez derrotó al campeón de ajedrez, Garry Kasparov con resultado de 3.5 a 2.5. Deep Blue empleaba un proceso de búsqueda previamente programado apoyado en una base de datos de juegos conocidos. Veinte años más tarde AlphaGo Master, creado por DeepMind, derrotó a Ke Jie, el entonces jugador número uno de Go del mundo. AlphaGo también contaba con una base de datos de movimientos del juego. Posteriormente fue creado AlphaZero, un sucesor de este programa cuya principal diferencia era que la única información inicial eran las reglas del juego. Finalmente AlphaZero fue capaz de derrotar a su predecesor AlphaGo con un marcador de 100-0. Si bien ambos se consideran hitos de la inteligencia artificial, Deep Blue se podría catalogar como lo que Turing llamaría un comportamiento completamente disciplinado, mientras que a AlphaZero, el cual aprendió únicamente jugando, se le podría denominar un verdadero comportamiento inteligente.

Los juegos han sido un entorno conveniente para experimentar en la construcción de agentes que simulan la inteligencia humana. El primer intento de un programa capaz de ganarle a un humano en un juego puede atribuirse a Samuel [20], quien lo terminó en 1955 y posteriormente lo mostró en 1956 en la televisión. Su programa implementa principios del aprendizaje por diferencia temporal TD (temporal difference learning) incluso antes de que este principio fuera analizado y descrito. Su programa al igual que AlphaZero adquiría habilidad jugando solo y recibía como única información inicial las reglas del juego.

Con este mismo principio, en 1992 Tesauro [24] crea un programa capaz de jugar Backgammon, evaluando mediante el uso de una red neuronal la posición del juego, entonces este programa llamado TD-Gammon fue el primer programa en alcanzar el nivel de juego que tenían los jugadores humanos, la red neuronal actualizaba los valores de las acciones que predecía usando TD y actualizaba los pesos con propagación hacia atrás.

En el campo de los videojuegos, Tetris es en un importante punto de referencia para la investigación en inteligencia artificial. Las primeras aproximaciones realizadas por Bertsekas y Tsitsiklis (1996) [26] mediante programación dinámica, lograron un puntaje superior a 2.800. Szita y Lorincz [23] plantearon una mejora sobre el algoritmo y posteriormente en 2009 el agente de Thiery and Scherrer [25], llamado BCTS basado en el trabajo de Szita y Lorincz, era capaz de lograr 35 millones de puntos. BCTS. Ganó una competencia de aprendizaje por refuerzo RL (Reinforcement learning) [6].

En 2013 se presentó el Arcade Learning Environment (ALE) [7]. Un entorno que hace uso del emulador de la consola de videojuegos Atari 2600, sobre el cual se aplican algoritmos estándar de RL con una función de aproximación lineal entre otros. Este entorno también permite leer directamente los 128 bytes de memoria la RAM de la consola.

En 2013 DeepMind, una compañía dedicada a la investigación y desarrollo en inteligencia artificial, propone un algoritmo llamado Deep Q-learning Network (DQN), que combina el Q-learning con redes neuronales convolucionales y lo utiliza para jugar varios juegos de la consola Atari 2600 [16] utilizando el ALE. En los programas realizados hasta el momento los valores de entrada del agente eran adaptados para representar características específicas del problema a resolver. En cambio en este proyecto el agente era capaz de jugar 7 juegos de la consola usando como datos de entrada los píxeles de la entrada de vídeo (210 $\times$ 160 RGB video a 60Hz), es decir, veía de la misma manera que un humano vería un juego.

Dos años más tarde los investigadores de DeepMind presentaron los resultados del algoritmo DQN más estable gracias a algunas mejoras y mostraron su desempeño incrementando el número de juegos a 49 [17]. El algoritmo DQN aprendía una política diferente con cada juego pero la arquitectura de la red y los parámetros de entrada eran los mismos. El resultado de este algoritmo fue que superó el desempeño del nivel de un humano en 29 de los 49 juegos.

En 2016 DeepMind nuevamente presenta un algoritmo denominado Doble DQN (DDQN) [29] el cual incorpora una red neuronal adicional en su arquitectura con la idea de reducir la sobreestimación. Muestran que es más eficiente y más estable y en [14] señalan algunos problemas que DQN pueden presentarse con DQN y proponen algunas soluciones; además usan un aproximador lineal para realizar un benchmark comparándolo con DQN. Finalmente introducen otra nueva arquitectura de redes neuronales llamada Dueling DQN que en combinación con otras mejoras lleva a un nuevo nivel el estado del arte [30]. Posteriormente en 2018 estudian empíricamente la combinación de las extensiones de DQN [12].

En 2017 DeepMind también realizó pruebas con una versión asíncrona y que funciona en paralelo de 4 algoritmos [18]. Los denomina Asynchronous one-step Q-learning, Asynchronous one-step Sarsa, Asynchronous n-step Q-learning y Asynchronous advantage actor-critic (A3C) destacando un importante desempeño y eficiencia de este último.

También en 2016, Sygnowski y Michalewski [22] aplican para 3 juegos y usan el mismo algoritmo DQN inicial con algunas mejoras, con la diferencia de que como entradas no utilizan la entrada de video, si no que en su lugar como entrada se utiliza la información de la memoria RAM de los juegos de Atari 2600 y obtienen resultados similares.

OpenAI hace público el entorno OpenAi Gym [11] en 2016 el cual es un kit de herramientas para la investigación y desarrollo de algoritmos de RL. Dentro de sus entornos se encuentra entre otros el ALE. Además en 2017 OpenAI propone una familia de métodos de optimización para RL mas sencillos de implementar comparados con los tradicionales: PPO, PPO2, ACER,y TRPO. Además integra su implementación al entorno OpenAI Gym.

En 2017 se introdujo otro entorno, el Retro Learning Enviroment (RLE) [10], el cual permite el uso de otros emuladores de consolas de videojuegos como la Super Nintendo (SNES) o Megadrive entre otras. Con la introducción de este entorno se realizan pruebas con los algoritmos DQN, DDQN y Dueling DQN en los juegos F-Zero, Gradius 3, Mortal Kombat, Super Mario World y Wolfenstein de SNES.

En 2018 se presenta el entorno Gym Retro [19] el cual apunta a la flexibilidad y versatilidad para crear múltiples entornos de RL mediante el uso de emuladores de las consolas, Atari2600, TurboGrafx-16 Game Boy,Game Boy Color,Game Boy Advance, Nintendo Entertainment System (NES), Super Nintendo Entertainment System (SNES), GameGear, Mega Drive y Master System. Gym Retro está integrado en OpenAI Gym.

En 2019 OpenAI Five se convierte en la primera inteligencia artificial capaz de derrotar a los campeones mundiales de un juego por equipos de e-sports, Dota2. Esto se considera un gran hito en el avance del campo de la inteligencia artificial dado que esta victoria ha mostrado características complejas como el trabajo colaborativo en equipo.

Capítulo 3

Aprendizaje por refuerzo

El psicólogo Edward Thorndike formulaba a comienzos del siglo XX que cuando individuo realiza una acción y obtiene una respuesta satisfactoria, tiene a futuro mayor probabilidad de realizar dicha acción, y en caso de percibir un estímulo negativo, esa probabilidad de repetir esta acción disminuye [15]. A este concepto le llamó la ley del efecto. Décadas mas tarde Alan Turing plateaba ideas para hacer que una maquina fuera inteligente aplicando algo parecido a la ley del efecto, a este tipo de máquinas les llamaba «sistemas pleasure-pain» [27]. Harry Klopf en 1972 se refería a este concepto como la hipótesis de la «neurona hedonista» [13], refiriéndose a que en el aprendizaje, las neuronas trabajan para maximizar el placer y disminuir el dolor.

Cuando se habla de aprendizaje de maquina normalmente se menciona que hay dos tipos, supervisado y no supervisado, sin embargo, esta idea mencionada arriba no encaja como tal en ninguno de los dos. El aprendizaje por refuerzo es un tipo de aprendizaje que cuenta con algunos elementos de los anteriores y cuyo objetivo es que dicha maquina a la cual se le denomina *Agente* sea capaz de tomar decisiones para alcanzar una meta determinada.

3.1. Elementos del aprendizaje por refuerzo

En el aprendizaje por refuerzo se denomina como agente al software creado para aprender. Este agente actúa sobre un entorno sobre el cual percibe observaciones o estados. A partir de las observación o *estado* que hace el agente, toma una *acción* o decisión la cual es retroalimentada mediante un nuevo *estado* u observación del entorno y una respuesta o estimulo el cual es llamado *recompensa*. Este ciclo se repine durante lo que se denomina un episodio.

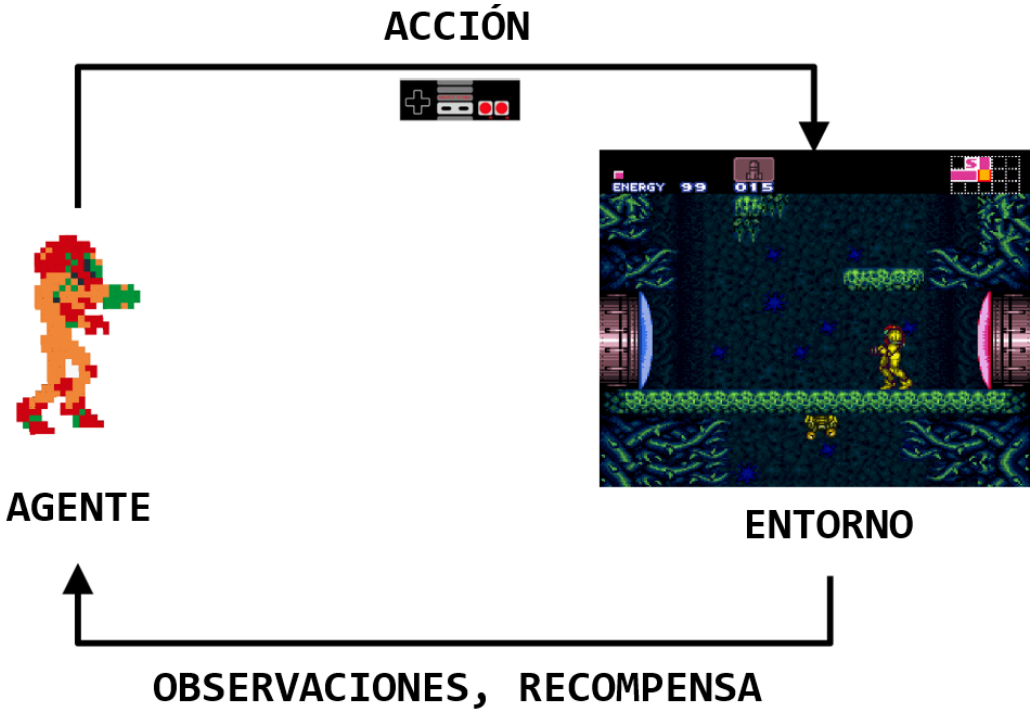


Figura 3.1: *Elementos del aprendizaje por refuerzo.*

Entre un estado y el siguiente, el agente recibe una recompensa, este valor indica al agente que tan buenas son sus acciones inmediatamente. La recompensa acumulada que se espera en el largo plazo se denomina función de valor. La función de valor representa la calidad a largo plazo de un conjunto de estados. El objetivo del agente es maximizar la recompensa total acumulada durante el episodio o periodo de juego. El criterio con el cual el agente toma las decisiones sobre cual acción debe tomar se denomina *política*.

3.2. Q-Learning

El algoritmo Q-learning intenta predecir el valor esperado de las recompensas las acciones. Así como en tiempo futuro el dinero no vale lo mismo que en el presente, las recompensas futuras tienen un valor ligeramente menor que se estima utilizando una tasa o factor de descuento γ. Esta predicción es una función de valor que recibe un estado s (state) y una acción a (action), y se calcula:

$$Q(s,a) = r_t + \gamma^1 \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + ... + \gamma^n \cdot r_{t+n} \tag{3.1}$$

$$Q(s,a) = r + \underbrace{\gamma \cdot r_{t+1} + \gamma \cdot r_{t+2} + ... + \gamma \cdot r_{t+n}}_{\gamma\ Q(s',a')} \tag{3.2}$$

siendo s' el siguiente estado y a' la acción optima. Esta ecuación puede es equivalente a la ecuación de Bellman [8] [9].

$$Q(s,a) = r_t + \gamma \cdot \max_{a'} Q(s',a') \tag{3.3}$$

En un entorno determinístico, la ecuación de Bellman es suficiente para predecir el valor esperado de la recompensa total, sin embargo, cuando no se sabe como se comporta exactamente el entorno y solo se conocen estados y acciones se utiliza la diferencia temporal (TD) estimando el valor de los cambios que han ocurrido entre un estado y el estado siguiente.

$$TD = [r + \gamma \cdot \max_{a'} Q(s',a')] - [Q(s,a)] \tag{3.4}$$

$$Q_t(s,a) = Q_{t-1}(s,a) + \alpha \cdot TD \tag{3.5}$$

$$Q(s,a) = Q(s,a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s',a') - Q(s,a)] \tag{3.6}$$

3.2.1. Deep Q Networks

Cuando hay un número muy alto de características, por ejemplo en una imagen, se tiene un nivel dimensional de los estados que con Q-learning no es posible manejar. Para solucionar este problema [16] introdujo Deep Q-Networks lo cual consiste en una red neuronal convulucional para aproximar la función de valor Q.

La idea es básicamente que dada la función de valor Q*, con la cual se obtiene el mejor valor posible dada cualquier política:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{3.7}$$

Q* puede aproximarse con una red neuronal profunda (Deep Network) y la función de pérdida $L(w)$ se calcula con el error cuadrático medio:

$$L(w) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q^*(s',a',w) - Q(s,a,w)\right)^2\right] \tag{3.8}$$

Siendo $L(w)$ la diferencia entre el valor objetivo y el valor estimado Q y el gradiente de

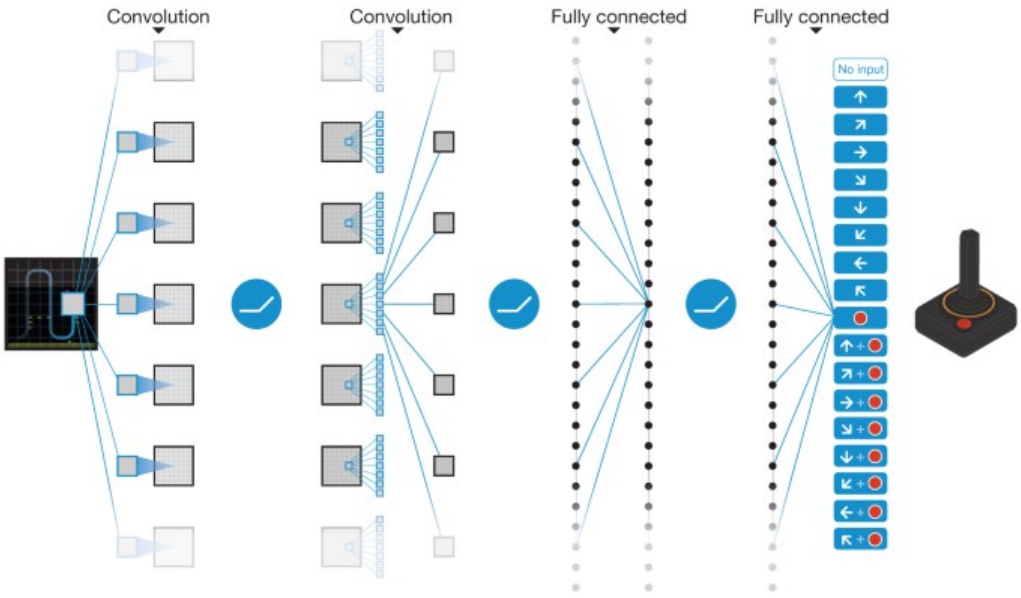


Figura 3.2: Esquema de DQN presentado en [17].

pérdida:

$$\nabla_w L(w) = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q^*(s', a', w) - Q(s, a, w)\right) \nabla_w Q(s, a, w)\right] \tag{3.9}$$

3.2.1.1. Experience Replay

Uno de los problemas del algoritmo Q-learning es que utiliza las experiencias que ha obtenido de forma aleatoria una sola vez para ajustar la red neuronal y luego se desecha, esto hace que el algoritmo no sea eficiente ya que de alguna manera está desperdiciando conocimiento, por ejemplo experiencias que son raras o que puede implicar una recompensa negativa son difíciles de obtener nuevamente.

Esta técnica consiste en que el agente usa estados pasados y repetidamente le presenta estos estados al algoritmo de aprendizaje, de esta manera se acelera el proceso de aprendizaje y el agente puede reforzar lo que ya ha aprendido.

La implementación que proponen en [17] es tomar tuplas comprendidas por observacíon, acción, recompensa, nueva observación y almacenarlas en un memoria, luego de ellos se toma una muestra aleatoria, está muestra pasa por la red neuronal, y dado que estas tuplas no tienen correlación, se genera una verdadera función genérica.

El tamaño de estas muestras (batch size) es importante dado que si es muy grande puede generar problemas con los recursos (memoria) y si es demasiado pequeño realmente no se estaría usado los beneficios de esta técnica.

Otro aspecto a tener en cuenta es que es conveniente usar únicamente experiencias recientes

ya que las acciones de un pasado lejano normalmente normalmente no fueron tan buenas.

3.2.1.2. Fijar red objetivo

Al incrementar el valor en Q(s,a), Q(s',a) se incrementa para todas las acciones y la actualización de Q y los objetivos están correlacionados. Para evitar esta correlación los pesos de la red no se actualizan de forma inmediata. Se utilizan parámetros antiguos, en términos de implementación se utiliza una red que es copia de la original la cual se actualiza cada n iteraciones.

3.2.2. Doble DQN

En un trabajo mas reciente [29] mostraron que el algoritmo DQN tiende a sobreestimar los valores q, debido a que siempre está calculando el valor máximo. La solución que proponen consiste en no actualizar los pesos de la red inmediatamente, sino usar una red para calcular el valor q y otra red para seleccionar la mejor acción. Los pesos de la segunda red son actualizados cada cierto número de iteraciones.

$$Y_t^{DoubleDQN} \equiv r + \gamma Q(s, \underset{a}{\operatorname{argmax}}\, Q(s', a'; \theta_t), \theta_t^-) \tag{3.10}$$

red valor total — Q — red selecciona la mejor acción — $Q(s', a; \theta_t)$

$$Y_t^{DoubleDQN} \equiv r + \gamma \; Q \; (s', \underset{a'}{\operatorname{argmax}}\; Q(s', a; \theta_t), \theta_t^-) \tag{3.11}$$

$$Y_t^{DoubleDQN} \equiv r + \gamma Q_{\text{next}}(s, \underset{a}{\operatorname{argmax}}\, Q_{\text{eval}}(s', a; \theta_t), \theta_t^-) \tag{3.12}$$

3.2.3. Dueling DQN

En Dueling DQN cambia la estructura del modelo, el cual se crea con la estructura:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|A|} \sum_{a=1}^{|A|} A(s,a) \tag{3.13}$$

$V(s)$ representa el valor del estado s y A es la ventaja de realizar esa acción en el estado s. El valor del estado no depende de la acción.

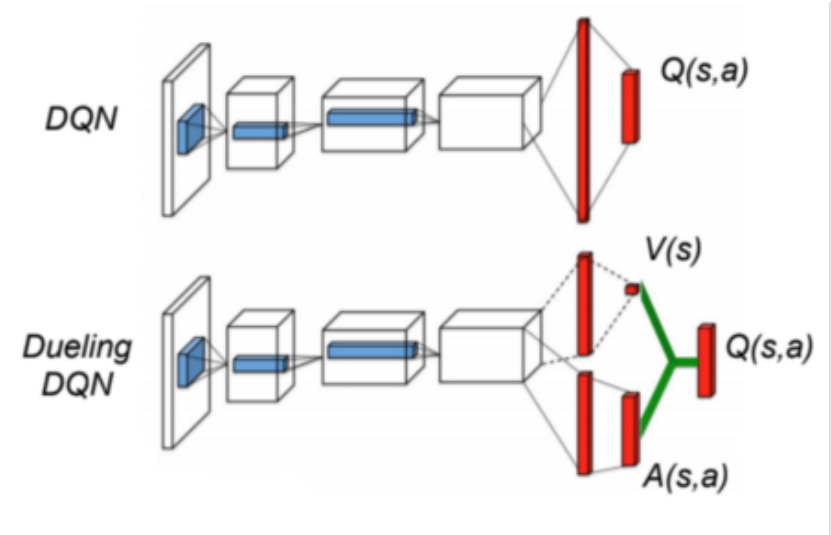


Figura 3.3: Esquema de Dueling DQN presentado en [30].

Capítulo 4

Juegos de NES

4.1. NES

Nintendo Entertainment System (NES) fue una consola de videojuegos producida por Nintendo en 1983. La consola vendió 61.91 milliones de unidades alrededor del mundo. Dentro de su catálogo se encuentran 715 juegos que fueron lanzados entre 1983 y 1992, el más conocido y vendido fue Super Mario Bros. Las especificaciones técnicas de la consola comparadas con la consola Atari 2600 se puede ver en la tabla 4.1

Especificaciones	Atari	NES
Catalogo de juegos	565	715
Velocidad del CPU	1.19 MHz	1.79 MHz
Tamaño memoria RAM	128 bytes	2 kB
Tamaño ROM (capacidad cartucho)	2-4 kB	8 kB a 1 MB
Profundidad de color	8 bit	8 bit
Resolución de pantalla	160×210	256×240
Botones en el control	5	8
Combinaciones de botones posibles	18	35

Tabla 4.1: Especificaciones técnicas de las consolas Atari 2600 y NES

Los botones de la consola NES son: [▲], [◄], [►] , [▼], [SELECT],[START], [B] y [A]. El botón [START] se usa para pausar el juego, [SELECT] normalmente no se usa durante el transcurso de una partida. Dependiendo del juego las acciones ejecutadas pueden variar al presionar uno o varios botones. Existen combinaciones de botones imposibles como [◄ + ►] mientras que puede encontrarse combinaciones de botones equivalentes, es decir, que ejecutan la misma acción. Por este motivo el número de acciones puede reducirse al punto que solo con este conjunto mínimo de acciones es posible completar el primer nivel de cada uno de los juegos

enunciados a continuación.

4.1.1. Donkey Kong

Es un juego de plataforma 2D en el cual el personaje principal "Jumpman" (después conocido como Mario) debe llegar a lo más alto de una construcción evitando obstáculos como bolas de fuego y barriles para rescatar a Pauline quien ha sido secuestrada por un gorila llamado Donkey Kong. El personaje además de caminar hacia la derecha o izquierda puede subir o bajar escaleras y está en la capacidad de saltar. Si un barril o una bola de fuego lo toca, o si cae de una gran altura el personaje pierde una vida.

(a) (b)

Figura 4.1: Donkey Kong

En este juego el botón [B] no tiene uso. El botón [▼] podría no usarse, si bien llevaría al jugador a no poder revertir equivocaciones con las escaleras, es posible jugar y terminar el nivel. Las acciones mínimas con las que el personaje puede completar el primer nivel están descritas en la tabla 4.2.

Acción	Descripción
▲	Subir escalera
◄	Moverse a la izquierda
►	Moverse a la derecha
◄ + A	Saltar a la izquierda
► + A	Saltar a la derecha

Tabla 4.2: Lista mínima de acciones posibles en Donkey Kong.

4.1.2. Ice Climber

Es un juego de plataforma 2D publicado en 1984 en el cual el personaje que solo está equipado con un martillo debe abrirse paso para llegar a la cima de la montaña. El personaje puede moverse hacia la derecha e izquierda, así como saltar para llegar a un nivel superior. El personaje también puede usar su martillo para evitar ser atacado por los "Topi" [4].

(a) (b)

Figura 4.2: Ice Climber

En Ice Climber los botones [▲] y [▼] no tienen uso. Mientras que el personaje usa el martillo no puede ejecutar ninguna otra acción. El conjunto mínimo de acciones que puede ejecutarse está descrito en la tabla 4.3.

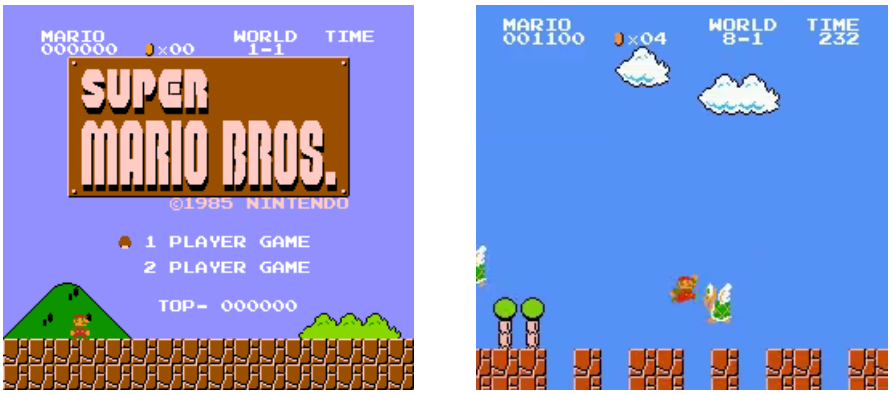
Acción	Descripción
◄	Moverse hacia la izquierda
►	Moverse hacia la derecha
A	Saltar
◄ + A	Saltar a la izquierda
► + A	Saltar a la derecha
B	Dar un martillazo

Tabla 4.3: Lista de acciones en Ice Climber

4.1.3. Super Mario Bros

En este juego la pantalla avanza gradualmente hacia la derecha en la cual el personaje principal, Mario debe avanzar a medida que sortea obstaculos como tubos, tortugas y abismos. Al final de cada área, se encuentra una gran escalera y una bandera. El puntaje se contabiliza

por la cantidad de monedas obtenidas, enemigos derrotados y altura alcanzada en la bandera. También se contabilizan puntos extras por el tiempo faltante en el reloj.



(a) (b)

Figura 4.3: Super Mario Bros

El botón [▲] no se utiliza durante el juego. El uso del botón [▼] es esporádico. Dado que la pantalla siempre está avanzando hacia la derecha, el uso del botón [◀] para moverse hacia la izquierda no es indispensable. De esta manera, puede jugarse y terminarse el primer nivel del juego únicamente moviéndose hacia la derecha y saltando. El botón [B] se usa normalmente para acelerar la velocidad de movimiento de Mario, puede asumirse que si este botón siempre está oprimido es equivalente a un juego a mayor velocidad pero con una acción menos que ejecutar. Así, únicamente son necesarias dos acciones para completar el primer nivel de Super Mario Bros. Tabla 4.4.

Acción	Descripción
▶ + B	Correr hacia adelante
▶ + B + A	Saltar hacia adelante

Tabla 4.4: Lista de acciones en Super Mario Bros

4.1.4. Kung Fu

Kung Fu Master es un juego del género beat 'em up 2D de Arcade. El juego fue adaptado a varias consolas, entre ellas la Atari 2600 y NES. La versión de NES en Japón es conocida como Spartan X mientras que en América y Europa fue nombrada Kung Fu. El personaje (Thomas) a medida que avanza debe derrotar con habilidades de artes marciales a sus oponentes, cada vez que Thomas es atacado su energía disminuye.

(a)

(b)

Figura 4.4: Kung Fu

En el primer nivel el personaje avanza cuando se mueve hacia la izquierda, los oponentes se acercan desde ambos lados de la pantalla y para atacarlos con los botones [B] o [A] es necesario que el personaje se encuentre de frente a ellos. Los seis botones de acción [▲], [◀], [▶] , [▼], [B] y [A] son necesarios. Dado que el personaje deja de avanzar cuando lanza un golpe las acciones [◀ + A] y [▶ + A] son equivalentes a [A]. De igual forma [◀ + B] y [▶ + B] tienen el mismo efecto que [B]. Las combinaciones [▼ + B] , [▼ + A] ,[▲ + B] y [▲ + A] no se consideran indispensables para completar el primer nivel del juego. Se listan las acciones mínimas para completar el primer nivel en la Tabla 4.5.

Acción	Descripción
◀	Moverse hacia la izquierda
▶	Moverse hacia la derecha
▲	Saltar
▼	Agacharse
A	Dar un puñetazo
B	Dar una patada

Tabla 4.5: Lista de acciones en Kung Fu

4.1.5. Metroid

El personaje principal (Samus Aran) llega al planeta Zebes con la función de encontrar y destruir a Mother Brain, en este juego 2D es indispensable adquirir ciertos items para poder avanzar hacia determinadas áreas. En varias ocasiones es necesario retroceder y explorar hasta que encuentra el elemento que le permite avanzar. Además en el entorno hay varios enemigos que

pueden hacerle daño con facilidad, algunos de ellos pueden ser eliminados mediante disparos. Es necesaria la precisión tanto para saltar hacia lugares mas altos como para eliminar los enemigos.

(a) (b)

Figura 4.5: Metroid

Normalmente cualquier jugador humano al iniciar el juego, si tiene alguna experiencia con juegos de plataformas, se dirigiría hacia la derecha, atravesaría la puerta llegando a la habitación contigua observando que hacia abajo hay enemigos pero no es posible acceder. Si el jugador continúa hacia la derecha después de unas pantallas se daría cuenta que no puede avanzar mas debido a que hay un tunel en el cual Samus no puede entrar. Este tunel solo puede atravesarse una vez que Samus ha adquirdo el power-up denominado como Maru-Mari (Morphing Ball), el cual solamente puede encontrar a la izquierda de la posición inicial. El juego tiene un alto componente de exploración.

En Metroid se utilizan durante una partida casi todos los botones del control. Incluso en determinado punto del juego, cuando se adquieren los misiles, el botón [SELECT] cumple la función de activar o desactivar su uso. Al menos hasta este punto, el uso del botón [SELECT] no es necesario.

Por limitación propia del juego, Samus no puede apuntar ni disparar en diagonal, por lo tanto acciones como [▲ + ▶], [▲ + ◀], [▲ + ▶ + B] y [▲ + ◀ + B] no se usan, así como la combinación de [▼] con cualquier otro botón. Dado que el personaje puede avanzar mientras dispara se prefiere la acción [▶ + B] sobre [▶] y de manera análoga [◀ + B] sobre [◀]. Se describe el mínimo conjunto de acciones para completar la primera meta en la Tabla 4.6.

Acción	Descripción
▼	Agacharse (Transformarse en esfera)
◄ + A	Saltar hacia la izquierda
► + A	Saltar hacia la derecha
◄ + B	Moverse y disparar hacia la izquierda
► + B	Moverse y disparar hacia la derecha
▲ + B	Disparar hacia arriba

Tabla 4.6: Lista de acciones en Metroid

Capítulo 5

Implementación

En esta sección se describen los requerimientos del proyecto, el modelo utilizado y el diseño de las funciones de recompensa.

5.1. Herramientas

La implementación se llevó a cabo en un computador de escritorio con sistema operativo Windows 10 Home 64-bit, CPU Ryzen™ 5 1500X, 16 GB de RAM y una GPU Nvidia GTX 1050 Ti.

El software se desarrolló usando la versión 4.8.4 de Anaconda en el lenguaje de programación Python. Además se usó la API de programación de gráficos de uso general NVIDIA CUDA 10.2. La versión de Python utilizada es la 3.7.4 sobre el entorno de desarrollo Spyder. El código Python importa varios paquetes y módulos:

- *Gym Retro*: paquete diseñado para la investigación en aprendizaje por refuerzo en videojuegos. Provee integración con emuladores de Atari y NES entre otros.
- *PyTorch*: paquete diseñado para realizar cálculos numéricos haciendo uso de la programación de tensores. Permite su ejecución sobre la GPU para acelerar los cálculos.
- *Numpy*: paquete que provee a Python con arreglos multidimensionales y operaciones de álgebra lineal para aplicaciones en ciencia de datos.
- *Matplotlib*: paquete para realizar gráficos en 2D y 3D.

5.2. Modelo utilizado

En este proyecto se ha adaptado el algoritmo de Deep Q-learning del experimento realizado por Deepmind [16] con las mejoras propuestas en [17] a cinco juegos de la consola NES. Se espera

que los metodos basados en valores como DQN sean suficientes para completar por lo menos el primer nivel de cada juego. Por otro lado, dado que Gym Retro ya cuenta con algoritmos predefinidos e implementados como A2C o PPO los cuales están optimizados, se descartó crear un agente desde cero implementado estos algoritmos.

A partir de los experimentos realizados por DeepMind se ha establecido prácticamente como regla la lectura de los estados a partir de las imágenes generadas en pantalla. En este proyecto se realiza la lectura desde la memoria RAM de cada juego. Se realizó un experimento similar con juegos de Atari 2600 [22] únicamente con el algoritmo DQN. Se investiga la efectividad del aprendizaje de los algoritmos DQN, Doble DQN y Dueling DQN usando unicamente la memoria RAM de la NES, la cual es ocho veces mas grande que la de Atari.

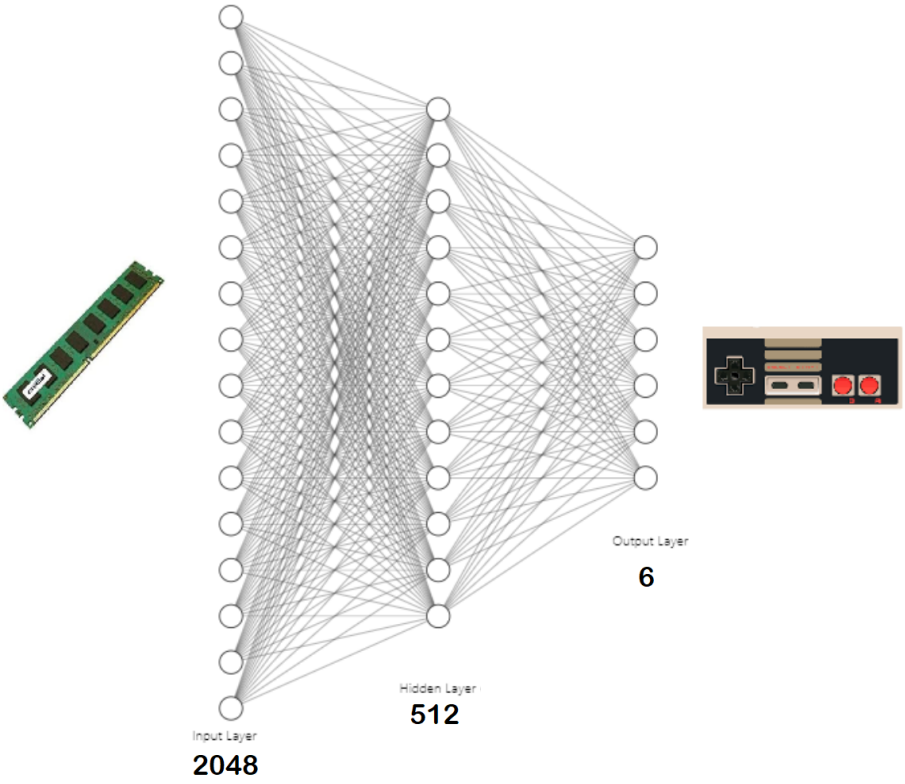


Figura 5.1: Red Neuronal utilizada en Algoritmo DQN.

5.3. Pre-proceso de datos

Cuando leen los estados directamente de la RAM del emulador no es necesario pre-procesar los datos. La NES cuenta con una RAM de 2 KB más 8 KB de RAM que almacena en el cartucho. En cada byte se puede almacenar un valor entero entre 0 y 255. Un estado en un determinado cuadro está compuesto por un vector de 10240 valores, es decir, se está explorando

en un espacio de 255^{10240} estados posibles. Debido a la limitación de tiempo y recursos, se hace lo posible para reducir el tamaño de este espacio así el numero de salidas (acciones) posibles. Una primera medida consiste en verificar si el emulador en efecto está usando la memoria RAM del cartucho (memoria wram).

Se experimenta con un agente que ejecuta acciones aleatorias para verificar la cantidad de bytes que cambian durante la ejecución de un episodio. Se obtiene una lista de los bytes en memoria que cambiaron en algún momento. Con esta información y con el mapa de la RAM se realiza un esquema de la memoria que es utilizada mientras se juega.

En todos los juegos, excepto Metroid, se activan bytes que se encuentran en los primeros 2 KB de RAM. Basándose en esto cada estado, el cual puede interpretarse como un vector de características de 10240 dimensiones, puede reducirse a 2048. Podría reducirse aún más basándose solamente en aquellos bytes utilizados por el agente aleatorio, pero no hay certeza que una vez se vaya avanzando en el juego se activen mas bytes. Los valores de las observaciones son normalizados entre 0 y 1.

El número de acciones posibles que podrían ejecutarse con un control de 8 botones es de 2^8, de estas acciones se consideran posibles 36, y dentro de estas no todas son efectivas dependiendo del juego (por ejemplo, presionar arriba en Super Mario Bros). Se propone reducir al mínimo el número de acciones efectivas y suficientes para que el agente sea capaz de terminar el primer nivel del juego.

5.4. Funciones de recompensa

Gym Retro por defecto tiene predefinidas unas funciones de recompensa básicas y una condición de terminación de los episodio para los juegos que están integrados. La mayoría de estas funciones se basan en la obtención de puntos durante el juego. Esto no siempre es lo ideal ya que un alto puntaje no implica que se culmine el juego o que sea completada un nivel y en cambio puede llevar a que el agente ejecute acciones repetitivas con el único fin de acumular puntos, este fenómeno se conoce como (*Farming*). Durante el transcurso del proyecto se proponen funciones de recompensa alternativas para los juegos Donkey Kong, Ice Climber y Kung Fu. Para el juego Metroid, puesto que no está integrado a Gym Retro, ha sido necesario proponer una función de recompensa y condición de episodio finalizado.

5.4.1. Donkey Kong

En Donkey Kong, por defecto está definida la recompensa como el puntaje obtenido (*score*). Durante el juego se contabilizan puntos que se obtienen cuando el personaje salta sobre un barril (100) o cuando el personaje llega a la plataforma de Pauline (el tiempo restante). Para

un jugador humano es evidente que se encuentra en la plataforma mas alta porque la ve a simple vista.

Un agente no cuenta con un incentivo directo para que el personaje se acerque a la plataforma de Pauline. Como el objetivo es obtener puntos podría fácilmente quedarse saltando barriles hasta que el tiempo se termine. Se investigó en el mapa de la memoria RAM del juego [1] que hay una variable llamada *platform* la cual indica la plataforma en la que se encuentra el personaje.

Algoritmo 1 Recompensa en Donkey Kong

```
if score or platform has changed then
  return Δscore + 1000 × Δplatform
else
  return 0
end if
```

La recompensa descrita en el Algoritmo 1 ahora incentiva cuando el agente logre ascender una plataforma. Como abreviatura se utiliza el símbolo Δ al lado de una variable para denotar el cambio en dicha variable del estado $n-1$ a el estado n.

$$\Delta score = score_n - score_{n-1} \tag{5.1}$$

5.4.2. Ice Climber

En este juego el puntaje solo es calculado al finalizar el nivel, así que la recompensa es contabilizada durante el juego para ser luego descontada de tal forma que iguale al puntaje. La recompensa usa las variables:

- *bricks_hit*: el número de bloques que el personaje rompe.
- *ice_hit*: el número de bloques de hielo que el personaje rompe.
- *bird_hit*: el número de pajaros que el personaje golpea.
- *score*: se calcula al finalizar el nivel, lo componen las variables anteriores multiplicadas por constantes, el numero de berenjenas recogidas en la fase de bonus y puntos extra si el personaje alcanza a el condor que vuela sobre la montaña.

En Ice Climber al igual que Donkey Kong no hay un incentivo directo para avanzar hacia arriba. En la el mapa de la RAM del juego [**?**] se encuentran dos variables que podrían usarse

para este objetivo: *level* que indica el piso o nivel de la montaña en la que está el personaje, y *y_axis* que corresponde a la posición vertical del personaje en pantalla.

La recompensa que primero se intenta implementar consiste en premiar al agente en la medida que *level* sea incrementada y/o que *y_axis* indique que el personaje está subiendo. Los problemas con estas dos variables surgen en el momento que el personaje pierde una vida, puesto que en ese momento el personaje cae haciendo que estas variables incluso tengan valores negativos. Durante los cuadros de animación en que el personaje está perdiendo la vida, el agente estaría leyendo los estados generados mientras va recibiendo una recompensa negativa independientemente de la acción que esté tratando de ejecutar posiblemente generando ruido en los datos. Un segundo problema ocurre en el momento de monitorear los resultados de un episodio, dado que no importa lo que haga el agente, si pierde vidas, la recompensa acumulada se perdería.

La solución que se propone consiste en mantener una función de recompensa similar a la planteada por defecto con una modificación, la variable *level* ahora sería un multiplicador. Por ejemplo, romper un bloque en el primer piso tiene una recompensa inferior a hacerlo en el tercer piso. De esta forma la recompensa de un episodio se acumula sin perderse y el agente debería desarrollar preferencia para ejecutar acciones que producen recompensa en pisos superiores. Algoritmo 2.

Algoritmo 2 Recompensa en Ice Climber

```
if score or level or bird_hit or ice_hit or bricks_hit has changed then
  return Δscore + (Δbird_hit + Δice_hit + Δbricks_hit)×0,1× level
else
  return 0
end if
```

5.4.3. Super Mario Bros

En Super Mario Bros la recompensa no está dada por el puntaje obtenido, la variable *xscroll* indica el avance hacia la derecha del personaje. Como en este juego el personaje siempre debe moverse hacia la derecha, la recompensa acumulada no es mas que simplemente la distancia recorrida desde el origen.

Algoritmo 3 Recompensa en Super Mario Bros

```
return Δxscroll
```

5.4.4. Kung Fu

La recompensa en Kung Fu está calculada directamente con el valor del puntaje. Se obtienen puntos a medida que se derrotan oponentes. Se premia con puntos extra al alcanzar el final de un nivel a partir de la energía y tiempo restante. Como antes se ha mencionado, el agente podría quedare en un solo lugar haciendo *farming*.

La función de recompensa propuesta usa la variable *xpos* encontrada en la RAM [2], la cual indica la posición del personaje en el eje x. Al igual que en Super Mario Bros es la recompensa estaría dada por la variación de *xpos*, aunque a diferencia de Mario, no se mueve a la derecha siempre. En los niveles impares (1,3 y 5) el personaje debe ir de derecha a izquierda iniciando en *xpos*=68 y dirigiendose hacia *xpos*=2. En el segundo y cuarto nivel el personaje va desde *xpos*=2 hasta *xpos*=68. Este problema es fácil de corregir si se calcula una nueva variable que cambie o mantenga el signo de $\Delta xpos$ dependiendo de la posición de origen x_0 y la posición de destino x_t.

$$\text{orientacion}(x_0, x_t) := \begin{cases} -1 & ;\text{si } x_0 < x_t, \\ 0 & ;\text{si } x_0 = x_t, \\ 1 & ;\text{si } x_0 > x_t. \end{cases} \tag{5.2}$$

La recompensa se calcularía con base en los valores $\Delta xpos$, $\Delta health$ para que evite recibir daño y $\Delta score$ en una proporción del 10 %.

Algoritmo 4 Recompensa en Kung Fu

```
orientation = calcular orientacion ( x_0 , x_t )
if score or health or xpos has changed then
    return Δscore ×0,1 − 5 × Δhealth + orientation ×Δxpos
else
    return 0
end if
```

5.4.5. Metroid

Como ya se ha mencionado, Metroid no está integrado a Gym Retro. Dentro de las variables de la RAM no hay definido puntaje. Tampoco está inicialmente claramente definido el objetivo ya que a diferencia de los juegos anteriores no hay niveles, hay 5 zonas que pueden visitarse de acuerdo a los power-ups adquiridos [5]. Se listan algunas variables encontradas en el mapa de la RAM del juego.

- *Health*: Indica la energía de Samus, una vez es 0 el juego termina.

- *SamusGear*: Es un valor binario que indica cuales power-ups han sido adquiridos.
- *TankCount*: Capacidad de tanques de energía.
- *MaxMissiles*: Capacidad máxima de misiles.
- *SamusAge*: Tiempo de juego
- *MapPosX*: Posición X en el mapa.
- *MapPosY*: Posición Y en el mapa.

La variable *Health* es suficiente para establecer una condición de finalizado del episodio. También en este apartado puede usarse *SamusAge* para limitar el tiempo de una sesión de juego. Si *SamusGear*, *TankCount* o *MaxMissiles* cambian durante el juego, indicaría que el jugador ha encontrado un objeto por lo cual es una buena idea de recompensa.

Dando al agente la ubicación de un power-up se estaría estableciendo un objetivo y el hecho de alcanzarlo se podría considerar como culminar un nivel. Una meta inicial puede ser por ejemplo *conseguir los misiles* (Figura 5.2). Esta información no vendría directamente del juego sino de conocimiento externo como una guía estratégica [3], que puede transmitirse al agente como una lista de ubicaciones.

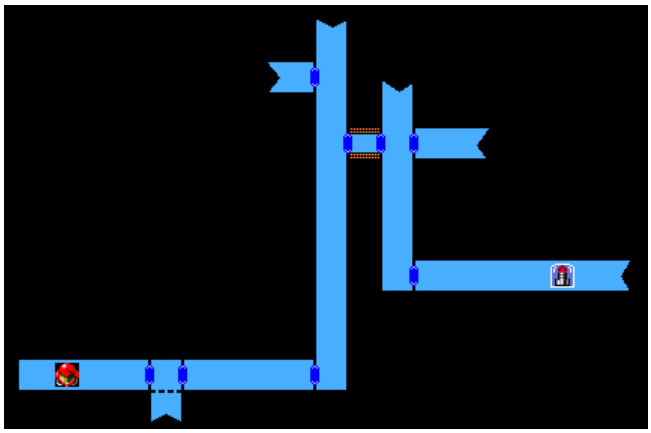


Figura 5.2: Primer objetivo, encontrar los misiles.

Si se elabora un mapa del juego basado en las locaciones posibles a las que puede acceder Samus se obtiene un laberinto delimitado por una zona de 30×30 habitaciones. Samus comienza en la posición (3,14) de la cuadrícula (marcado como 0 en la Figura 5.3), desde allí se debe dirigir a la posición (1,14) en la cual encuentra la Morphing Ball (1). Finalmente puede ir por los misiles ubicados en (18,11), marcado como 2. Después de llegar a la meta en la posición (18,11) podría dirigirse hacia (6,5) en la cual encontraría el rayo de largo alcance (3).

Puede establecerse una función de recompensa similar a la del juego Kung Fu, en este caso hay movimiento en X e Y, siendo la posición de origen (x_0, y_0) y la posición de destino (x_t, y_t).

Estos valores deben actualizarse a medida que Samus llega a un objetivo. La lista de destinos es un conjunto ordenado de puntos: $G = \{(1, 14), (18, 11), (6, 5), (25, 7), (25, 5)\}$. Para calcular la orientación su utiliza la misma función de la ecuación 5.2. Esta función tambíen sirve para indicar que Samus ha llegado al objetivo cuando retorna 0.

La variable n denota el objetivo a alcanzar, $n = 1$ al iniciar el episodio. Cuando Samus cambie de posición en el mapa recibirá una recompensa positiva o negativa dependiendo si se está acercando o alejando del objetivo. En el momento que Samus llega a la ubicación del objetivo, no necesariamente encuentra de inmediato el power-up y la recompensa que recibe es 0. En el momento que el power-up es encontrado recibe una recompensa de 100 y el objetivo es actualizado incrementando la variable n.

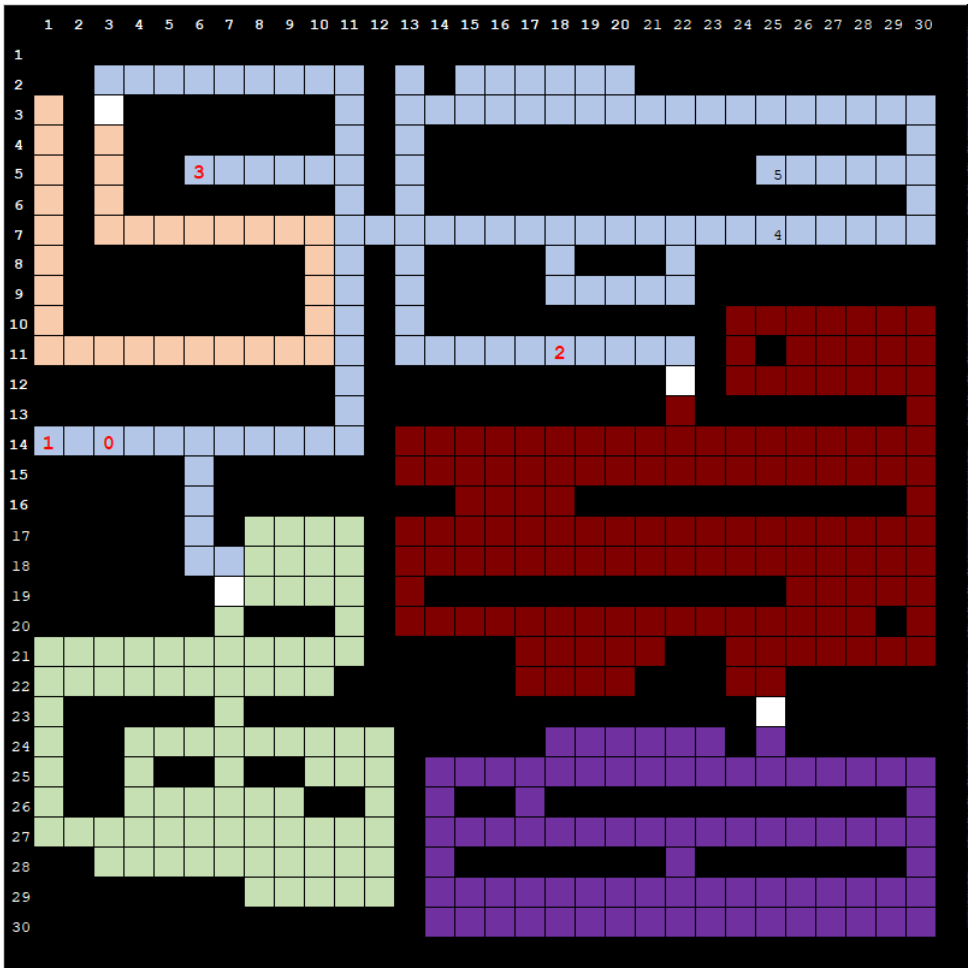


Figura 5.3: Mapa aproximado del juego Metroid.

Algoritmo 5 Recompensa en Metroid

$(x_t, y_t) = G[n]$
x_orientation = calcular orientacion ($MapPosX$, x_t)
y_orientation = calcular orientacion ($MapPosY$, y_t)
if MapPosX **or** MapPosY has changed **then**
  return x_orientation $\times\Delta$MapPosX + y_orientation $\times\Delta$MapPosX
end if
if SamusGear **or** TankCount **or** MaxMissiles has changed **then**
  $n = n + 1$
  return 100
else
  return 0
end if
if $x\_orientation = 0$ **and** $y\_orientation = 0$ **then**
  $(x_t, y_t) = G[n]$
end if

5.5. Ajustes realizados al algoritmo

5.5.1. Frame skip

La NES tiene una tasa de refresco de 60 FPS, esto significa que un agente sería capaz de ejecutar 60 acciones en 1 segundo, jugando a la velocidad normal del juego. Para un humano esta tarea es imposible y si se piensa en una persona que tiene muy poca habilidad para jugar, además de ejecutar las acciones mas básicas, puede mantener presionado un botón incluso durante medio segundo. Si este comportamiento se recrea en un agente, realizaría una lectura similar a la de una persona con menor capacidad de reacción. A diferencia del trabajo de DeepMind no se apilarían todos los frames en un paso. Se ha implementado como un parámetro el número de cuadros de animación en los que el agente repite la acción.

5.5.2. Política ϵ greedy

El valor de la probabilidad con la que se selecciona una acción aleatoria decrece a medida aumenta el número de pasos del experimento. En este proyecto se ha implementado la ecuación que asegura que para cierto número de acciones el agente explore antes de empezar a aprender y asegura una transición mas suave entre la fase de exploración y explotación. Este valor no es

calculado en cada paso sino al final de cada episodio.

$$\pi(t) = (\epsilon - \epsilon_0)\left(1 - \frac{2e^{e^{-t}}}{e^{2e^{-t}} + 1}\right) - \epsilon_0 \tag{5.3}$$

siendo t el valor del episodio ajustado por el parámetro ϵ_s.

5.6. Experimentos

5.6.1. Ajuste de parámetros

En aprendizaje por refuerzo hay tantos parámetros que es difícil encontrar una combinación optima en poco tiempo. Como punto de referencia se usaron inicialmente parámetros de los experimentos de [17] descritos en la Tabla 5.1. En este caso el agente fue entrenado 50 millones de frames en cada juego. En [22] se usaron los mismos parámetros con algunas excepciones, $lr = 0{,}0002$, $\gamma = 0{,}95$ $M = 100000$.

Parámetro		Valor	Descripción
Learning rate	lr	2.5×10^{-4}	Tasa de aprendizaje del optimizador de la red neuronal
Batch Size	bs	32	Tamaño de los lotes de datos que optimiza el modelo
Frame skip	fs	4	Número de cuadros de animación (frames) durante los cuales el agente repite la acción
Gamma	γ	0.99	Factor de descuento
ϵ Inicial	ϵ_0	1	Probabilidad de exploración al comienzo del entrenamiento
ϵ Final	ϵ	0.1	Probabilidad de exploración mínima durante el entrenamiento
Explore steps %	ϵ_s	0.02	Porcentaje de pasos en los que Epsilon decae hasta el valor mínimo
Memory size	M	1×10^6	Tamaño de la memoria usada para Experience Replay
Replay start size		50000	Frame en el cual el agente comienza a aprender

Tabla 5.1: Parametros de referencia.

Dado que los recursos y el tiempo son limitados en comparación, se realizan grupos de experimentos variando solo un par de parámetros a la vez con cada juego. Se establece $M = 50000$ debido a la limitación del hardware y que la RAM de la NES tiene un tamaño mayor a la de la Atari 2600.

5.6.2. Comparación de algoritmos

Hay mejoras propuestas al algoritmo DQN mencionadas en la sección (—). En trabajos en lo que se ha usa la memoria RAM [22] y [**?**] se ha experimentado con el algoritmo DQN y el

DDQN respectivamente. En este proyecto se compara el desempeño de los algoritmos DQN, DDQN , Dueling DQN y la combinación de ambos (Dueling DDQN) en los cinco juegos. Se espera que el algoritmo Dueling DDQN tenga un mejor desempeño.

Se aprovechará la etapa de entrenamiento para medir en los cinco juegos cuales son los bytes activados durante el proceso para disminuir aún mas el número de dimensiones de los estados del juego. Posteriormente se comparará el algoritmo Dueling DDQN entrenado con este conjunto reducido de bytes. Se espera que los resultados sean similares.

5.6.3. Evaluación

Para evaluar el desempeño del agente DeepMind usó un agente entrenado para jugar durante 30 episodios [17]. Se utilizará el mismo procedimiento comparando la recompensa obtenida por los agentes.

Capítulo 6

Resultados

6.1. Ajuste de parámetros

En la mayoría de los casos se trató de reajustar los parámetros *lr* y *fs*, ya que son los que mayor impacto tienen en la recompensa obtenida. En la medida que los experimentos fueron avanzando se usaron valores mas cercanos teniendo en cuenta resultados obtenidos con el juego anterior. La idea era buscar un conjunto de parámetros que fuese usado con todos los juegos. En todos los casos $\epsilon_0 = 1{,}0$ y $\epsilon = 0{,}1$, lo que se reajustó fue el valor de ϵ_s. usando la ecuación 5.3 en el primer grupo de experimentos. En la Tabla 6.1 se describen los parámetros evaluados por cada juego.

Juego	Parametros evaluados
Donkey Kong	fs, lr, ϵ_s
Ice Climber	fs, lr
Kung Fu	fs, lr, bs
Super Mario Bros	fs, bs, γ
Metroid	fs, lr, ram

Tabla 6.1: Parámetros evaluados por juego.

Para determinar el mejor resultado en cada caso se tuvo en cuenta la recompensa obtenida durante el entrenamiento, su promedio y el valor mas alto de recompensa alcanzado. Para medir la estabilidad se tuvo en cuenta el valor *qmax* durante el entrenamiento y los pesos en las acciones del agente. Las tablas de resultados se presentan con la siguiente información:

- *episodes*: número episodios.
- *steps*: número total de veces en que el agente ejecutó una acción

- *avg*: recompensa promedio obtenida durante toda la fase de entrenamiento.
- *hi*: el máximo valor de la recompensa obtenido durante un episodio.
- *qmax*: máximo valor q de la última capa de la red neuronal.
- *action*: acción a la cual corresponde *qmax*.
- *time*, tiempo del entrenamiento en horas.

6.1.1. Donkey Kong

Se realizaron 12 experimentos de 200000 pasos (steps). El número de cuadros en los cuales la acción se repite, *fs*, tuvo valores de 4, 8, 15 y 30 combinándose con diferentes tasas de aprendizaje, *lr*. Los valores de *lr* fueron 2×10^{-4} , 1×10^{-3} y 1×10^{-2} . El valor de *epsilon* siempre fue desde 1.0 hasta 0.1 y si bien se utilizó la misma función, el parámetro $epsilon_s$ cambió en cada caso. Figura 6.3.

Se presentan los resultados de la recompensa obtenida en los 12 experimentos en la Figura 6.1, la evolución en el valor *qmax* en la Figura 6.2, el cual gráficamente es un indicador de la estabilidad del algoritmo.

Las acciones que el agente puede ejecutar son: [▲], [◀], [▶] , [◀+A] y [▶+A]. Los pesos en la última capa de la red correspondiente a las acciones se presenta en la Tabla 6.2.

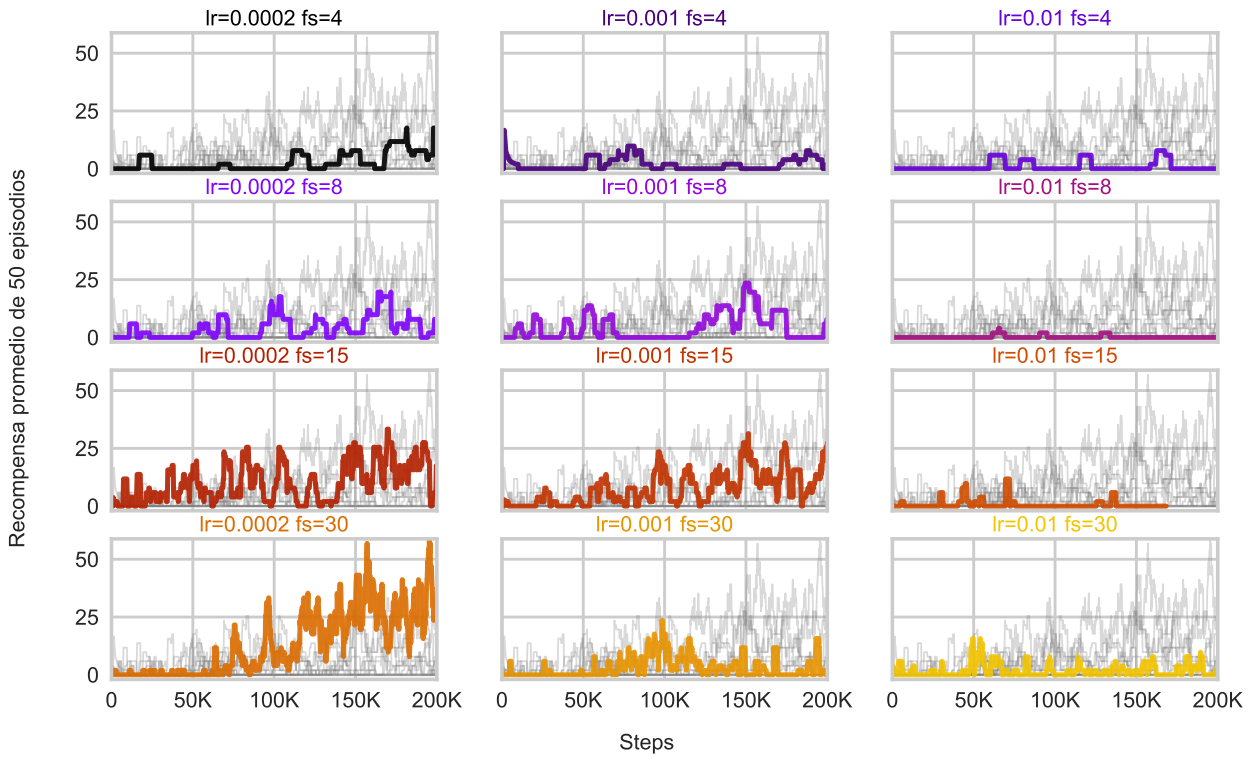


Figura 6.1: Promedio de recompensa de 50 episodios en Donkey Kong para el algoritmo DQN.

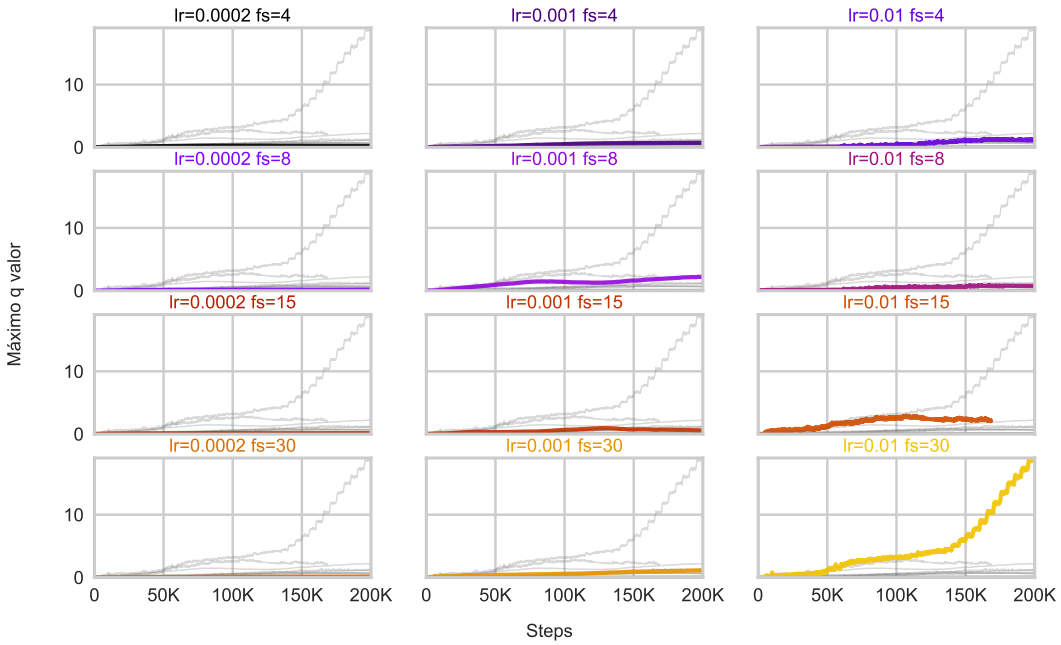


Figura 6.2: Variación del máximo valor de q en Donkey Kong para el algoritmo DQN.

fs	lr	▲	◄	►	◄+A	►+A
4	0.0002	0.145873	0.096845	0.132583	0.209052	0.088774
4	0.001	0.676311	0.705544	0.710001	0.685583	0.686774
4	0.01	1.109719	1.109785	1.106192	1.102818	1.108612
8	0.0002	0.123385	0.054084	0.179973	0.025008	0.010347
8	0.001	1.962595	1.991026	2.164751	2.020838	2.146529
8	0.01	0.710521	0.696482	0.698298	0.721729	0.706617
15	0.0002	0.015055	-0.008474	0.074166	0.127351	0.078465
15	0.001	0.500015	0.568263	0.581740	0.561885	0.263153
15	0.01	2.072543	2.011470	2.064319	2.049829	2.062313
30	0.0002	0.093899	0.061301	0.135100	0.045722	0.074014
30	0.001	1.140577	0.579508	0.463973	0.320182	0.656318
30	0.01	17.722286	18.194397	17.783361	17.967335	18.720324

Tabla 6.2: Pesos de la última capa de la red por acción para el juego Donkey Kong.

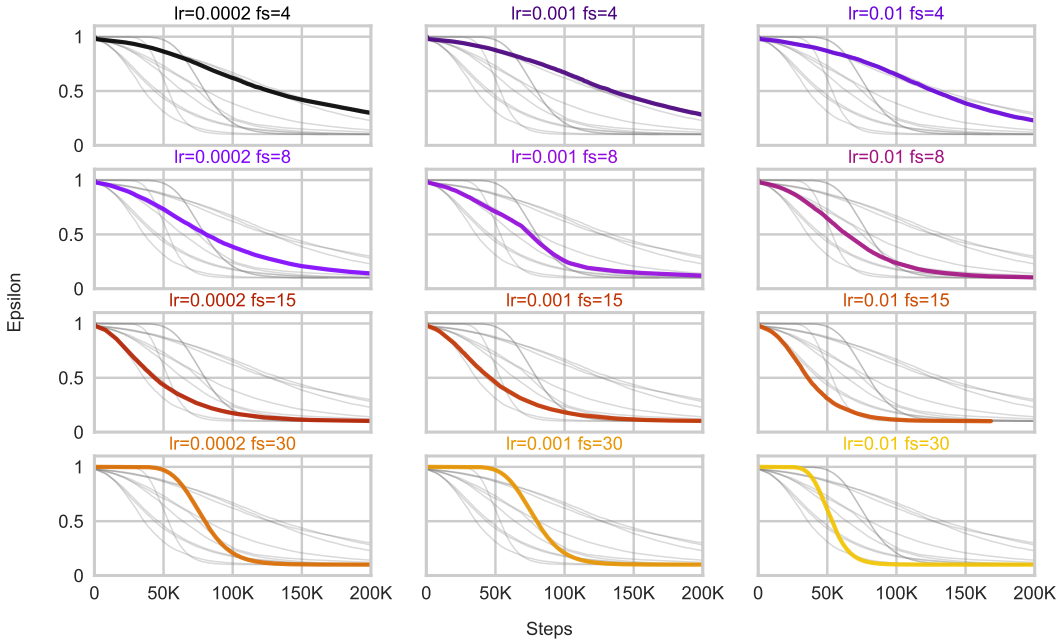


Figura 6.3: Evolución del valor del parámetro *epsilon* en Donkey Kong para el algoritmo DQN.

fs	lr	episodes	steps	avg	hi	qmax	action
30	0.0002	5535	200056	10.205925	800.0	0.135100	▶
15	0.0002	2452	200021	9.747145	700.0	0.127351	◀ + A
15	0.0010	2423	200039	6.974825	600.0	0.581740	▶
8	0.0002	1613	200093	3.719777	800.0	0.179973	▶
8	0.0010	1828	200128	3.391685	500.0	2.164751	▶
4	0.0002	1085	200065	2.857143	600.0	0.209052	◀ + A
30	0.0010	5489	200013	2.641166	600.0	1.140577	▲
4	0.0010	1122	200069	1.871658	300.0	0.710001	▶
30	0.0100	5787	200045	1.727713	600.0	18.720324	▶ + A
4	0.0100	1240	200077	0.967742	300.0	1.109785	◀
15	0.0100	3000	168322	0.800000	600.0	2.072543	▲
8	0.0100	2315	200101	0.172786	100.0	0.721729	◀ + A

Tabla 6.3: Comparación de parámetros para Donkey Kong

La mejor configuración es lr=0.0002 y fs=30. Al observar al agente jugando se descubre que al parecer encuentra un bug del juego ya que consigue en un solo salto mas de 100 puntos.

6.1.2. Ice Climber

Se realizaron 12 experimentos de 100000 pasos cada uno. El valor *fs* se contrastó con diferentes tasas de aprendizaje *lr*. Los valores de *fs* fueron de 4, 8 y 15 mientras los valores para *lr* fueron de 2×10^{-4}, 2.5×10^{-4}, 5×10^{-4} y 1×10^{-3}. Los resultados de la recompensa obtenida se presentan en la Figura 6.4, la evolución en el valor *qmax* en la Figura 6.6 y El valor de los pesos correspondientes a las acciones pueden verse en la Figura 6.7. Las acciones que el agente puede ejecutar son: [▲], [◄], [►] , [◄+A] y [►+A].

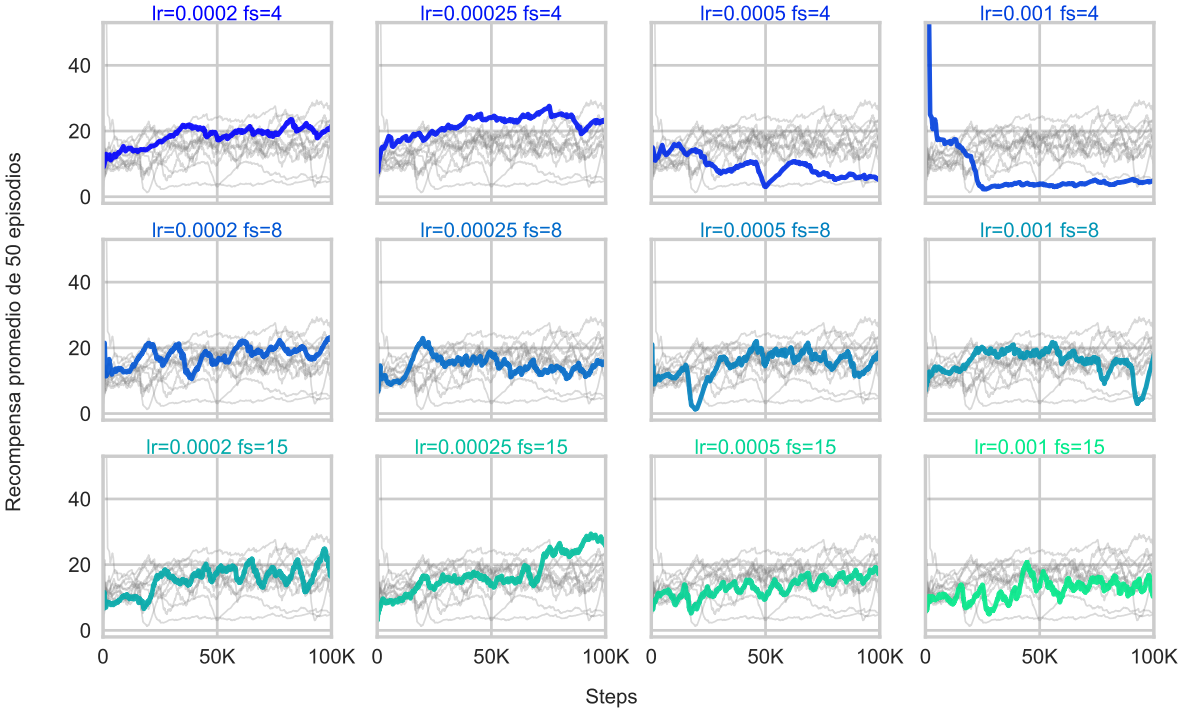


Figura 6.4: Promedio de recompensa de 50 episodios en Ice Climber para el algoritmo DQN.

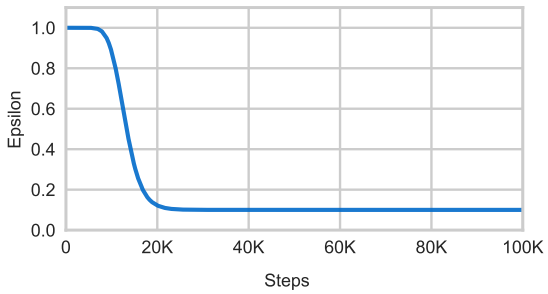


Figura 6.5: Evolución del valor del parámetro *epsilon* en Ice Climber para el algoritmo DQN.

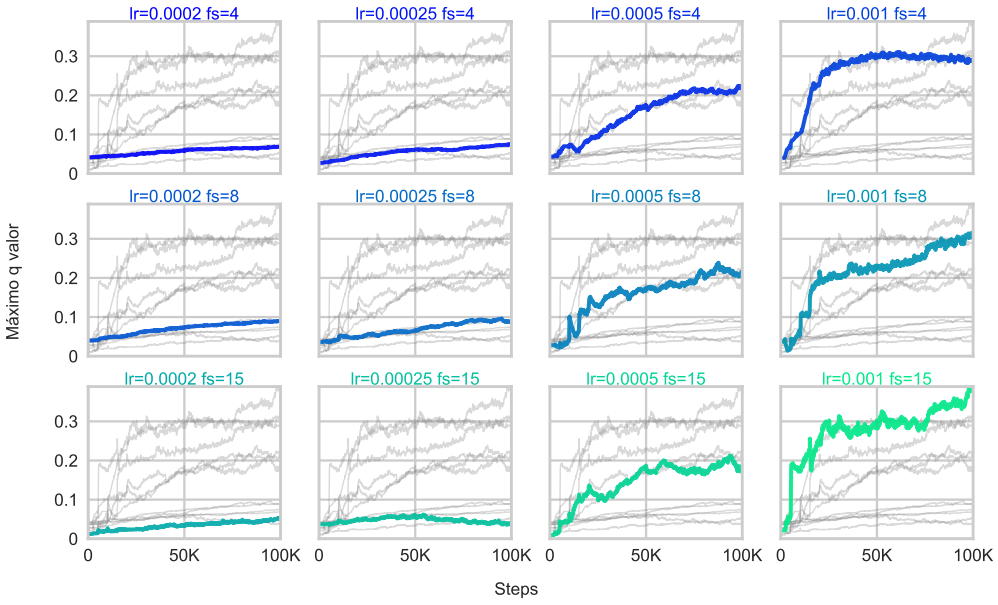


Figura 6.6: Variación del máximo valor de q en Ice Climber para el algoritmo DQN.

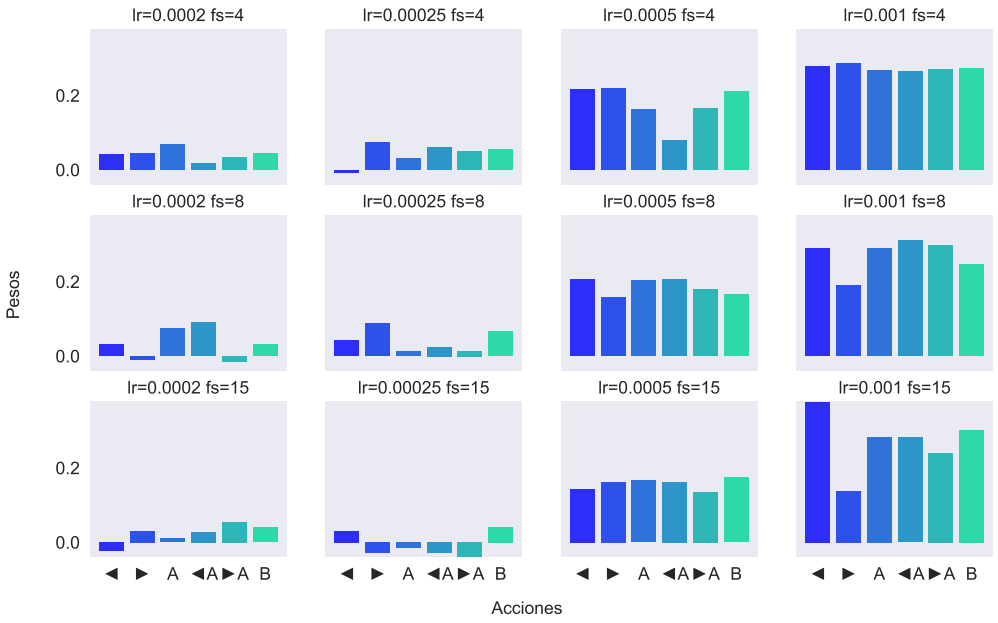


Figura 6.7: Pesos de la última capa de la red por acción para el juego Ice Climber.

.

Los mejores resultados se observan con lr=0.00025 y fs=4. La gráfica de fs=15 y lr=0.00025 también muestra una ligeramente mejor evolución que fs=4, sin embargo, se descarta esta posibilidad dado que su media y valor mas alto son comparativamente menores.

fs	lr	episodes	steps	avg	hi	qmax	action
4	0.00025	268	100031	23.137546	68.0	0.074540	▶
4	0.00020	346	100424	19.570605	68.0	0.068166	A
8	0.00020	797	100058	17.973684	65.0	0.089812	◀ + A
15	0.00025	1412	100069	16.921444	58.0	0.040372	B
8	0.00100	975	100170	15.428279	62.0	0.309644	◀ + A
15	0.00020	1487	100028	15.188172	62.0	0.054438	▶ + A
8	0.00025	1054	100023	14.678673	73.0	0.088089	▶
8	0.00050	1059	100001	14.450000	77.0	0.205546	◀
15	0.00050	1567	100003	12.987245	65.0	0.174411	B
15	0.00100	2064	100022	11.811138	60.0	0.377830	◀
4	0.00050	560	100172	7.786096	46.0	0.219596	▶
4	0.00100	629	100189	4.585714	53.0	0.287579	▶

Tabla 6.4: Comparación de parámetros para Ice Climber

6.1.3. Super Mario Bros

6.1.3.1. Frame skip vs batch size

Inicialmente se realizaron 9 experimentos de 200000 pasos. La tasa de aprendizaje *lr* se mantuvo fija en 0.00025. Los valores de *fs* fueron 4, 6 y 12 mientras *bs* fue 32, 64 y 128.

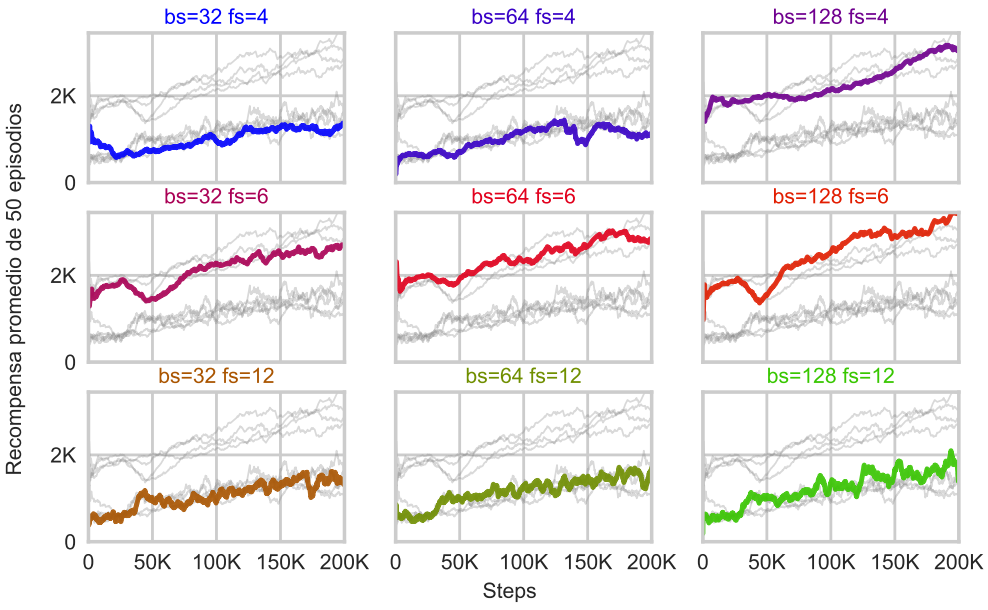


Figura 6.8: Promedio de recompensa de 50 episodios en Super Mario Bros para el algoritmo DQN.

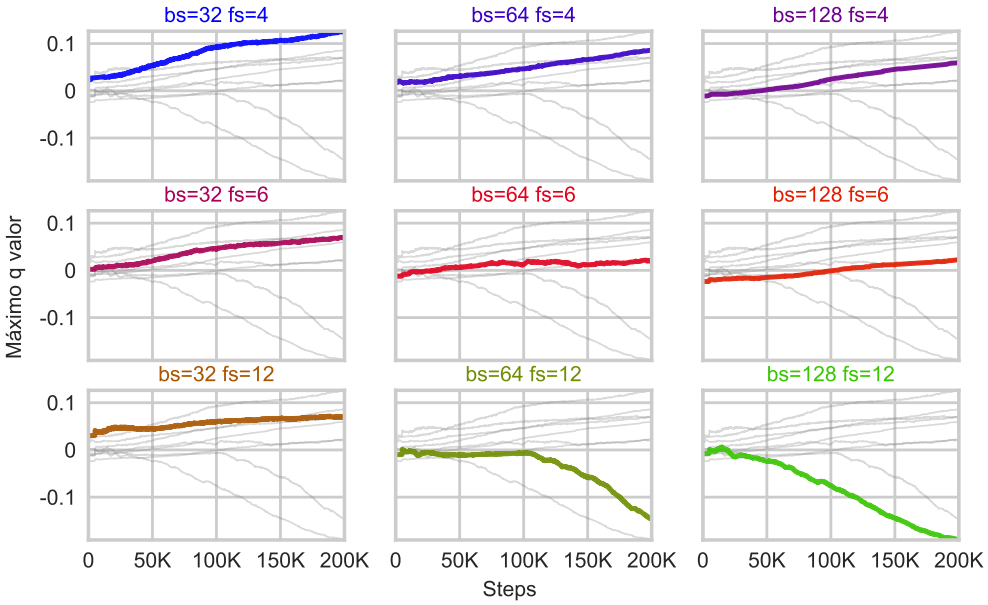


Figura 6.9: Variación del máximo valor de q en Super Mario Bros para el algoritmo DQN.

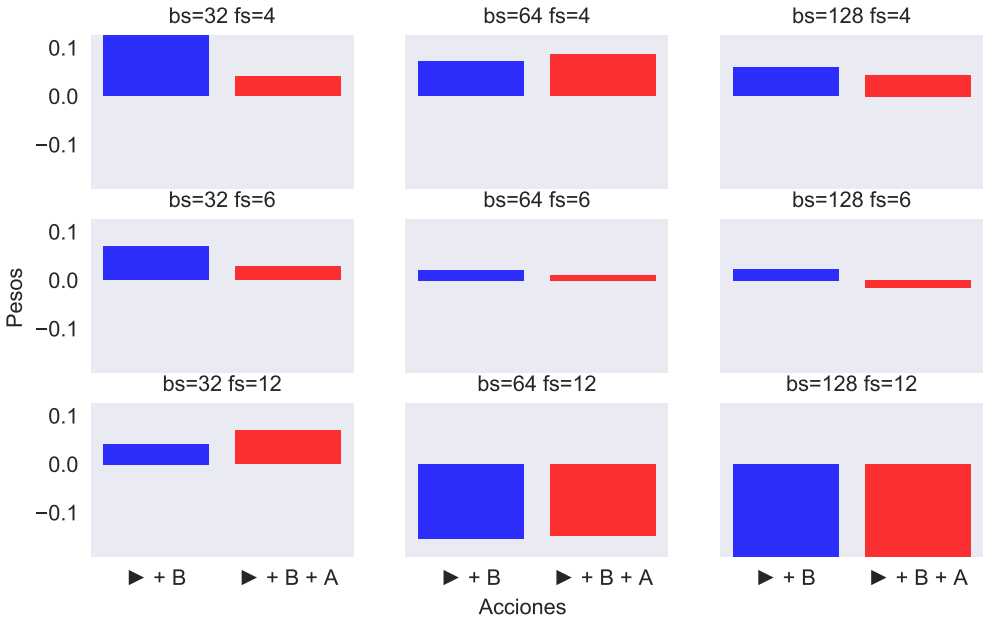


Figura 6.10: Pesos de la última capa de la red por acción para el juego Super Mario Bros.

Se presentan los resultados de la recompensa obtenida en los 9 experimentos en la Figura 6.8, la evolución en el valor *qmax* en la Figura 6.9. Las acciones que el agente puede ejecutar son [▶+B] y [▶+B+A]. El valor de los pesos en la última capa de la red neuronal que corresponde a estas acciones pueden verse en la Figura 6.10.

fs	bs	episodes	steps	avg	hi	qmax	action
6	128	397	202356	2460.977387	6547.0	0.023054	▶ + B
6	64	423	200056	2413.132075	5915.0	0.020693	▶ + B
4	128	298	200747	2359.558528	5725.0	0.059601	▶ + B
6	32	478	200121	2149.672234	5839.0	0.069152	▶ + B
12	128	2590	200086	1144.617522	4137.0	-0.190486	▶ + B
12	64	2612	200118	1106.516265	3946.0	-0.146664	▶ + B + A
12	32	2714	200232	1010.284715	4282.0	0.069667	▶ + B + A
4	64	874	200128	998.460571	3930.0	0.086290	▶ + B + A
4	32	907	200144	988.992291	3837.0	0.125498	▶ + B

Tabla 6.5: Comparación de parámetros para Super Mario Bros.

El promedio, el máximo valor alcanzado y la estabilidad del valor q permiten destacar los resultados con fs=6 y bs=128. En este caso cuando el agente juega se observa que tiene éxito completando el nivel 1-1 algunas veces.

6.1.3.2. Factor de descuento γ

Se realizó un segundo grupo de experimentos, 4 , con el fin de comparar el desempeño del algoritmo variando el parámetro γ. La tasa de aprendizaje lr se mantuvo fija en 0.00025, fs se fijó en 6 y bs en 128.

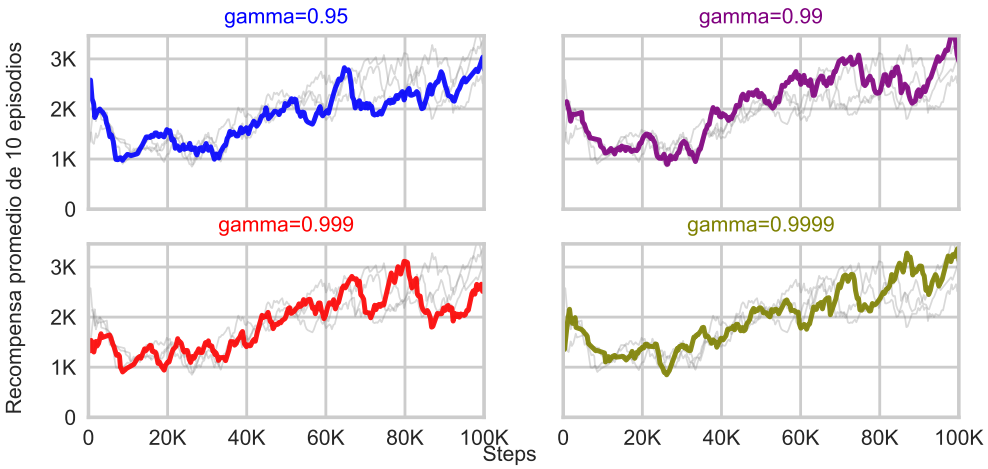


Figura 6.11: Promedio de recompensa de 10 episodios en Super Mario Bros para el algoritmo DQN.

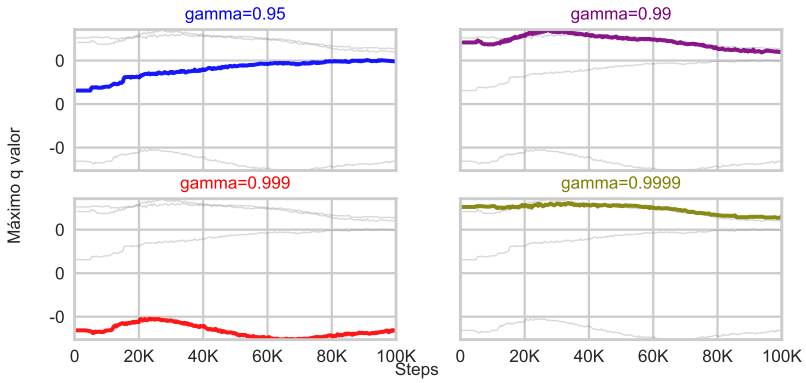


Figura 6.12: Variación del máximo valor de q en Super Mario Bros para el algoritmo DQN.

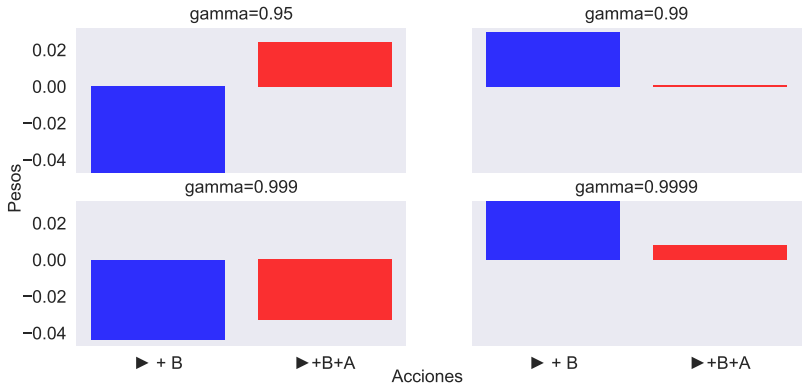


Figura 6.13: Pesos de la última capa de la red por acción para el juego Super Mario Bros.

fs	bs	γ	episodes	steps	avg	hi	qmax	action
6	128	0.99	242	100332	2008.057613	5872.0	0.029555	► + B
6	128	0.9999	231	100044	1994.198276	5895.0	0.031780	► + B
6	128	0.999	253	100029	1861.822835	5183.0	-0.032944	► + B + A
6	128	0.95	249	100042	1842.412000	5480.0	0.024214	► + B + A

Tabla 6.6: Comparación de γ para Super Mario Bros.

Se observan mejores resultados con $\gamma = 0{,}99$, el cual es el valor utilizado por defecto en el resto de experimentos.

6.1.4. Kung Fu

6.1.4.1. Frame skip vs learning rate

Se realizaron 9 experimentos. Los valores de *fs* fueron 4, 8 y 12; mientras que *fs* fue 1.5×10^{-4}, 2×10^{-4} y 2.5×10^{-4} y *bs* se mantuvo fijo. El valor de *epsilon* siempre fue desde 1.0 hasta 0.1.

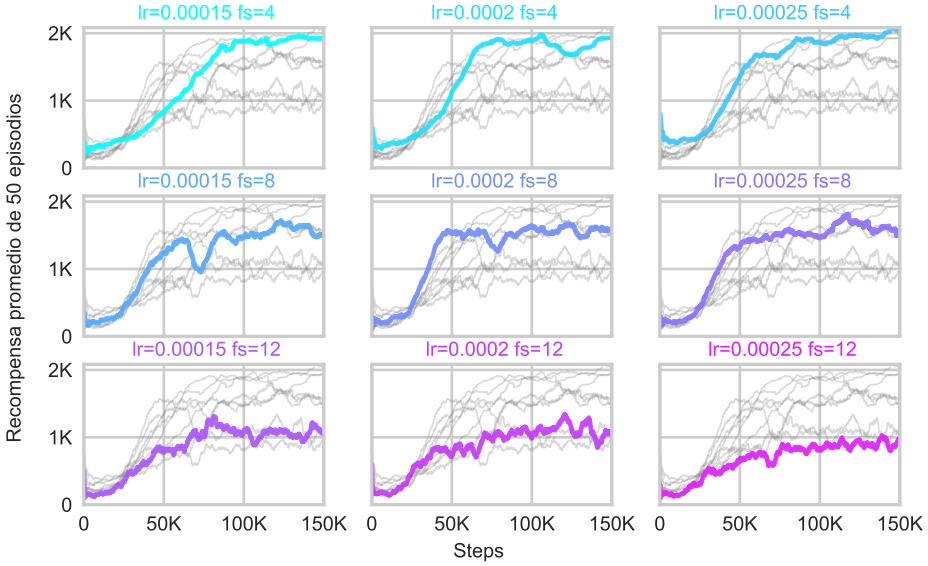


Figura 6.14: Recompensa para el juego Kung Fu.

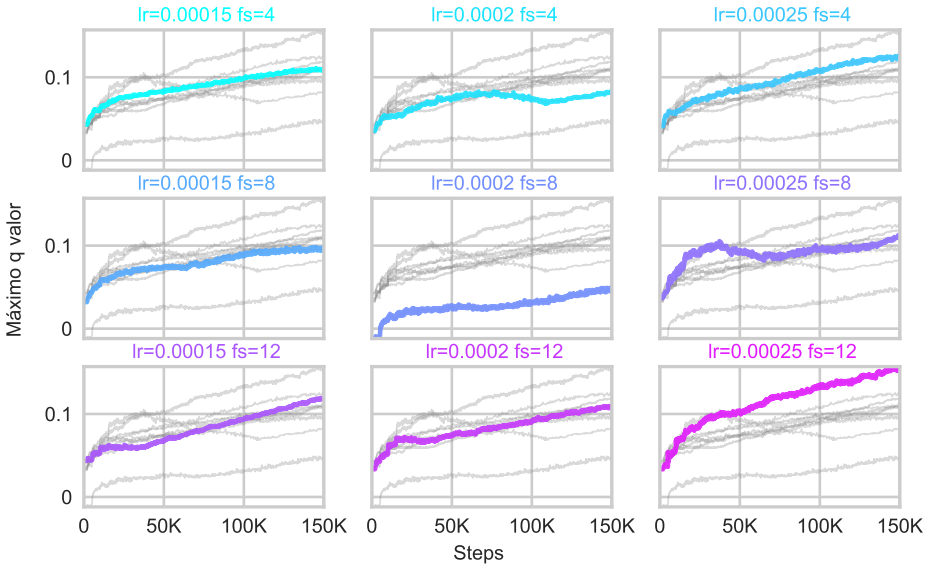


Figura 6.15: Variación del máximo valor de q en Kung Fu para el algoritmo DQN.

Los resultados de la recompensa obtenida en los primeros 9 experimentos pueden verse en la Figura 6.14 y la evolución en el valor *qmax* en la Figura 6.15. Las acciones que el agente puede ejecutar son [▼],[◄],[►],[A],[B] y [▲]. El valor de los pesos de estas acciones pueden verse en la Figura 6.16. La evolución del valor del parámetro ϵ se encuentra en la Figura **??**

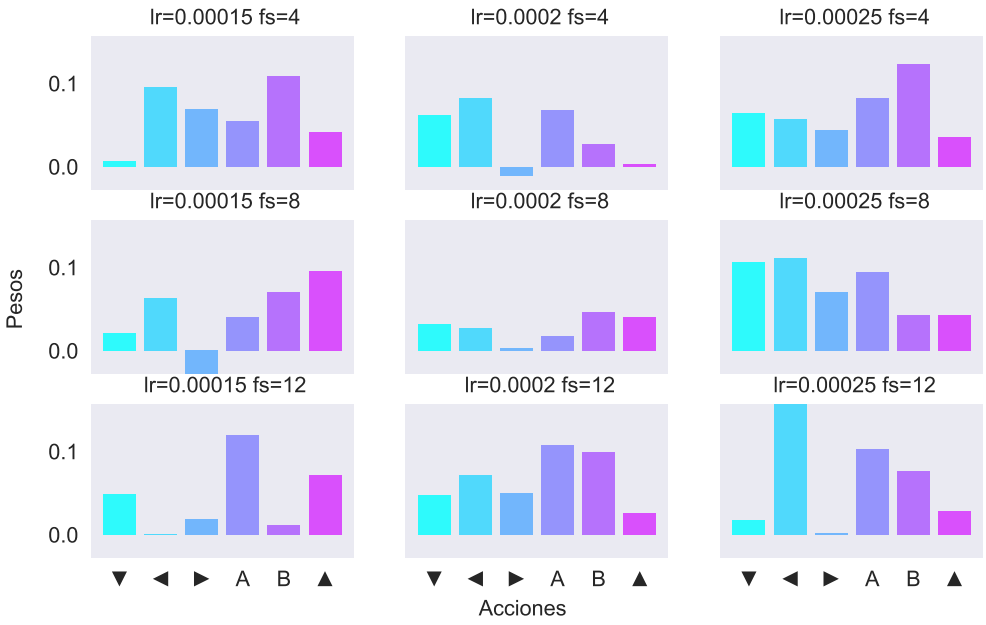


Figura 6.16: Pesos de la última capa de la red por acción para el juego Kung Fu.

fs	lr	episodes	steps	avg	hi	qmax	action
4	0.00025	382	150342	1481.723238	5200.0	0.123472	B
4	0.00020	381	150151	1408.376963	2200.0	0.082522	◄
4	0.00015	387	150245	1306.185567	5200.0	0.108338	B
8	0.00020	839	150085	1252.738095	4400.0	0.046797	B
8	0.00025	832	150080	1236.974790	5100.0	0.110713	◄
8	0.00015	851	150119	1150.938967	3400.0	0.095322	▲
12	0.00020	1438	150103	839.958304	3300.0	0.107937	A
12	0.00015	1407	150088	813.281250	3600.0	0.118957	A
12	0.00025	1493	150023	672.021419	2100.0	0.156191	◄

Tabla 6.7: Comparación de parámetros para Kung Fu

Los resultados indican que el mejor desempeño se da con el agente con parámetros fs=2.5 $\times$ 10^{-4} y lr=4.

6.1.4.2. Batch size

Nuevamente se entrena el agente variando *bs* entre 32, 64 y 128. Se mantienen fijos fs=2.5×10^{-4} y lr=4. Además se utiliza la nueva función de recompensa y se registra el tiempo de entrenamiento.

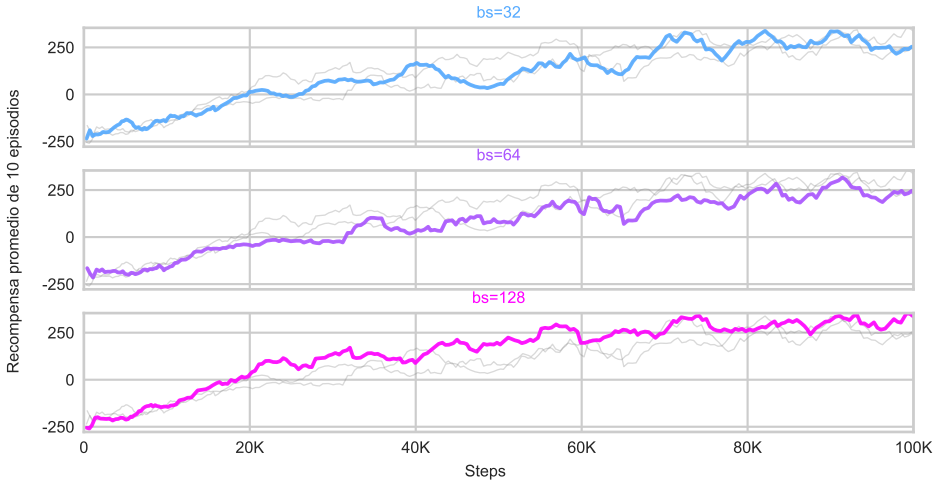


Figura 6.17: Recompensa para el juego Kung Fu.

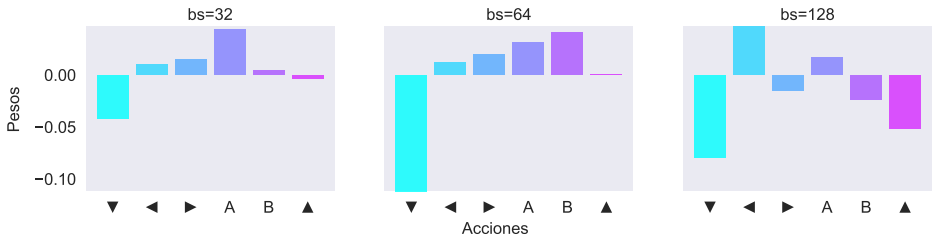


Figura 6.18: Acciones en Kung Fu.

bs	episodes	steps	avg	hi	qmax	action	time(h)
128	238	100011	144.079498	765.0	0.046621	◄	2.06
32	258	100246	95.231660	835.0	0.043563	A	0.72
64	246	100122	69.979757	1075.0	0.040381	B	1.16

Tabla 6.8: Comparación de parámetros para Kung Fu

De acuerdo con la Tabla 6.8 hay un promedio mas alto cuando bs=128 pero es el mas lento. El tiempo de entrenamiento es casi el triple que cuando bs=32 y el valor promedio es cerca de

un 150 %. Observando la Figura 6.18 puede afirmarse que desde el punto de vista jugable tiene mas sentido bs=128.

6.1.5. Metroid

6.1.5.1. Frame skip vs learning rate

Se comparó fs=15 y fs=36 con la tasas de aprendizaje lr=0.00015 y lr=0.00025. Se utilizó la primera función de recompensa. Se fijó bs en 64.

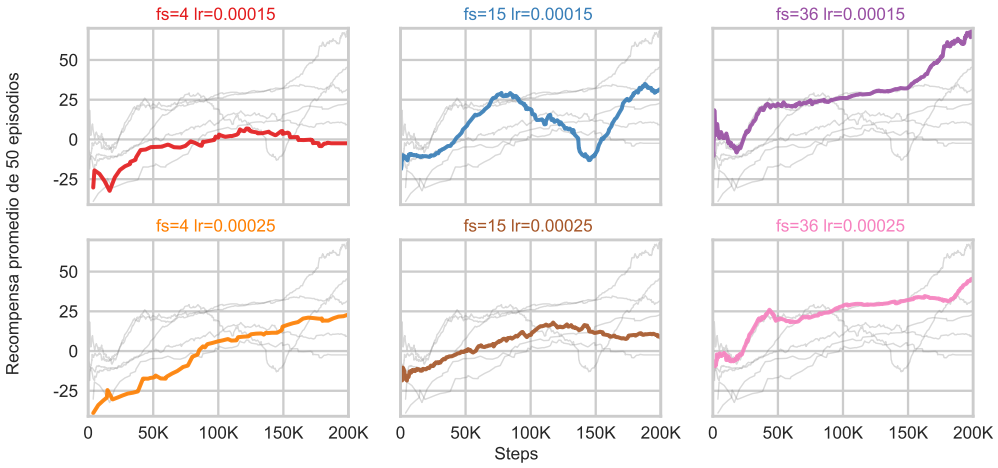


Figura 6.19: Comparación de la recompensa obtenida en Metroid.

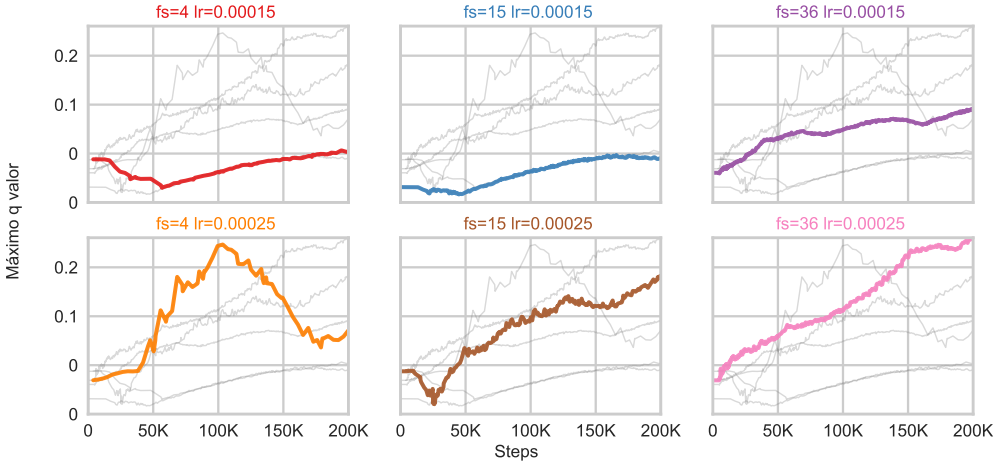


Figura 6.20: Comparación de la evolución del valor q en la capa de salida de la red.

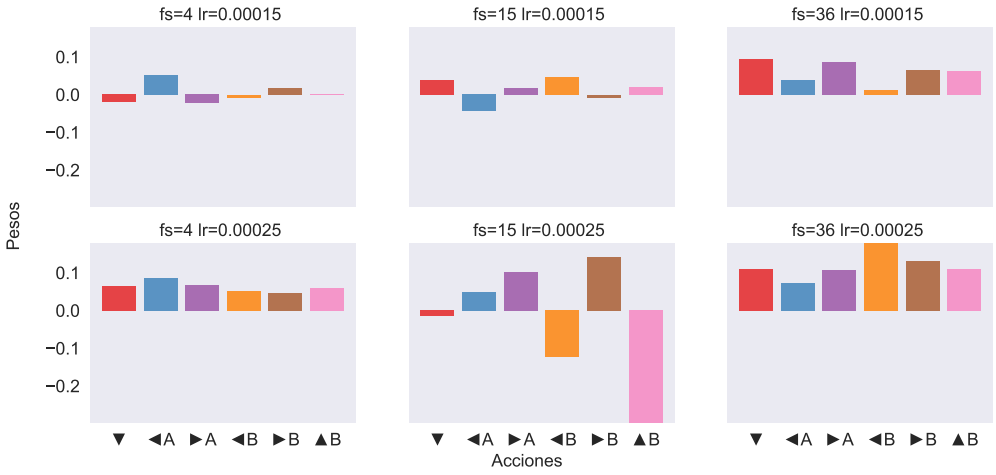


Figura 6.21: Comparación los pesos de la última capa de la red.

fs	lr	episodes	steps	avg	hi	qmax	action
36	0.00015	587	200110	30.519268	93.303000	0.094452	▼
36	0.00025	565	200266	21.325369	85.311000	0.179654	◀ + B
4	0.00025	68	200182	12.532333	142.588999	0.085898	◀ + A
15	0.00015	309	200069	11.190525	96.356000	0.045511	◀ + B
15	0.00025	278	200025	5.047774	83.506000	0.141235	▶ + B
4	0.00015	132	203282	0.046721	90.677000	0.052658	◀ + A

Tabla 6.9: Comparación de parámetros para Metroid

Si bien el puntaje mas alto obtenido ocurre cuando *fs=15* y *lr = 0.00015*, el promedio es mas bajo en comparación con *fs = 30*. A su vez el valor q máximo es mas estable con una tasa de aprendizaje *lr=0.00015*. El mejor de los 4 modelos es y *fs=36* y *lr=0.00015*.

6.1.5.2. Learning rate vs ram mode

A diferencia de los otros juegos Metroid hace uso tanto de la RAM de la consola, como de la memoria RAM del cartucho (WRAM). También durante el transcurso del proyecto se ha encontrado un mapa de la memoria del juego en el que pueden destacarse unos bytes *claves* (157 b).

Se denomina *ram mode* o simplemente *ram* a la cantidad de bytes leídos y usados como observaciones. Por defecto, leer solo los 2 KB de la consola se denomina como *ram*, leer los 10 KB (consola + cartucho) se denomina como *wram* y cuando sean leidos solo los bytes clave se denomina como *map*.

Se mantuvo bs=64 y fs=36, la tasa de aprendizaje lr con 0.00015 y 0.00025 y se compararon los 3 métodos de lectura de la RAM. Se utilizó la primera función de recompensa.

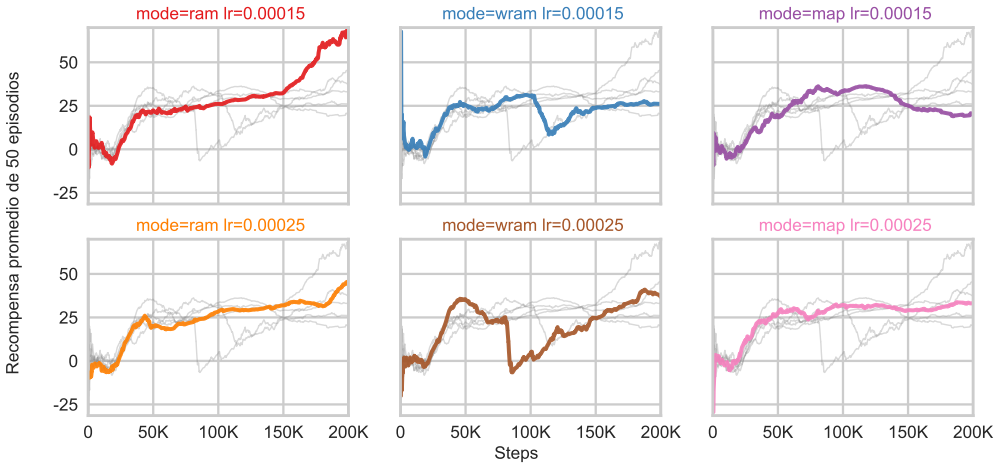


Figura 6.22: Comparación de la recompensa obtenida para ram, wram y map.

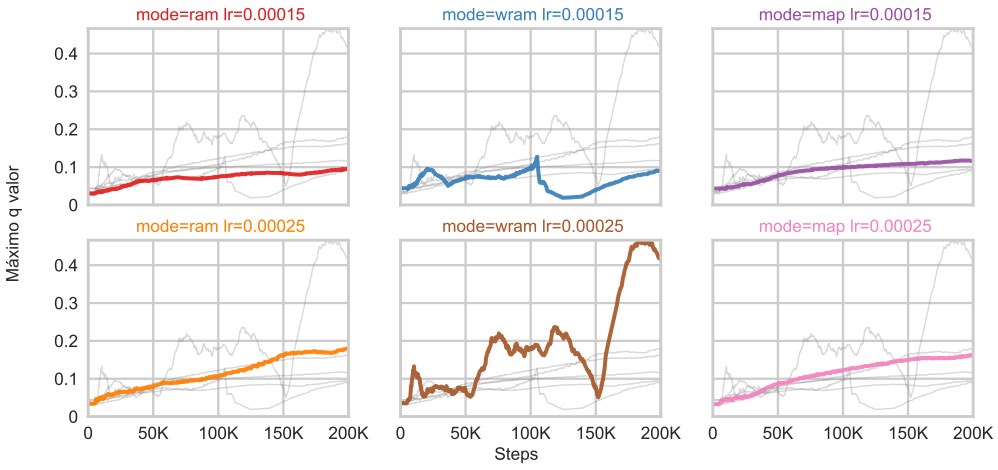


Figura 6.23: Comparación de la evolución del valor q en la capa de salida de la red.

El modo *wram* fue el que peor desempeño tuvo, mayor cantidad de información de entrada en la red también implica mas ruido y cálculos que realizar. Además aumenta considerablemente la velocidad de entrenamiento. El método *map* toma como entrada 20 valores que se encuentra en esa parte de la memoria y al parecer esa información clave es suficiente. En este modo se logró la mayor recompensa durante algún episodio.

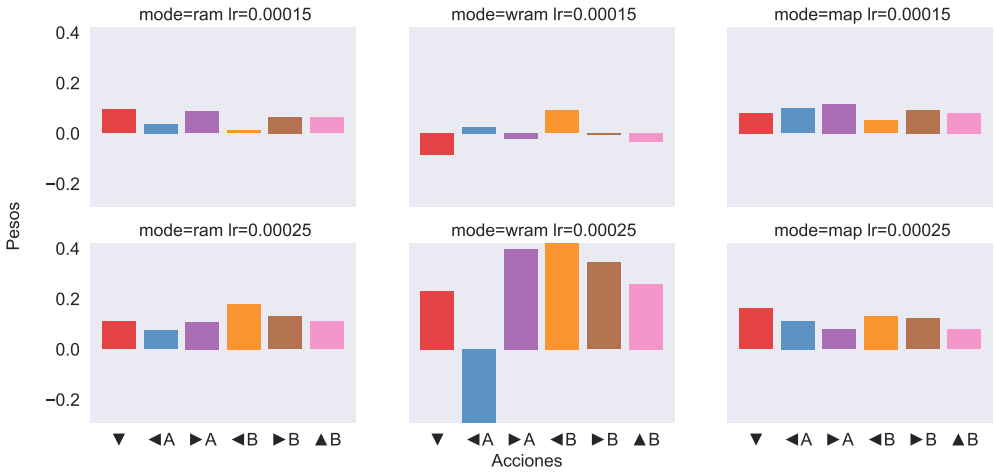


Figura 6.24: Comparación los pesos de la última capa de la red.

lr	mode	episodes	steps	avg	hi	qmax	action
0.00015	ram	587	200110	30.519268	93.303000	0.094452	▼
0.00025	map	530	200223	23.798235	95.579000	0.161584	▼
0.00025	ram	565	200266	21.325369	85.311000	0.179654	◀ + B
0.00015	map	539	200267	20.179996	86.788000	0.116066	▶ + A
0.00015	wram	544	200136	19.285408	88.998000	0.090780	◀ + B
0.00025	wram	649	200058	15.967601	86.153999	0.422575	◀ + B

Tabla 6.10: Comparación de parámetros para Metroid: ram, wram y map.

6.2. Entrenamiento

Los parámetros ajustados de los experimentos anteriores con el algoritmo DQN están listados en las Tablas 6.11 y 6.12. Se usaron los mismos parametros en la ejecución de los algoritmos DDQN, Dueling DQN y Dueling DDQN.

Parámetro	Valor
Memoria	5×10^5
Batch size	32
Gamma	0.99
ϵ_0 inicial	1.0
ϵ' final	1.0
% exploración	0.1
Learning rate	2.5×10^{-4}
Frame skip	depende del juego

Tabla 6.11: *Parametros.*

Juego	Frame skip
Donkey Kong	30
Ice Climber	15
Kung Fu	4
Super Mario Bros	6
Metroid	30

Tabla 6.12: *Frame skip.*

6.2.1. Donkey Kong

Se realizaron 4 experimentos de 3×10^6 pasos. Se presentan los resultados de la recompensa obtenida en la Figura 6.25, la evolución en el valor *qmax* en la Figura 6.26 y los pesos por acción en la Figura 6.27 . Además de usar la función de recompensa propuesta se reemplazaron las acciones [◄+A] y [►+A] por la acción [A]. Así, las acciones que el agente puede ejecutar son [▲], [◄], [►] y [A].

6.2.1.1. Recompensa

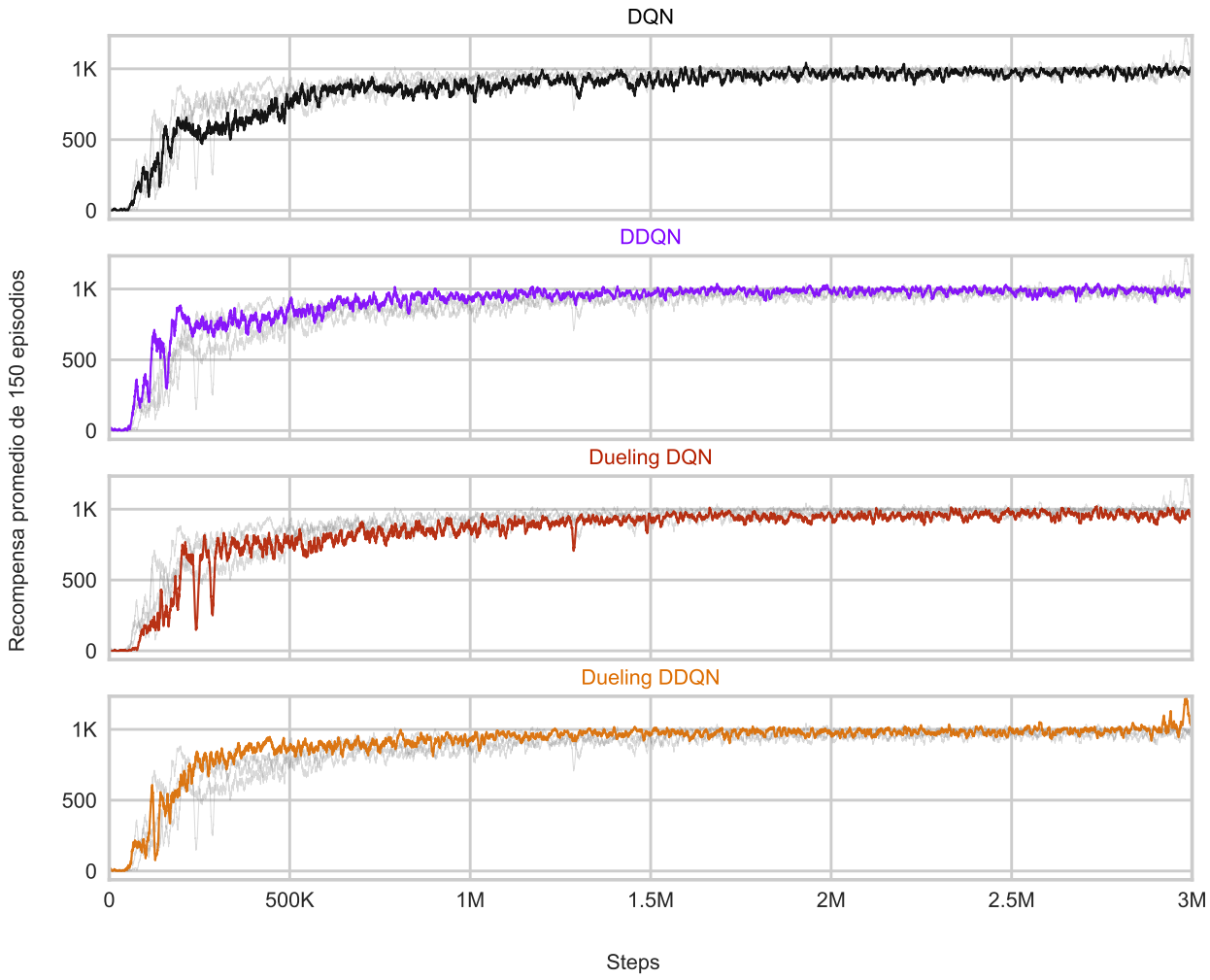


Figura 6.25: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Donkey Kong.*

Agent	episodes	steps	avg	hi	qmax	action
DDQN	66599	3000026	887.185070	7200.0	0.408860	▲
Dueling DDQN	67899	3000044	885.934348	6000.0	0.521262	▲
DQN	68244	3000014	837.137164	7200.0	0.604941	A
Dueling DQN	68263	3000044	825.254138	6400.0	0.689409	▲

Tabla 6.13: *Comparación de Algoritmos para Donkey Kong .*

Los algoritmos DDQN y Dueling DDQN muestran mejor desempeño. Al parecer en este juego el uso de la doble red neuronal. Por los resultados se puede predecir que el agente lograría subir al menos una plataforma cada vez que juegue.

6.2.1.2. Valores Q

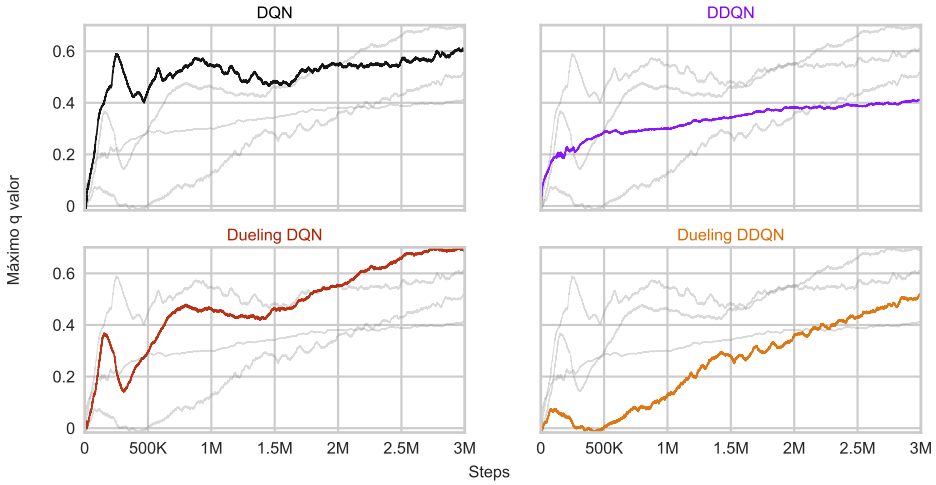


Figura 6.26: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Donkey Kong.*

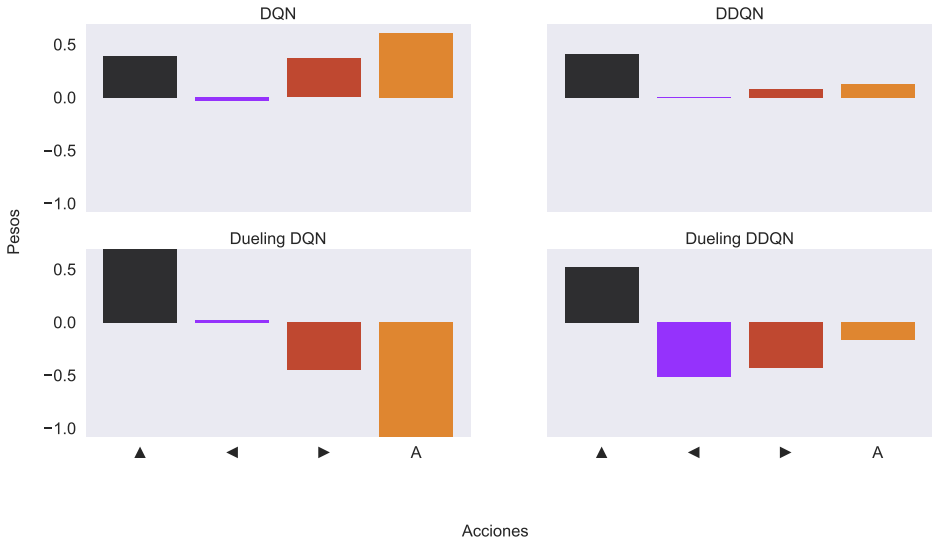


Figura 6.27: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Donkey Kong.*

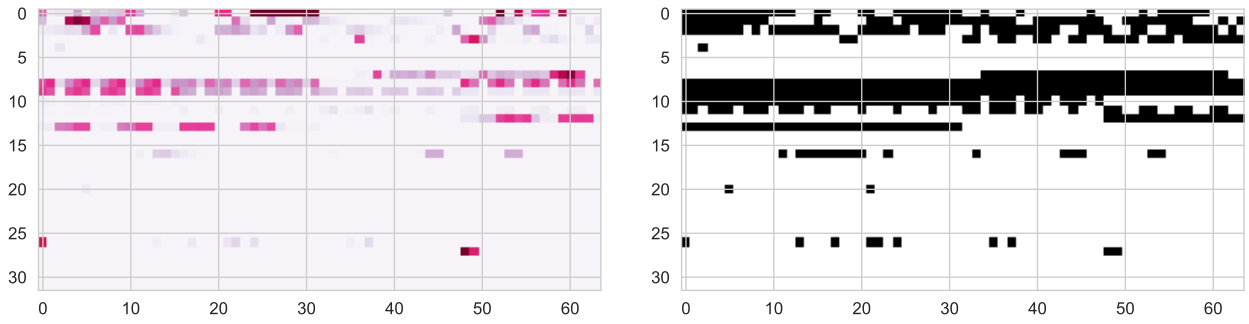
Dueling DQN muestra un desempeño mas estable comparado con los otros algoritmos, también se observa estabilidad en el valor *qmax*.

6.2.1.3. Uso de la RAM

Durante el entrenamiento se midió el número de veces en que se detectó algún cambio en los valores de entrada de la red, es decir, los bytes de la RAM. A estos bytes se le llaman bytes activos, y se relaciona por cantidad en cada algoritmo en la Tabla 6.14. Posteriormente se promediaron las frecuencias de los cuatro algoritmos para hacer una representación gráfica y obtener un único vector de los *bytes activos*. Figura 6.28.

Algoritmo	Bytes
DQN	450
DDQN	458
Dueling DQN	454
Dueling DDQN	456
Todos los algoritmos	448
Un algoritmo o más	460

Tabla 6.14: *Número de bytes activos en RAM durante el entrenamiento en Donkey Kong.*



(a) Frecuencia de bytes activados

(b) bytes activos al menos una vez

Figura 6.28: Bytes en RAM activados durante entrenamiento en Donkey Kong.

6.2.2. Ice Climber

6.2.2.1. Recompensa

Se presentan los resultados de la recompensa obtenida en la Figura 6.29. Las acciones que el agente puede ejecutar son [◄],[►],[A],[◄ + A],[► + A] y [B].

Agent	episodes	steps	avg	hi	qmax	action
DQN	24299	3000047	4.13	141.700000	0.701527	◀ + A
DDQN	24448	3000186	4.10	267.100002	0.458658	◀ + A
Dueling DQN	24645	3000140	2.84	33.300001	0.054018	▶
Dueling DDQN	20107	3000064	1.96	167.600001	-0.020930	▶

Tabla 6.15: *Resultados de entrenamiento Ice Climber .*

Una de las variables influyen en la recompensa es *score*, este solo es sumado si el personaje logra terminar el nivel, excepto Dueling DQN, los agentes logran completar el primer nivel. DDQN lo logra al menos 10 veces mientras que DQN lo hace 3 veces pero es el único que mantiene una recompensa promedio mayor al resto. A partir de 1 millón de pasos todos los algoritmos tienden a degradarse. Esto puede deberse a que el juego cambia radicalmente cuando el personaje sube 10 pisos y que la función de recompensa no contempla este cambio.

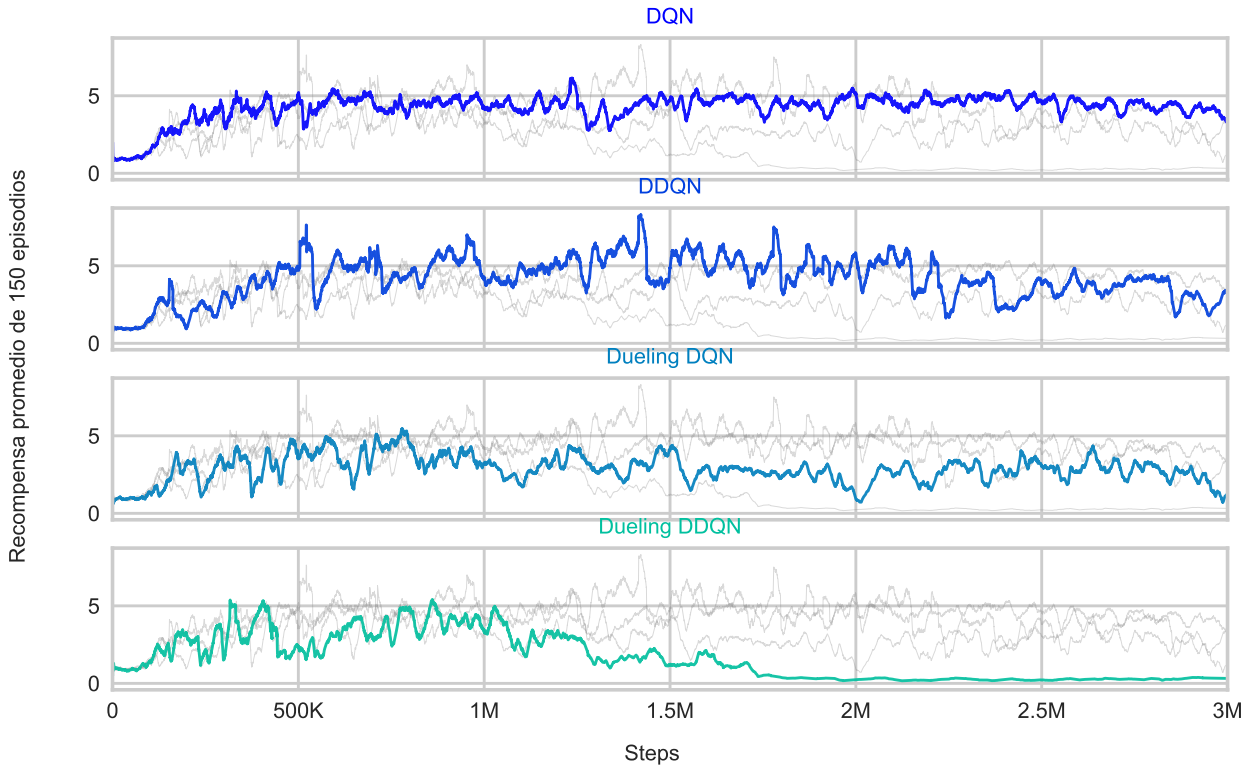


Figura 6.29: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Ice Climber.*

6.2.2.2. Valores Q

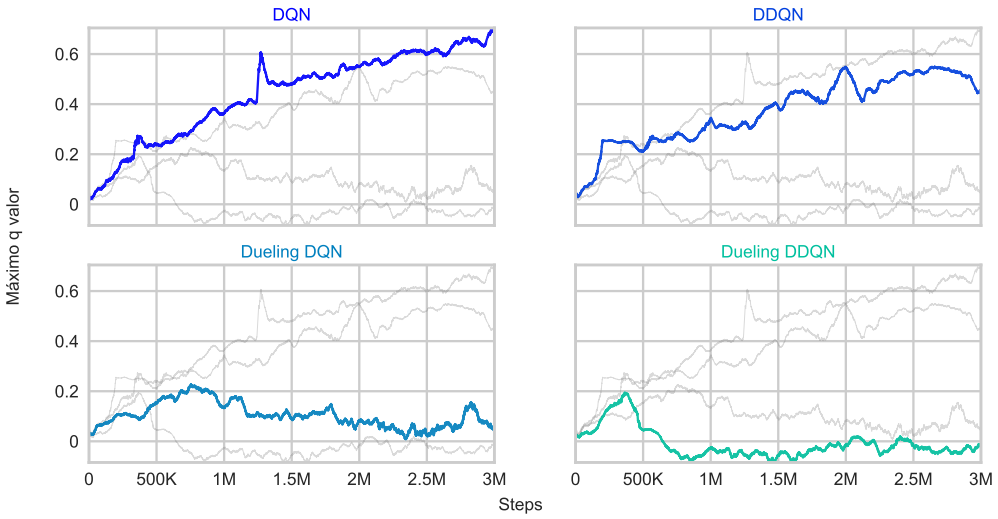


Figura 6.30: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Ice Climber.*

De lejos DQN hace una sobre-estimación de los pesos de las acciones con el valor q siempre creciente. Los pesos de Dueling DQN y Dueling DDQN son negativos.

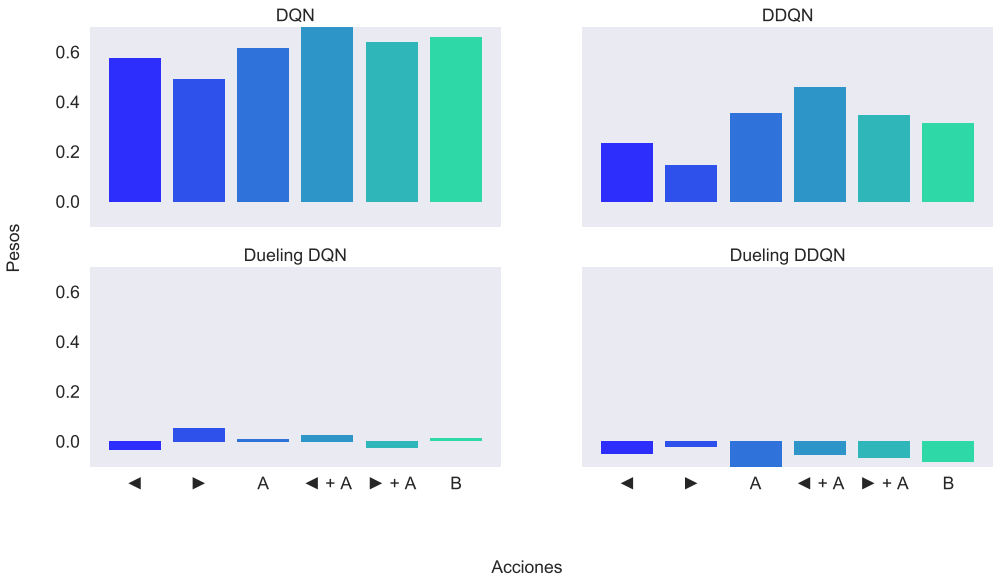


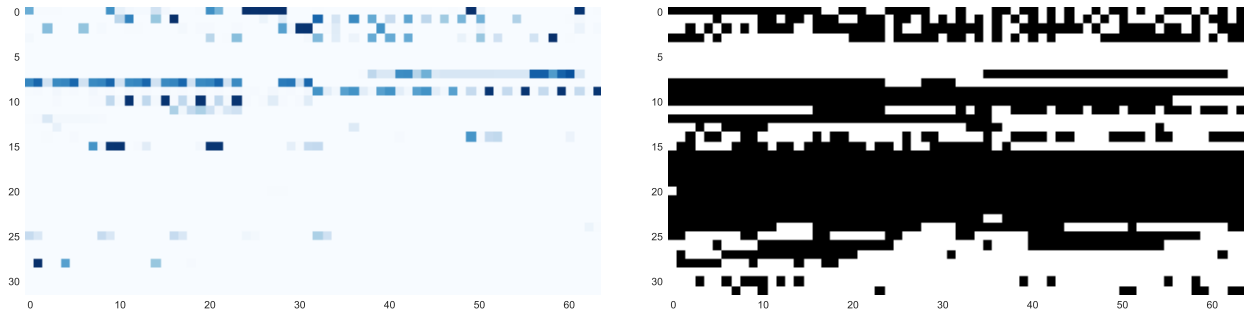
Figura 6.31: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Ice Climber.*

6.2.2.3. Uso de la RAM

Algoritmo	Bytes
DQN	640
DDQN	1102
Dueling DQN	619
Dueling DDQN	436
Todos los algoritmos	424
Un algoritmo o más	1113

Tabla 6.16: *Número de bytes activos en RAM durante el entrenamiento en Ice Climber.*

DDQN al completar el nivel activa el doble de bytes que los otros algoritmos.



(a) Frecuencia de bytes activados

(b) bytes activos al menos una vez

Figura 6.32: Bytes en RAM activados durante entrenamiento en Ice Climber.

6.2.3. Super Mario Bros

6.2.3.1. Recompensa

Agent	episodes	steps	avg	hi	qmax	action
Dueling DQN	9404	3000285	2104.02	5488.0	0.083661	▶ + B
Dueling DDQN	9538	3000237	2092.84	5466.0	0.003037	▶ + B
DQN	9638	3000450	1978.62	4882.0	-0.242963	▶+B+A
DDQN	8852	3000259	1964.91	4884.0	0.433738	▶ + B

Tabla 6.17: *Resultados de entrenamiento Super Mario Bros .*

No se observan diferencias significativas entre los cuatro algoritmos, puede influir en esto que el agente puede ejecutar solamente dos acciones. El mejor desempeño se observa con Dueling DQN y Dueling DDQN.

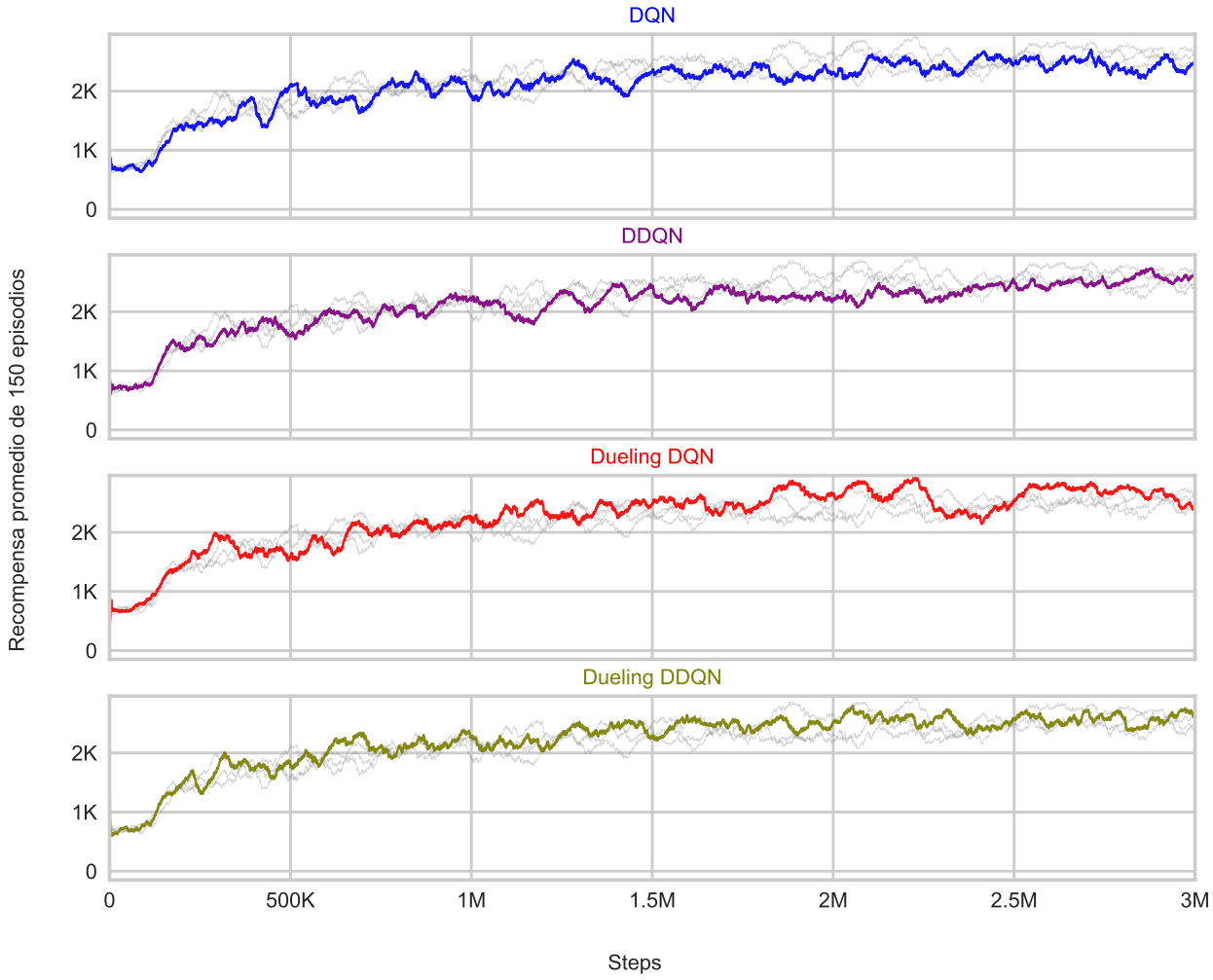


Figura 6.33: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Super Mario Bros.*

6.2.3.2. Valores Q

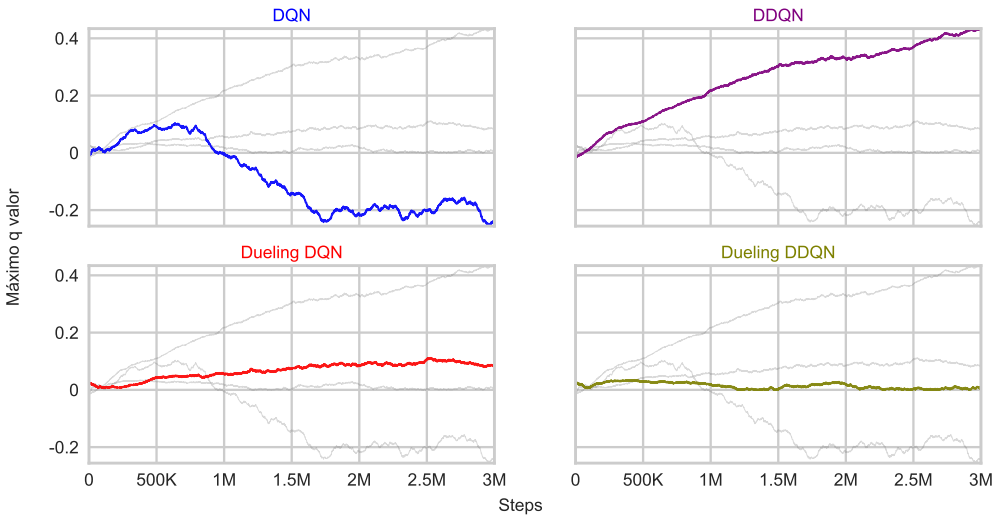


Figura 6.34: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Super Mario Bros.*

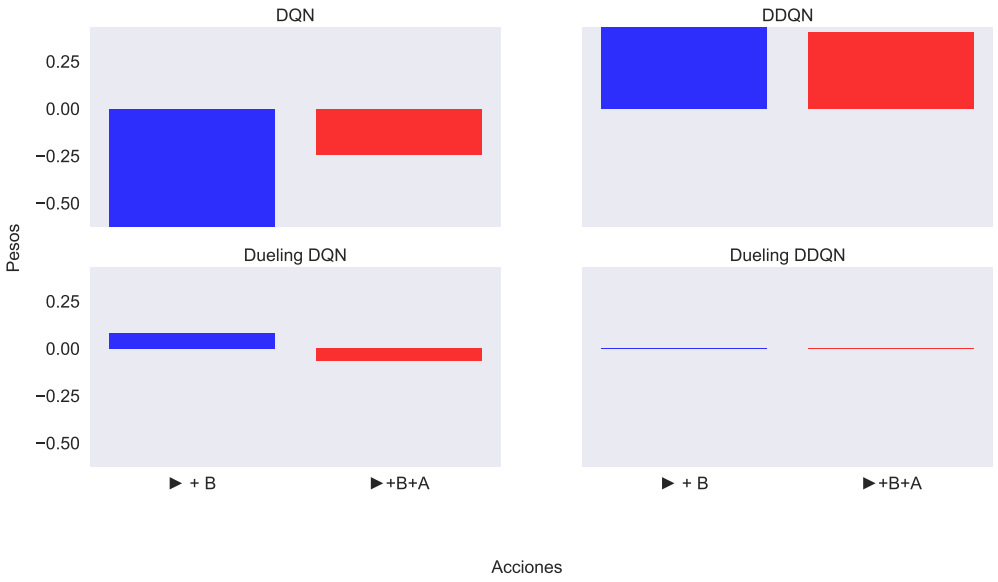
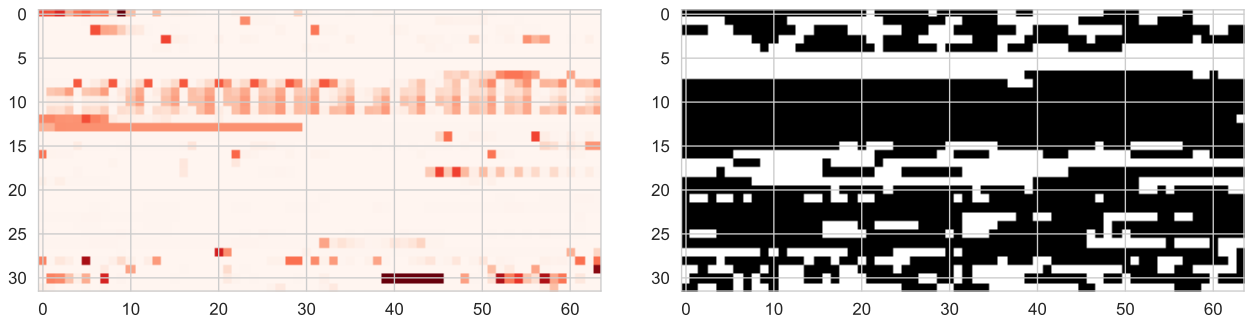


Figura 6.35: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Super Mario Bros.*

6.2.3.3. Uso de la RAM

Algoritmo	Bytes
DQN	1289
DDQN	1277
Dueling DQN	1313
Dueling DDQN	1256
Todos los algoritmos	1255
Un algoritmo o más	1318

Tabla 6.18: *Número de bytes activos en RAM durante el entrenamiento en Super Mario Bros.*



(a) Frecuencia de bytes activados

(b) bytes activos al menos una vez

Figura 6.36: Bytes en RAM activados durante entrenamiento en Super Mario Bros.

6.2.4. Kung Fu

6.2.4.1. Recompensa

Agent	episodes	steps	avg	hi	qmax	action
Dueling DDQN	5544	3000491	489.08	3726.0	0.140356	A
Dueling DQN	5492	3000249	487.47	3234.0	0.188051	◀
DDQN	5646	3000027	464.32	3153.0	0.416421	A
DQN	5690	3000102	456.11	1415.0	0.389699	B

Tabla 6.19: *Resultados de entrenamiento Kung Fu .*

Dueling DDQN y Dueling DQN tienen un desempeño bastante similar en promedio, Dueling DDQN muestra que tuvo en determinado momento una mayor recompensa.

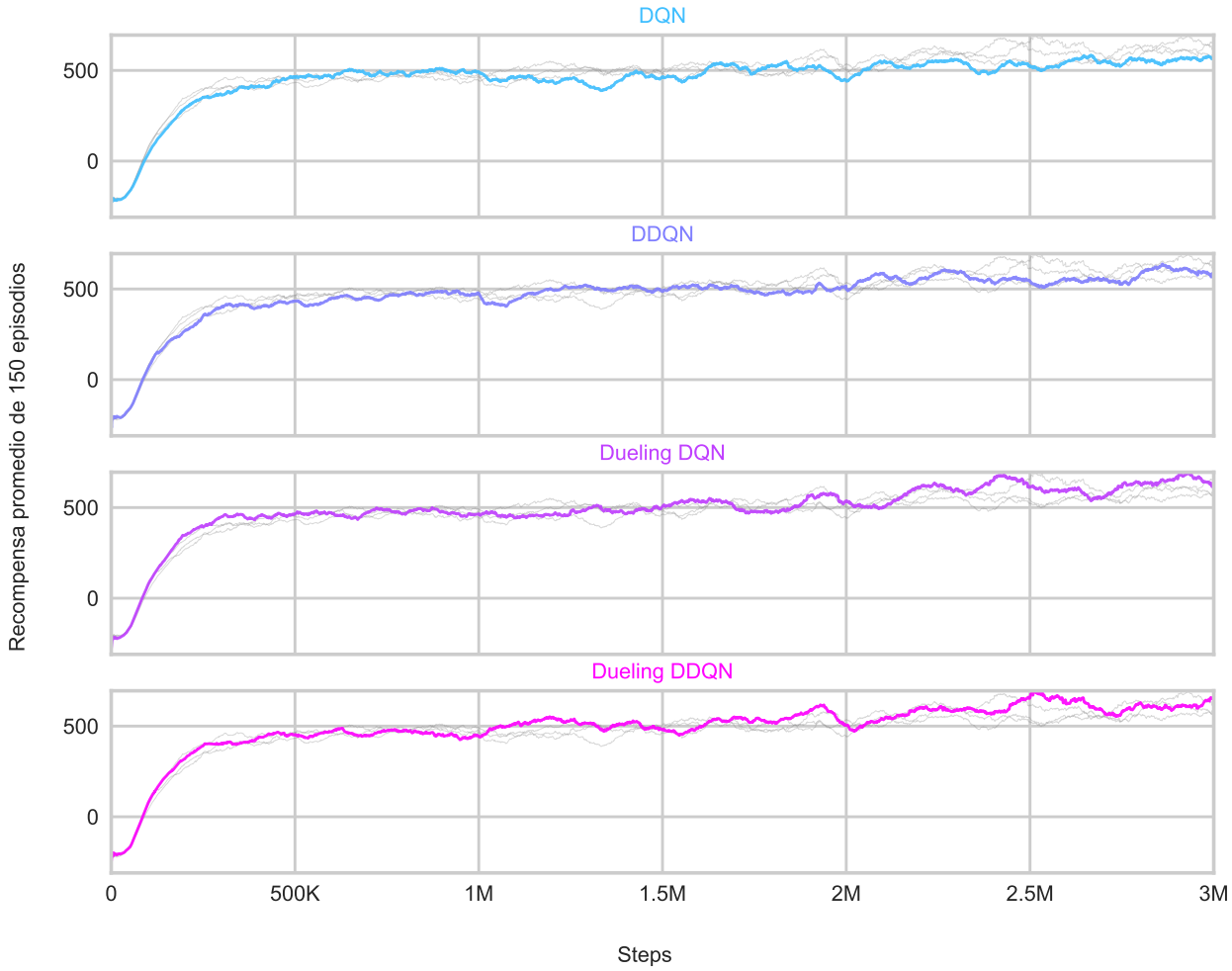


Figura 6.37: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Kung Fu.*

6.2.4.2. Valores Q

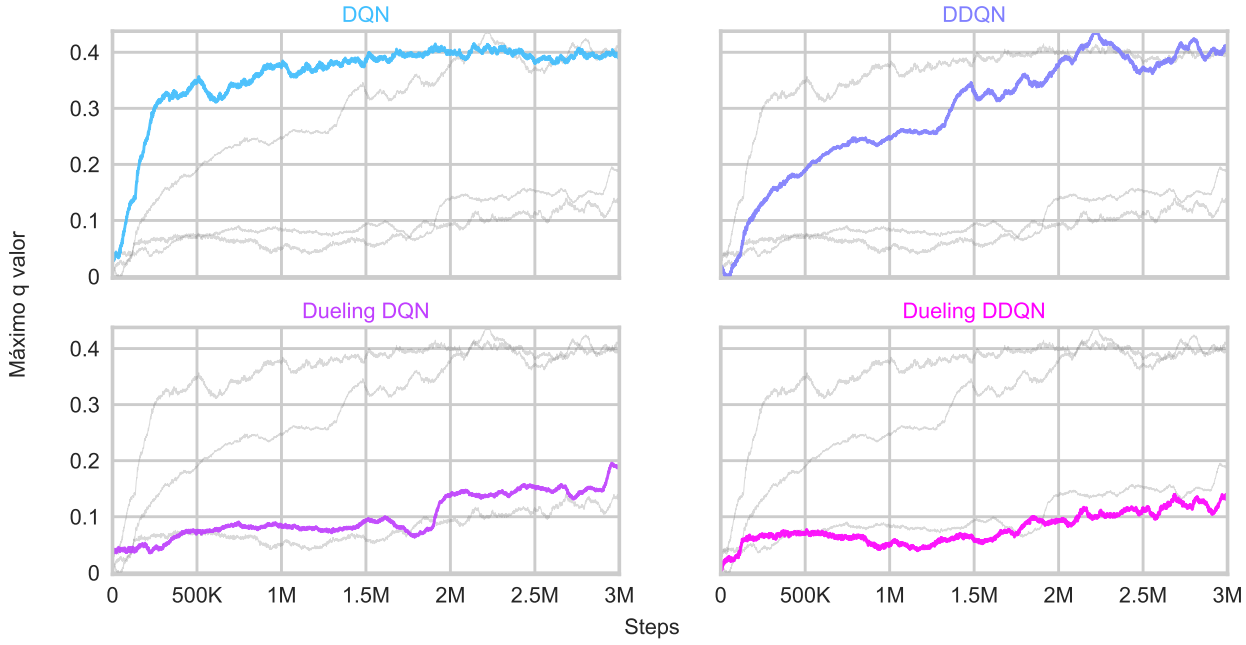


Figura 6.38: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Kung Fu.*

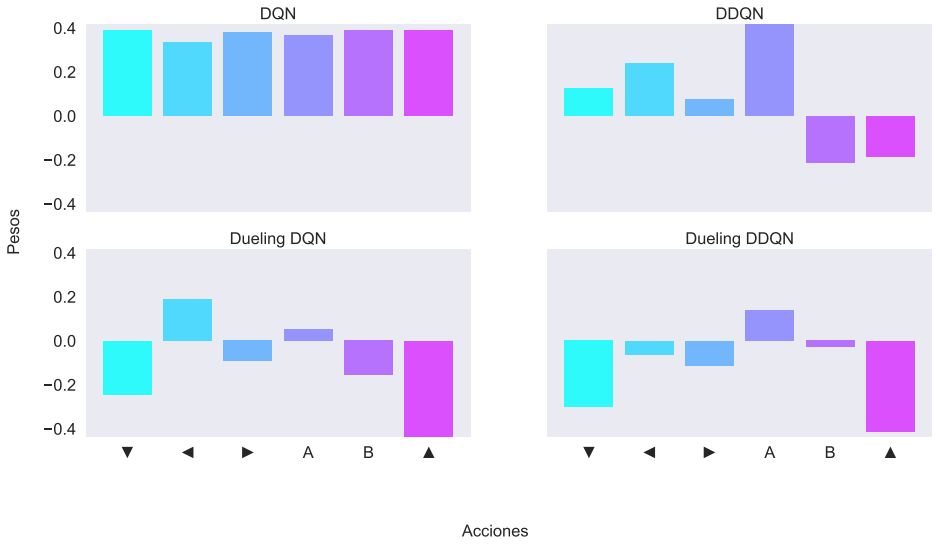
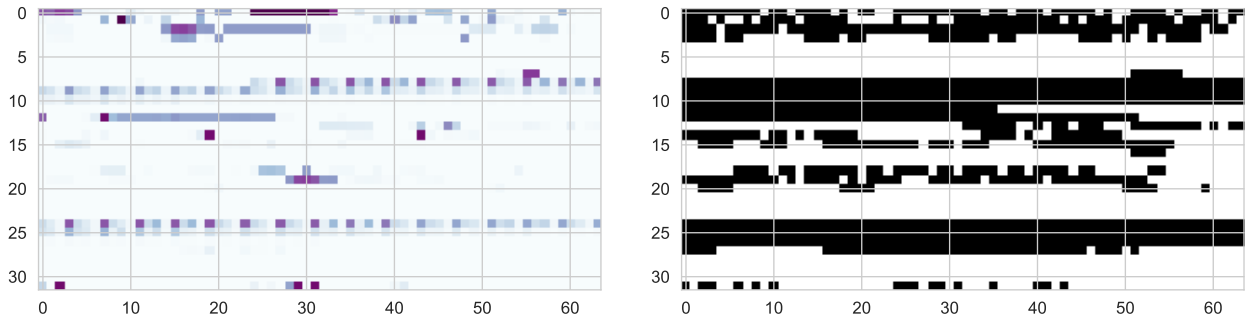


Figura 6.39: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Kung Fu.*

6.2.4.3. Uso de la RAM

Algoritmo	Bytes
DQN	836
DDQN	498
Dueling DQN	895
Dueling DDQN	893
Todos los algoritmos	498
Un algoritmo o más	900

Tabla 6.20: *Número de bytes activos en RAM durante el entrenamiento en Kung Fu.*



(a) Frecuencia de bytes activados

(b) bytes activos al menos una vez

Figura 6.40: Bytes en RAM activados durante entrenamiento en Kung Fu.

6.2.4.4. RAM modo map

Con los bytes que fueron activados de la RAM durante la fase de entrenamiento, se entrenó un agente Dueling DDQN con esos 900 bytes como entrada de la red neuronal (modo map). Se presentan los resultados comparándolo con el agente usando 2 KB (modo ram). Los pesos indican una acción más parecida a lo que se espera en este juego. El agente en modo map obtiene mejores resultados en menor tiempo de entrenamiento.

Agent	ram	episodes	steps	avg	hi	qmax	action
Dueling DDQN	map	5471	3000588	508.66	3360.0	0.451518	◀
Dueling DDQN	ram	5544	3000491	489.08	3726.0	0.140356	A

Tabla 6.21: *Resultados de entrenamiento Kung Fu modo map .*

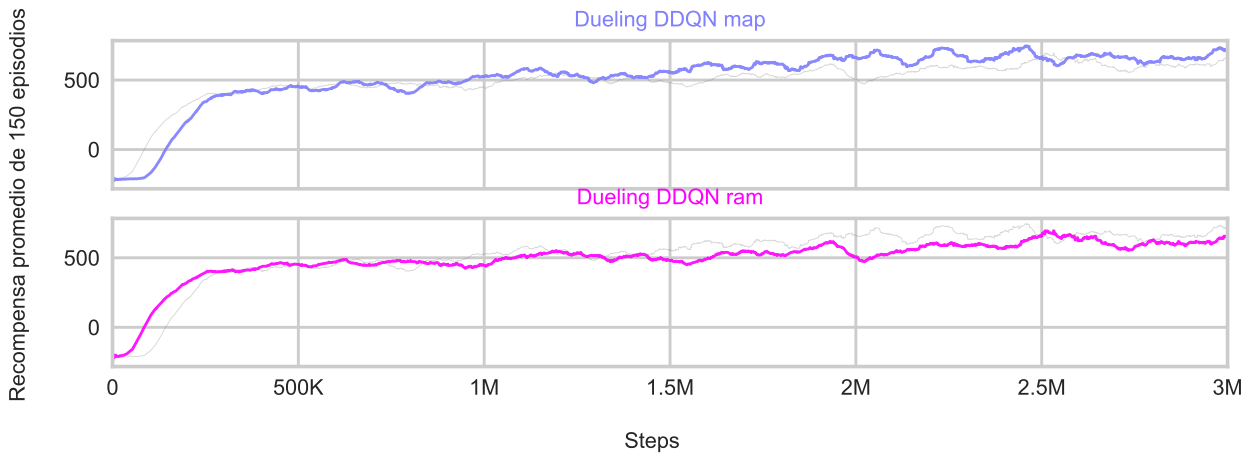


Figura 6.41: *Promedio de recompensa de 150 episodios del algoritmo Dueling DDQN en Kung Fu en modos map y ram.*

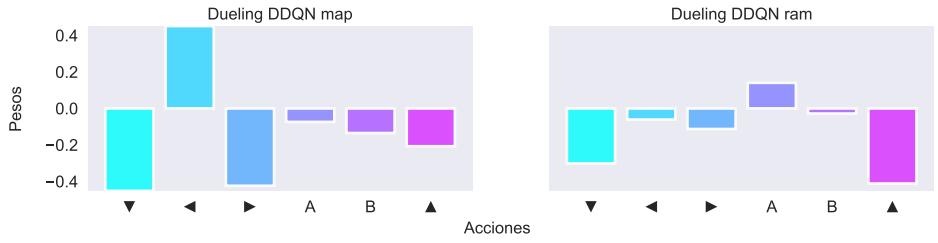


Figura 6.42: *Pesos de la última capa de la red neuronal de los algoritmos Dueling DDQN en modos map y ram en Kung Fu.*

6.2.5. Metroid

6.2.5.1. Recompensa

Agent	mode	episodes	steps	avg	hi	qmax	action
Dueling DQN	ram	11846	3000124	109.19	129.0	0.325789	▼
DDQN	ram	12036	3000196	108.27	128.0	0.883983	► + A
Dueling DDQN	ram	12385	3000085	107.17	127.0	0.206919	▲ + B
DQN	ram	12282	3000180	106.92	128.0	0.455567	► + A

Tabla 6.22: *Resultados de entrenamiento en Metroid .*

Como se mencionó anteriormente, en este juego el agente debe dirigirse primero hacia la izquierda por un power up, e inmediatamente debe dirigirse hacia la derecha. Se observa un desempeño bastante similar con los cuatro algoritmos, después de 1.5 millones de pasos se obtiene la misma recompensa. Esto indica que el agente logra el primer objetivo (morphing ball) pero luego al ir hacia la derecha no está avanzando más allá del punto de partida inicial.

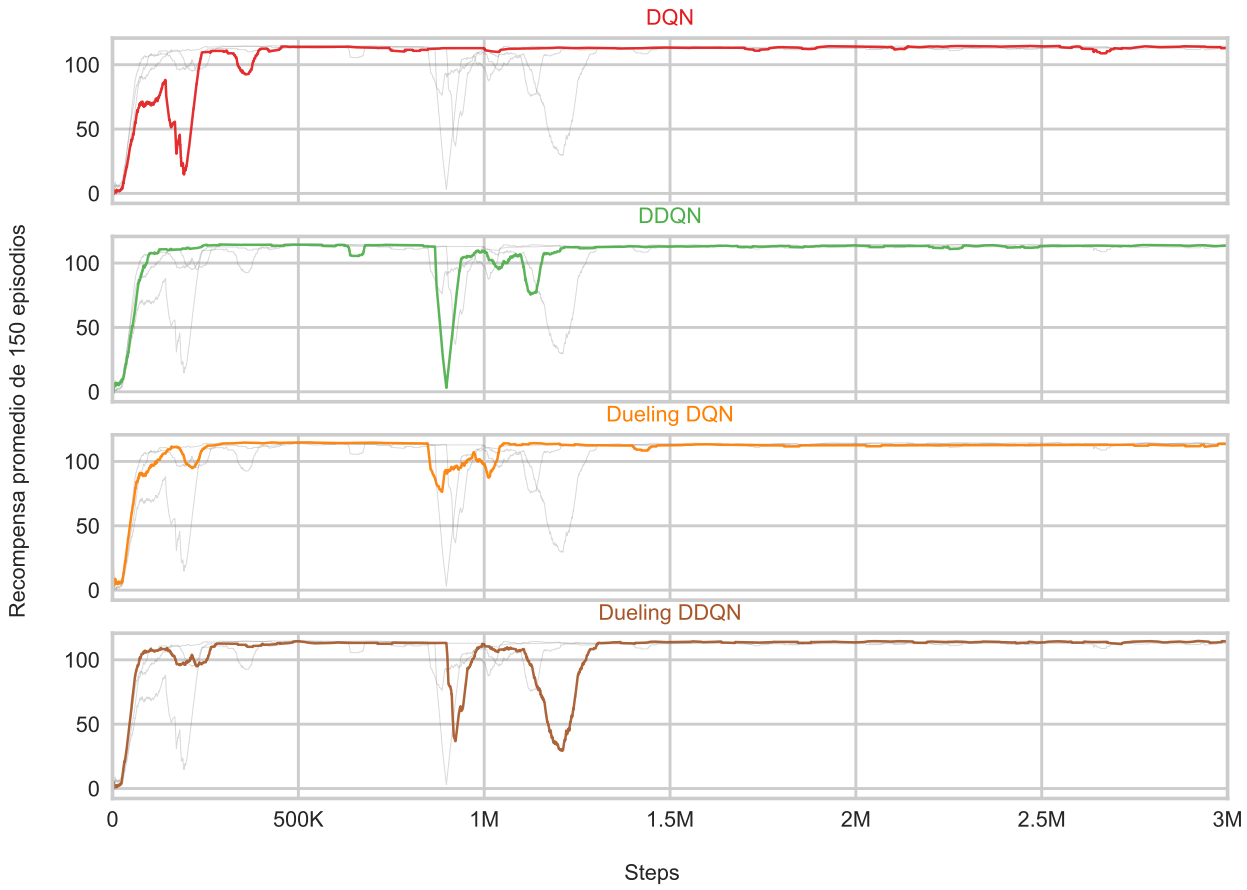


Figura 6.43: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid.*

6.2.5.2. Valores Q

El algoritmo DDQN presenta algún problema cerca del primer millón de pasos, tarda casi otro millón de pasos para estabilizarse, esta anomalía podría explicar el hecho que, contrario a la teoría sea el algoritmo que mas sobre-estima los pesos de las acciones.

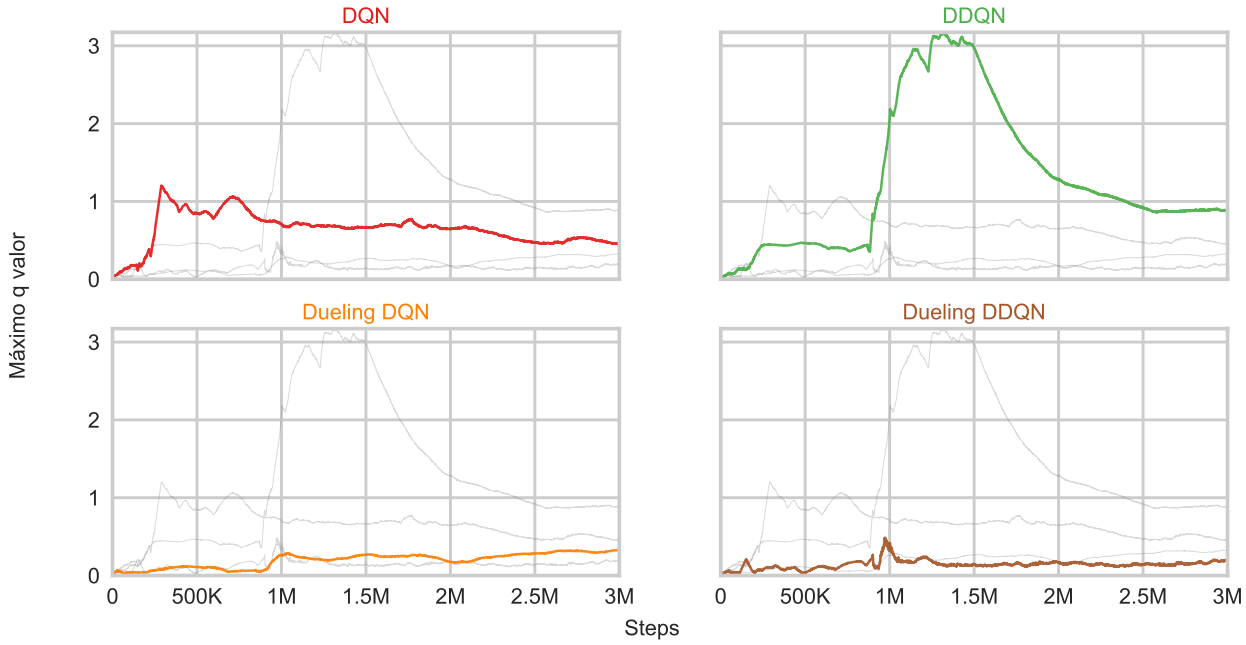


Figura 6.44: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid.*

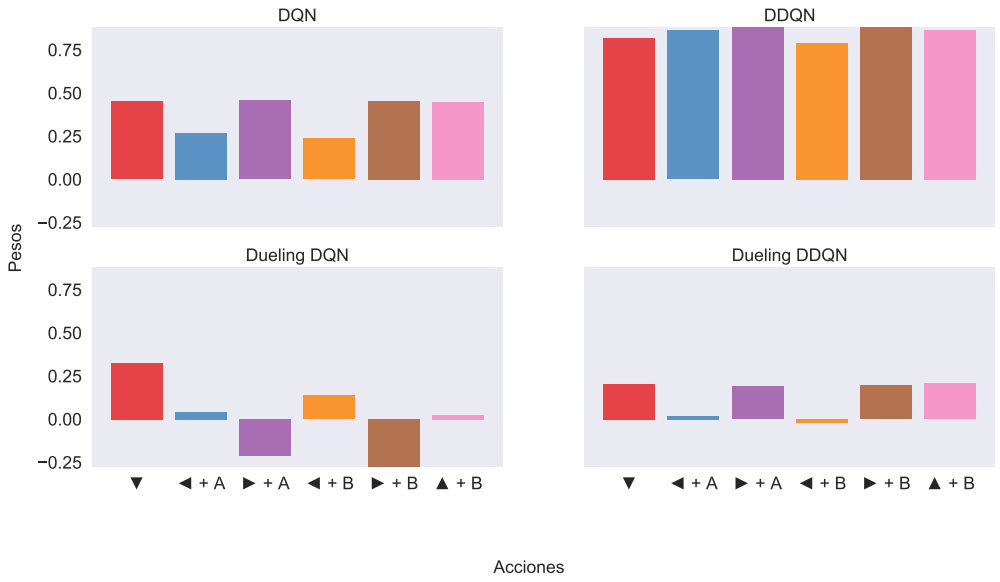
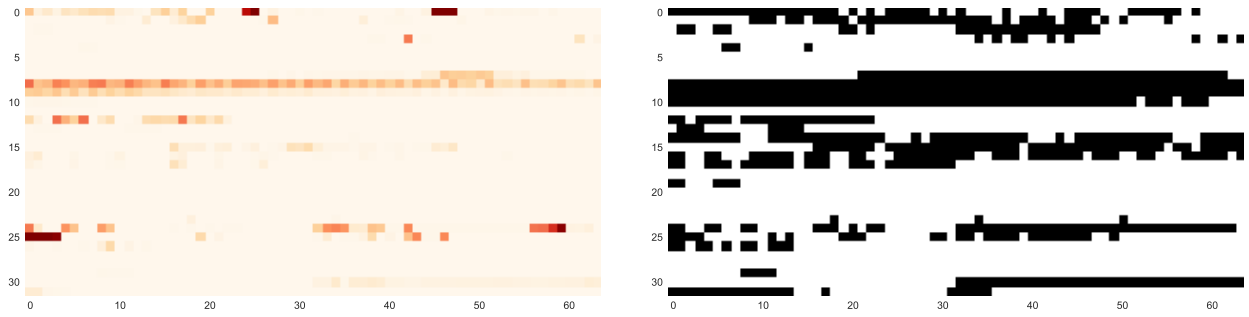


Figura 6.45: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid.*

6.2.5.3. Uso de la RAM

Algoritmo	Bytes
DQN	668
DDQN	459
Dueling DQN	639
Dueling DDQN	635
Todos los algoritmos	459
Un algoritmo o más	670

Tabla 6.23: *Número de bytes activos en RAM durante el entrenamiento en Metroid.*



(a) Frecuencia de bytes activados

(b) bytes activos al menos una vez

Figura 6.46: Bytes en RAM activados durante entrenamiento en Metroid.

6.2.5.4. RAM modo map

Durante la fase inicial de la investigación se encontró un mapa de la RAM mas detallado para este juego. Con base en ello se construyó un vector de características con bytes tanto de la RAM como de la memoria WRAM. Esto permitió reducir el número de entradas de la red a 157 bytes claves. Con estos bytes se entrenó el agente usando los 4 algoritmos. Se presentan los resultados comparados con los del apartado anterior (modo ram).

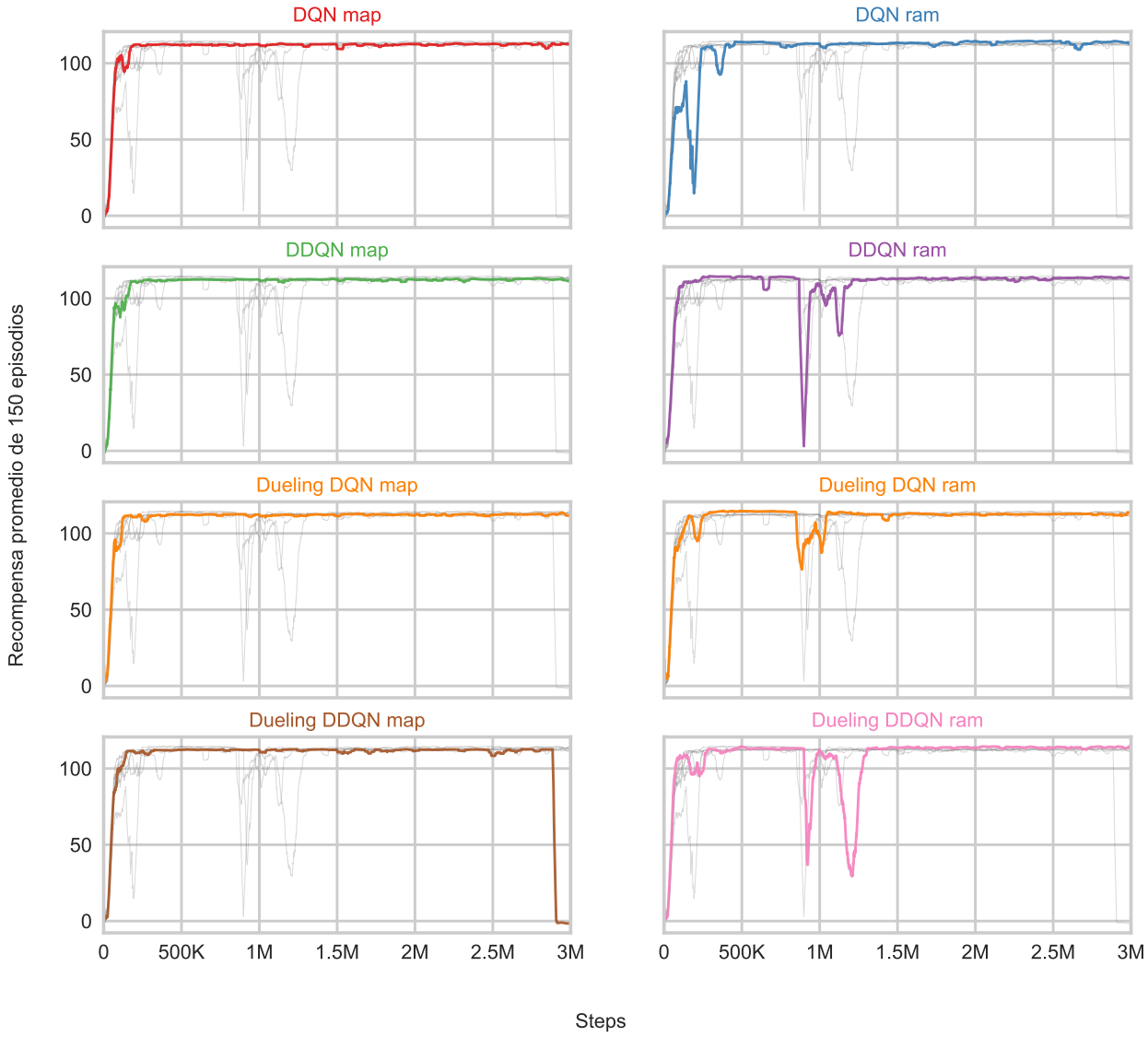


Figura 6.47: *Promedio de recompensa de 150 episodios de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid en modo map.*

Durante uno de los experimentos (Dueling DDQN) se presentó una interrupción cerca del final, en este caso no recuperó lo aprendido luego de retomar el entrenamiento. En general con esta excepción, los agentes en modo map muestran mejores resultados que los que usan los 2048 bytes de la memoria RAM.

6.2.5.5. Recompensa

Agent	modo	episodes	steps	avg	hi	qmax	action
Dueling DQN	map	11233	3000053	109.82	127.0	0.139381	▲ + B
DDQN	map	11280	3000151	109.58	126.0	0.337362	◀ + A
DQN	map	11287	3000031	109.32	115.0	0.471640	◀ + A
Dueling DQN	ram	11846	3000124	109.19	129.0	0.325789	▼
DDQN	ram	12036	3000196	108.27	128.0	0.883983	▶ + A
Dueling DDQN	ram	12385	3000085	107.17	127.0	0.206919	▲ + B
DQN	ram	12282	3000180	106.92	128.0	0.455567	▶ + A
Dueling DDQN	map	11319	3000154	104.07	125.0	0.243624	◀ + A

Tabla 6.24: *Resultados de entrenamiento en Metroid .*

6.2.5.6. Valores Q

Los algoritmos en modo map muestran mayor estabilidad a los algoritmos en modo ram.

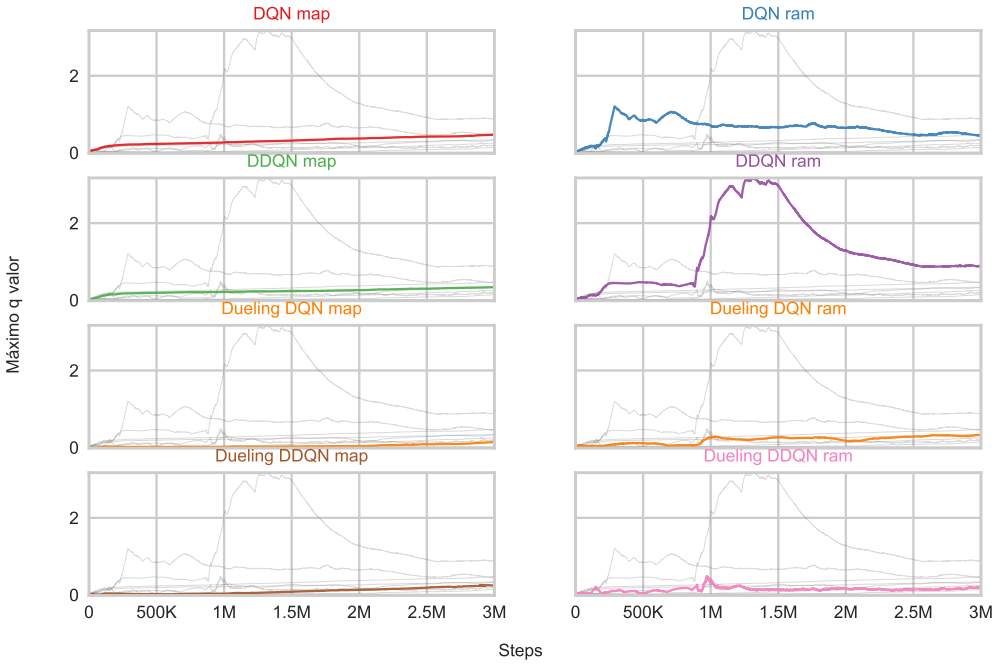


Figura 6.48: *Variación qmax en para los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid en modos ram y map.*

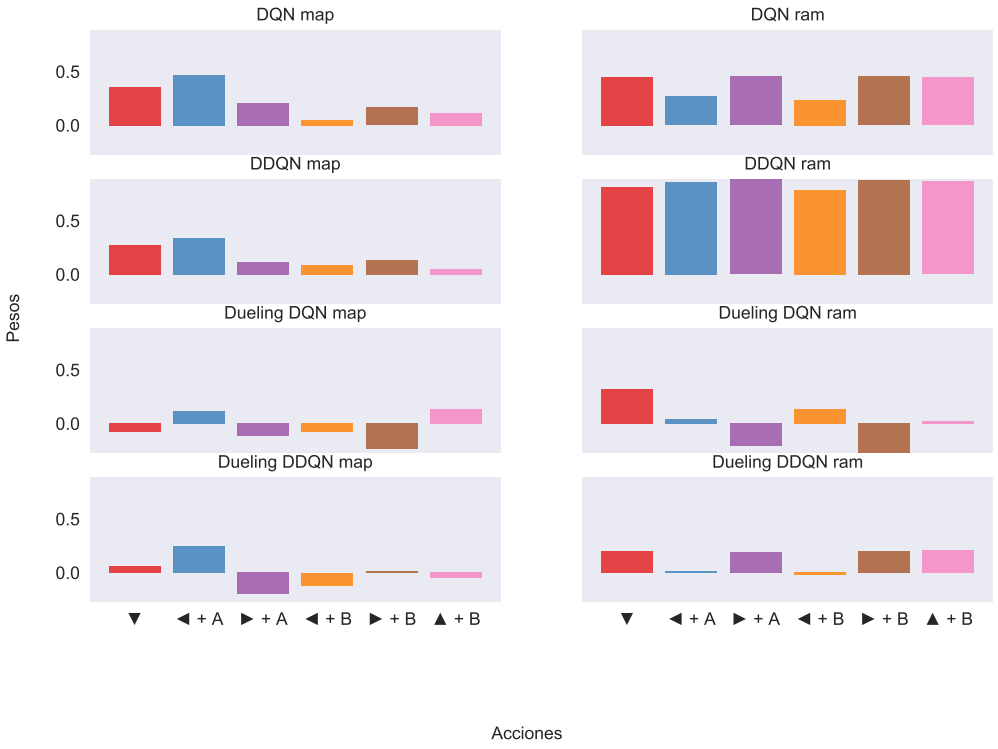


Figura 6.49: *Pesos de la última capa de la red neuronal de los algoritmos DQN, DDQN, Dueling DQN y Dueling DDQN en Metroid en modos ram y map.*

6.3. Pruebas de juego

Después de entrenar al agente se utilizó el último modelo entrenado por algoritmo para que el agente jugara durante 30 episodios con el valor fijo ϵ=0.05. Los resultados obtenidos por cada juego son presentados a continuación.

6.3.1. Donkey Kong

El personaje fue capaz de subir a la segunda plataforma al menos una vez en la mayoría de los casos. DQN fue mas consistente en esta tarea. Puede observarse en la Figura 6.50 y en la Tabla 6.25 que Dueling DDQN presentó mayor varianza pero logró el mayor valor de recompensa. Ocasionalmente el agente logró subir mas allá de la segunda plataforma.

Agente	avg	std	hi
DQN	2803.33	569.49	3200.0
Dueling DDQN	2410.00	951.96	4100.0
Dueling DQN	2186.67	808.59	3100.0
DDQN	573.33	609.88	2000.0

Tabla 6.25: *Resultados de 30 episodios de juego en Donkey Kong.*

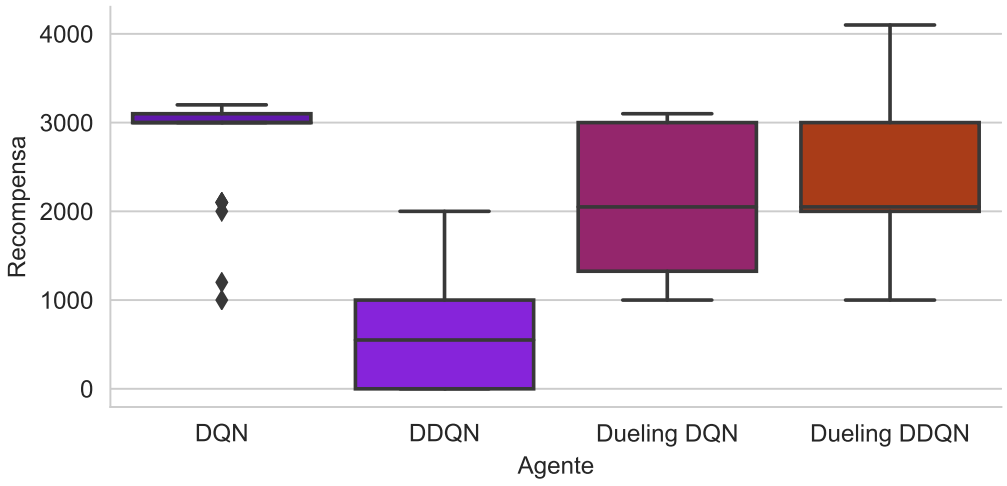
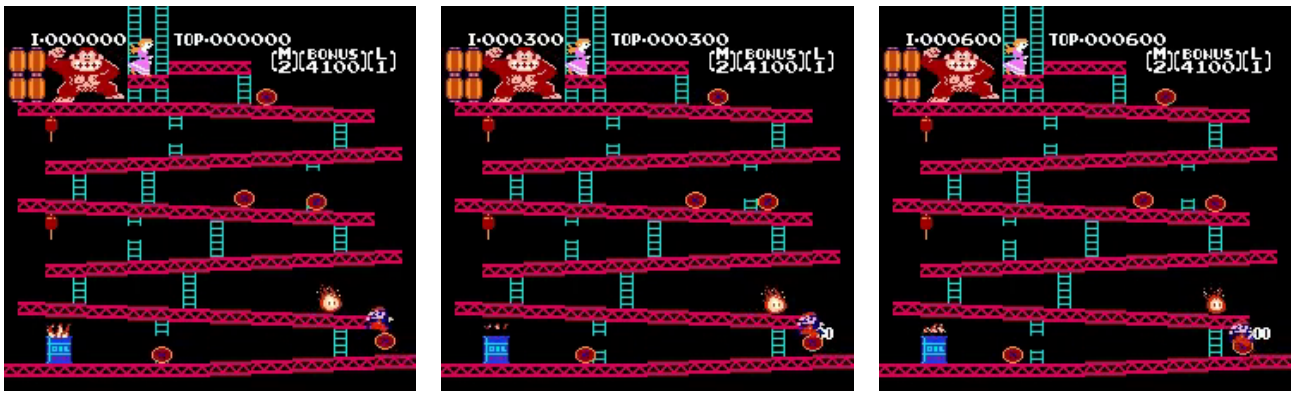


Figura 6.50: *Resultados de 30 episodios de juego en Donkey Kong.*

Durante la fase inicial de ajuste de parámetros se observaba que el agente lograba hasta 600 puntos durante un episodio. El agente al parecer descubrió un bug durante la fase de entrenamiento. Cuando el personaje salta sobre un barril normalmente consigue 100 puntos,

sin embargo, si lo hace en la esquina y en la misma dirección que el barril se dirige, se contabiliza el puntaje varias veces (Figura 6.51).



(a) (b) (c)

Figura 6.51: El agente hace que el personaje salte en la misma dirección que va el barril y consigue 600 puntos.

6.3.2. Ice Climber

Resultado 30 episodios de juego

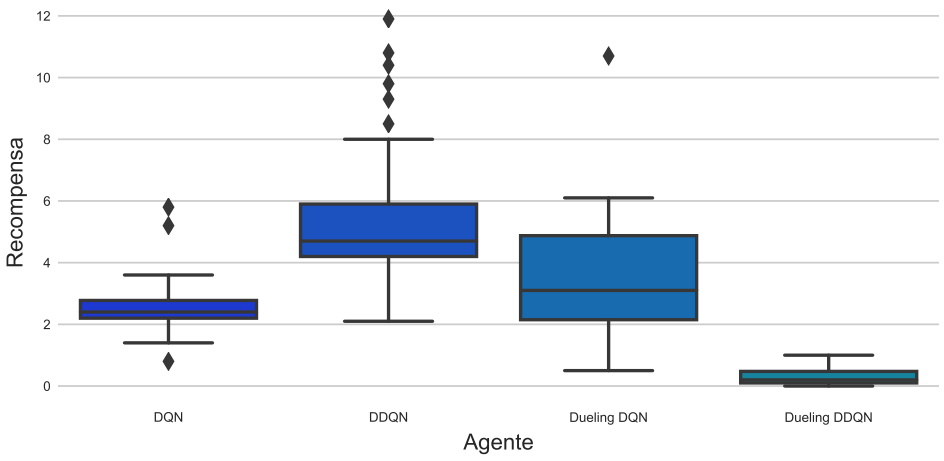


Figura 6.52: *Resultados de 30 episodios de juego en Ice Climber.*

Agente	avg	std	hi
DDQN	5.65	2.50	11.9
Dueling DQN	3.50	2.10	10.7
DQN	2.59	0.96	5.8
Dueling DDQN	0.33	0.26	1.0

Tabla 6.26: *Resultados de 30 episodios de juego en Ice Climber.*

En entrenamiento el algoritmo con mejores resultados era DQN. En juego DDQN logra una mayor recompensa pero aún está muy lejos de mostrar una política que le permita llegar hasta arriba.

6.3.3. Super Mario Bros

Agente	avg	std	hi	success
Dueling DDQN	3471.80	752.05	4074.0	80.0 %
Dueling DQN	3047.07	1029.04	4301.0	60.0 %
DQN	2604.47	1402.91	3950.0	50.0 %
DDQN	2041.03	1304.06	3880.0	33.3 %

Tabla 6.27: *Resultados de 30 episodios de juego en Super Mario Bros.*

El agente Dueling DDQN logra completar el nivel el 80 % de las veces.

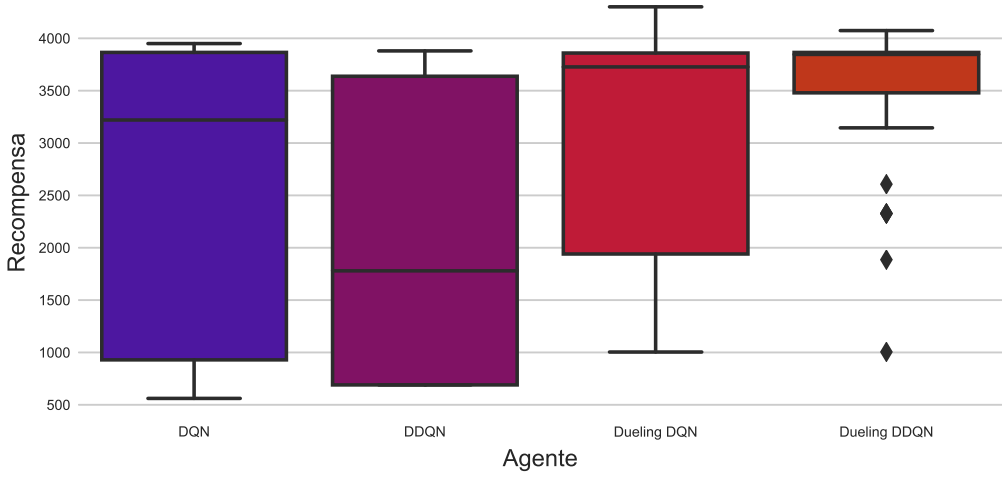


Figura 6.53: *Resultados de 30 episodios de juego en Super Mario Bros.*

En experimentos iniciales, durante un episodio el agente contaba con tres intentos (vidas). Se descubrió una técnica empleada por el agente para aumentar la recompensa (distancia recorrida). El agente recorre el escenario normalmente hasta que llega al último abismo, allí habiendo recorrido el 80 % decide caer, sacrificando una vida. El agente reaparece en el checkpoint del escenario (cerca del 40 %) y continúa hasta llegar al final. Durante el episodio con esta estrategia el agente ha recorrido aproximadamente un 140 % del escenario, se ilustra la distancia recorrida 2 veces en la figura 6.54.

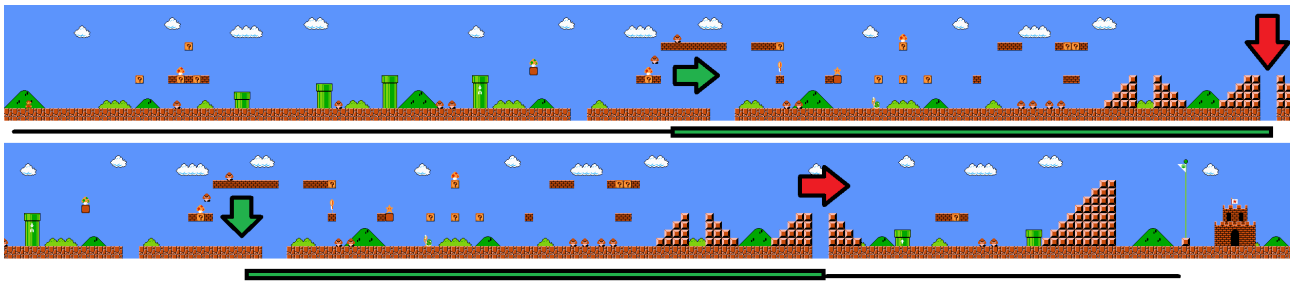


Figura 6.54: *Estrategia empleada para obtener mayor distancia recorrida en el primer nivel de Super Mario Bros.*

6.3.4. Kung Fu

Se probaron jugando los cuatro agentes entrenados en la sección anterior y el agente que usó menos bytes de la memoria RAM para entrenar (Dueling DDQN ram).

Agente	avg	std	hi	success
Dueling DDQN map	2903.27	801.18	4688.0	43.33 %
Dueling DDQN ram	2255.10	398.48	3988.0	3.33 %
Dueling DQN	2045.33	340.77	2945.0	0.00 %
DDQN	2008.07	455.09	3997.0	3.33 %
DQN	1161.33	305.00	1935.0	0.00 %

Tabla 6.28: *Resultados de 30 episodios de juego en Kung Fu.*

El agente Dueling DDQN ram no solamente fue el mejor sino que logró completar el nivel 13 de las 30 sesiones de juego.

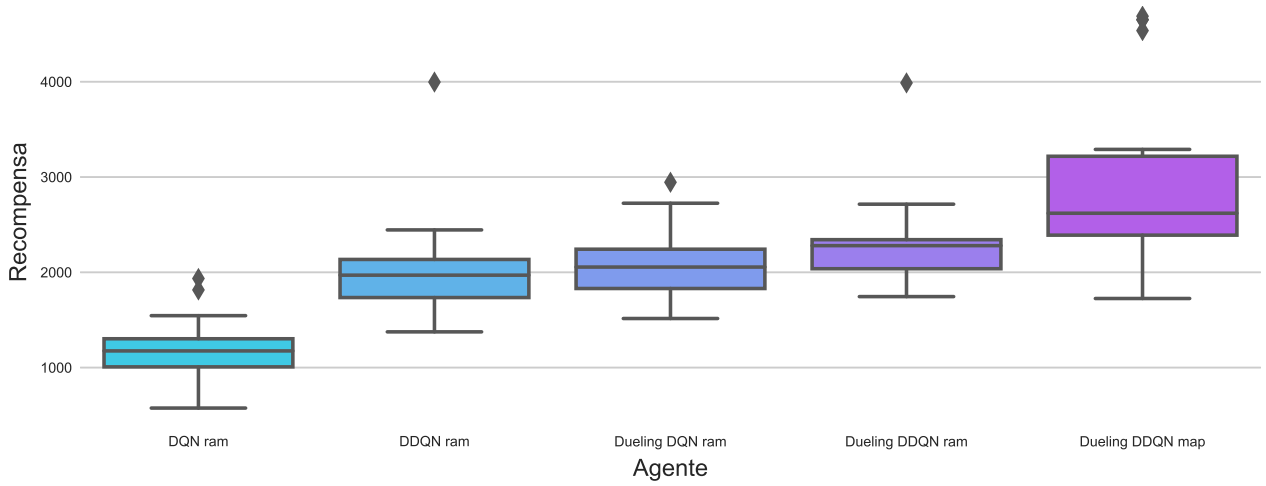


Figura 6.55: *Resultados de 30 episodios de juego en Kung Fu.*

Durante las pruebas iniciales se encontró una curiosidad. Por defecto, un golpe con una patada voladora consigue 200 puntos. Existe un truco para conseguir puntos extra, en el cual si se da una patada voladora al oponente morado número 12 no se obtienen 200 puntos sino 5000. El agente descubrió esta técnica. Figura 6.56.

(a) (b) (c)

Figura 6.56: El agente hace que el personaje salte en la misma dirección que va el barril y consigue 600 puntos.

6.3.5. Metroid

El modo map en Metroid demostró ser ineficiente en el momento de jugar, al tener objetivos cambiantes el agente parece olvidar lo primero que aprendió.

Agente	avg	std	hi
Dueling DQN ram	112.97	0.18	113.0
DQN ram	112.87	1.23	115.0
Dueling DDQN ram	112.67	0.54	113.0
DDQN ram	111.43	0.56	113.0
Dueling DQN map	49.73	56.43	112.0
DQN map	0.00	0.86	1.0
DDQN map	-0.30	0.69	1.0
Dueling DDQN map	-1.70	0.46	-1.0

Tabla 6.29: *Resultados de 30 episodios de juego en Kung Fu.*

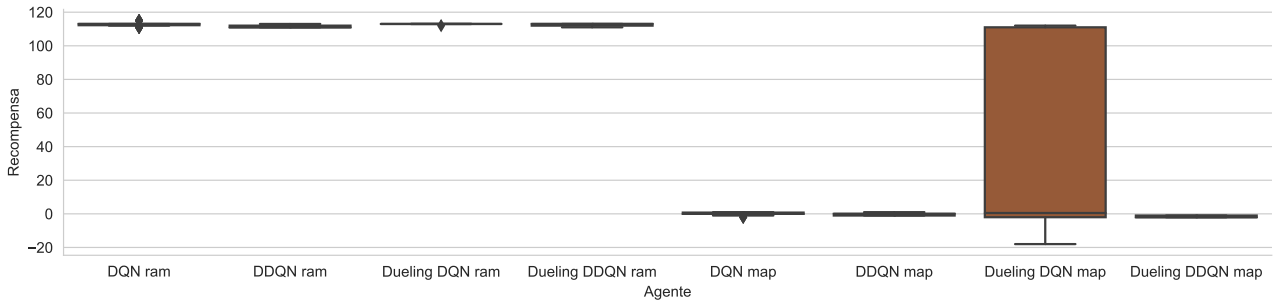


Figura 6.57: *Resultados de 30 episodios de juego en Metroid.*

Capítulo 7

Conclusiones

En este proyecto se ha construido un agente capaz de jugar cinco juegos de NES usado la información de la memoria RAM. En este documento se ha presentado un estado del arte del aprendizaje por refuerzo en vídeo juegos así como los algoritmos DQN, Doble DQN y Dueling DQN. Además se han propuesto funciones de recompensa para cuatro juegos de NES y se han desarrollado los elementos necesarios para incluir el juego Metroid en el entorno Gym Retro.

En un comienzo el proyecto solo iba a abarcar Metroid, dada la necesidad de probar la efectividad del agente se incluyeron otros juegos en los cuales el agente fue entrenado con el objetivo de terminar al menos el primer nivel. En dos de los cuatro juegos se logró el objetivo (Super Mario Bros y Kung Fu), mientras que en los otros dos se observó que los algoritmos conducen al agente en la dirección correcta y necesitarían mayor tiempo de entrenamiento.

En Metroid el algoritmo y la función de recompensa tienen una limitante, al ser un mapa relativamente pequeño (30×30), el personaje tiene un alto margen para moverse en una habitación sin que su posición de referencia cambie, esto hace que al obtener recompensa con menor frecuencia sea un problema mas complejo. El modo map en Metroid resultó ser ineficiente. Si se va a reducir las dimensiones de las observaciones es mas eficiente explorar y extraer los bytes activos.

Algo que se observó con todos los juegos fue una tendencia relativamente rápida, 500.000 pasos, para alcanzar valores altos de recompensa, de ahí en adelante si bien el agente seguía mejorando parece estancarse. Este fenómeno es relativamente común con los algoritmos DQN pero en la bibliografía consultada se observaba que tiende a ocurrir más tarde. Esto puede deberse a que la red es mas sencilla ya que no procesa pixeles.

Incluir la tasa de refresco de la animación (frame-skip) como un parámetro y experimentar permitió observar que tiene un impacto importante en el desempeño de los algoritmos y su elección se ajusta mas a la velocidad de reacción necesaria para jugar.

Un aspecto que debido a el tiempo no se pudo experimentar fue comparar el desempeño de

los Algoritmos DQN con los algoritmos de política de gradiente. Como trabajo futuro podría plantearse este objetivo así como realizar otras mejoras al algoritmo DQN como PER o HER. Por lo observado este último podría tener mejores resultados en un juego como Metroid.

Otras mejoras en la implementación pueden encaminarse a la eficiencia y velocidad. Técnicas como entrenamiento en paralelo desde diferentes checkpoints así como el aprovechamiento optimo de Pytorch y el hardware especializado de las recientes GPUs (Tensor Cores).

Bibliografía

[1] https://datacrystal.romhacking.net/wiki/Donkey_Kong:RAM_map.

[2] https://datacrystal.romhacking.net/wiki/Kung_Fu:RAM_map.

[3] https://strategywiki.org/wiki/Metroid/Walkthrough.

[4] Ice Climber Instruction Booklet.

[5] Metroid Instruction Booklet.

[6] Simón Algorta and Özgür Şimşek. The Game of Tetris in Machine Learning. 2019.

[7] Marc G Bellemare, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. Technical report, 2013.

[8] Richard Bellman. The Theory of Dynamic Programming, 1954.

[9] Richard Bellman. A Markovian Process. 1957.

[10] Nadav Bhonker, Shai Rozenberg, and Itay Hubara. Playing SNES in the Retro Learning Environment. *5th International Conference on Learning Representations, ICLR 2017 - Workshop Track Proceedings*, nov 2016.

[11] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. jun 2016.

[12] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pages 3215–3222, 2018.

[13] A. H. Klopf. *The hedonistic neuron: a theory of memory, learning, and intelligence*, volume 17. jan 1983.

[14] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael Bowling. State of the art control of atari games using shallow reinforcement learning. *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS*, pages 485–493, 2016.

[15] S. A. McLeod. Edward Thorndike - Law of Effect. Simply Psychology.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. dec 2013.

[17] 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, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[18] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. Technical report, 2016.

[19] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta Learn Fast: A New Benchmark for Generalization in RL. apr 2018.

[20] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1-2):207–219, 2000.

[21] S G Sterrett. Bringing up Turing ' s ' Child-Machine '. pages 1–10, 2012.

[22] Jakub Sygnowski and Henryk Michalewski. Learning from the memory of Atari 2600. *Communications in Computer and Information Science*, may 2016.

[23] István Szita and András Lorincz. Learning tetris using the noisy cross-entropy method. *Neural Computation*, 18(12):2936–2941, dec 2006.

[24] Gerald Tesauro. TD-Gammon: A Self-Teaching Backgammon Program. In *Applications of Neural Networks*, pages 267–285. Springer US, 1995.

[25] Christophe Thiery and Bruno Scherrer. Improvements on learning tetris with cross entropy. *ICGA Journal*, 32(1):23–33, 2009.

[26] John N. Tsitsiklis and Dimitri P. Bertsekas. Neuro-Dynamic Programming.

[27] A. M. Turing. Intelligent Machinery, 1948.

[28] A. M. Turing. Computing Machinery and Intelligence. *Mind*, 59:433–460, 1950.

[29] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-Learning. *30th AAAI Conference on Artificial Intelligence, AAAI 2016*, pages 2094–2100, 2016.

[30] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Frcitas. Dueling Network Architectures for Deep Reinforcement Learning. *33rd International Conference on Machine Learning, ICML 2016*, 4(9):2939–2947, 2016.