Proyecto de Grado
Ingeniería en Computación

Jugador de ZX Spectrum utilizando aprendizaje por refuerzos profundo

Rodrigo Laguna, Diego Melli, Ezequiel Sanchez
{rodrigo.laguna, diego.melli, ezequiel.sanchez}@fing.edu.uy

Tutores:
Mág. Diego Garat
Dr. Guillermo Moncecchi

Facultad de Ingeniería
Universidad de la República
23 de octubre de 2017

Resumen

Uno de los desafíos del Aprendizaje Automático desde sus comienzos ha sido la resolución de juegos. Los juegos proveen entornos controlados en donde es posible desarrollar y probar los resultados de distintos algoritmos. En particular, los videojuegos muchas veces presentan problemas que para su resolución requieren de secuencias complejas de acciones: tomar objetos en cierto orden, llevarlos a lugares específicos, entre otros.

Este trabajo se plantea como objetivo la creación de un jugador artificial para el videojuego Manic Miner sobre la plataforma ZX Spectrum mediante Aprendizaje profundo. Para esto se desarrollan y comparan agentes que aprenden directamente de los píxeles de la pantalla, entrenados mediante la aplicación de distintas variantes de *Deep Q Learning*: *Deep Q-Network* (DQN), *Double Deep Q-Network* (DDQN) y *Dueling Network*, combinándolas con técnicas de aprendizaje como *Human Checkpoint Replay*.

También se implementa un entorno para esta plataforma, inexistente hasta el momento, que permite su interacción con el agente y una interfaz interactiva para generar *checkpoints* y editar niveles.

A pesar de la complejidad del juego y del tiempo de entrenamiento que requieren estos algoritmos, se logra superar al primer nivel en más de una oportunidad, al asistir a la exploración del agente mediante *Human Checkpoint Replay*. Además, se analiza cualitativa mente la capacidad del agente entrenado para aplicar sus conocimientos en escenarios ligeramente distintos al de entrenamiento gracias al editor de niveles interactivo desarrollado.

Palabras clave

Aprendizaje por refuerzos profundo, Deep Q-Learning, Human Checkpoint Replay

Índice general

Capítulo 1

Introducción

Históricamente los juegos tales como las Damas [34], el Ajedrez [16], el Poker [13] y el GO [36] han servido a la inteligencia artificial como entornos de aprendizaje y pruebas controlados. Los videojuegos extienden este abanico de desafíos, presentando diferentes grados de dificultad. Muchos de ellos suelen recrear virtualmente entornos y situaciones reales. Estos juegos han sido diseñados para ser jugados por humanos, logrando un equilibrio entre su complejidad y lo atractivos o divertidos que pueden resultar a los jugadores.

Durante la última década, los investigadores han intentado que una computadora aprenda a jugar videojuegos [28, 10]. Una de las técnicas más utilizadas para entrenar un jugador artificial, al que llamaremos agente, ha sido el aprendizaje por refuerzos, empleada frecuentemente para la adquisición de comportamientos en robots [25]. Con esta técnica el agente intenta distintos movimientos, de los que no conoce a priori su resultado, y obtiene por ellos recompensas, positivas o negativas, que utiliza para aprender un comportamiento [26].

Algunos de los resultados más relevantes en resolución de videojuegos son los obtenidos por Mnih et al. [29] usando aprendizaje por refuerzos en conjunto con redes neuronales. A esta combinación de tecnologías se le denomina aprendizaje por refuerzos profundo. En particular, estos autores utilizan redes neuronales convolutivas, un tipo de red que ha tenido gran éxito en el procesamiento de imágenes [31]. Al método desarrollado se le denomina *Deep Q-Network* (DQN) y permite utilizar una misma arquitectura para aprender múltiples juegos. Pequeñas modificaciones a este método han dado surgimiento a variantes como *Double Deep Q-Network* (DDQN) [42] y Dueling [43], por lo que a veces se les denomina conjuntamente como *familia de métodos DQN*.

Gran parte del esfuerzo en la resolución de videojuegos se ha centrado sobre la consola ATARI [24, 35, 8, 43, 42]. Esto se debe principalmente al *Arcade Learning Environment* [1], comúnmente denominado ALE: una plataforma que emula la consola ATARI permitiendo la interacción del agente con el juego escogido. Sin embargo, para otras plataformas como ZX Spectrum no existen trabajos previos. Esta última plataforma es la que se utiliza en este proyecto, tomando el juego Manic Miner.

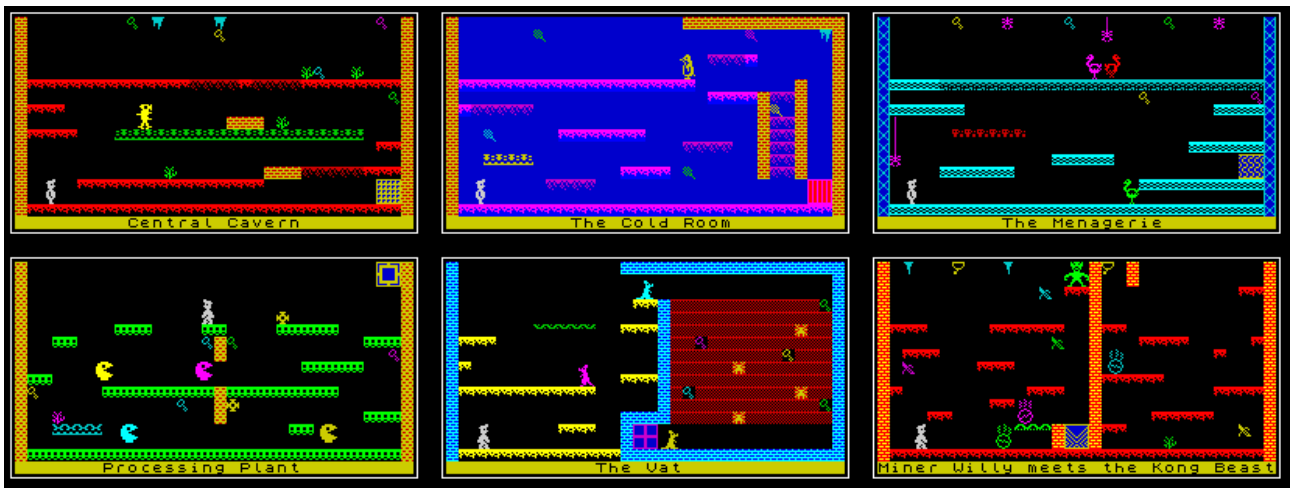


Figura 1.1: Diferentes niveles del juego. De izquierda a derecha, de arriba a abajo son los niveles 1, 2, 3, 6, 7 y 8.

1.1. Manic Miner

Manic Miner es un videojuego originalmente creado para ZX Spectrum, una computadora lanzada en la década de los 80 con un procesador de 8 bits y una cantidad entre 16 y 48 Kilobytes de RAM. Este videojuego nos pone en el papel de un minero llamado Willy, quien debe recorrer 20 cavernas en el interior de una mina.

Inicialmente se cuenta con tres vidas y una cantidad de oxígeno, que se va agotando a media que transcurre el tiempo. En cada una de las cavernas hay hasta cinco objetos parpadeantes que el jugador debe recolectar, aumentando en 100 su puntaje al tomar cada uno. Un portal comienza a titilar luego de recolectados todos los objetos, indicando que está abierto. Al alcanzarlo, el aire que no fue consumido se convierte proporcionalmente en puntos, y se pasa al siguiente nivel. En el camino se debe evadir guardias, que se mueven a lo largo de rutas predefinidas con velocidades constantes, y diversas trampas estáticas: en caso de tomar contacto con alguna de ellos, Willy muere. Si quedan vidas, se vuelve a comenzar el nivel desde el principio, sin ninguna modificación en el puntaje pero con una vida menos y la totalidad del aire. También se puede morir por asfixia o por caídas que superen determinada altura.

Existen, además, tres tipos de pisos, con comportamientos distintos: los brillantes estáticos son firmes, los brillantes con movimiento arrastran a Willy hacia la izquierda o derecha cuando está sobre ellos, y los opacos se desmoronan al ser pisados por el personaje. En la figura 1.1 pueden verse algunas de las cavernas.

Este proyecto se centra en la resolución del primer del nivel, llamado Caverna Central. Para superarlo se deben recoger cinco llaves titileantes. Las trampas en este nivel son cuatro plantas y dos picos de hielo. Hay un solo guardia situado en el centro de la pantalla que se mueve de izquierda a derecha y viceversa. El portal se ubica en la esquina inferior derecha. El piso verde brillante que se encuentra en el centro de la pantalla arrastra al personaje hacia la izquierda. En la imagen 1.2 se identifican los objetos del nivel y se propone una numeración a las llaves, que se corresponde al orden en el que un jugador humano suele recogerlas.

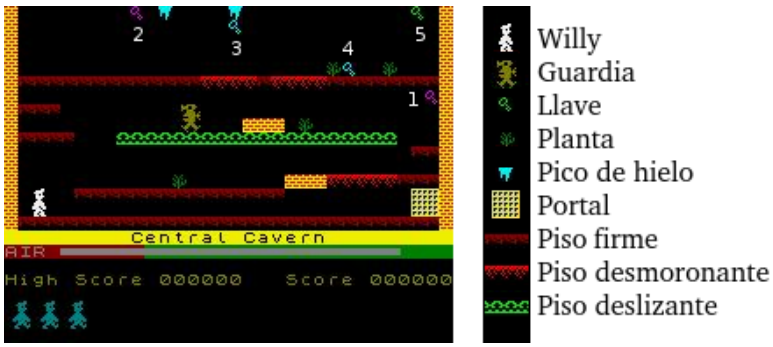


Figura 1.2: Numeración de las llaves e identificación de objetos del primer nivel.

1.2. Alcance del proyecto

Este proyecto tiene como objetivo desarrollar un jugador artificial capaz de aprender a jugar al videojuego Manic Miner de la consola ZX Spectrum.

Dado que no existe una plataforma que permita la interacción de un agente con un juego de la plataforma ZX Spectrum esta debe ser implementada para desarrollar el jugador. Para esto, se toma como punto de partida un emulador de la plataforma y una ROM que contiene el juego, y se implementan las operaciones que el Agente necesita para su interacción.

De las técnicas relevadas, en este trabajo no se desarrollan aquellas que utilizan algoritmos evolutivos como forma de aprendizaje [10], ya que la propuesta inicial requiere el uso de redes neuronales profundas. También se descartan las que toman como insumo la memoria RAM del emulador [1, 39] porque se priorizan los métodos que aprenden a partir de las imágenes generadas sin extraer características de la imagen de forma explícita [28, 43, 21].

Dentro de la familia de métodos DQN, se opta por probar *Deep Q-Network* (DQN) [28], *Double Deep Q-Network* (DDQN) [42] y *Dueling Network* [43] para el desarrollo del Agente. Para mejorar su aprendizaje se combinan estos agentes con técnicas como *Prioritized Experience Replay* (PER) [35] y *Human Cheeckpoint Replay* (HCR) [15]. Si bien dentro del objetivo inicial se encuentra la resolución de todos los niveles de Manic Miner, durante el desarrollo del trabajo se decide, teniendo en cuenta el alto tiempo de ejecución de cada prueba, priorizar la experimentación de una mayor cantidad de métodos en el primer nivel en detrimento de la exploración de todos los niveles del juego.

1.3. Estructura del informe

El resto del informe tiene la siguiente estructura: en el segundo capítulo se presenta un abordaje teórico del aprendizaje por refuerzos, redes convolutivas y técnicas específicas para la resolución de videojuegos mediante aprendizaje por refuerzos profundo. El capítulo tres expone los experimentos realizados para comparar distintos agentes y técnicas, con un análisis, tanto a nivel cuantitativo como cualitativo, de los resultados obtenidos. El cuarto capítulo presenta el ambiente en que se desarrollaron las pruebas: por un lado se detalla la adaptación que se hizo del emulador para convertirlo en un entorno capaz de interactuar con el agente, y por otro se describe la implementación del agente. Por último, el capítulo cinco consta de las conclusiones y trabajo a futuro.

Capítulo 2

Marco teórico

Para un jugador artificial, jugar videojuegos se puede resumir en observar la pantalla y, a partir de ella, tomar acciones válidas en cada instante, buscando maximizar algún criterio de avance definido. Cada acción realizada recibe una recompensa, positiva o negativa, que refleja qué tan buena o mala es la secuencia de acciones tomada hasta el momento. Aprender a jugar videojuegos puede verse como utilizar estas recompensas para aprender a elegir en cada momento la acción que maximiza el total de recompensas que se obtienen durante el juego.

El aprendizaje por refuerzos le permite a un agente, partiendo de un estado de total desconocimiento del entorno y del efecto de sus acciones, adquirir paulatinamente una estrategia de control a partir de una señal de recompensa. Aprender a controlar agentes tomando insumos de alta dimensionalidad, tales como imágenes o audios es uno de los retos de larga data del aprendizaje por refuerzos [26].

Los recientes avances en la visión por computadora y el reconocimiento de voz, se basan en el entrenamiento de redes neuronales profundas las cuales son capaces de aprender automáticamente características de alto nivel [33, 5]. Particularmente se utilizan redes con capas neuronales convolutivas ya que logran detectar características directamente de insumos de alta dimensionalidad [31], tales como las imágenes generadas por los juegos de video. Estos éxitos motivan a conectar algoritmos de aprendizaje por refuerzos con redes neuronales profundas, siendo esta combinación denominada aprendizaje por refuerzos profundo [28].

A continuación se presenta la teoría general que sustenta el aprendizaje por refuerzos en la primera sección. La segunda se dedica específicamente a la resolución de videojuegos utilizando aprendizaje por refuerzos profundo, proporcionando una introducción general a las redes neuronales convolutivas y haciendo especial hincapié en la familia de métodos *Deep Q-Network* (DQN).

2.1. Aprendizaje por refuerzos

El aprendizaje por refuerzos, afronta el desafío de un agente autónomo que percibe su entorno y actúa en consecuencia, aprendiendo a escoger acciones para lograr sus metas [26]. El agente interactúa con un entorno mediante una secuencia de observaciones, acciones y recompensas como se detalla en la figura 2.1.

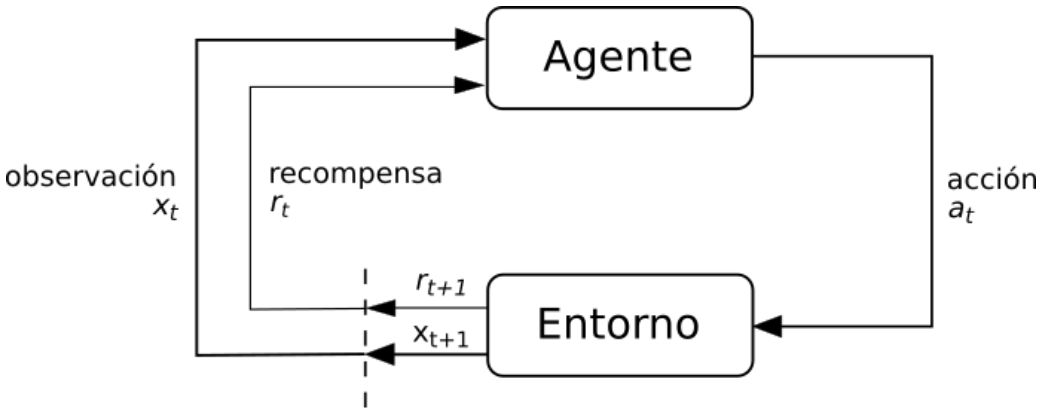


Figura 2.1: Modelo de interacción agente-entorno. Primero el agente selecciona una acción que se ejecuta en el entorno y luego como consecuencia de ella obtiene una nueva observación representativa del mismo junto a la recompensa. Imagen tomada del libro de Richard S. Sutton [38]

En cada paso de tiempo t el agente examina la observación x_t que obtiene del entorno y, con base en esta observación, selecciona una acción válida, recibiendo del entorno una recompensa r_{t+1} y una nueva observación x_{t+1}. La recompensa representa qué tan bien se desempeñó el agente específicamente en el paso t. En el contexto de un videojuego, la observación puede ser una matriz de píxeles que representan la imagen de la pantalla actual o la memoria RAM de la consola. La recompensa se puede corresponder con la diferencia entre el puntaje anterior y posterior a la acción efectuada.

La tarea del agente consiste en aprender una política π que le indique cómo actuar en cada momento con el objetivo de maximizar la suma de la recompensa obtenida en el juego. El estado s_t es una representación de la realidad que hace el agente, que condensa la información relevante del entorno en un determinado momento t. Usualmente s_t es igual a la observación x_t, aunque, como se ve más adelante, también puede ser representado de otras formas. Esta política no es más que una función π que, para cada estado s_t, le asigna la acción a tomar a_t, es decir, $\pi(s_t) = a_t$.

La recompensa obtenida en cada paso no es suficiente para determinar qué tan buena fue la acción tomada. Reconocer la influencia de una determinada acción en el desenlace final (perder o ganar) no es fácil ya que en algunos casos recién puede ser apreciada después de transcurridos cientos de pasos: una secuencia inicial de movimientos buenos puede llevar a una recompensa negativa si luego se toman malas decisiones, o viceversa. A este proceso se lo conoce como asignación de crédito [26].

Se define el valor acumulado o retorno partiendo de un estado inicial s como la suma descontada de recompensas obtenidas siguiendo una política arbitraria π:

$$V^{\pi}(s) = \mathsf{r}_1 + \gamma^1 \mathsf{r}_2 + \ldots + \gamma^{k-1} \mathsf{r}_t = \sum_{k=0}^{\infty} \gamma^k \mathsf{r}_{k+1}$$

La constante γ, entre 0 y 1 es usada para determinar el peso de las recompensas inmediatas contra las obtenidas a largo plazo. Cuanto más cercana a 1 es la constante, más énfasis cobran las recompensas lejanas. Se desea encontrar una política que obtenga el mayor retorno a largo plazo para todos los estados, denominada política óptima π^*.

¿Cómo puede un agente aprender una política óptima para un entorno arbitrario? Para lograr

este objetivo se define la función de acción-valor $Q(s, a)$ de modo que su valor corresponda al retorno máximo acumulado, que se logra partiendo del estado s, aplicando la acción a y siguiendo una política óptima a partir del siguiente estado s'.

$$Q(s, a) = r(s, a) + \gamma * V^*(s')$$

Conociendo la función Q podemos deducir la política óptima, tomando la acción que maximiza esta función en cada estado. El problema radica ahora en encontrar una forma de estimar los valores para Q, dada sólo una secuencia de recompensas inmediatas r_t distribuidas a lo largo del tiempo. Este objetivo suele lograrse mediante una aproximación iterativa. La función V puede ser definida a partir de Q, permitiendo reescribir Q en forma recursiva.

$$V^*(s) = max_{a \in Acciones} Q(s, a)$$

$$Q(s, a) = r(s, a) + \gamma * max_{a' \in Acciones} Q(s', a')$$

Esta definición recursiva proporciona la base para algoritmos que aproximan Q iterativamente, estimandola con una tabla de búsqueda mediante programación dinámica. En cada entrada de la tabla se almacena el valor actual estimado para la función Q, denominado $\hat{Q}$. Inicialmente la tabla puede ser llenada con valores aleatorios o ceros. El agente observa repetidamente su estado actual, elige y ejecuta una acción, observa la recompensa resultante y su nuevo estado. Luego actualiza la entrada de la tabla correspondiente siguiendo la fórmula:

$$\hat{Q}(s, a) = r + \gamma * max_{a' \in Acciones} \hat{Q}(s', a')$$

Con suficiente entrenamiento, y asegurando visitar repetidamente todos los estados, la información se propaga a lo largo de la tabla, convergiendo $\hat{Q}$ a Q [26]. El agente influye directamente en la distribución de ejemplos de entrenamiento con la secuencia de acciones tomadas, por lo que es el quien define qué estados se visitan durante el aprendizaje, dando lugar al dilema de la exploración - explotación que se presenta en la sección 2.1.2.

La representación mediante una tabla es factible solamente cuando el problema tiene un pequeño espacio de estados. En caso de problemas no triviales, se debe usar una función de aproximación para generalizar: redes neuronales, funciones de aproximación según el vecino más cercano, o regresión ponderada local [25].

Se sabe que el aprendizaje por refuerzo es inestable o incluso diverge cuando se usa una función no lineal, como una red neuronal, para aproximar Q. Esta inestabilidad se explica por varias causas como lo son la fuerte correlación entre los elementos de la secuencia de observaciones y que pequeñas actualizaciones en Q pueden cambiar significativamente la política y por tanto la distribución [7]. Para mitigar estos efectos sobre la función de aprendizaje no lineal, Lin y Long-Ji [25] proponen el uso de un mecanismo llamado *experience replay*, detallado en la sección 2.1.3, junto a una actualización tardía de la función de aproximación Q. Además se acota el valor de la función de recompensa en pro de la estabilidad.

2.1.1. Ejemplo ilustrativo

Para mostrar el aprendizaje de la función Q, supongamos que se cuenta con un agente capaz de moverse en un entorno con forma de grilla, tomando acciones: izquierda, derecha, arriba y abajo. La función de recompensa inmediata es definida por el entorno: devuelve 0 en todas las transiciones salvo las que llegan al estado que contiene la llave o la planta, retornando 100 y -100 respectivamente. Estos estados son absorbentes, por lo que al alcanzarlos termina el episodio y se vuelve a comenzar en el estado inicial s_1. Consideremos además la aproximación $\hat{Q}$ de la función Q en forma de tabla. En la figura 2.2 se muestra cómo evoluciona esta función a lo largo del entrenamiento.

Con todos los valores $\hat{Q}$ inicializados en cero, el agente sólo realiza cambios en las entradas de la tabla cuando llega a un estado final. Recibe una recompensa distinta de cero y actualiza el valor $\hat{Q}$ para la transición que conduce al estado final. En el siguiente episodio, si el agente pasa por este estado, adyacente al estado final, su valor $\hat{Q}$, ahora distinto de cero, permite actualizar el valor de las transiciones que se encuentran a dos pasos de distancia del estado final. Con suficientes episodios de entrenamiento, la información se propaga de las transiciones con recompensa distinta de cero a lo largo de todo el espacio estado-acción disponible para el agente, resultando en una tabla $\hat{Q}$ con los valores de la figura 2.3.

2.1.2. Exploración vs Explotación

En el aprendizaje por refuerzos, existen dos estrategias principales con las que el agente toma sus decisiones durante la etapa de entrenamiento. Cuando utiliza el conocimiento que ha adquirido para tomar la mejor decisión conocida y obtener una mayor recompensa, se dice que sigue una estrategia de explotación. Por otra parte, dado que inicialmente el agente desconoce todos los estados y transiciones posibles, la exploración es necesaria; esta estrategia consiste en optar por estados que el agente aún desconoce para recabar información sobre ellos: durante la exploración el agente opta por sacrificar recompensa inmediata conocida en pos de mejorar la recompensa a largo plazo [37, 32]. Un balance correcto entre exploración y explotación es crucial a la hora de aprender cualquier tarea mediante ensayo y error, sin importar de qué tipo de agente se trate [37].

Para problemas con espacios complejos como son los videojuegos, aún hoy existen grandes desafíos para explorar eficientemente el entorno, siendo esta una rama de investigación activa. Existen juegos como *Montezuma's Revenge*, para los que debido a su complejidad, solo recientemente se han desarrollado agentes que juegan a niveles humanos [2, 14].

A continuación se analizan algunos de los ejemplos clásicos de estrategias de exploración.

Optimismo de cara a la incertidumbre

En esta estrategia se asume que aún no se ha encontrado la mejor política, por lo que es conveniente visitar los estados desconocidos. Esto se traduce en un enfoque optimista: lo mejor está por venir. Se inicializa al agente de manera que crea que los estados no visitados son razonablemente buenos, asegurando que todos los estados se recorren al menos una vez. Es sumamente útil cuando la cantidad de estados es pequeña [32], pero en videojuegos, donde la cantidad de estados es muy grande, produce un efecto de exploración perpetua [8].

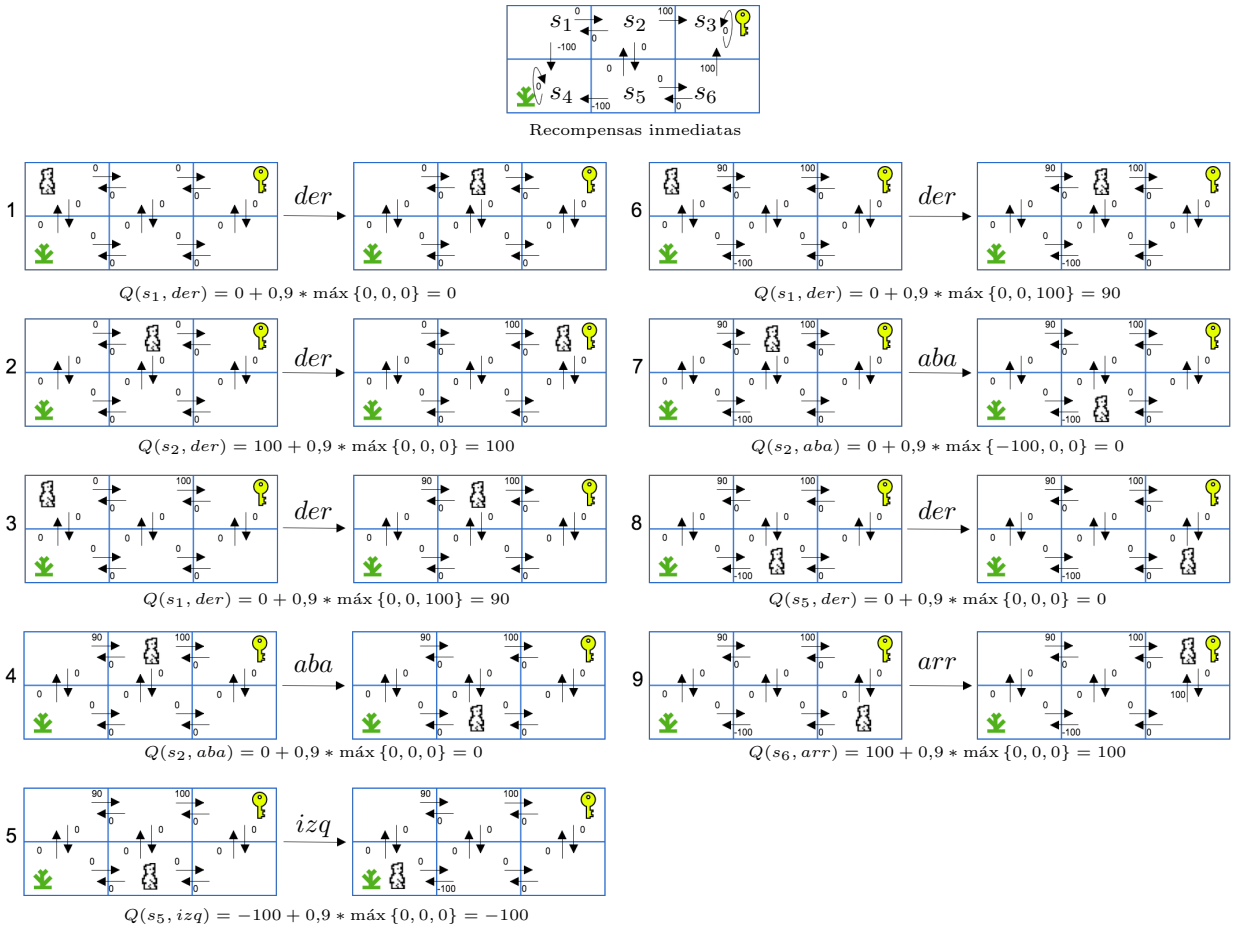
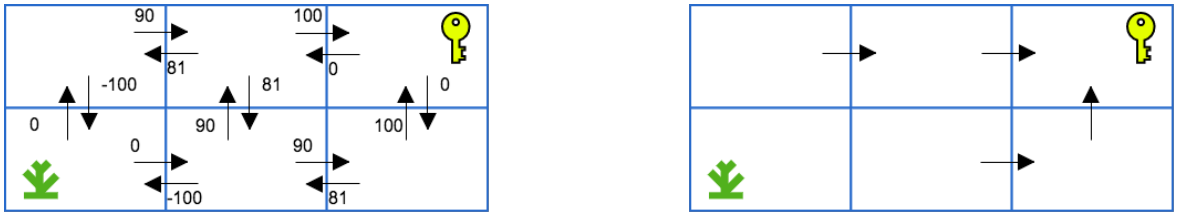


Figura 2.2: Ejemplo simple para ilustrar los conceptos básicos de aprendizaje por refuerzos. Cada entrada de la tabla representa un estado distinto y cada flecha una acción. Se muestra la función de recompensa inmediata y los cambios en la tabla $\hat{Q}$ producidos por cada una de las acciones usando un factor de descuento de 0,9.



Valores $Q(s, a)$ Una política optima

Figura 2.3: Se muestra la tabla de la función $\hat{Q}$ luego de converger y una política óptima, correspondiente a las acciones con máximo valor $\hat{Q}$.

Épsilon-*greedy*

Épsilon-*greedy* es una estrategia en la que, en cada paso se toma una acción aleatoria con una probabilidad ε y en caso contrario se toma la mejor opción (*greedy*). El parámetro ε puede ser fijo durante todo el proceso de aprendizaje o puede ir decreciendo con el tiempo hasta alcanzar cierto umbral prefijado, generalmente de forma lineal con el número de pasos.

Defazio [8] reporta que este método aplicado a videojuegos tiene problemas para tratar con el gran espacio de estados. Cada acción modifica el estado del sistema mínimamente, resultando en muy poca exploración. Además, para juegos de alta complejidad, se requieren secuencias de acciones específicas para que la exploración pueda llegar a nuevos estados que no son fácilmente accesibles.

Pese a sus falencias conocidas, es ampliamente utilizado en la resolución de videjuegos debido a su simpleza, aunque en varios de ellos se obtuvieron como resultado pobres políticas de exploración [29, 28, 24].

Política *softmax*

Un problema con la estrategia de exploración épsilon-*greedy* es que durante la exploración todas las acciones tienen la misma probabilidad de ser elegidas. Esto implica que la peor de ellas tiene la misma probabilidad de ser seleccionada que una óptima. En tareas donde la peor acción tiene resultados muy adversos, este comportamiento es insatisfactorio.

Para mitigar este problema, la estrategia *softmax* propone asociar las acciones a una probabilidad de manera tal que las mejores acciones tengan mayor chance de ser seleccionadas. Para ello se transforman los Q-valores asociados a cada acción en probabilidades mediante la función *softmax*:

$$P(s,a) = \frac{\exp(\frac{Q(s,a)}{\tau})}{\sum_{i=1}^{i=n} \exp(\frac{Q(s,i)}{\tau})}$$

Este enfoque evita tener que reducir el parámetro de exploración, pero requiere un escalar positivo, τ, llamado temperatura. Cuando este parámetro es grande, todas las acciones son prácticamente equiprobables; sin embargo, cuando se acerca a cero, la política *softmax* se convierte en una política *greedy*. Una alternativa posible es que la temperatura se modifique dinámicamente, comenzando con valores altos para favorecer la exploración inicial y que luego vaya disminuyéndo con el número de pasos para aumentar la explotación. Este parámetro debe ser elegido específicamente para cada experimento ya que depende fuertemente de la naturaleza del problema. Bellmare et al. [1] reportan esto como impracticable, principalmente por la alta sensibilidad a la temperatura.

Long-Ji Lin [25] utiliza este enfoque de temperatura dinámica en el aprendizaje de un robot que explora el espacio en que se mueve. Ji He et al. [12] utilizan esta estrategia en la resolución de juegos cuyo estado y espacio de acciones están representados en lenguaje natural.

2.1.3. *Experience replay*

Una experiencia se define como la tupla $e = (s, a, r, s')$, donde s representa el estado actual, a la acción, r la recompensa obtenida por dicha acción y s' es el estado siguiente. Se puede ver que las experiencias modelan las transiciones entre los estados al realizar cierta acción.

En el aprendizaje por refuerzos tradicional se utiliza el aprendizaje en línea: cada experiencia se usa para entrenar una única vez al momento que se genera y luego se descarta. La principal ventaja de este método es que no requiere memoria para almacenar las experiencias. Sin embargo, cuando se aprende en línea, el conocimiento actual del agente determina directamente, mediante las acciones que realiza, los estados que serán visitados y sobre los que el agente será entrenado. Por ejemplo, si la acción que maximiza la recompensa es mover a la izquierda, luego los ejemplos de entrenamiento serán dominados por los del lado izquierdo, pudiendo ajustarse a un subconjunto de estados, estancándose en un mínimo local pobre, o incluso llegar a divergir. Los algoritmos *Q-learning* son ineficientes cuando se entrena de esta forma [25]. Utilizar experiencias una sola vez es un derroche, ya que algunas pueden ser raras o costosas de obtener, tales como aquellas que involucran daños materiales o estados difícilmente accesibles. Además del costo que tiene generar cada experiencia, su uso inmediato para el entrenamiento crea una alta correlación entre los ejemplos ya que el estado del entorno cambia muy poco entre pasos sucesivos.

En contraposición, la técnica de *Experience replay* consiste en almacenar las experiencias para luego seleccionar un conjunto de ellas de forma aleatoria sobre el cual entrenar. Seleccionar aleatoriamente los ejemplos rompe la correlación entre experiencias sucesivas y reduce la varianza de las actualizaciones [7]. Usando *Experience Replay*, la distribución del comportamiento se promedia sobre muchos de los estados previos, suavizando el aprendizaje, evitando oscilaciones o divergencia de los parámetros y acelerando el proceso de propagación de asignación de créditos [25]. Además, el agente tiene la oportunidad de refrescar lo que ha aprendido antes: durante el entrenamiento, si un patrón de entrada no se ha presentado por mucho tiempo, la red normalmente olvida lo que ha aprendido para ese patrón y por lo tanto tiene que volver a aprenderlo cuando este se vea nuevamente más adelante, si es que se ve. A este problema se le denomina *re-learning*.

Una condición para que la *experience replay* sea útil, es que las leyes que gobiernan el entorno del agente no cambien con el tiempo; si las leyes han cambiado, las experiencias pasadas pueden ser irrelevantes o incluso engañosas [25].

Para poblar inicialmente la memoria de *replay* se ejecuta una fase de *warmup* o calentamiento. Esta consiste en ejecutar cierta cantidad de acciones aleatorias hasta tener almacenadas la cantidad de experiencias deseadas, por una única vez antes de comenzar el entrenamiento.

A continuación se presentan algunas variantes de *experience replay* y la forma en que se seleccionan las experiencias para entrenar. Cuando se aplica una política que prioriza algunas experiencas sobre otras, se dice que es un método PER, aludiendo a sus siglas en inglés *Prioritized Experience Replay*. Ninguno de los dos métodos que se presentan determina un criterio para reemplazar las experiencias cuando se alcanza su límite de almacenamiento. Como opción más simple, se sigue la estrategia de remover las experiencias más antiguas para darle espacio a las más nuevas. Schaul et al. [35] indican que la elección de una buena política de reemplazo es importante, aunque no profundizan en ello porque enfocan su trabajo en la selección de las experiencias a utilizar. Tim de Bruin et al. [7] plantean que mantener las experiencias iniciales siempre en memoria puede contribuir a la estabilidad del aprendizaje cuando los espacios son

pequeños. Sin embargo advierten que mantener las experiencias demasiado tiempo en memoria puede inducir al agente a sobreajustarse a ellas.

Prioridad uniforme

En esta variante, a cada experiencia almacenada en memoria se le asigna la misma probabilidad de ser escogida para entrenar. Es ampliamente utilizada debido a su sencillez. Fue propuesta por Lin y Long-Ji [25] hace más de una década. Este enfoque implica que los ejemplos que no tienen mucho que enseñar tengan la misma prioridad para ser seleccionados que un ejemplo que si lo tiene.

Priorized Experience Replay: PER

En este enfoque se intenta dar mayor prioridad a los ejemplos más útiles, tratando de repetir aquellos que tengan más para enseñar.

Suponiendo que se cuenta con una medida que indica la cantidad que el agente puede aprender de una experiencia, convendría priorizar las que tengan este valor más elevado. Dado que esta medida no es directamente accesible, una aproximación razonable es utilizar el error TD, tal como se define a continuación.

$$errorTD(s) = r + \gamma V(s') - V(s)$$

Donde s es el estado actual, s' es el siguiente estado y r es la recompensa obtenida por pasar de s a s'. Cuando V converge, el error es cero. Este error mide la diferencia entre dos estimaciones de la función V: una partiendo del estado actual, y otra partiendo del siguiente estado. Si bien ambas son la misma función, estimarla sobre el siguiente estado es más preciso ya que la recompensa obtenida es un dato real. Aquellas experiencias que tienen un error TD grande, son las que aún no se aproximan de forma correcta, por lo que se les asigna una probabilidad mayor de ser seleccionadas para entrenar. Esto es análogo a lo que hace un estudiante que dedica mayor tiempo de estudio a los temas que le son más difíciles de comprender.

El enfoque es presentado por Schaul et al. [35], reportando mejores resultados que trabajos previos [29] y alcanzándolos con un menor tiempo de entrenamiento. Schaul et al. también plantean que esta técnica se puede utilizar en problemas de aprendizaje supervisado, sobre todo cuando se cuenta con un conjunto de datos desbalanceado [35].

Inicialmente al ingresar una experiencia en la memoria se desconoce el error TD, por lo que se le asigna un valor máximo para asegurar que se ejecuten al menos una vez. Por otro lado, cada vez que el agente entrena con una experiencia, el error TD asociado cambia y se lo actualiza en la memoria.

La forma más sencilla de definir la prioridad de una experiencia en función del error TD es priorizar siempre las que presentan el mayor error de todas las almacenadas: con este enfoque, las prioridades son deterministas. Esto tiene la desventaja de que las experiencias que inicialmente poseen un error TD bajo pueden no ser repetidos nunca. También es muy sensible al ruido cuando las recompensas son estocásticas o a los mismos errores de estimación que incurre la aproximación del error TD. Además de esto, al tratarse de un enfoque *greedy*,

los estados de mayor error TD se repetirán con mayor frecuencia, llevando inevitablemente a sobreajustar.

Otra posibilidad es utilizar prioridades estocásticas: Schaul et al. [35] proponen un muestreo estocástico que interpole entre una priorización completamente *greedy* y una totalmente estocástica. El método asegura que la probabilidad de que una experiencia sea elegida es monótona en la prioridad, al tiempo que garantiza probabilidades no nulas para todas las experiencias.

Se define la probabilidad de muestrear la experiencia i como:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

donde $p_i > 0$ es la prioridad de la experiencia i. El exponente α determina cuánta priorización se utiliza. Puede observarse que α igual a cero corresponde al caso uniforme. A su vez, para definir estas prioridades se consideran dos variantes: la primera es directa, consiste en una priorización proporcional donde $p_i = |\delta_i| + \varepsilon$ y ε es una pequeña constante positiva que evita el caso borde de las experiencias que tengan error cero. La segunda es indirecta, consiste en una priorización basada en rangos donde: $p_i = \frac{1}{rango(i)}$ y $rango(i)$ es el la posición de la transición i cuando la memoria de repetición se ordena decrecientemente por $|\delta_i|$.

Los autores no reportan ventajas significativas entre estas alternativas para definir la priorización. [35].

2.2. Resolución de videojuegos

En esta sección se presentan algunos de los métodos aplicados a la resolución de videojuegos, introduciendo conceptos básicos y técnicas propias de este dominio. Los métodos de aprendizaje por refuerzos profundo son una variante del aprendizaje por refuerzos con la peculiaridad de que utilizan redes neuronales profundas para aproximar la función Q, aprovechando su poder para procesar imágenes y computar el valor de la función con una única red.

Como insumo para un aprendiz de videojuegos existen dos grandes vertientes: utilizar la imagen generada por el emulador [29, 28, 42, 24, 3, 15] o su memoria RAM [1, 39]. A su vez, se puede dividir a los primeros, entre los que realizan algún tipo de preprocesamiento en búsqueda de características [24, 1, 40] y los que usan las imagenes originales [28, 29, 42, 39]. Debido a que se busca un agente que aprenda en base a insumos similares con los que aprende un jugador humano, se elige únicamente presentar la última aproximación, introduciéndose técnicas para el tratamiento de las imágenes y cómo se representa un estado a partir de ellas.

Entre los métodos de aprendizaje por refuerzos profundo se decide hacer especial hincapié en el método *Deep Q-Network* desarrollado por Mnih et al [28]. Este es concebido específicamente para la resolución de videojuegos, constituyendo la primera experiencia exitosa con un nivel de generalidad tal que permite utilizar una misma arquitectura de red e hiperparámetros para entrenar sobre una gran variedad de juegos.

En algunos videojuegos en los que la complejidad es alta y la recompensa dispersa, como es

el caso de *Montezuma's revenge*[1] o *Private eye*[2], recientemente se ha logrado que un aprendiz artificial pueda desempeñarse mejor que un agente aleatorio [14, 2][3]. Un jugador humano cuenta con un modelo previo de los objetos presentes en el juego con una semántica asociada a ellos, mientras que un aprendiz artificial carece de estos conocimientos a priori. Esto le permite a un jugador humano que se enfrenta por primera vez a determinado juego intuir que una escalera sirve para subir o bajar, o que tomar un arma le da la habilidad de disparar, de manera que puede comenzar por probar estos movimientos en vez de intentarlos todos. Esto determina que, como el agente no obtiene fácilmente ninguna recompensa positiva que guíe sus acciones, cualquier acción puede potencialmente conducir a una recompensa mayor, derivando en un comportamiento aleatorio [15]. Debido a la dificultad de exploración de algunos juegos, se introduce la técnica de *Human Checkpoint Replay* que asiste al agente en la exploración.

2.2.1. Manipulación de la recompensa

Definir una función de recompensa adecuada es fundamental para resolver cualquier tarea mediante aprendizaje por refuerzos [26]. Esta función es en gran medida la que va a determinar el comportamiento del agente, por lo que definirla de forma incorrecta puede llevar a resultados inesperados [6]. No siempre es sencilla su definición para todas las tareas: por ejemplo si se busca enseñar un robot a caminar, no está claro a priori qué acciones deben ser premiadas o castigadas. Para estos problemas donde resulta difícil definir explícitamente una función de recompensa, existen enfoques que buscan aprender esta función siguiendo criterios humanos paralelamente al entrenamiento: el agente le presenta a un humano dos alternativas en las que está aprendiendo a resolver la tarea, y el humano le indica cuál se asemeja más a la correcta según su criterio. Esta elección del humano se utiliza luego para redefinir la función de recompensa [6].

Tradicionalmente en la resolución de videojuegos la recompensa de una acción queda determinada por la diferencia entre el puntaje total previo y posterior a realizarla [1], aunque existen trabajos donde además se asigna recompensa para beneficiar la exploración [21] o para modificar conductas que no se ven reflejadas en el puntaje [6], como puede ser perder una vida. Esta recompensa es calculada por el entorno, aunque en ocasiones se la transforma antes de ser enviada al agente para alentar o desalentar diferentes conductas.

La naturaleza heterogénea de la recompensa en un mismo juego, donde pueden haber acciones que generen recompensas con diferencias significativas en cuanto a su magnitud o no estar acotadas, genera problemas de estabilidad para métodos de aprendizaje por refuerzos profundo [41]. Este problema se agudiza cuando se pretende una solución única para entrenar sobre varios videojuegos, ya que las escalas de recompensas entre distintos juegos serán aún más diversas [28, 41].

Para evitar estos problemas, Mnih et al. [29] plantean acotar la recompensa dentro del intervalo $[-1, 1]$, de manera que todas tengan el mismo orden de magnitud. Dado que la recompensa está definida por la diferencia del puntaje, y que su valor absoluto es mayor a uno o es cero, en la

[1] Es uno de los primeros juegos que combina búsqueda de tesoros, múltiples cuartos y resolución de puzles. El jugador controla un personaje que se mueve de un cuarto a otro dentro de un laberinto subterráneo.

[2] En este juego el jugador controla a un detective que viaja en un auto por la ciudad, atrapando ladrones que luego debe llevar a la carcel.

[3] Estos son trabajos publicados luego de avanzado el proyecto por lo que están fuera de alcance. Para obtener estos puntajes Hester et al. [14] comienzan su entrenamiento utilizando experiencias humanas, y Bellmare et al. [2] utilizan una estrategia de exploración que lleva una noción de estados visitados para hacerla más eficiente.

práctica la recompensa termina siendo 0, 1 o -1. Como consecuencia inmediata, dos acciones distintas con recompensas positivas son igual de buenas ante los ojos del agente, lo que no siempre es deseable.

Hado Van Hasselt et al. [41] proponen una técnica llamada POP-ART, en donde se plantea aprender otra función para reescalar linealmente los objetivos de manera que sea viable utilizar distintos valores de recompensas, manteniendo la estabilidad. Esto le permite al agente discriminar entre dos acciones buenas cuál es mejor (análogamente para las malas). Los juegos donde se explotan más estos beneficios, son aquellos en que la recompensa puede variar mucho de una acción a otra, como en *Pac-Man*, donde comer un fantasma tiene una recompensa bastante mayor que comer una bolita. En otros juegos en que las recompensas son naturalmente muy similares, como en *Pong*[4], la estrategia de acotar la recompensa resulta mejor.

2.2.2. Preprocesamiento de la observación

Trabajar directamente con los cuadros sin procesar generados por el emulador puede ser computacionalmente muy exigente. Para acelerar el entrenamiento se suele aplicar un preprocesamiento básico destinado a reducir la dimensionalidad de las imágenes, a diferencia de otros enfoques, donde la finalidad del procesamiento de la imagen es reconocer objetos o extraer características [10, 24]. Por ejemplo, en el trabajo de Mnih et al. [29], realizado sobre juegos de la plataforma ATARI, las imágenes generadas de 256×192 píxeles en RGB se convierten a escala de grises y luego se realiza un reescalado de las imágenes a 84×84. Este preprocesamiento representa una disminución del número de parámetros de entrada a sólo un 5 % de los originales.

También toman el máximo entre el valor del píxel actual y el píxel del cuadro anterior para remover el cambio de colores, o "titileo", que tienen muchos de los juegos de *Arcade* para hacerlo visualmente más atractivo.

2.2.3. Representación de estados

Si el agente observa únicamente la imagen actual de la pantalla, no es posible que comprenda completamente la situación. Tomemos por ejemplo el juego *Breakout*: con una sola imagen no se puede saber siquiera para dónde se mueve la pelota. Para solucionar esto, se consideran secuencias de acciones y observaciones $z_t = x_1, a_1, x_2, a_2, \ldots, a_{t-1}, x_t$ para aprender a partir de ellas. Se asume que todas las secuencias en el emulador terminan en un número finito de pasos de tiempo. Con este formalismo se da lugar a un proceso de decisión Markoviano, extenso pero finito, en el que cada secuencia corresponde a un estado distinto del que se puede deducir el siguiente estado al realizar cierta acción. Como resultado, es posible aplicar métodos de aprendizaje por refuerzo estándar, simplemente usando la secuencia completa z_t como la representación del estado en el tiempo t.

Debido a las dificultades que representa el uso de entradas de tamaño variable para una red neuronal, se utiliza una representación simplificada de largo fijo. Esta aproximación consiste en truncar la secuencia z_t tomando únicamente los últimos m cuadros, que denominaremos s_t. En el caso particular en que m es 1 volvemos al escenario inicial en que sólo se contempla

[4]Este juego de ATARI consiste en mover un barra vertical ubicada a un lado de la pantalla para impedir que una pelota llegue a este lado, al tiempo que se intenta que la pelota llegue al otro lado, que es defendido por otra barra controlada por la consola.

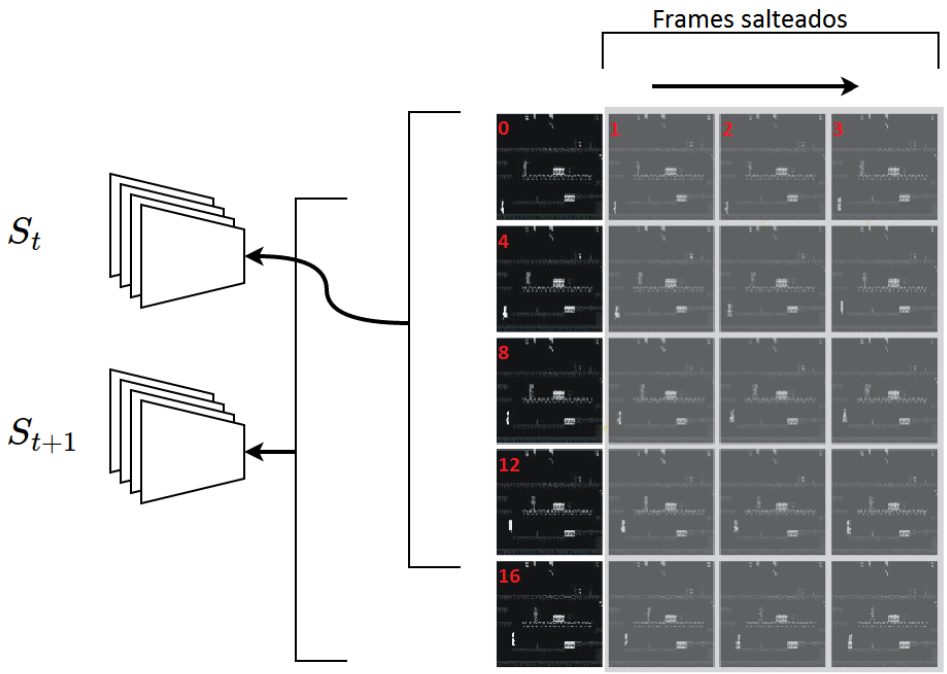


Figura 2.4: Los cuadros opacos no forman parte de la entrada para el agente. Se ejemplifican dos estados consecutivos; se observa que entre un estado y el siguiente hay una superposición de $m - 1$ cuadros.

la observación actual, por lo que se toma un m mayor. En términos prácticos, si por ejemplo se considera el juego *Pong*, dado un único cuadro sólo podemos observar la posición de la pelota, pero no hacia dónde se dirige. Accediendo a los dos últimos cuadros podemos deducir su velocidad y dirección. Observando los últimos tres cuadros se puede deducir la aceleración de la pelota. Mnih et al. utilizan los últimos cuatro cuadros, aunque puede ser otro valor, como Hausknecht y Stone [11] que utilizan los últimos díez.

2.2.4. Salteo de cuadros

El salteo de cuadros es una técnica ampliamente utilizada en la resolución de videojuegos que busca acelerar el aprendizaje introduciendo una mayor diferencia entre dos cuadros consecutivos. Una vez que el agente obtiene una observación y selecciona una acción, esta es repetida durante los siguientes k cuadros. Esto tiene dos efectos: por un lado, acelera las ejecuciones ya que se toma una decisión cada k cuadros, y generalmente simular un paso en el entorno es menos costoso que decidir [29]. Por otro, se asemeja más a la forma en que juega un humano, ya que es difícil que este pueda interactuar con la consola en cada uno de los cuadros. Su valor óptimo depende de cada juego, existiendo enfoques que ajustan dinámicamente este parámetro durante el juego o intentan aprender un valor óptimo para un juego particular [3].

Se hace notar que este parámetro influye directamente en el estado que recibe el aprendiz. En la figura 2.4 se observa la ejecución con salteo de cuadros igual a 4, en donde las imágenes opacas son descartadas, y se toman los cuadros 0, 4, 8 y 12 para formar el estado s_t y los cuadros 4, 8, 12 y 16 para constituir el estado s_{t+1}.

2.2.5. Formas de evaluación

Muchas veces es difícil evaluar a un agente, sobre todo cuando se utilizan redes neuronales. Las principales dificultades están dadas por el uso de *Deep learning* que, al disponer de tantas neuronas, permite al algoritmo memorizar una secuencia de acciones sin necesidad de generalizar. Esto se agudiza cuando el estado inicial es siempre el mismo y el juego es determinista, como en el problema planteado.

Otra parte de la dificultad es intrínseca al aprendizaje por refuerzos. En otras tareas como las de clasificación, el conjunto de datos se separa en entrenamiento, validación y testeo, dejando sus roles bien definidos. Sin embargo en el aprendizaje por refuerzos no existen datos de antemano ya que se generan durante la propia ejecución, por lo que separar estos conjuntos no es trivial.

En tareas de aprendizaje supervisado se suelen utilizar épocas para cuantificar el entrenamiento, definidas como una iteración completa de entrenamiento sobre todo el conjunto de datos de entrenamiento. Sin embargo, esta definición no tiene sentido en el aprendizaje por refuerzos ya que a cada paso de tiempo se genera una nueva experiencia sobre la que eventualmente se entrena. Como los datos de entrenamiento se generan dinámicamente, es impracticable determinar cuándo se ha completado una iteración completa de entrenamiento sobre todo el conjunto de datos. Pese a esto, algunos autores igualmente definen una época como una cantidad fija de pasos de entrenamiento [29].

Un concepto importante es el de episodio, que se define como la secuencia de experiencias que van desde un estado inicial a uno terminal. Esta es una forma de cuantificar el entrenamiento en el contexto de aprendizaje por refuerzos, junto con la cantidad de pasos.

El agente tiene dos modos de interactuar con el entorno: puede hacerlo en modo entrenamiento, donde juega tomando decisiones según alguna estrategia de exploración para generar experiencias sobre las que entrenar, o hacerlo en modo evaluación, donde cada decisión la toma intentando maximizar las recompensas obtenidas.

Dado un agente entrenado, la evaluación más sencilla consiste en reportar el promedio de la recompensa que obtiene en una cantidad fija de episodios ejecutando en modo evaluación. A pesar de su simplicidad, esta técnica tiene la desventaja de no contemplar el nivel de sobreajuste de un agente en la medida final. Es difícil considerar cuantitativamente el sobreajuste, por lo que normalmente se usan técnicas que reducen el determinismo del emulador: las más usuales son la utilización de un comienzo humano o aleatorio.

El comienzo humano consiste en obtener puntos muestreados de una partida iniciada por un humano experto. Comienza la evaluación del episodio en uno de esos puntos y se ejecuta el emulador por una cantidad máxima de pasos. Cada agente se evalúa únicamente sobre las recompensas acumuladas obtenidas luego de tomar el control del juego.

Otra posibilidad es comenzar cada episodio de entrenamiento y evaluación con una cantidad aleatoria de operaciones *no-ops*[5] de modo que el juego no comience exactamente igual en todos los episodios.

Debido a las largas ejecuciones que involucra el entrenamiento, es deseable visualizar su progreso en tiempo real a los efectos de decidir sobre la continuidad del experimento. Mnih et al. [29] comentan que una posibilidad es ejecutar el agente durante una cantidad fija de episodios al término de cada época de entrenamiento, utilizando el agente en modo de evaluación, y

[5] *No-op* quiere decir *no operation*, el agente no realiza ninguna acción.

promediar las recompensas obtenidas durante estas ejecuciones. Sin embargo, esta métrica tiene la desventaja de ser muy oscilante debido a la naturaleza de la exploración: una pequeña modificación en los pesos puede inducir a un cambio completo en los estados que se visitan, dando la impresión de que por momentos se está "desaprendiendo".

Como consecuencia de lo anterior, proponen una métrica que en sus experimentos resulta más estable. Consiste en tomar el promedio de la función V en un conjunto de estados seleccionados aleatoriamente. Recuérdese que la función V retorna la mayor recompensa alcanzable desde un estado, siguiendo una política óptima. Al utilizar esta métrica, el conjunto de estados sobre los que se calcula V es análogo a emplear un conjunto de datos de validación en los métodos supervisados[6]. Esta medida no siempre es útil para comparar diferentes agentes. Depende de si se usa el mismo conjunto de estados para evaluar todos los agentes, en cuyo caso la comparación tiene sentido, o si se toma un conjunto nuevo de estados para evaluar en cada ejecución. En este último se tiene una idea del progreso del aprendizaje durante el entrenamiento, pero no es comparable a otras ejecuciones.

Otro valor a tomar en cuenta es la pérdida. Esta es un medida de qué tan bien se ajusta la red durante el entrenamiento a los estados visitados. Es un valor de suma importancia para monitorear el estado de la red, pero no se puede guiar el entrenamiento únicamente por ella porque carece de información respecto del desempeño del agente. Una rápida convergencia puede ser síntoma de sobreajuste.

2.2.6. Redes convolutivas

Las redes convolutivas son un tipo de red neuronal que se utilizan para el procesamiento de imágenes, y representan el estado del arte en esta área [31]. Si bien este tipo de arquitectura tiene aplicación en otros campos, como el procesamiento de lenguaje natural [19], es en el reconocimiento de imágenes donde tiene su origen.

Pese a que teóricamente una red neuronal completamente conectada con una única capa oculta y una cantidad finita de neuronas es capaz de aproximar cualquier función, es ineficiente para procesar imágenes, dado que no contempla la estructura espacial de las imágenes [31], tratando píxeles lejanos y cercanos de igual forma. Esto hace que la propia red deba inferir la relación espacial existente entre los píxeles. Para evitar este problema las redes convolutivas plantean una arquitectura de red que toma ventaja de la estructura de las imágenes.

A continuación se exponen los dos componentes principales de una red convolutiva, con el objetivo de entender cómo funcionan. Para guiar la explicación, se toma el problema de reconocimiento de dígitos escritos a mano MNIST [23], que consiste en identificar, a partir de una imagen de 28×28 píxeles en escala de grises, de cuál dígito se trata.

Campos receptivos locales

Las redes completamente conectadas modelan las neuronas de entrada como un arreglo vertical de píxeles. Sin embargo, en una red convolutiva, la capa de entrada se modela como una matriz de 28×28 neuronas cuyos valores se corresponden con los 28×28 píxeles. Estas neuronas de

[6] No es exactamente un conjunto de validación debido a que no se puede asegurar que durante el entrenamiento el agente no visite estos estados.

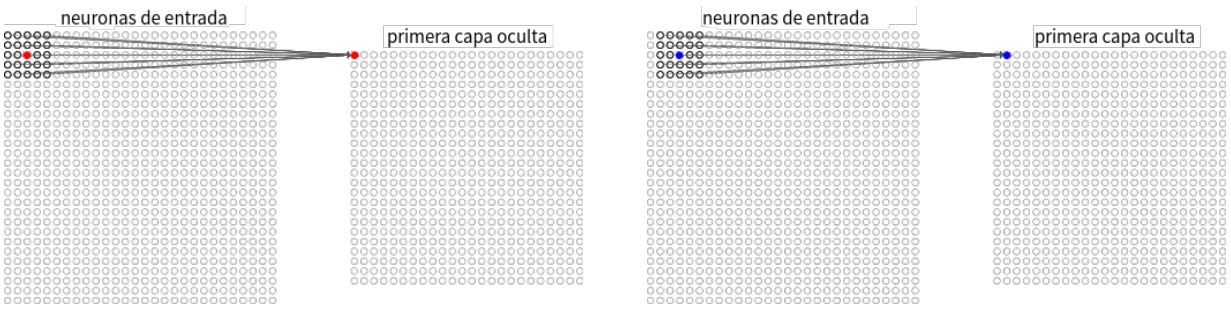


Figura 2.5: Estas figuras representan cómo se desplazan horizontalmente los campos receptivos locales. Análogamente se desplazan de forma vertical cubriendo así la totalidad de la imagen. Se puede interpretar la capa oculta como una nueva imagen donde cada uno de sus píxeles se corresponde con un píxel en la imagen original. Imágenes tomadas del libro de Michael A. Nielsen [31].

la capa de ingreso se conectan a una capa oculta, en regiones pequeñas y localizadas de 5×5, o sea, 25 píxeles, como se muestra en la figura 2.5. Esa región se denomina campo receptivo local (*local receptive field*). Este campo receptivo se desliza a través de la entrada como si fuera una ventana, hasta cubrir toda la imagen. Cada campo receptivo local se conecta a una única neurona en la capa oculta, por lo que esta capa mantiene la estructura matricial, como se muestra en la figura 2.5. Una imagen de 28×28 píxeles, como la del ejemplo, con campos de 5×5, se transforma en 24×24 unidades ocultas. Para entender a qué se debe esta reducción en la dimensión de la imagen, se puede pensar en la capa oculta como una nueva imagen donde el píxel central de cada campo receptivo local queda redefinido en función de sus vecinos. Como los píxeles del borde de la imagen no tienen vecinos hacia el exterior, al aplicar los campos receptivos estos píxeles se pierden resultando en una imagen más pequeña. Si se quieren evitar estas pérdidas se puede agregar un borde a la imagen para que la capa oculta mantenga el tamaño de la imagen original. La forma más común de hacerlo es rellenarla con ceros, aunque también se puede hacer repitiendo los píxeles del borde.

Debido a que los desplazamientos del campo para cubrir la imagen se realizan de a un píxel, se dice que el largo del paso (*stride length*) es de 1. Se pueden utilizar otros valores, por ejemplo 2 o 3.

Una mejora a esta estructura es compartir los mismos pesos en todos los campos receptivos; esto se puede interpretar como que un mismo campo receptivo local se va desplazando por la imagen. Como consecuencia, todas las neuronas de la primera capa detectan exactamente la misma característica, pero en diferentes posiciones, haciéndolas invariantes a la traslación. La imagen de un cero contiene un cero sin importar en qué parte de la imagen se encuentre.

Este mapeo de entrada a la primera capa oculta se llama mapa de características (*feature map*), y a la capa que lo implementa se la denomina capa convolutiva. Esta capa define un *kernel* o filtro idéntico a los que se utilizan en el procesamiento tradicional de imágenes, pero que se aprende automáticamente con el entrenamiento. Esta estructura reconoce una única característica de la imagen, por lo que se utilizan varios filtros. En el ejemplo de la figura 2.8, se muestran tres, con los que se detectan tres características a lo largo de toda la imagen. En la práctica se usan muchos más: en la primera versión de resolución del MNIST, LeCun et al. [22] utilizaron 6 mientras que Krizhevsky et al. [20] sobre ImageNet[7] utilizan 96 mapas de

[7]Es una base de 1,2 millones de imágenes etiquetadas a ser clasificadas en 1000 categorías

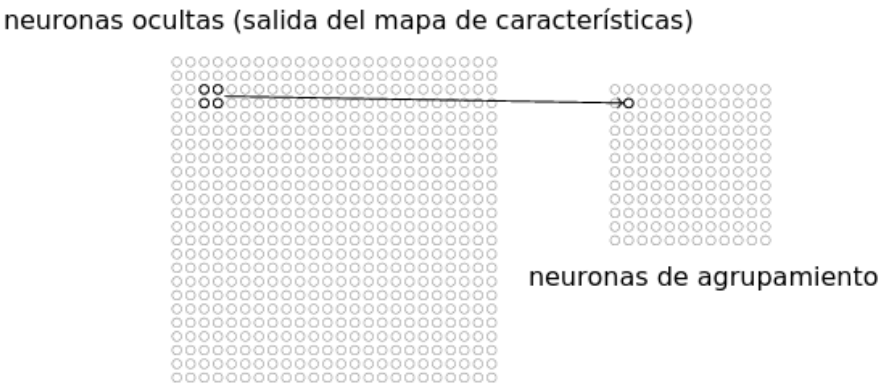


Figura 2.6: Capa de agrupamiento: la información de cuatro neuronas se condensa en una sola. Tomada del libro de Michael A. Nielsen [31].

características[8].

Una ventaja de los pesos compartidos es que reducen el número de parámetros de la red. Para cada mapa se necesitan 25 pesos compartidos y un único sesgo. Con 20 mapas es un total de 520 parámetros definiendo la capa convolutiva. En una red completamente conectada, con 30 unidades ocultas, son 23550 parámetros; en otras palabras, 40 veces más que en el ejemplo convolucional. Intuitivamente al compartir parámetros el entrenamiento llevará menos tiempo y posibilitará construir redes neuronales profundas con capas convolutivas más complejas. Una red con gran cantidad de parámetros para ajustar es, además, propensa a sobreajustar esos parámetros al conjunto de entrenamiento, perdiendo capacidad de generalización.

Capas de agrupamiento

Las capas de agrupamiento (*pooling layers*) con frecuencia son usadas inmediatamente después de una capa convolutiva, ya que simplifican la información en la salida de la capa convolutiva, preparando un mapa de características condensadas. Por ejemplo, en la figura 2.6, cada unidad en la capa de agrupamiento resume una región de 2×2 de la capa anterior. Un ejemplo concreto de unidad de agrupamiento es el *max pooling* donde la unidad se queda con el máximo del valor de activación de la región. Se puede pensar el *max pooling* como una forma para la red de preguntar si una característica dada se encuentra en alguna parte en la región de la imagen. Si bien esta es la más usada de las técnicas no es la única: otros ejemplos son el agrupamiento L2 (*L2 pooling*) donde se toma la raíz cuadrada de la suma de los valores de activación de la región, o el *avg pooling* que consiste en tomar el promedio de estos valores en la región [31].

Al definir esta capa, se debe especificar tanto el tamaño de la región como el largo del paso (*stride length*), de forma análoga a como se los define para los campos receptivos locales.

Una vez que la característica es hallada, no importa tanto su localización exacta sino su aproximación relativa a las demás características. Como ventaja, hay menor cantidad de características agrupadas: de las 24×24 neuronas de la capa convolutiva, quedan 12×12 neuronas de agrupamiento (*pooling neurons*[9]). Esto reduce el número de parámetros necesarios en las

[8]Considerando únicamente los de la primera capa oculta para ambos problemas. La arquitectura completa de LeCun et al. utiliza dos capas convolutivas con 32 mapas mientras que la de Krizhevsky et al. emplea 5 capas con 1376 mapas en total.

[9]Utilizando un paso de tamaño uno.

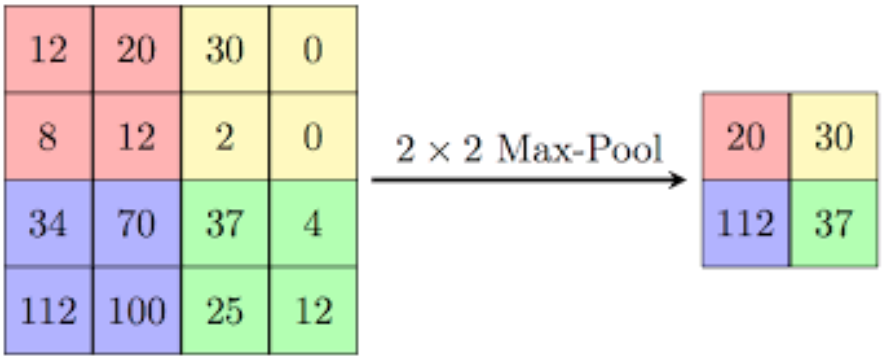


Figura 2.7: Unidad *max pooling* aplicada con 2 de *stride* a regiones de 2×2. Se reducen los parámetros en un 75 % pasando de 16 a 4.

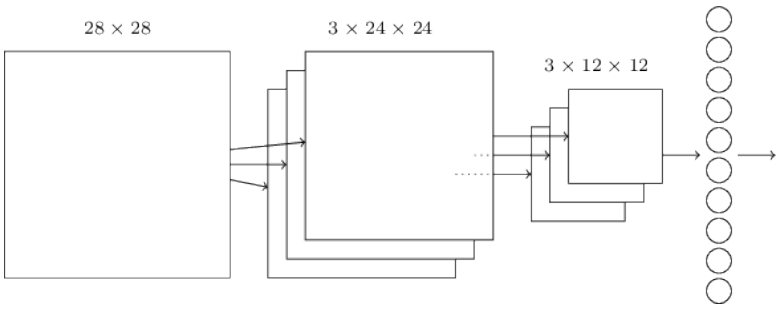


Figura 2.8: Esquema de la arquitectura convolutiva completa. Tomada del libro de Michael A. Nielsen [31].

capas siguientes. Frecuentemente se utilizan regiones de 2×2 con paso de tamaño dos, como se muestra en la figura 2.7. Esta configuración permite disminuir en un 75 % la cantidad de pesos necesarios, independientemente de qué unidad de agrupamiento se utilice. Se suele aplicar una capa de agrupamiento a cada mapa de características.

Para formar una red convolutiva completa se unen los componentes previamente referidos, agregando una capa de salida completamente conectada. En ella se asigna una neurona para cada dígito, como se observa en la figura 2.8. Para problemas más complejos de procesamiento de imágenes, la arquitectura típica de red consiste en intercalar varias veces una capa convolutiva con una de agrupamiento, terminando con una o varias capas completamente conectadas.

2.2.7. *Deep Q-Network*: DQN

DQN es el algoritmo propuesto por Mnih et al. [28, 29] para aproximar a la función Q mediante redes neuronales profundas. La idea central del algoritmo es utilizar una red neuronal que reciba los últimos cuadros, los cuales definen el estado, y retorne en su capa de salida el valor de la función Q para cada posible acción.

Como se menciona anteriormente, aproximar Q con una función no lineal, como es el caso de una red neuronal, genera problemas de inestabilidad y eventualmente divergencia, por lo que se han propuesto dos ideas fundamentales para mitigar sus efectos [25, 28]. Primero, la utilización de *experience replay* en su versión de prioridad uniforme. Segundo, se dispone de una red neuronal auxiliar, igual a la red principal Q, pero que se mantiene fija durante C iteraciones, para lograr

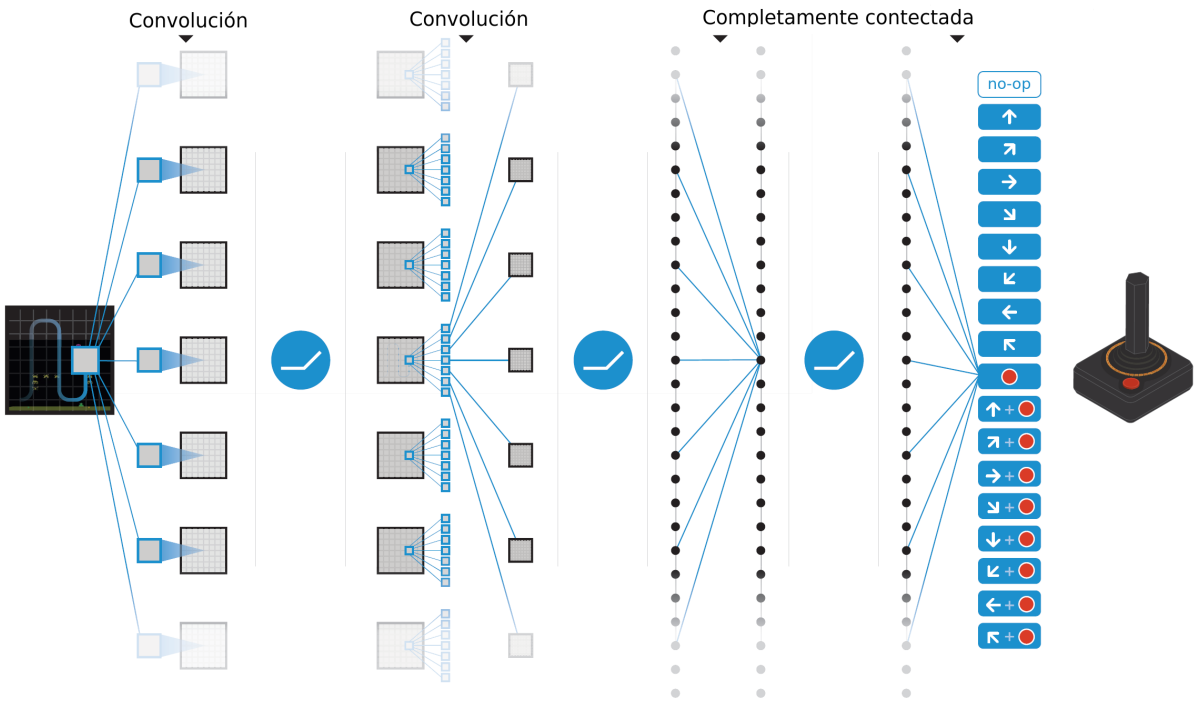


Figura 2.9: Esquema de la arquitectura DQN, presentada por Mnih et al. [29]

así reducir correlaciones con la función Q objetivo. Cada C iteraciones, se clonan los pesos de la red principal Q a su función auxiliar, manteniéndola fija el resto del tiempo. Se indica que sin usar estos mecanismos a lo largo de cinco juegos, el puntaje disminuye a menos de un 30 % de los resultados alcanzados [29].

Arquitectura

El algoritmo DQN plantea utilizar una red neuronal profunda con una unidad de salida para cada posible acción, y únicamente la representación del estado como entrada. Las salidas se corresponden al valor de Q predicho para cada acción posible dado el estado de entrada. Esto hace que en una única pasada por la red se calcule el valor de Q para cada una de las acciones posibles en un estado dado. En la figura 2.9 se diagrama la arquitectura propuesta.

La red contiene una capa de entrada[10] de $84 \times 84 \times 4$, correspondiente a la imagen producida por el preprocesamiento, seguida de una capa convolutiva oculta con 32 filtros de 8×8 con un valor de *stride* de 4. La segunda capa oculta, también convolutiva, consta de 64 filtros de 4×4 con 2 de *stride*. La tercer capa convolutiva consta de 64 filtros 3×3 con *stride* en 1. A cada una de las capas anteriores se le aplica un rectificador no lineal en la salida [30, 17]. La última capa oculta está completamente conectada, consiste en 512 nodos. La capa de salida está completamente conectada, con una neurona por cada acción válida. Este número depende del juego y varía entre 4 y 18 para el caso de ATARI.

Una interpretación de esta arquitectura hecha por Yitao et al. [24] es ver la primera parte de la red, compuesta por capas convolutivas, como la encargada de procesar la imagen para obtener

[10]Se utilizan cuatro observaciones para formar un estado, tal como se describe en la sección 2.2.3.

Inicializar la memoria de replay D, con capacidad N
Inicializar la función Q con pesos aleatorios
for episodio = 1 to M **do**
    Inicializar la secuencia $s_1 = x_1$ y la secuencia preprocesada $\phi_1 = \phi(s_1)$
    for t = 1 to T **do**
        Con probabilidad ϵ elegir una acción aleatoria a_t
        Sino elegir $a_t = max_a Q(\phi(s_t), a; \theta)$
        Ejecutar a_t en el emulador y observar la recompensa r_t y la imagen x_{t+1}
        $s_{t+1} := s_t, a_t, x_{t+1}$
        Preprocesar $\phi_{t+1} = \phi(s_{t+1})$
        Almacenar la transición $(\phi_t, a_t, r_t, \phi_{t+1})$ en D
        Muestrear un *mini batch* de transiciones $(\phi_j, a_j, r_j, \phi_{j+1})$ de D
        $y_j := \begin{cases} r_j & \text{si el episodio termina en el paso j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'_j; \theta^-) & \text{si no} \end{cases}$
        Realizar un paso de descenso por gradiente sobre $(y_j - Q(\phi_j, a_j; \theta))^2$ respecto a la red de parámetros θ
        Cada C pasos, copiar $\hat{Q} = Q$
    end for
end for

Cuadro 2.1: *Deep Q-learning* con *Experience Replay* extraído de Mnih et al. [29].

un vector de características. Sobre este vector, la segunda parte calcula el valor de la función Q a partir de una red completamente conectada. Esta interpretación también se toma como partida en el trabajo de Ian Osband et al. [32].

Entrenamiento

Un pseudocódigo del algoritmo de entrenamiento se muestra en el cuadro 2.1. Durante el *loop* interno del algoritmo, se aplican actualizaciones *Q-learning* a un pequeño conjunto o *mini batch* de muestras de experiencias (s, a, r, s') tomadas aleatoriamente de la memoria de *replay*. Cada C actualizaciones la red Q es clonada para obtener una red objetivo $\hat{Q}$ y esta se usa para generar los objetivos y_j de las siguientes C actualizaciones de Q.

Cada agente se entrena durante cincuenta millones de ciclos, obteniendo una nueva observación y entrenando con un nuevo *mini batch* en cada ciclo.

Resultados obtenidos

Para que resultados obtenidos en distintos juegos sean comparables, se los normaliza de la siguiente manera:

$$\frac{DQN - Random}{Human - Random} \times 100$$

Esta normalización busca descontar el puntaje alcanzable por un jugador aleatorio y hacerlo relativo al desempeño de un humano experto. Cada juego maneja una escala diferente de

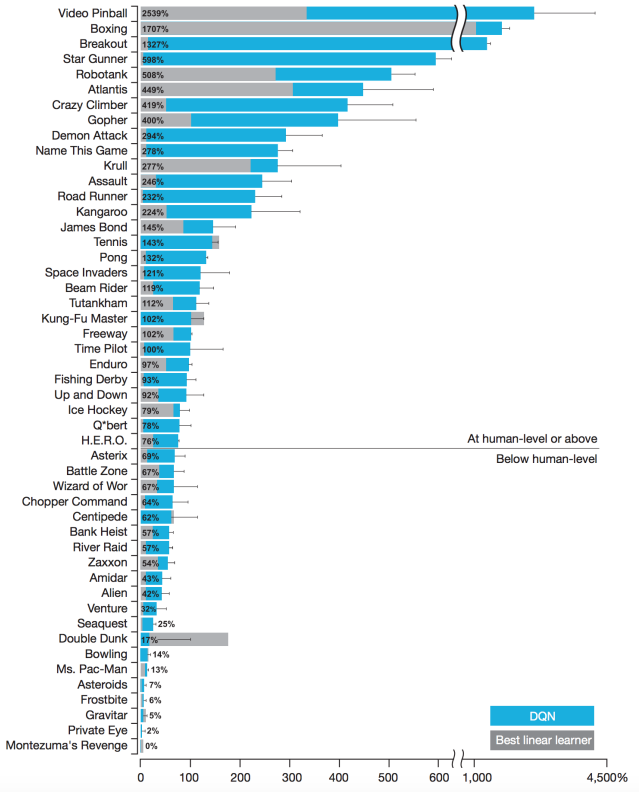


Figura 2.10: Resultados de la arquitectura DQN sobre 49 juegos testeados, presentados por Mnih et al. [29]. Se observa que para juegos con recompensa dispersa no se obtienen buenos resultados.

puntajes, por lo que esta normalización permite comparar fácilmente el nivel del jugador en diferentes juegos. Según estos autores, el agente alcanza un nivel humano si logra al menos un 75 % del puntaje de un humano experto. En la figura 2.10 puede verse que en la mayor cantidad de los juegos se supera el puntaje humano, en algunos por amplio margen, como es el caso de *Video Pinball*. El mal desempeño obtenido en algunos juegos es atribuido a la necesidad de tomar decisiones a largo plazo. Además, la solución propuesta es superior a las mejores implementaciones con aproximaciones lineales.

Críticas

Existen críticas a algunas decisiones tomadas por los autores planteadas por Yitao et al. [24], quienes identifican tres aspectos problemáticos en la metodología experimental utilizada.

La principal crítica es que se hacen comparaciones estadísticas con base en una única prueba. Pese a que los autores reportan la desviación estándar, esto sólo representa la variación observada durante la ejecución de una política elegida, y no la variación observada durante diferentes reentrenamientos independientes. La consistencia en los altos puntajes durante los distintos juegos sugieren que en general se tiene un buen desempeño, pero los resultados reportados en un juego particular pueden no ser representativos del nivel alcanzado.

Además, elegir la política con mejor desempeño de todo el período de aprendizaje y no la que se obtiene al finalizar el entrenamiento es una desviación significativa de la metodología típica en aprendizaje por refuerzos. Tal elección puede cubrir cuestiones de inestabilidad en casos donde

el desempeño del aprendiz es inicialmente bueno pero luego empeora a medida que avanza el entrenamiento, un efecto que es totalmente indeseado. Esto puede ocurrir incluso usando funciones lineales de aproximación comparativamente más estables.

Los autores restringen su agente al mínimo conjunto de acciones que tengan un único efecto en cada juego, información específica previa que no había sido explotada por otros trabajos anteriores sobre ALE. Dado que en la mayoría de los juegos el conjunto mínimo de acciones contiene menos acciones que el conjunto completo, los agentes con conocimiento de este subconjunto pueden tomar ventaja de esto.

También se modifica la finalización del episodio. En varios juegos Atari el jugador tiene una cantidad de vidas y el juego termina cuando estas se acaban. Sus autores utilizan el mismo criterio de finalización que Bellmare [1]: llegar al final del juego o luego de 5 minutos de ejecución[11], pero durante el entrenamiento también termina el juego cuando el agente pierde una vida, que es de alguna manera otra forma de información previa específica del juego.

En definitiva, el uso de conocimiento previo provisto mediante el diseño va en contra de la meta de evaluar capacidades generales. Además de estas críticas en cuanto a la forma que se reportan sus resultados, existen reportes de intentos de reproducción de los experimentos con resultados diferentes a los esperados [11].

2.2.8. *Double Deep Q-Network*: DDQN

Para aprender la función Q se sigue la siguiente actualización:

$$Q(s,a) = r + \gamma \max_a Q(s',a)$$

El principal problema en *Q-learning* está en el operador *Máximo*, ya que usa los mismos valores para elegir y para evaluar una acción. Esto tiende a sobrestimar el valor de la función Q para los estados [42].

La idea detrás de *Double* $Q-$*learning* es evitar esta sobreestimación desacoplando la selección de la evaluación. Para hacerlo, Hasselt et al. [42] plantean aprender independientemente dos versiones de la función, Q_1 y Q_2, una para determinar la acción que maximiza la función, y otra para estimar su valor. Cualquiera de las dos se actualiza ahora con una de las siguientes fórmulas:

$$Q_1(s,a) = r + \gamma Q_2(s', argmax_a Q_1(s',a))$$

$$Q_2(s,a) = r + \gamma Q_1(s', argmax_a Q_2(s',a))$$

En el caso particular de DQN, la función Q se encuentra parametrizada por los pesos θ, que son copiados cada ciertas iteraciones a θ^- donde permanecen incambiados, por lo que θ y θ^- son los candidatos naturales a utilizar como Q_1 y Q_2. Si bien estas funciones no son independientes entre sí, son suficientes para desacoplar la selección de la acción y la estimación de su valor.

[11] 18000 cuadros ejecutando a 60 cuadros por segundo.

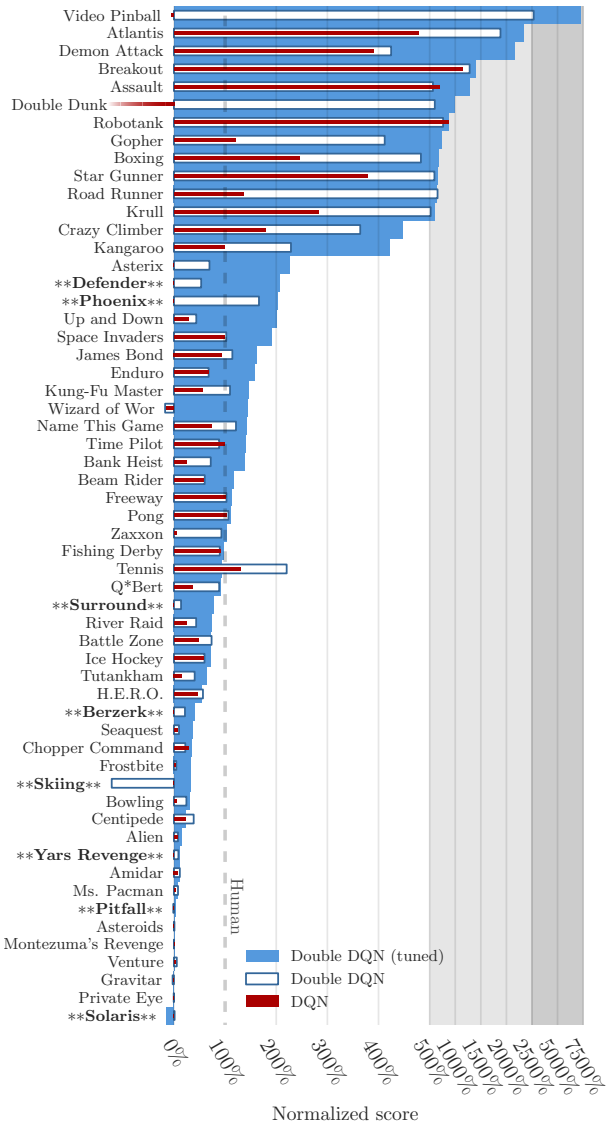


Figura 2.11: Resultados de la arquitectura DDQN presentado por Hasselt et al. [42]

Los resultados se pueden ver en la figura 2.11, donde se compara una implementación de DQN y una de DDQN, ambas utilizando los hiperparámetros usados por Mnih et al., y una tercera en la que se eligieron nuevos hiperparámetros para DDQN. Los puntajes fueron normalizados al igual en DQN, utilizando los mismos valores para el jugador humano y el aleatorio.

Esta propuesta arrojó resultados significativamente mejores que los reportados por Mnih et al. con DQN prácticamente en todos los juegos, incluso antes de realizar ajuste de parámetros.

2.2.9. *Dueling Network Architecture*

La arquitectura *Dueling* es una arquitectura propuesta por Hasselt et al. [43] que intuitivamente permite aprender cuáles estados son valiosos, sin tener que aprender el efecto de cada acción para cada estado. Esto es particularmente útil en los estados donde las acciones no afectan el entorno de forma relevante. Por ejemplo en el juego *Enduro* de Atari, que consiste en manejar un auto rebasando los demás, es necesario saber cuándo moverse para la izquierda o derecha sólo cuando una colisión es inminente. En muchos estados es de gran importancia conocer qué acción tomar mientras que en otros la elección de la acción no tiene repercusión.

Hasselt et al. plantean separar explícitamente la representación de valores de estado V utilizando una nueva función denominada Ventaja de acción A, definida relacionando a las funciones Q y V:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

Se puede interpretar el valor de la función V como una medida de cuán bueno es estar en un estado particular s, sin importar la acción que se elija. La función Q sin embargo es una medida del valor de tomar determinada acción a estando en el estado s. La función Ventaja de acción, A, representa un valor relativo a la importancia de cada acción.

En cuanto a la arquitectura, las capas bajas de la red son convolutivas al igual que el ejemplo de DQN original. Sin embargo, a continuación de las capas convolutivas, en lugar de usar una única secuencia de nodos totalmente conectados, se usan dos, como se muestra en la figura 2.12.

Las secuencias están construidas para que provean estimaciones separadas para la función valor y la función de ventaja. Por último las dos secuencias son combinadas, dando como salida final el valor de la función Q para cada acción válida.

Una de las secuencias tiene como salida un escalar $V(s;\theta,\beta)$, y la otra tiene por resultado un vector del tamaño del espacio de acciones válidas, $A(s,a;\theta,\alpha)$. Aquí, θ denota los parámetros de las capas convolutivas, mientras que α y β son los parámetros de las dos secuencias de las capas completamente conectadas.

El módulo final, marcado con verde en la imagen 2.12, sigue la siguiente ecuación:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{|A|}\sum_{a'}(s,a';\theta,\alpha))$$

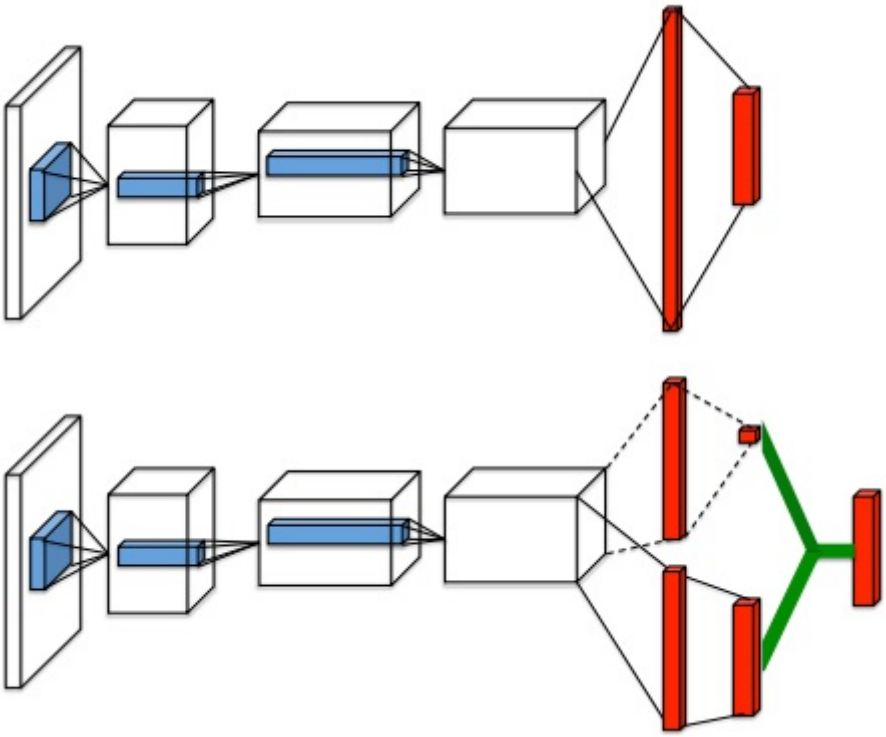


Figura 2.12: Arriba: Q-network estándar [29] con una única secuencia. Abajo: *Dueling Q-network*. Esta red tiene dos secuencias para estimar separadamente el escalar $V(s)$ y la ventaja para cada acción. Tomada del trabajo de van Hasselt [43].

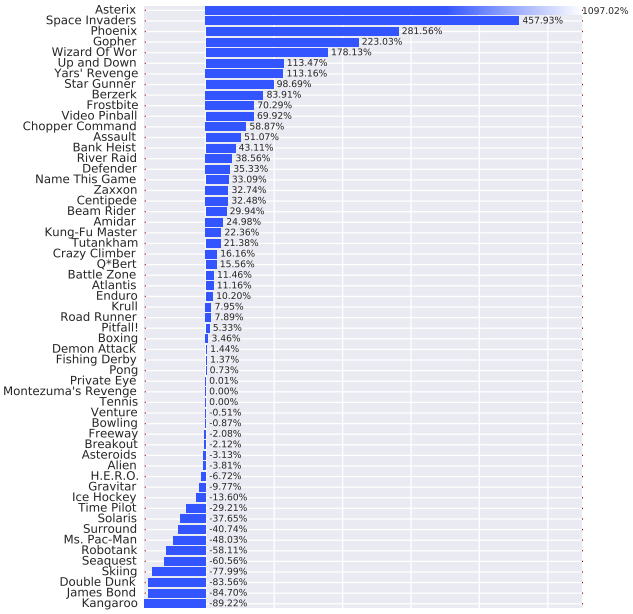


Figura 2.13: Resultados de la arquitectura Dueling reportados por van Hasselt [43]

Es importante observar que esta ecuación se implementa como parte de la red y no como un paso separado del algoritmo. El entrenamiento de la arquitectura *Dueling*, es realizado como con las redes Q estándar, y sólo requiere *back-propagation*. Además, las estimaciones de $V(s;\theta,\beta)$ y $A(s,a;\theta,\alpha)$ son calculadas automáticamente sin requerir supervisión extra o modificaciones de algoritmos.

Resultados obtenidos

La normalización de resultados propuesta por Hasselt et al. [43] tiene la misma motivación que la de Mnih et al. [29]: medir el desempeño del agente respecto a un humano, permitiendo comparar resultados de distintos juegos. Sin embargo, es ligeramente modificada:

$$\frac{Puntaje_{Agente} - Puntaje_{LineaBase}}{\text{máx}(Puntaje_{LineaBase}, Puntaje_{Humano}) - Puntaje_{Aleatorio}} \times 100$$

Esta modificación impide que cambios insignificantes aparezcan como grandes mejoras en los casos en que ni el agente ni la linea base tienen un buen desempeño.

Usando la arquitectura *Dueling* en combinación con DDQN y *Prioritized Experience Replay*, se obtuvieron los mejores resultados, los cuales se ven reflejados en la figura 2.13.

2.2.10. Asistencia humana

Sería deseable un agente que, por sí mismo y de forma independiente, pueda desempeñarse cumpliendo sus objetivos eficientemente en cualquier momento y situación, sin intervención alguna de agentes externos. Debido a que no siempre es posible, ya sea por la complejidad del problema o la dificultad para definir la función a maximizar [6], existen métodos que buscan asistir al agente mediante la intervención de un *Maestro*. Frecuentemente buscamos que el esfuerzo del maestro sea mínimo, pero que tenga un gran impacto en el aprendizaje del agente [44].

Las siguientes dos secciones presentan dos enfoques desarrollados por Hosu y Rebedea [15] para asistir en el proceso de exploración. A diferencia de los métodos y técnicas desarrolladas en secciones previas, estos enfoques brindan ayuda humana al agente, por lo que deja de ser un método general ya que la ayuda humana es particular para cada juego. Estas técnicas tienen la propiedad de colaborar en la exploración independientemente de la política exploratoria que se utilice, por lo que se espera que aceleren el aprendizaje y mejoren los resultados.

Human Checkpoint Replay

Este método consiste en almacenar posiciones o *checkpoints* del juego logradas por un humano en camino a una recompensa positiva. Estos *checkpoints* son almacenados para que luego el aprendiz, en vez de comenzar siempre desde el principio del episodio, empiece desde uno de estos puntos.

La intuición detrás de este método es que al menos un *checkpoint* estará lo suficientemente cerca de una recompensa positiva para ser alcanzada, sin importar la estrategia de exploración utilizada. De esta forma el agente es capaz de aprender lecciones relevantes, logrando así una mejor estrategia en la explotación a medida que el entrenamiento avanza.

Este método, probado con un agente DQN, reporta mejores resultados que los obtenidos con un agente totalmente aleatorio, cercano al doble de puntaje para *Montezuma's revenge* y varios órdenes de magnitud para *Private eye* [15]. Sin utilizar HCR, los trabajos previos no superan un agente aleatorio para estos juegos.

Se debe tener en cuenta que para iniciar cada episodio en un *checkpoint* se requiere la habilidad de seleccionar en el emulador un estado arbitrario del cuál comenzar.

Human Experience Replay

Este método alternativo utiliza directamente secuencias *offline* de juego humano almacenado en forma de experiencias (s, a, r, s'). Esto es similar a mostrarle al agente videos de un humano experto jugando. Para esto se debe disponer de una gran cantidad de partidas jugadas por un humano, en forma de experiencias, las que se almacenan de forma separada en la memoria del agente. Durante el entrenamiento se utilizan, además de la experiencia generada por el agente, las secuencias humanas. La forma de aprendizaje consiste en tomar repetida y simultáneamente experiencias humanas y experiencias generadas por el agente, en forma de *mini-batches*.

En el juego *Montezuma's Revenge* se pone a prueba el método y no se obtiene una mejora sustancial con respecto a un agente aleatorio. Debido a la naturaleza dispersa de la recompensa, las transiciones humanas no son suficientes para lograr que el agente aproveche estas secuencias con recompensa positiva de forma eficaz en la exploración.

Capítulo 3

Solución planteada

En este capítulo se presentan los experimentos realizados para el juego Manic Miner sobre la plataforma Spectrum. Si bien la mayor parte de los trabajos previos en el área se realizan sobre la plataforma ATARI, las técnicas relevadas deberían poder aplicarse a juegos de la plataforma Spectrum sobre la que se realizan los experimentos de esta sección.

Se realizan varias pruebas, entrenando agentes con arquitecturas DQN y Dueling, combinándolos con técnicas como DDQN *Priorized Experience Replay* y *Human Checkpoint Replay*. Los agentes utilizan exactamente la misma arquitectura de red que consta de tres capas convolutivas, seguidas de dos capas completamente conectadas, salvo en el caso de la arquitectura Dueling, donde se sustituyen las dos capas finales por dos secuencias independientes antes de la capa de salida. Ambas arquitecturas toman como entrada un estado conformado por los últimos cuatro cuadros, y tienen como salida el valor de la función Q para el estado de entrada y cada una de las siguientes acciones: izquierda, izquierda arriba, arriba, derecha arriba, derecha, enter[1] y *no-op*.

Debido a las diferencias existentes entre la plataforma utilizada en este proyecto y la consola ATARI que se emplea en los trabajos tomados como referencia, como por ejemplo en el tamaño de los cuadros, se pueden mejorar los resultados en la medida que se adapten más cantidad de parámetros y decisiones de diseño. Se deben probar varias configuraciones, comparando sus resultados. Sin embargo, debido a la gran cantidad de parámetros susceptibles de ser modificados y al tiempo que lleva cada prueba, sólo se modifican aquellos que tienen mayor impacto en el aprendizaje del jugador. En particular, los vinculados estrictamente a la red neuronal no han sido modificados. Estos son: la cantidad, tipo y dimensión de las capas, el algoritmo de descenso por gradiente y sus parámetros asociados, el tamaño del mini-*batch* de ejemplos empleados para entrenar y la tasa de aprendizaje de la red. Aquellos parámetros que no se modifican son tomados del trabajo de Mnih et al. [29], debido a los buenos resultados que los autores obtienen en una importante cantidad de juegos.

Para evitar el sobreajuste debido a que la consola es completamente determinista —cada nivel comienza en un mismo estado y no tiene ningún componente aleatorio—, al iniciar cada juego se obliga al agente a ejecutar una cantidad aleatoria, entre 0 y 30 de *no-ops*, tanto durante el entrenamiento como en las evaluaciones. Además se siguen los mismos criterios que proponen

[1]Esta acción puede no haber sido incluida ya que no produce efectos durante el juego. Se incluyó en las primeras pruebas y, para que los resultados sigan siendo comparables, no se removió luego.

Mnih et al. [29] para terminar un episodio: superar el nivel, morir o ejecutar una cantidad máxima de pasos.

Como algoritmo de exploración se escoge $\varepsilon-greedy$, comenzando todos los experimentos con un ε de 0,99, y disminuyéndolo linealmente durante el primer millón de pasos, hasta llegar a 0,1. Luego queda fijo en este valor hasta el final del experimento. Durante las evaluaciones también se utiliza este algoritmo con un valor fijo de ε (0,05) para evitar que el agente entre en posibles bucles infinitos.

En cada experimento existe una fase inicial denominada *warmup* o calentamiento. Durante esta etapa se ejecutan acciones aleatorias con el fin de poblar la memoria de *replay* para luego comenzar el entrenamiento propiamente dicho. Esta fase dura los primeros 50000 pasos. En la memoria de *replay* siempre se almacena el último millón de experiencias, y se utiliza la versión sin prioridades como base, a menos que se explicite lo contrario.

Cabe mencionar que entrenar un agente toma entre cinco y doce días. Si bien inicialmente se esperaba probar con varios niveles, durante el transcurso del proyecto se opta por experimentar únicamente con el primero, debido a que es el nivel más sencillo, y a que enfocarse en uno solo posibilita probar más configuraciones.

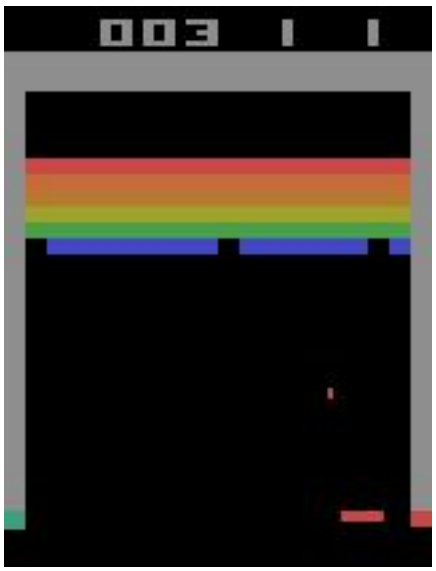
En el resto del capítulo se presentan la definición de algunos parámetros y decisiones tomadas para los experimentos desarrollados. Luego se introducen las métricas utilizadas, para finalmente exponer los experimentos realizados, analizando sus resultados. Sobre el final del capítulo se presenta una comparación de los distintos agentes entrenados, incluyendo pruebas sobre su capacidad de generalización.

3.1. Observaciones y estado

Dado el gran impacto que tiene el estado en el desempeño final del agente, se comienza el capitulo detallando las decisiones de su diseño, las que se mantienen constantes en todos los experimentos realizados.

A diferencia de ATARI donde las dimensiones de los cuadros son de 210×160 píxeles, la computadora ZX Spectrum genera imágenes de 256×192. Las observaciones son preprocesadas con el único objetivo de reducir su dimensión. Se opta por llevarlas a un tamaño de 84×84 en escala de grises, manteniendo así las dimensiones de trabajos previos [29]. Las consecuencias de este preprocesamiento pueden apreciarse en la imagen 3.2c. El tamaño de la imagen no se modifica proporcionalmente para mantener la arquitectura de red propuesta en DQN, por lo que inevitablemente se deforma y pierde resolución. Como la imagen original es rectangular y se la transforma en un cuadrado, se pierde más información en el eje horizontal que en el vertical ya que se debe comprimir más la imagen en esta dimensión. En contraste, casi no existe deformación al preprocesar la imagen para ALE por las dimensiones de la pantalla, como se aprecia en la figura 3.1.

Zahavy et al. [45] postulan que los píxeles correspondientes al puntaje, debido a su constante actualización, hacen que la red les preste más atención, restando importancia al resto de la imagen. Por tal motivo se recorta la parte inferior de la pantalla que contiene las vidas restantes, el aire y el puntaje. En nuestra implementación, este recorte se hace dentro del emulador: al inicializarlo se indica la cantidad de píxeles que se desea quitar a la observación antes de devolverla. Se puede apreciar en la figura 3.2b el resultado de este proceso.



(a) *Original.*

(b) *Preprocesada.*

Figura 3.1: Efectos del preprocesamiento sobre una observación en ALE.

Los cuadros que genera el emulador, una vez que pasaron por el preprocesamiento, se utilizan para formar los estados, que son los usados por el agente para decidir cuál es la mejor acción a tomar. En este proyecto un estado se forma tomando los últimos cuatro cuadros.

Para la representación de los estados iniciales, tipicamente se rellenan los cuadros faltantes con ceros, sin hacer especial hincapié en este asunto. No obstante, Zahavy et al. [45] establecen que es más estable el aprendizaje cuando se replica el primer cuadro en los faltantes, por lo que se sigue este último enfoque.

Esta representación del estado sirve de insumo al agente para tomar decisiones sobre cómo actuar. La red que aproxima a la función Q toma como entrada el valor de gris de cada uno de los píxeles de los cuatro cuadros procesados que conforman el estado. Esto determina que la red debe poseer, entonces, exactamente 28224 entradas ($84 \times 84 \times 4$).

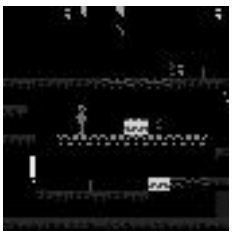
3.2. Política inicial

Debido a que el entorno en que se desarrolla el juego es completamente determinista, Mnih et al. [29] proponen iniciar cada episodio, tanto en el entrenamiento como en las pruebas con una política aleatoria, de manera de romper el determinismo de la consola. Para determinar esta política se analizaron dos enfoques: por un lado, utilizar una cantidad aleatoria de pasos de *no-ops*, entre 0 y 30, tomada según una distribución uniforme. Como alternativa, y en pos de aumentar la diversidad de estados iniciales, se decide tomar acciones aleatorias en vez de únicamente *no-ops*.

Este segundo enfoque aplicado al primer nivel tiene un problema: es posible que el agente muera durante esta etapa de inicialización aleatoria. Para evaluar el impacto, se registra la duración de los episodios para un jugador aleatorio, con salteo de cuadros igual a 1, 2 y 3 en el primer nivel. Con estos datos, se construye una tabla de frecuencias para aproximar la probabilidad de morir luego de ejecutar n acciones aleatorias. A partir de estas frecuencias se estima la probabilidad

(a) *Original*. Imagen de 256×192 píxeles generada por el emulador de ZX Spectrum. Puede verse el indicador de vidas restantes y aire remanente, junto al de puntaje.

(b) *Recortada*. Se pasa a una imagen de 240×128 píxeles que no incluye el puntaje, las vidas ni el aire remanente. Esta es la imagen que retorna el entorno implementado.



(c) *Preprocesada*. Esta es la observación tal cuál la recibe el agente: convertida a escala de grises y de tamaño 84×84 píxeles.

Figura 3.2: Efectos del preprocesamiento sobre el cuadro sobre una observación en ZX Spectrum.

de morir de la siguiente manera.

Sea $P(j)$ la probabilidad de que un agente aleatorio muera exactamente en j pasos. Este valor se estima siguiendo un enfoque frecuentista sobre el total de los 500 episodios:

$$P(j) = \frac{\#terminados_j}{total} \tag{3.1}$$

Donde $terminados_j$ representa los episodios que terminan en exactamente j pasos.

$$P_{morir}(M = i) = \sum_{j=1}^{j=i} P(j) \tag{3.2}$$

La ecuación 3.2 representa entonces, la probabilidad de morir antes de terminar M pasos de inicialización aleatoria.

Se observa en la figura 3.3 que al comenzar tomando un máximo de 30 acciones aleatorias, salteando de a un cuadro, existe una probabilidad de morir que ronda el 30 %, y con la misma cantidad, cuando se saltean de a 3 cuadros, alcanza el 70 %. Esto representa una gran cantidad de veces que el agente va a morir durante la inicialización del episodio, teniendo que, o bien reiniciarlo y volver a ejecutar los pasos aleatorios —lo cual es costoso ya que se desperdicia la primera inicialización— o bien reiniciar el episodio nuevo junto con el entrenamiento —que dejaría una cantidad de veces significativas el mismo estado inicial—. Dadas las importantes limitantes en la capacidad de cómputo, se decidió tomar la opción segura de utilizar una cantidad aleatoria entre 0 y 30 de *no-ops* para inicializar el episodio.

Vale la pena aclarar que el análisis realizado en esta sección fue únicamente para el primer nivel, que es en el que se centra este informe. Existen algunos niveles en donde realizar demasiados *no-ops* al inicio puede hacer que Willy muera, así como otros en los que tomar una pequeña cantidad aleatoria de acciones aleatorias podrían no matarlo.

3.3. Métricas utilizadas

Antes de comenzar las pruebas, se definen las métricas que se utilizan para medir su desempeño. Se diferencian dos tipos de métricas: unas se obtienen durante y otras luego del entrenamiento. Las primeras se emplean para monitorear el progreso del agente durante el entrenamiento ya que las ejecuciones son largas y es conveniente seguir su evolución. Las segundas se utilizan para evaluar la capacidad del agente una vez entrenado.

3.3.1. Durante el entrenamiento

Al finalizar cada episodio del entrenamiento se registran las métricas detalladas en el cuadro 3.1, que permiten monitorear el progreso del aprendizaje. Estas medidas obtenidas durante el entrenamiento tienen tendencia a ser ruidosas y no capturan exactamente la eficacia del método

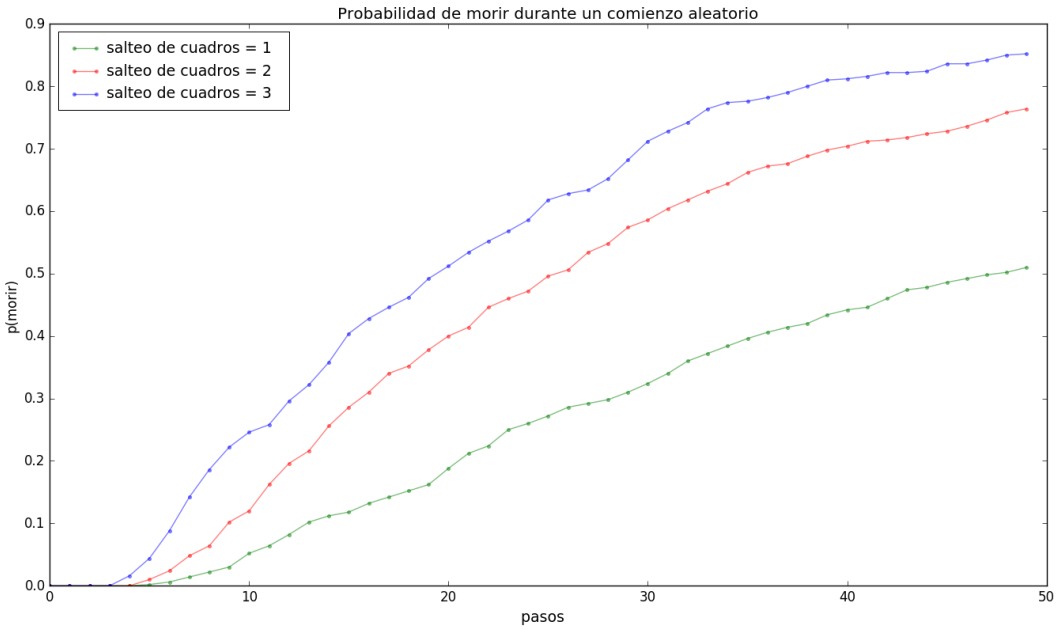


Figura 3.3: Probabilidad de morir en función de la cantidad máxima de movimientos aleatorios iniciales, M.

debido a que presentan una varianza alta. Esto se debe a que en cada paso estamos aproximando la función Q a partir de una versión previa de esta función. Al entrenar, cada paso modifica ligeramente los pesos, y con estos se modifica la política que sigue el agente.

Para paliar este problema, se suavizan las gráficas reportando la media móvil, lo que permite visualizar tendencias: en cada punto se toma una ventana de valores anteriores y posteriores, y se promedian. Una ventana pequeña considera un entorno más cercano al valor real; las grandes, por el contrario, tienden a generar gráficas suaves dando mayor peso al entorno pero disolviendo más en el promedio la recompensa obtenida en cada paso individual. La elección del tamaño corresponde a un compromiso entre qué tan suave se quiera la gráfica y la importancia dada a los episodios cercanos. En las siguientes pruebas generalmente se utiliza un valor de 1000; este valor fue elegido luego de realizar pruebas preliminares y observar que permite visualizar tendencias de forma clara.

Para reportar el valor de la función V promedio, que se menciona en el cuadro 3.1, se selecciona un conjunto de estados S, y al término de cada episodio se calcula el promedio de la función V sobre los estados del conjunto S, como se indica a continuación:

$$V_{promedio} = \frac{\sum_{s \in S} V(s)}{\mid S \mid}$$

Utilizar el promedio de la función V para monitorear el proceso de aprendizaje es algo que plantean Mnih et al. [29] pero no mencionan cómo se obtienen los estados de S, limitándose a decir que se hace de manera aleatoria, por lo que se plantean a continuación algunas formas de hacer esta selección.

Existen dos aproximaciones para esto: obtener un conjunto de estados aleatorios una única vez,

Recompensa del episodio: es la recompensa acumulada a lo largo del episodio. Es esperable que, en la medida que el agente aprende, este valor crezca. Hay que tomar en cuenta que la función de recompensa es modificada durante el desarrollo de distintas pruebas, por lo que se debe tomar recaudo al comparar este valor entre ejecuciones.

Puntaje global: es el puntaje total acumulado al finalizar el episodio. Este valor no se registra en las pruebas iniciales. Dado que la función de recompensa es cambiada a lo largo de las ejecuciones, se lo añade para comparar el desempeño de distintos agentes independientemente de la función de recompensa utilizada.

Pérdida: la pérdida al entrenar sobre una red neuronal es la diferencia entre la salida esperada y la salida obtenida, es decir, el error de la red. Es esperable que este valor converja a cero. Cuando esto sucede, la red ya no tiene más por aprender. Una convergencia rápida puede ser síntoma de sobreajuste.

V promedio: es el valor promedio sobre la función V para un conjunto aleatorio de estados al terminar un episodio.

Cuadro 3.1: Detalle de las métricas utilizadas.

almacenarlos y luego en cada entrenamiento utilizar el mismo conjunto para evaluar la función V, u obtener un nuevo conjunto S al comienzo de cada entrenamiento sobre el que evaluar la función. Para obtener estos estados, sin importar cuál de estos enfoques se tome, lo más sencillo es utilizar un jugador aleatorio, y para cada estado visitado decidir si se lo agrega o no con cierta probabilidad, hasta completar la cantidad deseada.

Una variante de este método es la selección manual del conjunto de estados S. Como principal ventaja, permite elegir estados mejor distribuidos en el nivel pero tiene la desventaja de que hay que generarlos para cada juego o nivel.

En este proyecto se opta por obtener aleatoriamente el conjunto S al comenzar cada entrenamiento, de manera de minimizar el trabajo al momento de añadir un nuevo juego o nivel. Se toman 10 estados de los que visita un jugador aleatorio, donde cada uno de los visitados tiene una probabilidad de 0,1 de ser incluido en el conjunto de referencia.

3.3.2. Postentrenamiento

Durante el entrenamiento del agente, cada determinada cantidad de pasos, se guardan los pesos de la red neuronal para comparar el rendimiento de un mismo agente con distintas cantidades de entrenamiento. En nuestro caso se realizan los respaldos cada 25000 pasos y, como cada agente entrena entre tres y doce millones de pasos, se obtienen entre 120 y 480 conjuntos de pesos a evaluar con los que, para medir su desempeño, se ejecutan varios episodios de evaluación.

Este proceso es muy costoso en tiempo, por lo que se utiliza una heurística para seleccionar sólo aquellos pesos que prometen mejores resultados a partir de su desempeño durante el entrenamiento, disminuyendo la cantidad de pesos a evaluar para cada agente. Tomando como referencia los datos del entrenamiento, se observa que los pesos con mejor desempeño tienden a ser aquellos que fueron almacenados cercanos a los puntos en que se producen las recompensas máximas durante el entrenamiento. De esta forma, se escogen los pesos almacenados en un entorno de los lugares en que se llegó a la máxima recompensa, como se observa en la figura

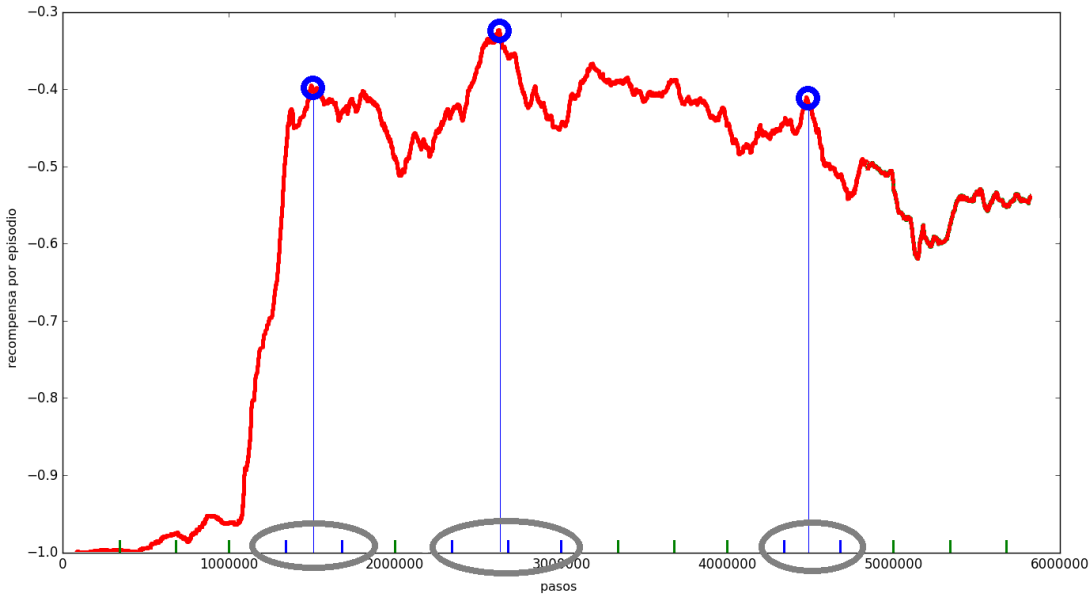


Figura 3.4: Para elegir los pesos prometedores se consideran aquellos que fueron almacenados en momentos cercanos a donde se alcanzan máximos locales durante el entrenamiento. Este ejemplo ilustrativo muestra la función de recompensa por episodio, los círculos azules señalan los máximos locales escogidos. Las marcas en el eje x, separados por cada 250 mil pasos, representan los puntos del entrenamiento en donde se realiza un respaldo de los pesos de la red. La cantidad de máximos locales elegidos y el tamaño de la ventana son aleatorios y dependen de la cantidad de puntos del entrenamientos que se desean evaluar.

3.4. Luego, para cada uno de esos conjuntos de pesos, se ejecutan pruebas con el agente en modo evaluación durante 30 episodios[2], registrando la recompensa que alcanza y la cantidad de pasos que le toma llegar a ella.

Finalmente se escogen los pesos que maximizan la recompensa promedio durante las 30 pruebas. En caso de que dos conjuntos logren la misma recompensa en promedio, se toma aquel que tenga menor cantidad de pasos. Este conjunto de pesos es el que se admite como el mejor para este agente y su método asociado. Se sigue una metodología análoga para elegir al conjunto de pesos capaz de superar más veces el nivel.

3.4. Procesamiento de recompensa

Durante el proyecto se manejan diferentes alternativas para la función de recompensa. Esta función es modificada en el transcurso de las pruebas cuando se detecta que induce dificultades al aprendizaje del agente.

En los primeros experimentos se asignan recompensas negativas al morir, sin importar la causa de la muerte. Esto lleva a problemas con las muertes por aire en donde el agente no recibe información relativa al aire remanente porque anteriormente se recorta esta información de la

[2]Esto implica utilizar ε en 0,05 para evitar *loops* infinitos y comenzando cada episodio con una cantidad aleatoria entre 0 y 30 *no-ops*

Figura 3.5: La llave marcada se encuentra muy cerca de la planta. Esto hace que el agente en un mismo paso pueda recolectar la llave y morir saltando desde el piso deslizante.

imagen. El agente puede pasar por dos experiencias exactamente iguales, en las que toma la misma acción y llega a un mismo estado, pero en uno obtiene una recompensa negativa por quedarse sin aire. Estos problemas se observan en las pruebas preliminares.

Como alternativa, se toma el enfoque tradicional de retornar únicamente la diferencia de puntaje, sin ningún procesamiento sobre la recompensa. Con esta decisión se comienzan las pruebas para determinar cuántos cuadros saltear.

Durante estas pruebas, al observar que el agente alcanza repetidamente a una segunda llave, se pone especial énfasis en ver cómo lo hace, ya que en principio la llave dos parece ser difícil de obtener, porque implica pasar por el suelo deslizante, esquivar una planta, saltar al guardia y saltar al final superior. Sin embargo, resulta que la llave cuatro es fácilmente alcanzable, saltando desde el piso deslizante, aunque implica una muerte inmediata. Esta llave se encuentra marcada con un círculo en la figura 3.5. El agente sistemáticamente recolecta la primera llave y seguidamente muere al tomar la cuarta llave y tocar la planta en el mismo paso.

Dado que el agente no tiene recompensas negativas al morir y como aún no ha explorado lo suficiente como para conocer que existen otras llaves, tomar la llave aunque implique un suicidio es la opción que le reporta mayor recompensa acumulada. Para que el agente tome una decisión distinta, debe encontrar otra trayectoria que le reporte mayor retorno. Llegar a la llave dos es difícil porque requiere una larga secuencia de pasos, que incluyen sortear varios obstáculos; puede ser entonces que ϵ-*greedy* no sea una política de exploración lo suficientemente efectiva.

Esto lleva a retomar la idea de penalizar las muertes para desalentar este tipo de comportamientos, exceptuando aquellas muertes que se producen por falta de aire para no volver a caer en el comportamiento anterior. Se observa que añadir recompensas negativas hace que la escala de recompensas deje de ser comparable con las versiones anteriores.

Luego de las pruebas utilizando la nueva función de recompensa se observa que el agente toma la primer llave y deja de intentar obtener la llave cuatro pero se queda inmóvil hasta el final del episodio. De todos modos, se decide quitar la llave cuatro del nivel para las demás pruebas, asegurando que el agente no tenga ninguna motivación para suicidarse.

Cabe destacar que no solo se eliminan las penalizaciones de las muertes por falta de aire, sino que se eliminan directamente estas muertes, sustituyéndolas por un final del episodio luego

de cierta cantidad fija de pasos. Se prueba inicialmente con un máximo de 18000 pasos por episodio[3], aunque finalmente queda establecido en 1800. Esta cantidad de pasos es bastante mayor que los 425 pasos requeridos para agotar el aire[4], dando más tiempo al agente para explorar el entorno antes de que termine el episodio.

En la figura 3.8 se grafica la recompensa obtenida, y la duración del episodio en pasos. Para este experimento, la mayor cantidad de pasos posibles en un episodio es 18000, y se alcanza una única vez, con lo cual los restantes episodios terminan porque el agente muere, obteniendo -1 como recompensa. Los episodios con recompensa 0 son aquellos en los que se consigue una llave, y con recompensa 1 cuando se consiguen dos. Como este experimento fue realizado sin la llave cuatro, obtener dos llaves implica necesariamente al menos una del nivel superior. Además, se puede ver un caso en que llega a recompensa 2, lo que implica tomar tres llaves, ya que por la duración del episodio se deduce que no muere por *time-out*.

3.5. Elección de salteo de cuadros

En la práctica, el salteo de cuadros es un parámetro que influye mucho en el desempeño final del agente y su valor óptimo depende de cada juego [3]. Se observa en experimentos preliminares que en el videojuego *Breakout*[5] el parámetro hace cambiar significativamente la calidad del jugador.

En esta sección se presentan los experimentos realizados con el fin de determinar la cantidad óptima de cuadros a saltar. Para realizarlos se entrenan paralelamente tres agentes DQN salteando de a 1, 2 y 3 cuadros cada uno. En base a los resultados de estas pruebas se decide el valor más adecuado para utilizar en las pruebas subsiguientes.

En las pruebas iniciales, se comienzan todos los episodios desde el mismo estado, es decir, sin inicializar cada episodio con una cantidad aleatoria de acciones. Naturalmente, partir siempre del mismo estado induce a que el agente se sobreajuste y pueda simplemente memorizar una secuencia de movimientos. Además, se utiliza como recompensa directamente la diferencia de puntaje, sin penalizar las muertes, utilizando todas las llaves y sin quitar las muertes por aire[6].

En la figura 3.6 se observan los resultados de las pruebas iniciales; cada uno de estos entrenamientos implican una ejecución de entre tres y cuatro días.

En vista de los resultados, el agente que saltea de a 3 cuadros comienza a aprender antes, ya que, a igual cantidad de pasos, recibe una mayor diversidad en el entrenamiento que los demás. Sin embargo, salteando de a 2 cuadros es cuando se obtiene una mayor recompensa. Se descarta utilizar el salteo de a 1 cuadro ya que muestra ser igual o más lento que saltear 2 cuadros, con resultados sensiblemente inferiores.

[3]Equivalente a 5 minutos ejecutando a 60 cuadros por segundo.

[4]Salteando de a tres cuadros durante la ejecución.

[5]Este juego de ATARI consiste en mover horizontalmente una barra ubicada en la parte inferior de la pantalla para impedir que se caiga la pelota. Al rebotar en ella, cambia su dirección y puede romper bloques de la parte superior, obteniendo puntos.

[6]En la práctica estas pruebas evidenciaron que el agente se suicida por tomar la llave cuatro, como se discute en la sección anterior.

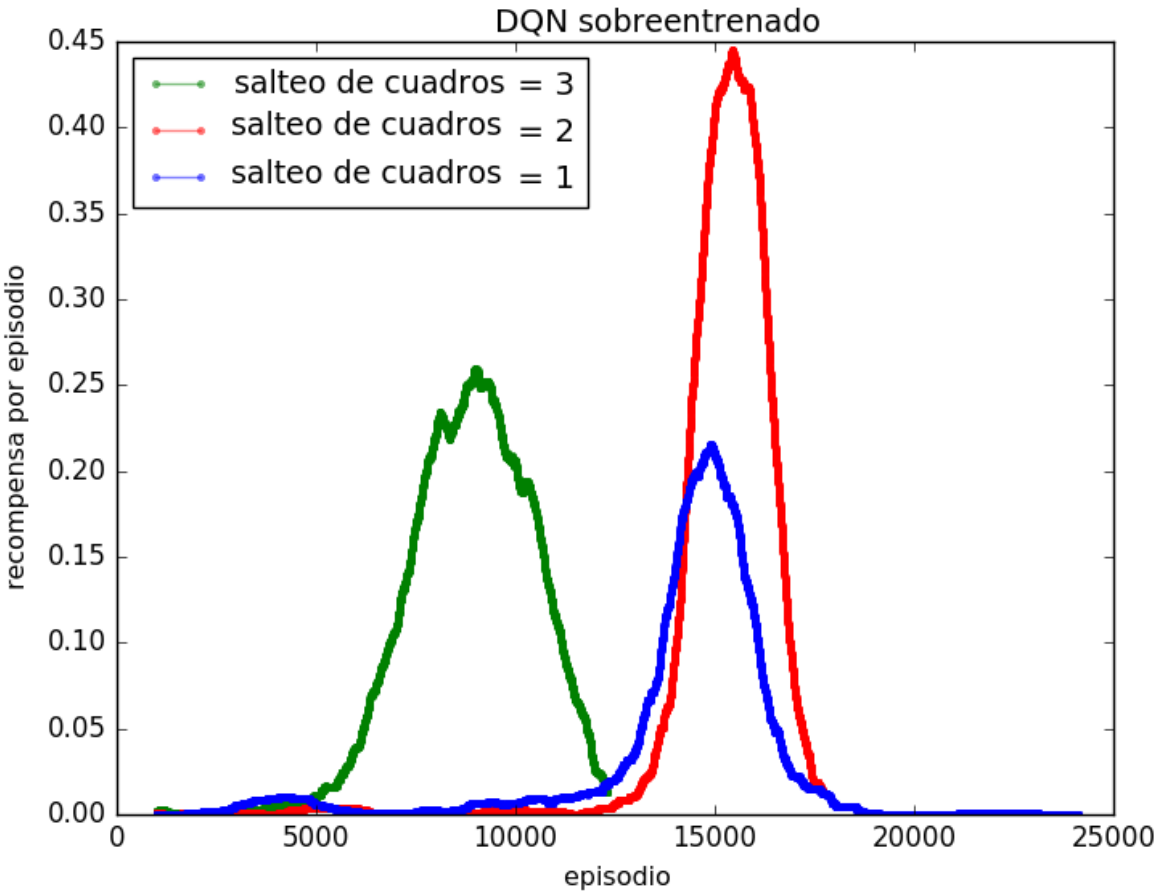


Figura 3.6: Recompensa en función de los episodios, con una media móvil de ventana 1000. Se observa que salteando de a 3 cuadros el aprendizaje comienza antes, mientras que salteando de a 2 se alcanza el mayor valor.

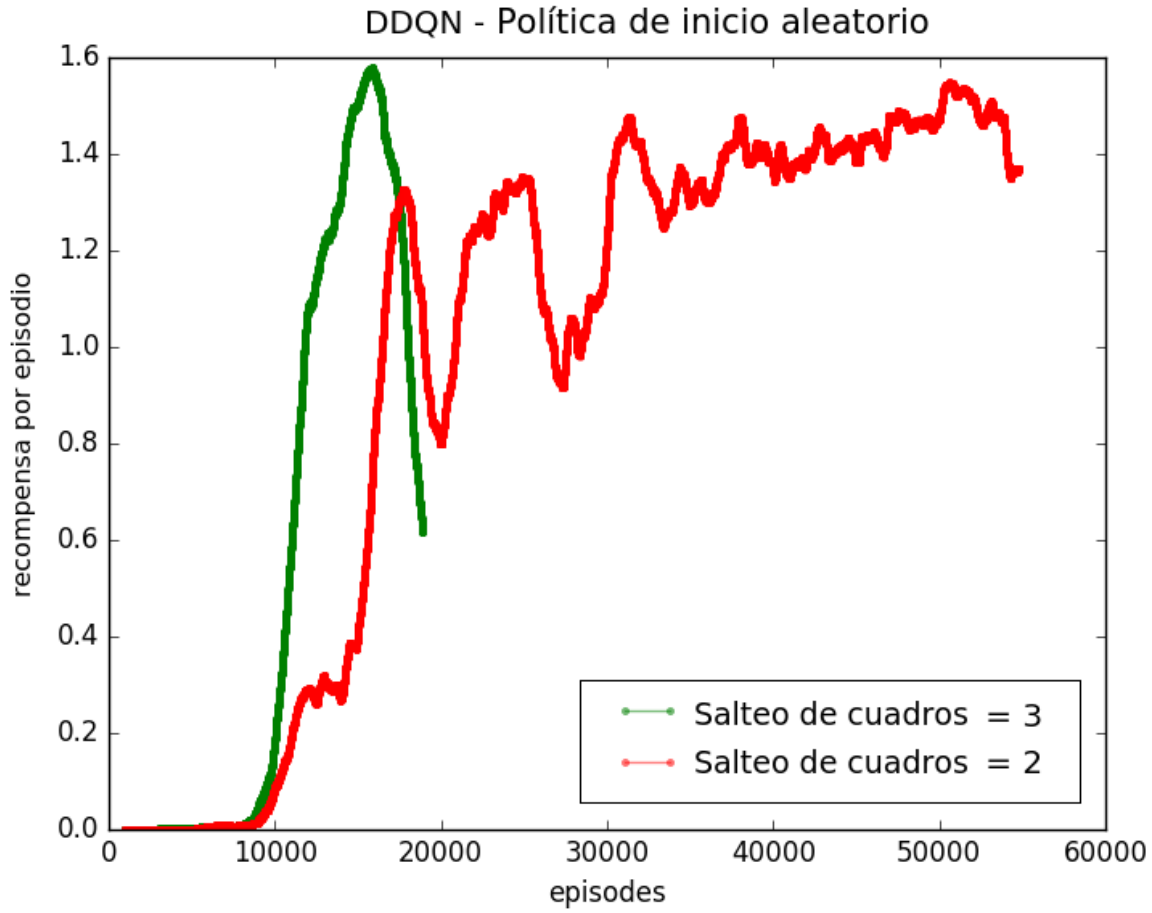


Figura 3.7: Recompensa en función de los episodios, con una media móvil de ventana tamaño 1000. Se corta la ejecución del agente con 3 cuadros de salteo debido a la abrupta caída en el desempeño. Para el salto de 2 cuadros, se observa la misma caída, sin embargo, como comienza a crecer se continúa su ejecución.

Con la prueba anterior, se valida la capacidad del agente para conseguir la primer llave, partiendo de un desconocimiento absoluto. Sin embargo aún no se concluye qué valor es mejor, dado que el agente posiblemente esté sobreajustado. Se decide repetir el experimento, esta vez cambiando DQN por DDQN e incluir el inicio aleatorio para disminuir la tendencia al sobreajuste. Únicamente se prueba saltear de a 2 y 3 cuadros, pudiendo ver los resultados en la figura 3.7. Para ambos, en muchos episodios la media móvil es superior a 1, llegando casi a 1,6: esto implica que prácticamente siempre toma la llave uno e incluso a veces una segunda: la llave[7] 4. Este resultado es ampliamente superior al del experimento previo, y es atribuible al uso de DDQN en vez de DQN.

Respecto al cambio en la forma de iniciar los episodios, el experimento anterior parte siempre del mismo estado inicial, por lo que la red perfectamente puede sobreajustar los parámetros, mientras que el nuevo experimento tiene un inicio aleatorio. Esto se ve reflejado en las recompensas que recibe cada agente durante su entrenamiento: el nuevo agente lleva casi 10000 episodios obteniendo cero, mientras que el anterior comienza a recibir recompensas positivas a partir de los 6000.

[7]Por causa de este experimento se decide quitar la llave cuatro, como se discute en la sección 3.4.

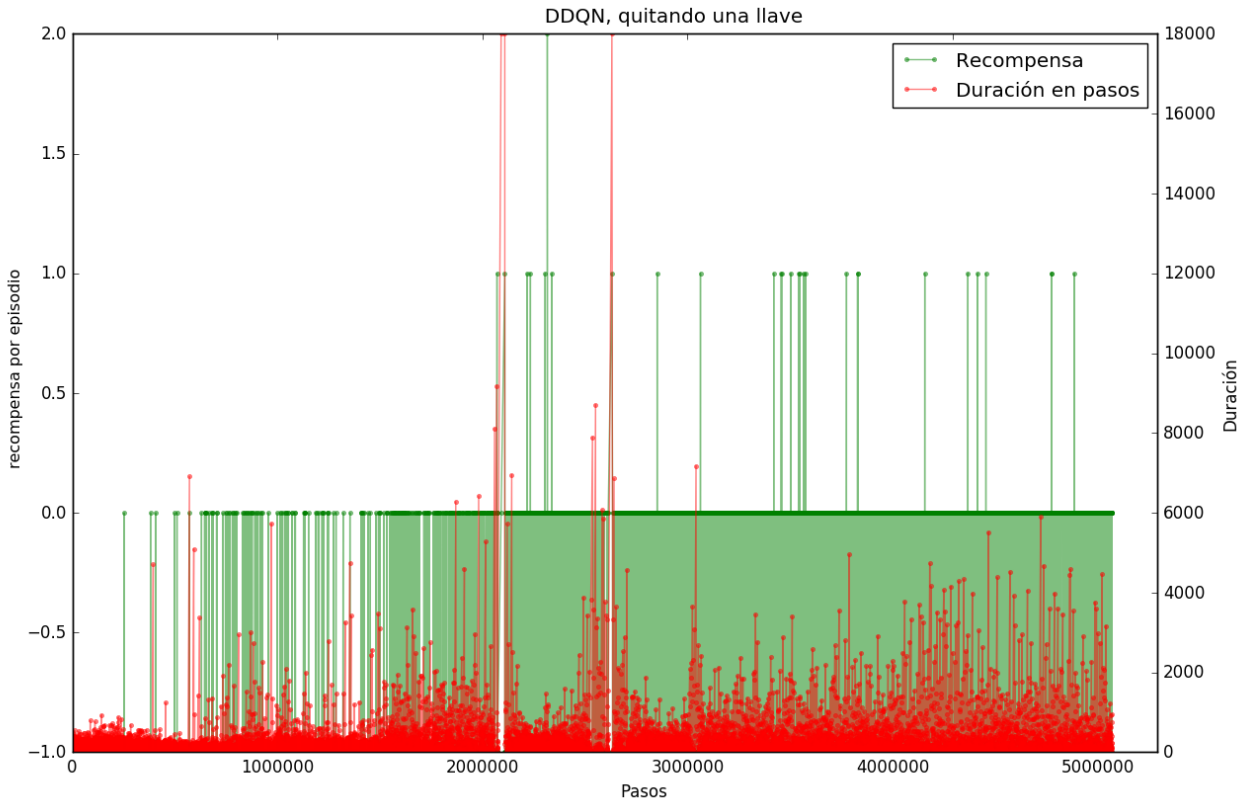


Figura 3.8: Recompensa obtenida y duración del episodio en pasos.

A partir de los resultados obtenidos, se fija el valor del salteo de cuadros igual a tres para los experimentos siguientes.

3.6. DQN, DDQN y Dueling

Teniendo definida la representación del estado, la política inicial y el valor de salteo de cuadros, se está en condiciones de probar la versión base de cada uno de los algoritmos a comparar: DQN, DDQN y Dueling. Para estas pruebas se continúa utilizando una memoria *Experience Replay* secuencial de un millón de entradas. Con base en estos experimentos se pretende concluir cuál de estos métodos se desempeña mejor en el juego Manic Miner.

Para entrenar el agente se utiliza el algoritmo DQN. DDQN y Dueling son pequeñas variantes de este algoritmo: en el caso de DDQN cambia la forma en que estimamos los valores de Q que se aprenden mientras que Dueling cambia la arquitectura de la red que se entrena. Cabe recordar que la arquitectura Dueling puede ser o no combinada con DDQN, pero dado que las ejecuciones son costosas, se prueba únicamente Dueling con DDQN.

En la gráfica de la figura 3.9 se observa la recompensa para cada una de las tres arquitecturas. Se puede ver que DQN requiere más pasos para comenzar a obtener recompensas que las otras dos alternativas. Además de ser más lento, las recompensas alcanzadas son menores a lo largo de todo el experimento, validando lo relevado en la bibliografía [42]. Comparando DDQN con Dueling, la diferencia es más sutil. Dueling comienza a crecer antes, aunque DDQN obtiene recompensas más altas cerca del final. Es de destacar que cada uno de los tres experimentos

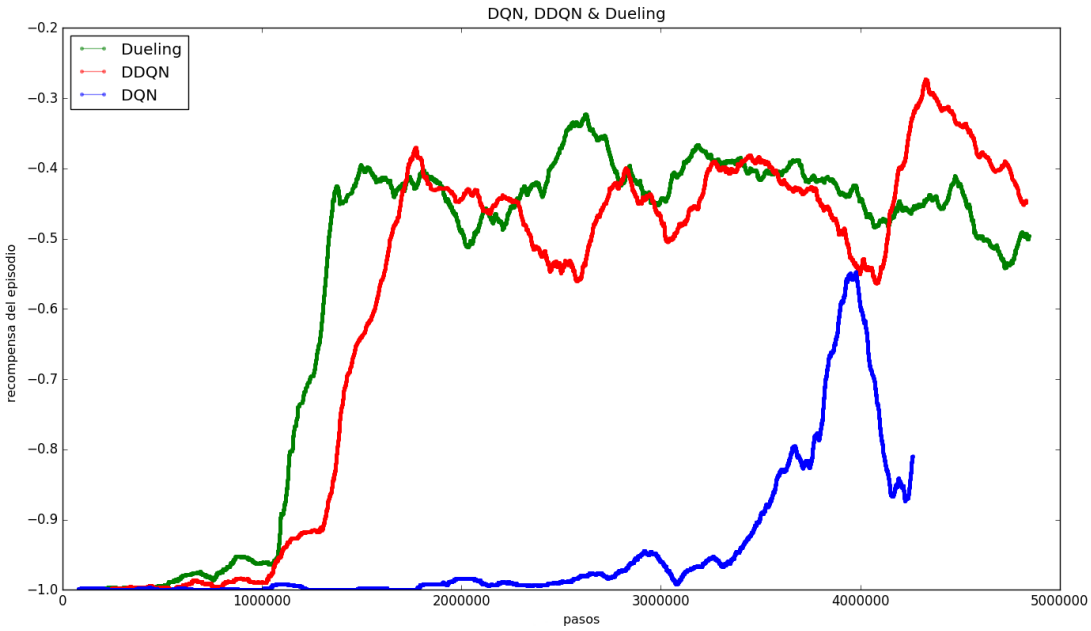


Figura 3.9: Recompensas por episodio para cada una de las tres arquitecturas consideradas. Se utiliza una ventana de tamaño 1000.

tomaron al menos seis días para su ejecución[8].

La figura 3.10 muestra la pérdida para cada arquitectura. Consistentemente con las observaciones anteriores, DQN mantiene un error alto casi hasta el final, mientras que DDQN y Dueling se encuentran siempre cercano a cero. De esto se deduce que DQN tarda más en ajustar su modelo de red al funcionamiento del entorno que los otros dos, requiriendo bastante más tiempo de entrenamiento para converger. Por una cuestión de escalas, se grafica en la figura 3.11 las pérdidas únicamente para DDQN y Dueling.

Al comparar DDQN con Dueling se puede ver que siguen siendo muy similares, aunque se notan pequeños matices. Dueling llega primero al máximo, lo que puede significar que aprende más rápido. Además, sobre la segunda mitad, su error está más cercano a cero, que desde el punto de vista de la red neuronal puede ser entendido de dos formas: como un mayor aprendizaje, o como que la red se está sobreajustando.

La figura 3.12 donde se grafica la función V confirma que las arquitecturas DDQN y Dueling tienen un comportamiento muy diferente al de DQN. En este caso se observa como DQN sobreestima los valores de la función V, cosa que DDQN y Dueling hacen en mucho menor medida. Recuérdese que Dueling también utiliza DDQN, lo cual explica sus similitudes.

Para confirmar la sobreestimación existente en DQN [42], se muestra en la figura 3.13 la evolución de la pérdida y de V promedio durante el entrenamiento para el agente DQN. Estas gráficas muestran cómo inicialmente ambos valores crecen. Como DQN está sobreestimando los valores de la función Q, los de V promedio también lo están, por lo que inicialmente el agente inicialmente cree que puede obtener una gran recompensa, pero en la medida que la red comienza a converger, V promedio decrece levemente y luego parece estabilizarse.

[8]6 días para DDQN y Dueling en un Intel core i5-4440 3.10GHz X4 con 16 GB de RAM y una GeForce GTX 1070, y 12 días para DQN en un Intel core i3-2100 3.10GHz con 12 GB de RAM y sin GPU.

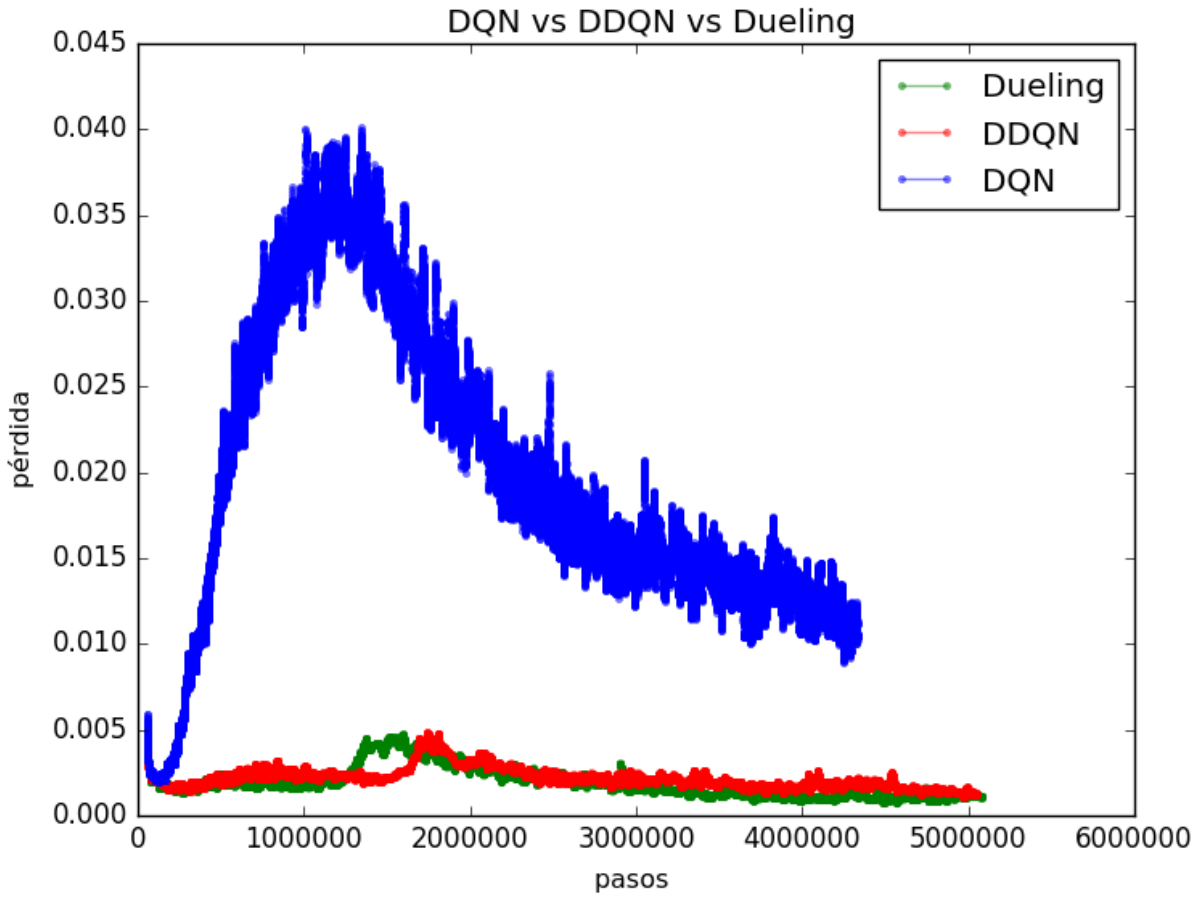


Figura 3.10: Función de pérdida para cada una de las tres arquitecturas consideradas. Se muestra la media móvil con ventana 50.

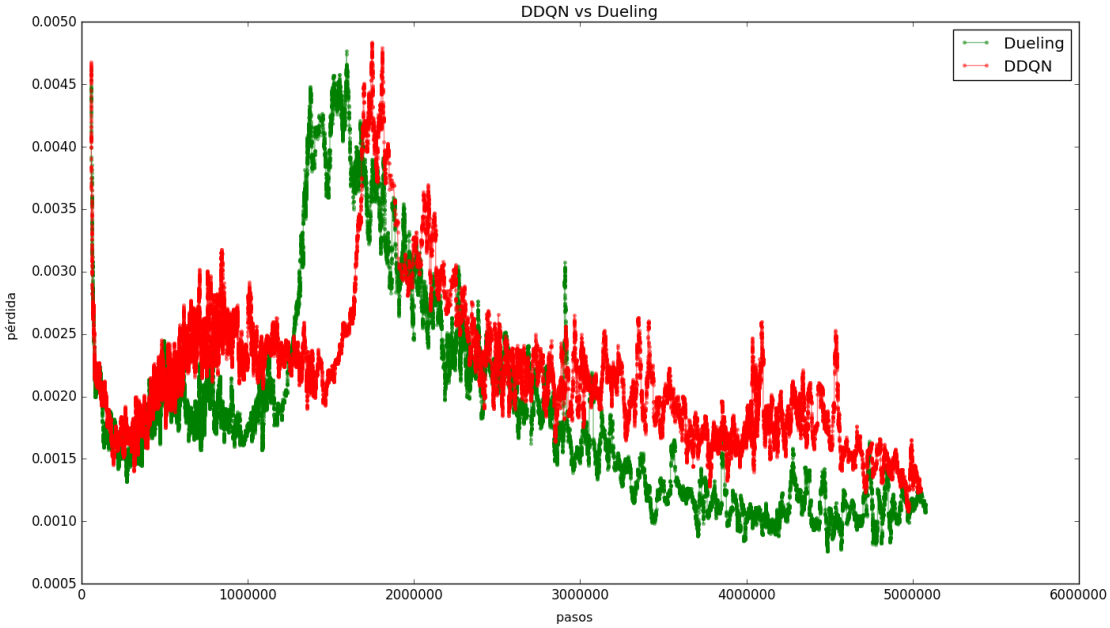


Figura 3.11: Función de pérdida para DDQN y Dueling. Se muestra la media móvil con ventana 50.

Hasta el momento, en los experimentos realizados el agente no llega a obtener llaves del nivel superior debido a que no puede superar al guardia. Durante el proceso de aprendizaje usualmente muere al intentarlo, porque requiere una secuencia de movimientos que aún desconoce para lidiar con el piso verde que tiene un comportamiento diferente a los demás. Se desprende de lo anterior que existe un problema asociado a una pobre exploración del entorno por parte del agente. En las siguientes secciones se aplican diferentes técnicas para intentar superar esta dificultad.

3.7. Priorización de experiencias en el entrenamiento

Al priorizar las experiencias almacenadas en la memoria, se puede entrenar más frecuentemente con aquellas que el agente aún no comprende del todo, a las que se les asigna mayor prioridad al almacenarlas. Esto es una mejora importante desde el punto de vista teórico respecto a la memoria en la que todas las experiencias tienen la misma prioridad. Se espera concluir si en la práctica, para el juego Manic Miner, esta priorización de experiencias influye positivamente en el desempeño del agente. En esta sección se comparan los efectos de utilizar una memoria que prioriza las experiencias sobre una que no lo hace, contemplando tres indicadores: recompensa, puntaje y pérdida.

El mecanismo de priorización consiste en asociar a cada estado almacenado en la memoria una probabilidad de muestreo proporcional al error obtenido la última vez que se lo utiliza para entrenar. Esto permite que se entrene con mayor frecuencia sobre los ejemplos que, a juzgar por el error, aún no son bien estimados por la red y se considera que tienen más para enseñar.

Para las pruebas de esta sección se emplean dos agentes que utilizan la arquitectura Dueling:

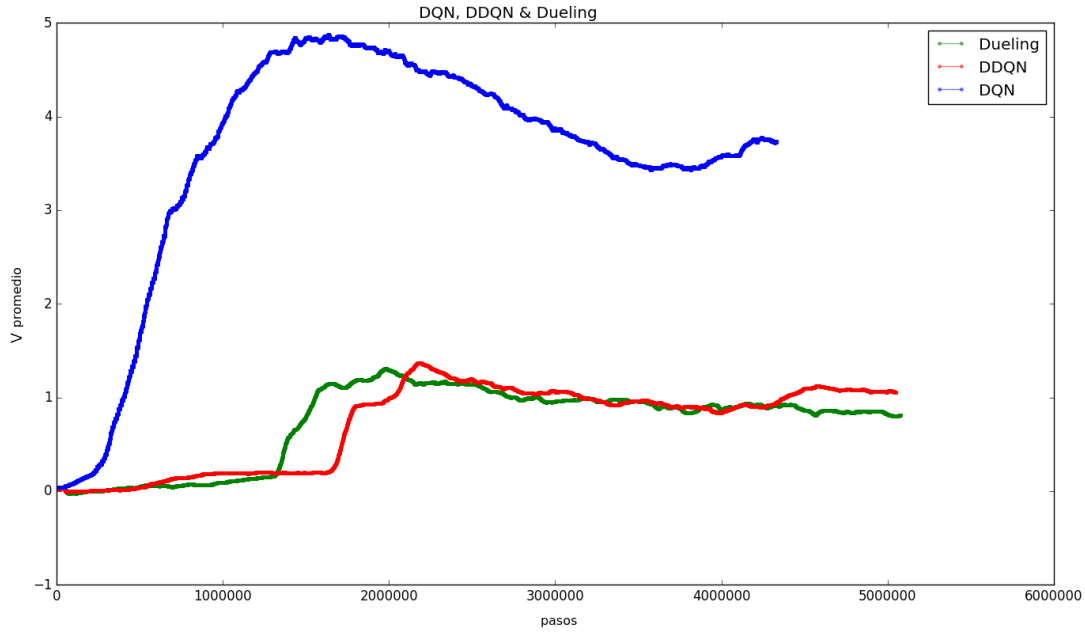


Figura 3.12: V promedio para cada una de las arquitecturas. Se muestra la media móvil con ventana 50.

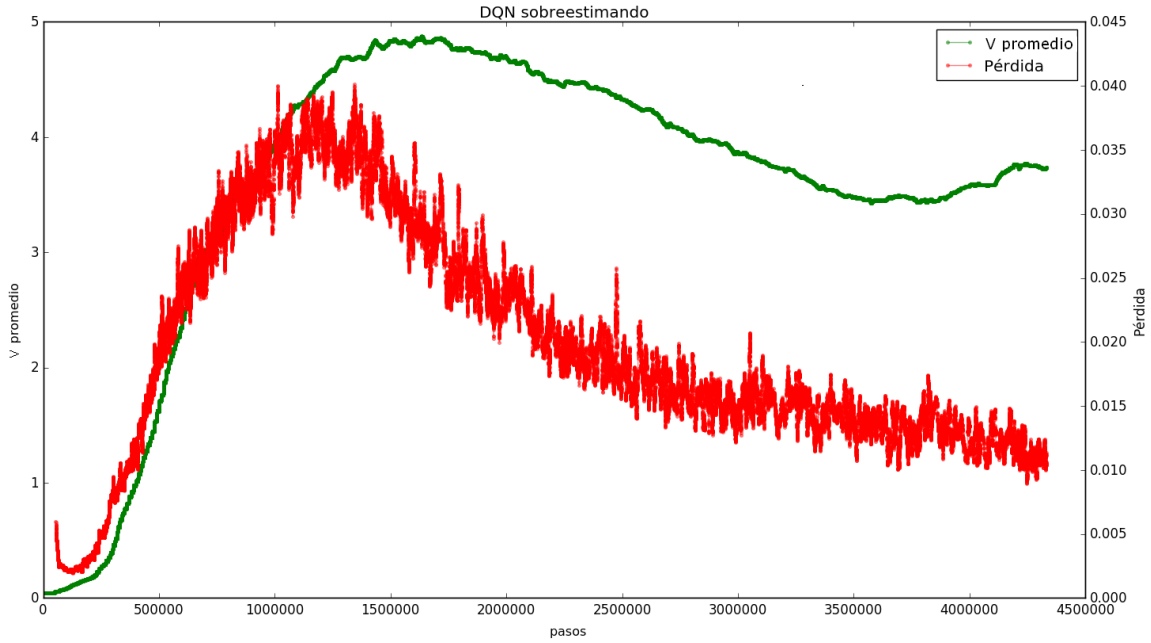


Figura 3.13: V promedio y pérdida para DQN. Puede notarse que a medida que la red se ajusta, el error disminuye y también desciende V promedio. Se muestra la media móvil con ventana 50.

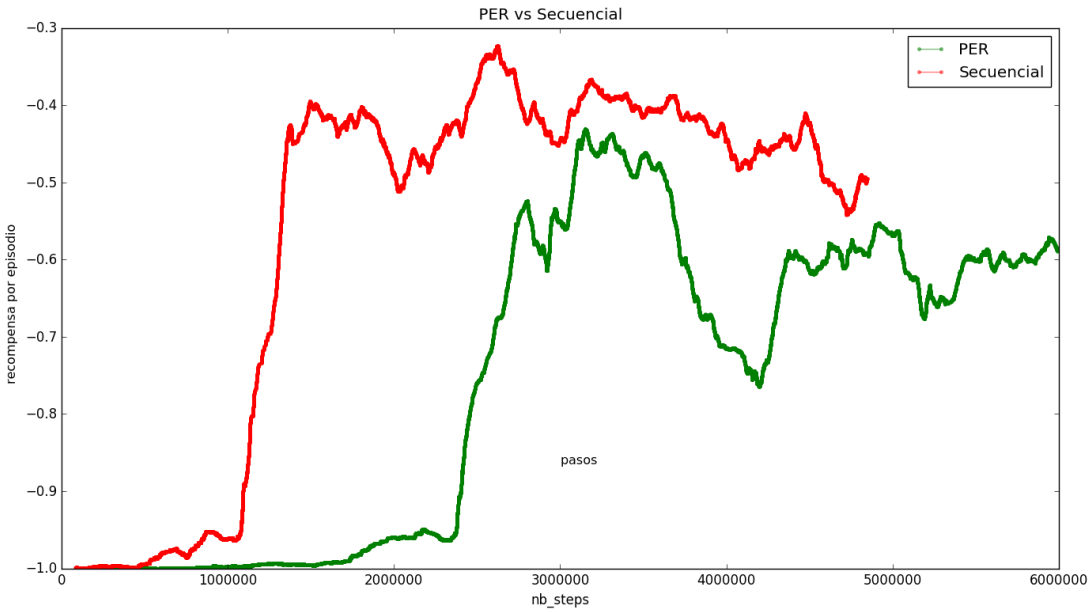


Figura 3.14: Recompensa para Dueling implementado con y sin PER, con media móvil tamaño 1000

uno emplea una memoria secuencial y otro una memoria PER, ambas con capacidad para almacenar el último millón de experiencias.

Como se mencionó anteriormente, dado que los experimentos previos no alcanzan todas las llaves por problemas atribuidos a la falta de exploración, se decide decrementar el valor de ε durante dos millones de iteraciones para el agente que utiliza PER, mientras que el otro agente continúa únicamente por un millón, como los demás agentes. Esto esencialmente prolonga la etapa de exploración, aumentando las posibilidades de llegar a recolectar otra llave. Esto implica una diferencia en la cantidad de pasos necesarios para comenzar a explotar el conocimiento por parte de los agente; sin embargo, el cambio hace que la comparabilidad entre los experimentos sea cuestionable.

Contemplando las recompensas obtenidas durante el entrenamiento, pese a lo que sugiere la bibliografía, PER parece obtener peores resultados en cuanto a la recompensa promedio comparado con la memoria secuencial, aunque por un pequeño margen. Sin embargo, como muestra la figura 3.15, PER presenta algunos episodios aislados en los que obtiene mayor recompensa que la memoria uniforme. El máximo puntaje lo alcanza en cinco oportunidades en las que logra obtener 300 puntos, equivalente a recolectar tres llaves.

En la figura 3.16 se observa la evolución del error. Pese a que ambas convergen, en el caso de la memoria secuencial la convergencia parece ser más rápida. Esta diferencia es esperable ya que PER le da prioridad a las experiencias que tienen más por enseñar. Como se toma el error TD para aproximar cuánto se puede aprender de una experiencia, las que tienen mayor error son las que se utilizan para entrenar con más frecuencia. Debido a esto, el uso de PER hace que para entrenar se tomen más cantidad de experiencias de mayor error respecto a la memoria secuencial, y en cada iteración se ajusta lentamente a ellas.

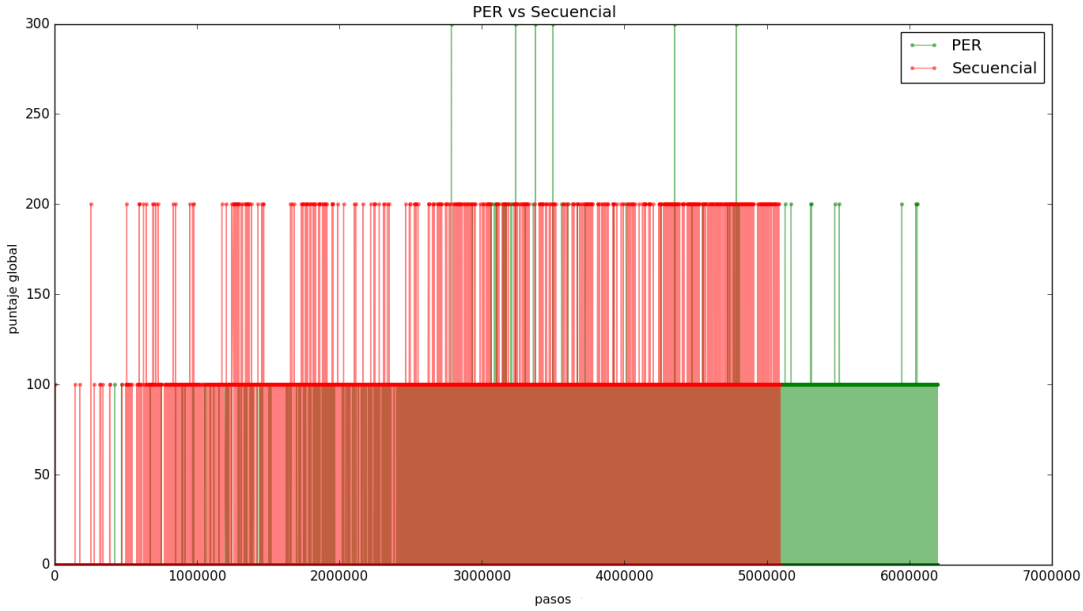


Figura 3.15: Puntaje para Dueling implementado con y sin PER

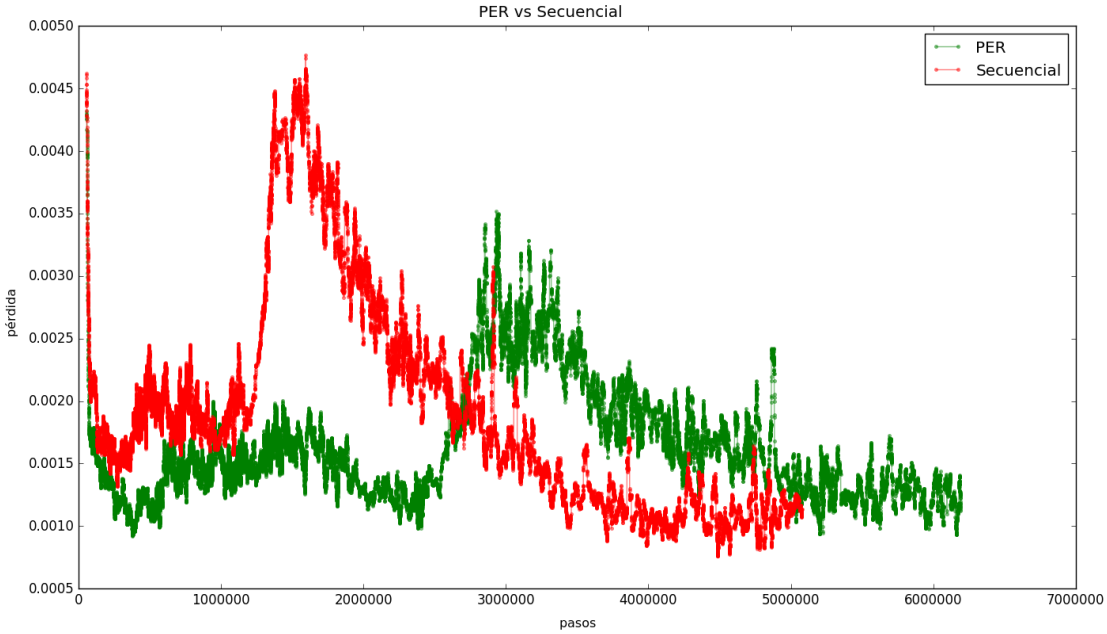


Figura 3.16: Pérdida para Dueling implementado con y sin PER.

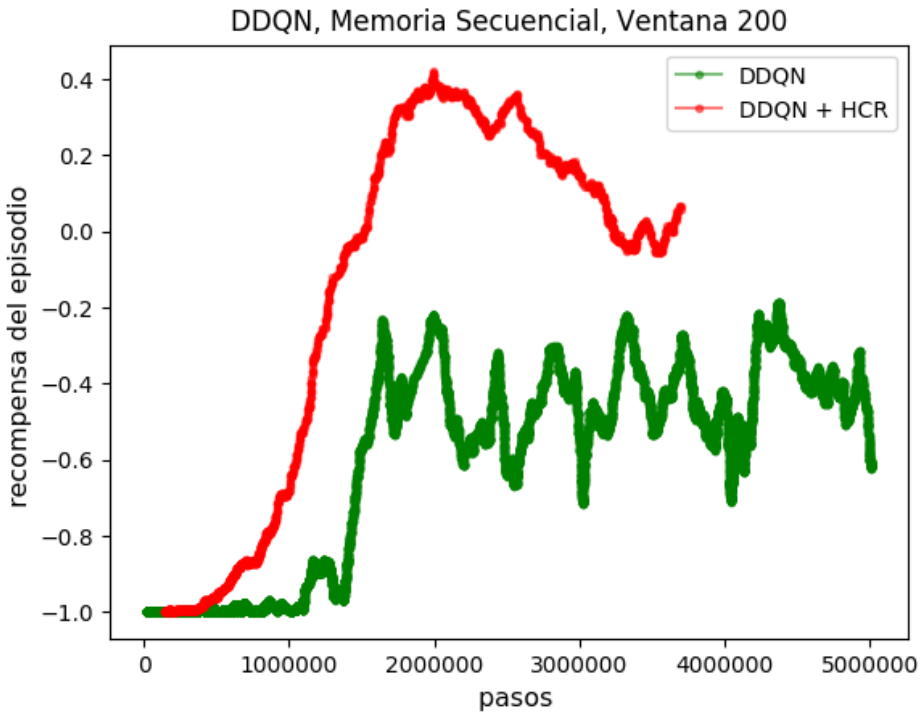


Figura 3.17: *Recompensa con y sin HCR.* Se muestran únicamente los episodios que empiezan en el *checkpoint* inicial para hacer ambos registros comparables.

.

3.8. *Human checkoint replay*: HCR

En vista que el agente no logra recolectar todas las llaves, obteniendo en contados episodios más de una, se intuye que el problema se debe a una pobre política de exploración. Debido a esto se decide experimentar con DDQN en conjunto con *Human checkoint replay* (HCR).

Esta técnica consiste en tomar algunos *checkpoints* por los que un jugador humano pasa al jugar, y proporcionárselos al agente para que comience su entrenamiento a partir de ellos. De esta forma, el agente puede llegar a estados a los que por sí solo no llegaría o tardaría más en hacerlo. Esto asiste al agente en la exploración y acelera el proceso de aprendizaje. Sin embargo, este método introduce conocimiento humano específico del juego en el entrenamiento, por lo que deja de ser un método general.

Se comparan dos agentes DDQN con y sin HCR, ambos con una memoria secuencial de un millón de experiencias. Para el agente que utiliza HCR se le proveen 17 *checkpoints* distribuidos a lo largo del recorrido de un jugador humano en el proceso de superar el nivel, con el primero en la posición inicial y el último cerca del portal tras recolectar todas las llaves. Se busca que entre ellos exista aproximadamente la misma distancia en pasos, aunque no es estrictamente necesario.

En la figura 3.18 se puede ver la pérdida a lo largo del entrenamiento. Esta gráfica arroja un resultado interesante: la pérdida para HCR se dispara para luego acercarse a cero. Esta divergencia inicial puede atribuirse a que, al visitar estados más diversos, puede percibir que su modelo no es el adecuado, adaptándolo para que cubra estas nuevas experiencias. En contra-

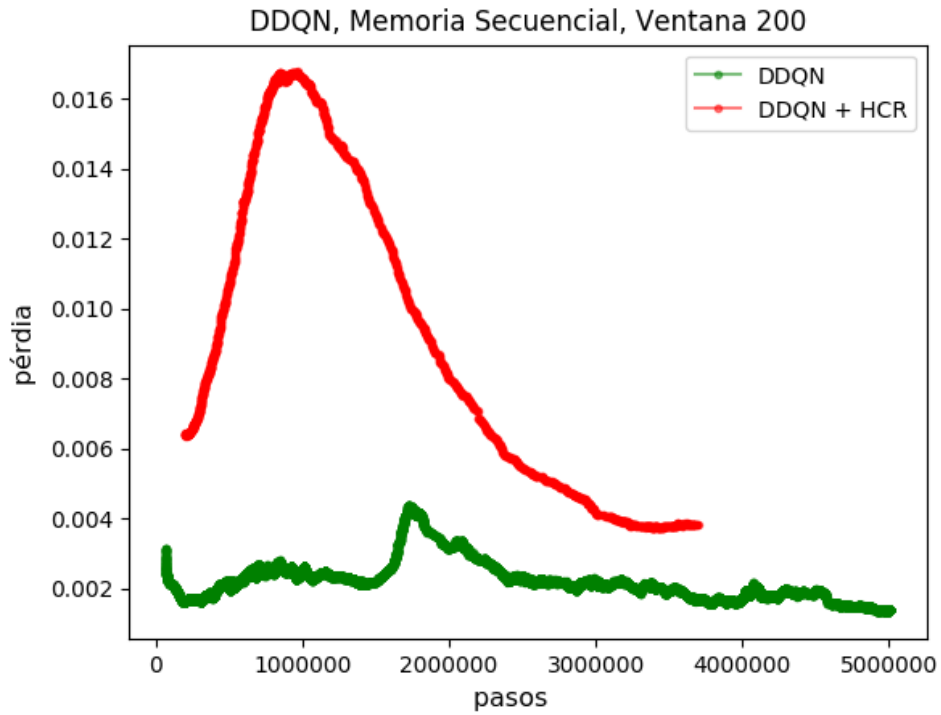


Figura 3.18: Pérdida con y sin HCR.

partida se observa que la pérdida sin HCR no se despega mucho del eje x, coincidiendo con que los estados que visita nunca se apartan mucho entre sí, convergiendo lentamente a cero para esos estados.

En la figura 3.19 se muestra la función V promedio para estados próximos al comienzo. Durante todo el entrenamiento la valoración de los estados iniciales[9] es mayor para HCR. Esto se atribuye a que se descubre antes que los estados iniciales pueden obtener retornos más elevados que sin HCR no se alcanzan.

Por último, en la figura 3.20 se observa que partiendo desde la posición inicial, sin utilizar HCR el agente no logra recolectar más que la primera llave. Al agregar HCR, no solo se toman todas las llaves, sino que se alcanza el portal en varias ocasiones. Resulta curioso ver que utilizando HCR nunca termina un episodio con 4 de recompensa. La explicación a esto es que para terminar con 4 debe haber recolectado todas las llaves y luego morir por *time-out*, que está fijado en 1800 pasos. Esto último nunca ocurre por lo que los posibles desenlaces son: alcanza el portal y se le suma uno a la recompensa total llegando a 5 o muere en el camino restándosele 1, pasando a tener 3.

Evaluando ambos agentes en el mejor juego de pesos y ejecutando el test 30 veces, se observa que el agente DDQN sin HCR en todos los episodios obtiene exactamente una llave y luego muere por *time-out*. Para el caso de HCR, se obtiene 2,8 de promedio, casi triplicando la cantidad del primero. De estas 30 veces llega 6 veces al portal, y en todas las ejecuciones obtiene al menos una llave. Se puede ver un histograma de la cantidad de episodios para el mejor conjunto de pesos del agente con HCR en la figura 3.21. Vale resaltar que para comparar ambos métodos se entrenan dos agentes exactamente iguales, excepto por la utilización o no de HCR.

[9]Se puede decir que son estados cercanos a la posición de inicio por la forma de ser recolectados.

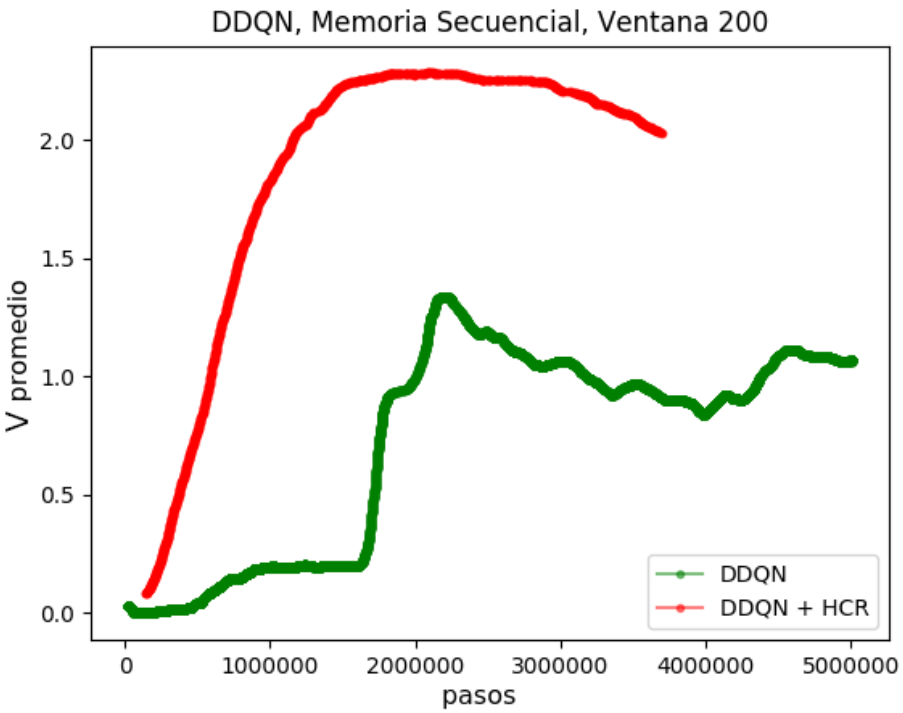


Figura 3.19: Recompensa con y sin HCR.

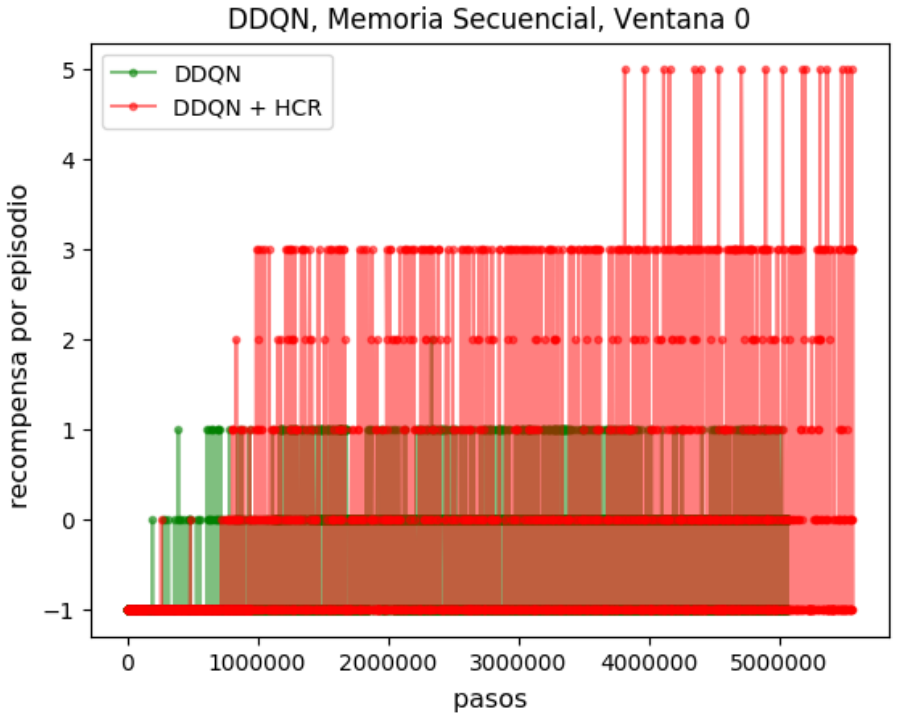


Figura 3.20: Recompensa con y sin HCR.

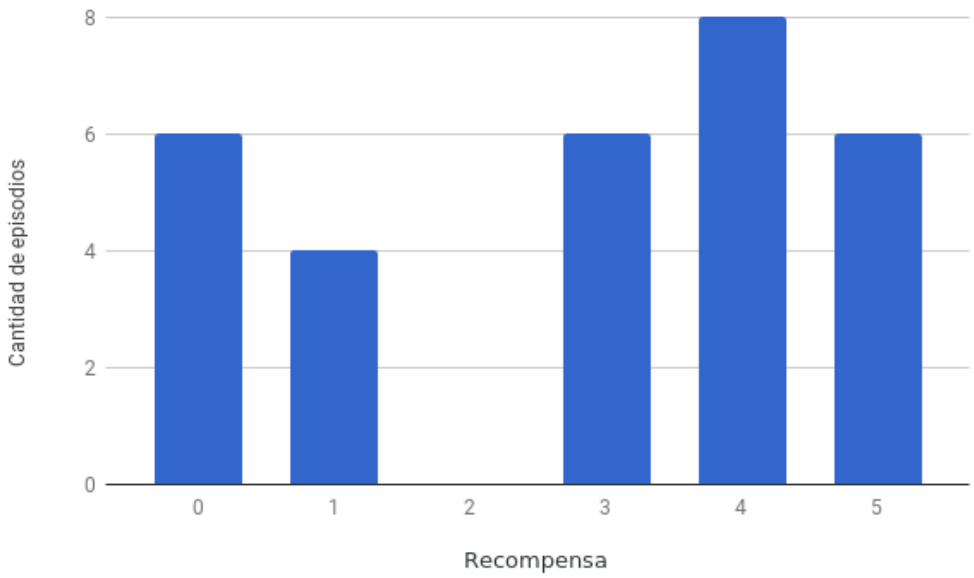


Figura 3.21: *Histograma con HCR.* Cantidad de episodios agregados según recompensa obtenida. No hay ocurrencias con recompensa en -1 debido a que todas las ejecuciones toman al menos una llave.

Se puede ver el impacto que tiene la técnica HCR sobre la recompensa obtenida comenzando desde el inicio del nivel. Se deduce de esto que una de las principales limitantes del agente es la exploración; la estrategia ϵ-*greedy* resulta insuficiente dada la complejidad que el juego plantea y el escaso tiempo de entrenamiento disponible. Por esta razón no se descarta que si se deja el agente entrenando unas 10 veces más, cincuenta millones en lugar de cinco millones de pasos, tal y como en trabajos previos [29], eventualmente aprenda a recolectar las llaves del nivel superior. Se debe considerar además el desafío no menor, ya que no cuenta con recompensas intermedias, de aprender a bajar hasta el portal.

3.9. Desempeño de los agentes

En esta sección se analiza y compara el desempeño de los agentes en función del puntaje alcanzado. Para cada método evaluado se utiliza la metodología descrita en la sección 3.3.2, ejecutando 30 episodios, para obtener los mejores pesos postentrenamiento.

En el cuadro 3.2 y 3.3 se muestran para cada método los resultados de los pesos que maximizan el promedio de recompensa obtenida y la cantidad de veces que se supera el nivel respectivamente. En la figura 3.22 se visualizan los datos del primer cuadro en forma gráfica.

En las gráficas se aprecia como los métodos que cuentan con HCR consiguen notoriamente más recompensa que los que no lo utilizan. Sin HCR no se logra recoger más de una llave en las pruebas. Además se ve cómo los métodos DDQN y Dueling son más eficientes en el entrenamiento que DQN simple.

DDQN en conjunto con PER y HCR, con diez millones de pasos de entrenamiento, es el agente con mejor desempeño, alcanzando superar el nivel en más de la mitad de las pruebas y

Método	Recompensa promedio	Desviación estándar	Pasos de entrenamiento	Entrenamiento total
DDQN PER HCR	4,13	0,85	9.550.000	9.550.000
DDQN HCR	2,30	0,70	2.600.000	6.650.000
DDQN	1,00	0,00	1.950.000	4.225.000
Dueling PER	1,00	0,00	2.725.000	6.175.000
DQN	0,97	0,18	3.875.000	4.325.000
Dueling	0,93	0,24	2.575.000	5.000.000

Cuadro 3.2: Recompensas obtenidas en el nivel uno para diferentes agentes y técnicas. En la columna *Pasos de entrenamiento* se muestran los pasos de entrenamiento con los que se obtiene la mejor recompensa promedio, la columna *Entrenamiento total* muestra el total de pasos de entrenamiento del experimento.

Método	Cantidad veces pasa nivel	Pasos de entrenamiento	Entrenamiento total
DDQN PER HCR	18	7.300.000	9.550.000
DDQN HCR	13	6.500.000	6.650.000
DDQN	0	-	4.225.000
Dueling PER	0	-	6.175.000
DQN	0	-	4.325.000
Dueling	0	-	5.000.000

Cuadro 3.3: Cantidad de veces que se supera el nivel uno. En la columna *Pasos de entrenamiento* se muestra la cantidad de pasos con que se supera más veces el nivel. La columna *Entrenamiento total* muestra el total de pasos de entrenamiento del experimento.

obteniendo más de 4 llaves en promedio[10].

A continuación se describe el comportamiento del agente con los pesos en el que más veces supera el nivel en las pruebas. Desde la posición inicial camina por debajo hasta pasar la primer planta, esto evita el riesgo de tocarla o tocar el guardia al esquivarla. Luego sube y toma la llave uno. Salta al piso deslizante y sube al muro amarillo. Sin detenerse, vuelve a bajar al piso deslizante e inmediatamente salta para evitar al guardia. Después de esto, asciende y toma las llaves 2 y 3 realizando saltos diagonales para ambas, lo cuál es más rápido que saltar verticalmente. Luego de tomar la llave cinco se deja caer en el piso desmoronante sobre el muro de ladrillos. Baja al piso deslizante por el lado izquierdo y de ahí salta nuevamente al guardia. Finalmente baja al nivel inferior para caminar en linea recta hacia el portal. Las veces que no llega a pasar el nivel muere generalmente por hacer contacto con el guardia tratando de saltarlo. En la imagen 3.23 se encuentra una ilustración de este recorrido.

En algunas ocasiones Willy se queda quieto unos segundos sin realizar ninguna acción para luego de algunos pasos, reanudar el juego. Respecto al tiempo que permanece inmóvil, se cree que el agente espera una observación del entorno conocida, con el guardia en alguna posición específica, de alguna forma *alineado* con un estado desde donde se alcanzan recompensas positivas. Probablemente las veces que muere se deba a que la función todavía no se ajusta completamente. Este número se reduce a medida que el agente entrena, a costa de sobreajustarse cada vez más, debido a la naturaleza determinista del problema.

[10]Contando el portal como una llave, 5 entonces en total.

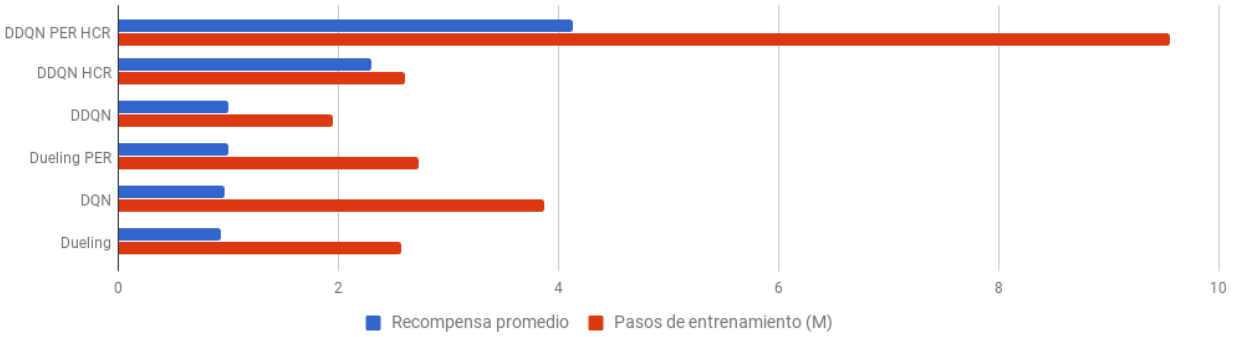


Figura 3.22: Recompensa promedio y pasos de entrenamiento.

Figura 3.23: Recorrido que generalmente hace el mejor jugador. El color azul corresponde al camino para recoger las llaves y el color verde al camino de vuelta hacia el portal.

3.10. Sobreajuste

Un punto interesante de los agentes desarrollados con métodos DQN, es conocer si son capaces de identificar los objetos en su entorno, clasificándolos como deseables o indeseables. Lo anterior es un buen indicador de si el agente generaliza su conocimiento del entorno o, por el contrario, simplemente memoriza una secuencia de acciones a realizar en cada instante.

Típicamente en un problema de aprendizaje se separa una parte de los datos sobre los que no se entrena, y se los utiliza para evaluar la calidad del aprendizaje luego del entrenamiento. Análogamente se puede seleccionar un conjunto de niveles sobre los que entrenar, y luego otro más pequeño sobre el que evaluar. Un jugador humano logra inferir rápidamente características de los niveles desconocidos luego de experimentar con unos pocos, pese a que las formas y colores de los objetos varíen ligeramente. Sin embargo, debido a que entrenar en un nivel hasta aprender un comportamiento aceptable requiere varios días de entrenamiento, hacerlo en varios niveles va a necesitar aun más tiempo, por lo que se descarta esta opción.

Como alternativa se opta por evaluar al jugador sobre variaciones del primer nivel. De esta forma se puede observar si el agente reconoce los diferentes elementos del nivel (como plantas, guardias, llaves, etc.) adecuando su juego a la nueva disposición planteada. Para esto, haciendo uso del editor de niveles detallado en la sección 4.1.3, y se pone a prueba el agente en el nuevo escenario.

Debido a que algunos de los nuevos escenarios tienen llaves que realmente son sencillas de tomar, no alcanza con analizar el puntaje obtenido por el agente en ellos para entender en profundidad su capacidad de generalizar: se debe observar su comportamiento para decidir si está o no sobreajustado. Basándose en las acciones tomadas por el agente durante cinco episodios, un observador intenta discernir si realmente tiene la capacidad de generalizar o está sobreajustado al nivel de entrenamiento. En los trabajos previos no se han relevado pruebas de esta naturaleza.

Para la realización de estas pruebas se considera al agente que supera el primer nivel la mayor cantidad de veces. Como se observa en el cuadro 3.3, este es un agente que utiliza DDQN con PER y HCR, entrenado durante 7300000 pasos. Se lo hace jugar durante cinco rondas en cada uno de los niveles editados, comenzando con una cantidad aleatoria de entre 0 y 30 *no-ops*. Se somete el agente a 4 modificaciones diferentes del primer nivel. Por sus características, bautizamos estos escenarios de la siguiente manera: Salida directa, Zig zag, Eliminación de trampas y Obstáculos añadidos.

Salida directa

El escenario que plantea la figura 3.24 es de los más sencillos que se puede concebir. Partiendo del estado inicial está claro que la política óptima es moverse en línea recta hacia la derecha, sin necesidad de saltar ni de correr ningún riesgo.

El agente solo recolecta la primera llave y salta al primer nivel. Luego comienza un paso errante como queriendo tomar las llaves de abajo, sin conseguir descender.

Figura 3.24: Se disponen todas las llaves de manera tal que para superar el nivel basta con caminar en linea recta hacia el portal.

Figura 3.25: Se disponen todas las llaves de modo tal que no sea necesario eludir al guardia para superar el nivel, pero si realizar algunos saltos.

Zig zag

En la figura 3.25 se tiene un escenario similar al primero salvo que las llaves están dispuestas en zig zag, de manera que no es suficiente con avanzar para superar el nivel, sino que además hay que saltar. Existe una llave que permanece en el lugar original, la llave uno.

Willy recolecta las dos llaves inferiores y se dirige al portal donde comienza a saltar y moverse dentro de esa región. En 3 de las 5 pruebas logra tomar la llave que se encuentra más a la derecha, y luego muere con el arbusto de la cinta transportadora. En ninguna de las pruebas toma la llave que está debajo del guardia, ni intenta subir al último nivel.

Eliminación de trampas

En el escenario planteado en la figura 3.26 se han removido todas las trampas del nivel: no hay plantas ni picos de hielo. Si bien el comportamiento general no debería cambiar mucho, al llegar al nivel superior sería innecesario saltar. Además, la llave cuatro, eliminada[11] durante el entrenamiento, puede ser perfectamente la segunda en ser recolectada.

Como resultado, en este escenario se obtiene todas las llaves en 3 de las 5 pruebas realizadas, aunque en ninguna consigue superar el nivel. La llave cuatro la recolecta cuando llega al nivel superior y no cuando entra a la cinta transportadora. Se puede observar que le es más difícil movilizarse a lo largo del nivel, quedándose quieto en varias oportunidades y volviendo hacia

[11] La llave que está resaltada en la figura 3.5.

Figura 3.26: Se eliminan todas las plantas y picos de hielo manteniendo el resto del entorno sin modificaciones.

Figura 3.27: Análogo al escenario Zig zag pero con obstáculos añadidos en el camino que deben ser evadidos para superar el nivel.

atrás lo que provoca la muerte por aire en 3 de 5 oportunidades. Donde había plantas aún continúa saltando.

Obstáculos añadidos

Para finalizar, se disponen las llaves también en zig zag, pero esta vez cercanas a plantas y picos de hielos, como se aprecia en la figura 3.27.

En todas las pruebas solo toma la primera llave y luego muere debido al primer pico de hielo o la primera planta.

Se desprende de cada escenario que existe una gran tendencia al sobreajuste del método DDQN. Probablemente el mal desempeño del agente en las pruebas propuestas se deba a varios factores: por un lado, durante todo el entrenamiento el nivel es relativamente estático, a excepción del guardia que se mueve de forma completamente predecible. A esto hay que sumarle que, como consecuencia del preprocesamiento del cuadro, las llaves meramente pasan a ocupar 5 píxeles cada una y las plantas 4, en un total de aproximadamente 7 mil (84×84) del total. Como consecuencia las llaves y las plantas, que son estáticas, sean tal vez irrelevantes a la hora de ajustar los pesos de la red, habiendo otros píxeles más importantes, como los de Willy que son 18 y además cambian constantemente de lugar. También cabe la posibilidad de que la red se esté ajustando al cuadro en su totalidad, sin diferenciar objetos, memorizando secuencias que impliquen llevar a Willy a determinados puntos de la caverna.

Aprender una secuencia fija óptima de teclas es una forma válida de superar el nivel, si se

consigue el objetivo de maximizar la recompensa obtenida. Quizá, si durante el entrenamiento tuviera la posibilidad de conocer distintos niveles, el agente podría aprender mejores representaciones de los objetos, aumentando su capacidad de generalización. Si se opta por entrenar en un único nivel, una posibilidad es cambiar los objetos de lugar para entrenar, de manera que la red tenga más oportunidad de reconocerlos y entender sus efectos. Un solo nivel de Manic Miner tiene muy pocos cambios durante los episodios: solo Willy cambia de forma notoria, ya que el guardia lo hace de forma completamente predecible. Esto no sucede en juegos como *Breakout* o *Enduro* en Atari: sobreajustarse a los cuadros es más difícil ya que estos juegos intrínsecamente presentan mayor variabilidad.

Capítulo 4

Ambiente de pruebas

El proyecto plantea la creación de un agente capaz de aprender a jugar Manic Miner. Para esto es necesario contar, por un lado, con un entorno que implemente el juego permitiendo su interacción con el agente y, por otro, con un agente que sea capaz de aprender a jugarlo. En este capítulo se presentan los principales componentes que dan soporte a las pruebas presentadas en el capítulo anterior, planteando los principales problemas enfrentados junto a las decisiones de implementación tomadas.

Dada la inexistencia de una plataforma que permita la interacción agente-entorno con juegos de ZX Spectrum, es necesario implementarla. Para esto se adapta un emulador de ZX Spectrum y se define la forma en la que agente y entorno se comunican. Durante este proceso se crea, además, un entorno interactivo que entre otras cosas facilita la realización de varias de las pruebas.

Este capítulo cuenta con dos secciones, una para cada una de los componentes utilizados. En la sección 4.1 se detalla la implementación del Entorno, mientras que en la 4.2 la del agente. En ambas componentes se reutilizaron trabajos previos. En la figura 4.1 se puede distinguir cuáles componentes fueron desarrolladas completamente durante el proyecto y cuáles se tomaron de trabajos previos.

4.1. Entorno

El entorno es el componente sobre el que un agente realiza acciones para lograr su objetivo. Por cada acción, el entorno devuelve un retorno que consiste en una recompensa y una observación correspondiente al nuevo estado. El entorno es quien define las reglas de juego y la función de recompensa, la cuál puede ser modificada antes de ser recibida por el agente.

En este proyecto el entorno se corresponde con el juego Manic Miner y la observación con el cuadro de la pantalla. Las acciones son los comandos válidos para el juego: saltar, izquierda, derecha, saltar-izquierda, saltar-derecha y no hacer nada. La recompensa se define a grandes rasgos como la diferencia de puntaje luego de realizar una acción. No es evidente pero el estado del entorno, que no es accesible para el agente, corresponde a la memoria RAM del emulador. Para contar con un entorno que represente un videojuego con el que un agente pueda interactuar,

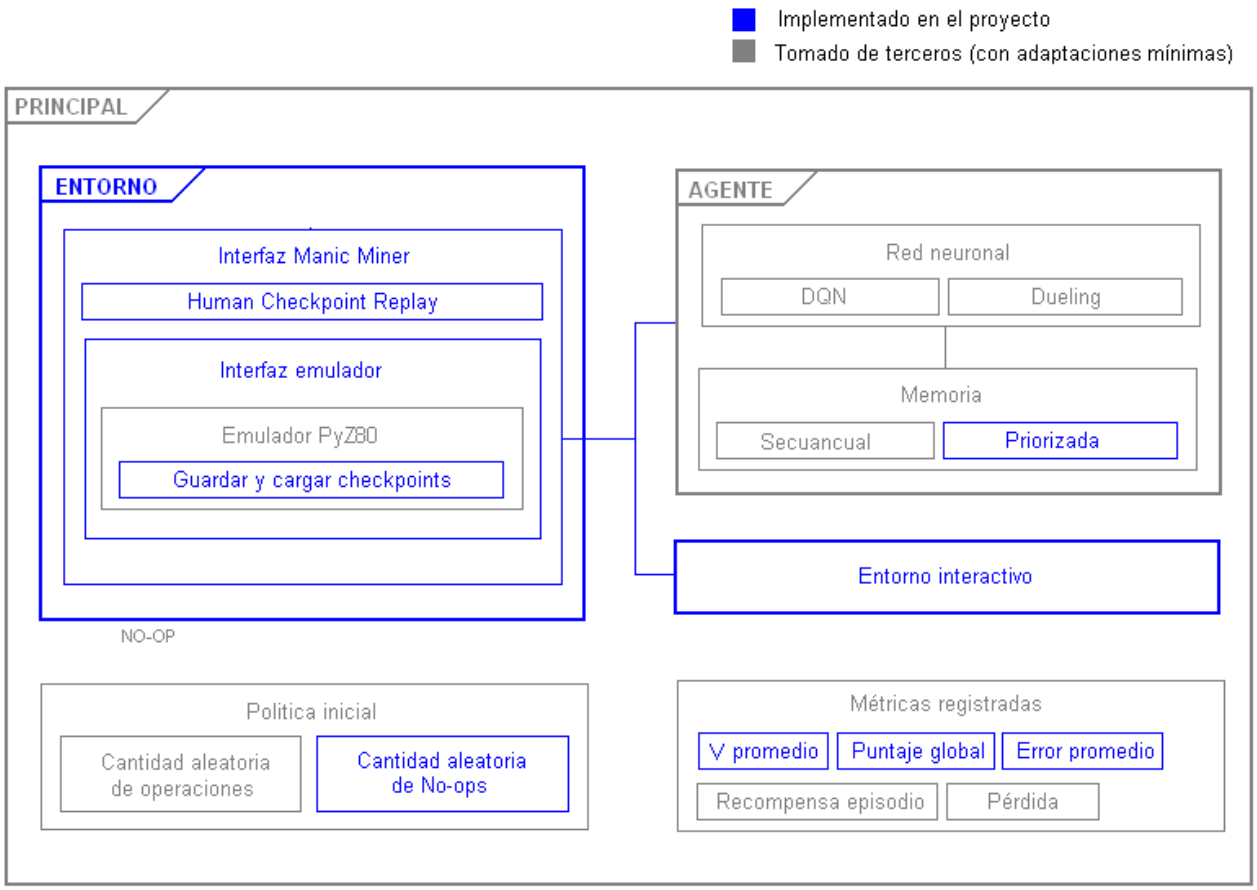


Figura 4.1: Estructura modularizada del entorno y agente. Cada componente está coloreado indicando si es desarrollado en el contexto de este proyecto o si es tomado de terceros.

no sólo es necesario disponer de un emulador funcional, sino que también se necesita tener una interfaz que exponga y encapsule únicamente las operaciones que el agente debe utilizar.

Existen diversos emuladores para ZX Spectrum pero ninguno con el que un agente pueda interactuar de la manera descrita. Luego de elegir un emulador base, es necesario desarrollar su interfaz correspondiente, lo que implica un esfuerzo extra pero que da mayor flexibilidad a la hora de modificar ciertos aspectos del juego.

Es crucial el trabajo de Richard Dymond [9], que mediante ingeniería inversa establece el mapa de la memoria de Manic Miner, revelando la ubicación de variables y rutinas invocadas. Esto permite, por ejemplo, conocer qué variables de la memoria consultar para saber si Willy está o no vivo, cuánto aire queda, o cómo modificar el nivel.

4.1.1. Emulador

El primer paso para contar con el entorno deseado consiste en encontrar un emulador de código abierto donde ejecutar Manic Miner. Para esto se prueban diferentes emuladores en Java, C++ y finalmente PyZX [18] en Python. Se opta por este último, en gran medida por la compatibilidad del lenguaje con las restantes herramientas a utilizar y la experiencia en el uso del lenguaje por parte de todos los integrantes del equipo.

Vadim Kataev es el desarrollador de PyZX, lo hace bajo la licencia GPL-2 y se basa en Jasper[1], un emulador programado en Java. PyZX utiliza la biblioteca PyGame para manejar los gráficos y la entrada por teclado.

En el marco del proyecto, el emulador es adaptado para ser controlado programáticamente, permitiendo introducir comandos por código en vez de únicamente por teclado. Además se lo adapta para generar observaciones a partir de los cuadros originales, se incluye renderizado opcional, se le agrega la funcionalidad de guardado y carga de estados completos del emulador, utilizado para crear *checkpoints* y se eliminan las interrupciones ya que no son necesarias en el juego.

Además del emulador de ZX Spectrum, es necesaria una ROM que contenga el juego Manic Miner en un formato compatible, ya que existen diversos emuladores y formatos diferentes de almacenamiento para los juegos. En la ROM se definen las variables, rutinas y el hilo principal del juego. Todas las ROMs de Manic Miner ejecutadas en el emulador PyZX, ocho en total, presentaron dos problemas:

1. Al iniciar el primer nivel, Willy salta continuamente hasta perder las 3 vidas por aire. Luego, el nivel inicia correctamente.
2. Tras recolectar todas las llaves el portal no cambia de color como debería, permaneciendo inmutado.

Para solucionar el problema vinculado al inicio del episodio, primero se intenta realizar las ejecuciones en *background*. Esto implica dejar que el emulador ejecute internamente la cantidad de ciclos necesarios para que Willy muera por falta de aire tres veces, momento en el que el nivel comienza correctamente. Esto se muestra muy ineficiente ya que, si bien se ejecuta una única vez al iniciar el entorno, toma unos cuantos ciclos terminar con las tres vidas agotando el aire.

[1] `https://github.com/begoon/jasper`

Finalmente se crea un *checkpoint* en un estado inicial válido una vez que el nivel inicia correctamente; cada vez que se inicia el entorno, se carga este *checkpoint*.

Para corregir el inconveniente del portal, como no se puede determinar cuál rutina es la responsable de hacerlo titilar, se implementa una nueva rutina para cambiar forzosamente sus colores. Esto es bastante complejo, e implica lograr entender en detalle el emulador y el juego, ya que tiene dos dificultades: detectar si el portal debe cambiar de color, y efectivamente cambiarlo. Para la primera, a cada paso se consulta si todas las llaves fueron recogidas. Para la segunda, se modifica el valor de la memoria correspondiente al color de los bloques que conforman el portal.

4.1.2. Interfaz del entorno

En esta sección se describe la interfaz que implementa el entorno para permitir su interacción con el agente. Esta interfaz surge de la necesidad de abstraer los detalles del entorno, exponiendo sólo el conjunto de métodos necesarios.

A pesar de que la implementación de la interfaz depende del juego elegido, ya que cada juego hace un manejo diferente de la memoria para manipular su estado, la definición de los métodos puede ser igual para todos. Acordar una interfaz común, lo suficientemente genérica y flexible, permite probar distintos entornos sin modificar al agente. Esta es la premisa que sigue OpenAI Gym [4], y es tomada como referencia, principalmente por proveer una interfaz sencilla y completa.

OpenAI Gym es una plataforma ideada para el desarrollo y comparación de algoritmos de aprendizaje por refuerzos. Para lograr esto define interfaces estándar que todo entorno debe respetar. Su principal clase se denomina *Enviroment*. Esta clase expone la interfaz del entorno con la que interactúa el agente. Su función principal ejecuta una acción en el entorno y cada acción ejecutada retorna los siguientes elementos:

- **Observación:** es el objeto que representa una observación del entorno luego de ejecutar la acción provista. En el contexto de videojuegos puede representar los píxeles de la pantalla o la memoria RAM. Para algunos problemas de control clásicos es simplemente un vector.
- **Recompensa:** es un escalar representando la recompensa obtenida por la acción realizada.
- **Terminado:** es una bandera que indica si se terminó el episodio, en cuyo caso es necesario reiniciar el entorno.
- **Información:** es un diccionario con datos adicionales que puede ser útil para depurar el proceso de aprendizaje. En las implementaciones oficiales de OpenAI Gym que forman parte del *ranking*[2] no se permite utilizar esta información durante el entrenamiento.

Para inicializar el entorno es necesario indicar por parámetro el juego a cargar y los cuadros a saltear entre otros valores. Existe también una función para reiniciar el entorno, en la que se puede especificar un *checkpoint* en el que comenzar[3], devolviendo una observación inicial.

[2]OpenAI Gym tiene disponible una plataforma en línea que permite a los investigadores y apasionados del tema subir sus algoritmos para ser comparados.

[3]Esta posibilidad de especificar un *checkpoint* en el que comenzar escapa a los estándares propuestos por OpenAI Gym. No está claro cómo utilizar HCR en los entornos provistos por esta plataforma. Agregando esta funcionalidad se puede entrenar con HCR sin problemas.

```
Cargar el Juego Manic Miner
for episodio = 1 to M do
    Se reinicia el nivel y se obtiene la observación inicial
    while no sea el fin del juego do
        acción = elegir_opción(Observación)
        Observación, Recompensa, Fin del juego, Información adicional = paso(acción)
    end while
end for
```

Cuadro 4.1: Pseudocódigo que ejemplifica el uso del entorno desarrollado. La acción es tomada a partir de la observación anterior, ya sea por un humano o un agente artificial que esté jugando.

En el cuadro 4.1 se presenta un pseudocódigo que ilustra cómo interactúa el agente con el entorno.

Cada entorno cuenta con un objeto llamado *Espacio* que define sus acciones válidas y la forma que tienen las observaciones que genera. Su propósito es estandarizar los distintos entornos permitiendo desarrollar agentes que se desenvuelvan en una gran cantidad de entornos sin necesidad de adaptarlos a cada uno. En el contexto del proyecto, el espacio de acciones es el conjunto de enteros entre 0 y 6 que representa los diferentes comandos del juego, y el espacio de las observaciones es una matriz de $256 \times 192 \times 3$, con enteros de un byte, que es la imagen de la pantalla en RGB. Esta observación puede ser de dimensiones menores si se le indica al entorno que recorte píxeles[4].

El entorno desarrollado en este proyecto no se integra completamente con OpenAI Gym, aunque su diseño se basa en la especificación que propone. Esto se debe a que se escoge implementar únicamente el subconjunto mínimo necesario de las operaciones de OpenAI Gym para no dilatar los tiempos de implementación dedicados al entorno.

Para implementar los métodos definidos por la interfaz, es necesario familiarizarse con la estructura del juego, siendo el manejo de sus rutinas una parte medular. Estas corresponden a fragmentos de código ordenados que son invocados por el hilo principal y sirven para modificar el estado del emulador. Entre las rutinas que se ejecutan se incluyen las que despliegan animaciones en pantalla, algunas de las cuales no se deben mostrar al agente como las pantallas del comienzo y final del juego. Para no mostrárselas, inicialmente se evitan estas rutinas mediante su ejecución en *background* cuando son detectadas. Sin embargo, esta solución no es viable debido a que estas rutinas se invocan con mucha frecuencia (una vez por episodio).

A esto se suma la dificultad de detectar el momento exacto donde finaliza un episodio y comienzan las rutinas. Un episodio puede terminar cuando el agente hace contacto con trampas o guardias, cae desde gran altura o se agota el aire. Pese a que estos casos tienen el mismo efecto, son realmente diferentes desde el punto de vista del emulador dado que no existe una variable para la muerte por falta de aire, pero sí para las restantes. El aire remanente no se almacena en una única variable: se utiliza la memoria que lo representa gráficamente en pantalla para llevar la cuenta del aire restante. Esta cantidad se puede aproximar a un número aunque con un pequeño error de redondeo, por lo que se establece un umbral mínimo para este valor a partir del cual se considera sin aire.

[4]Se debe indicar cuántos quitar de izquierda, derecha, arriba y abajo al crear el entorno. Esta posibilidad de recortar la observación es introducida en este proyecto y por tanto escapa a los estándares definidos por OpenAI Gym.

Finalmente se llega a la misma solución que se utiliza para el problema con el inicio del entrenamiento asociado al inconveniente con la ROM: cargar un *checkpoint* correspondiente al estado inicial del episodio.

4.1.3. Entorno interactivo

El entorno interactivo surge como consecuencia del desarrollo del emulador en el marco del proyecto. Inicialmente se piensa para realizar pruebas sobre el emulador pero finalmente se extienden sus funcionalidades, permitiendo guardar y cargar *checkpoints*, acceder al estado interno del emulador y editar niveles. Al iniciar este entorno el usuario indica cuál de estas funcionalidades desea utilizar. Además se provee de una interfaz gráfica que permite a una persona jugar un juego de ZX Spectrum sujeto a los efectos del salteo de cuadros, tal como lo haría el agente.

Jugar como lo haría el agente

En cada paso se muestra el cuadro sin procesar, y el usuario elige la acción a ejecutar. Sus principales beneficios son: permitir mover a Willy a una posición específica, crear y cargar un *checkpoint* y experimentar el impacto en la jugabilidad de los diferentes valores de salteos de cuadros y la discretización del tiempo.

Guardar y cargar *checkpoints*

Permite la creación, almacenamiento y posterior restauración del estado del emulador en forma de *checkpoints*. Esta funcionalidad no está presente en el emulador base, por lo que es implementada. Para esto se guarda en un archivo el estado completo de la memoria y los registros de la CPU del emulador. Además se almacena la imagen actual, para facilitar su identificación posterior.

Los archivos generados pueden ser usados en el ambiente interactivo o al momento del entrenamiento o evaluación como un *checkpoint*.

Editor de niveles

El editor de niveles permite de forma interactiva modificar los elementos estáticos de cada nivel, así como las posiciones de las llaves. Luego de editado el nivel, se crean los *checkpoints* que son cargados en las pruebas de sobreajuste del agente descritas en la sección 3.10, aunque perfectamente pueden ser usados para entrenar.

Cada nivel está representado por 512 recuadros[5], que pueden ser un elemento estático, como una planta o una llave, o elementos dinámicos, como Willy o los guardias. Estos últimos generalmente ocupan varios recuadros.

[5]Un recuadro es la unidad mínima que representa un objeto.

4.1.4. Tiempos de ejecución

Visto el largo tiempo que toman las ejecuciones de las pruebas presentadas en el capítulo 3, se realizaron mediciones con el fin de determinar su causa. Se encuentra que el entorno desarrollado tiene buena parte de la responsabilidad.

Las redes neuronales son lentas en su ejecución e involucran muchas operaciones para su ajuste, por lo que inicialmente se asume que llevan la mayor parte del cómputo, despreciando los efectos del tiempo de ejecución del emulador. Esto es coherente con lo que plantean los trabajos relacionados [29, 3], y explica en parte la utilidad del salteo de cuadros. Además, durante la mayor parte del entrenamiento, en cada paso se genera una nueva experiencia e inmediatamente después se entrena la red con 32 de ellas, lo que refuerza esta hipótesis.

Con el fin de verificar estos supuestos, se mide el tiempo que toma cada una de las principales tareas involucradas en el entrenamiento del agente: operaciones sobre la memoria, sobre el emulador y sobre las redes. Se considera en una primera etapa el *warmup* o calentamiento, consistente en los primeros pasos en los que el agente únicamente juega hasta tener una cantidad suficientemente grande de ejemplos para comenzar el entrenamiento. Esta etapa se caracteriza por un uso intensivo del emulador y prácticamente no utiliza las redes neuronales. La segunda etapa es la de entrenamiento propiamente dicho. En ella comienza el ajuste de las redes neuronales por lo que su uso se hace con más intensidad. Para este experimento se utiliza un agente que implementa DDQN con PER, ya que esta última técnica realiza más cantidad de operaciones en la memoria del agente. Se toman 50000 pasos para la etapa de calentamiento, y 550000 hasta el final de los experimentos. Ambos agentes utilizan el mismo valor de salteo de cuadros, y las ejecuciones se realizan en un Intel core i5-4440 3.10GHz X4, con 16 GB de memoria RAM y una GPU GeForce GTX 1070.

Esta medición arroja que durante el entrenamiento, más del 75 % del tiempo ocupado por estas tres tareas es utilizado en el emulador, y no en la red como se creía. Con este resultado se procede a realizar la misma medición sobre el entorno *Breakout* de OpenAI Gym. En este punto se debe tener en cuenta que ambos emuladores no tienen por qué ser comparables ya que implementan dos tareas completamente distintas, con sus lógicas particulares. Si en vez de optar por este juego, que además es de los más simples de ATARI, se elige otro más complejo, el resultado seguramente sería distinto. De todas formas parece interesante tomarlo de referencia. Como diferencia importante, se recuerda que *Breakout* está implementado en C++ mientras que ManicMiner está escrito completamente en Python.

Se observa que la ejecución del entorno ocupa menos del 25 % del tiempo, estando la mayor parte, casi el 75 %, ejecutándose las redes neuronales, lo que lo hace muy eficiente como entorno en comparación con Manic Miner.

Además del tiempo de cada tarea, se registra el tiempo total de cada experimento, tomando 57299 segundos para ZX Spectrum y 9931 para *Breakout*. A partir de estos datos se obtiene el tiempo que se están ejecutando otras tareas, como generar *logs*, guardar pesos y demás. En la figura 4.3 se nota que, a excepción del emulador, las demás tareas toman tiempos similares en ambos entornos. Hay una pequeña diferencia en las tareas categorizadas como otras, pero puede explicarse en el hecho de que, si bien ambos juegan la misma cantidad de pasos, no tienen por qué hacerlo en igual cantidad de episodios. El que ejecute más episodios va a tener más ejecuciones de tareas asociadas al cambio de episodio, por ejemplo, incrementar el *log*. De todas formas no es una diferencia significativa, como si lo es el tiempo en que se ejecuta el entorno.

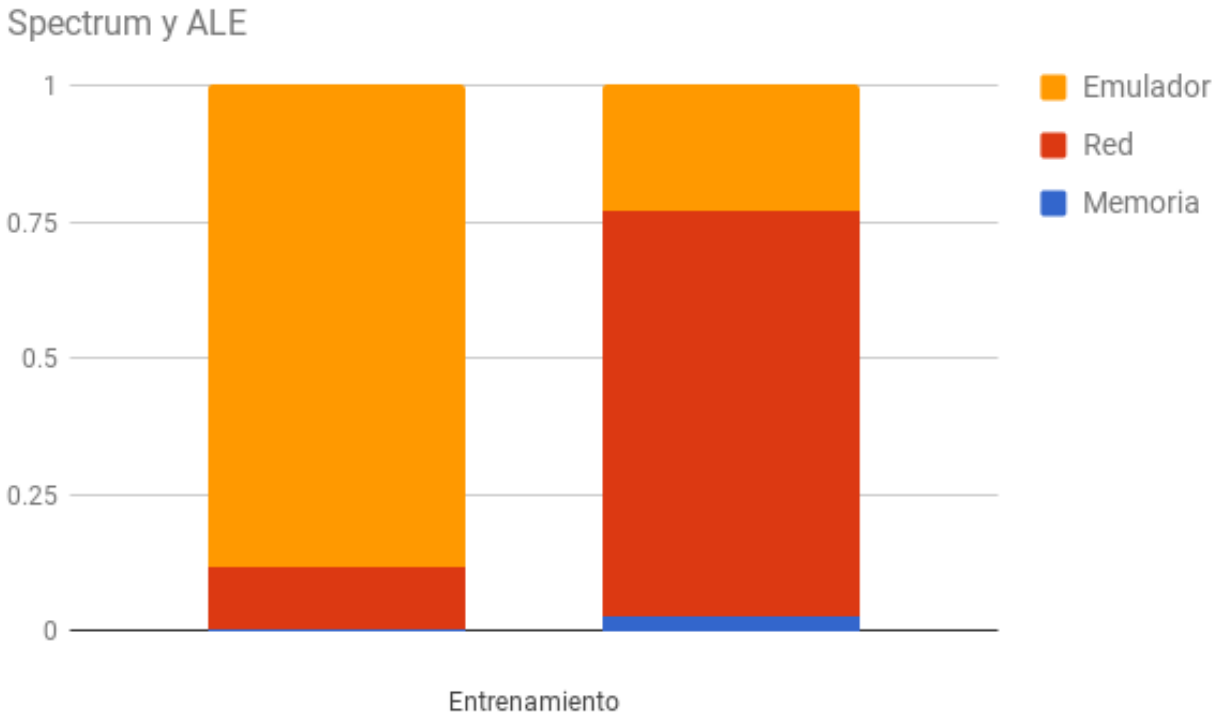


Figura 4.2: Se muestran los tiempos para cada emulador durante el entrenamiento. Los tiempos totales se encuentran discriminados proporcionalmente por tareas. La barra de la izquierda corresponde a ZX Spectrum mientras la de la derecha, a ALE.

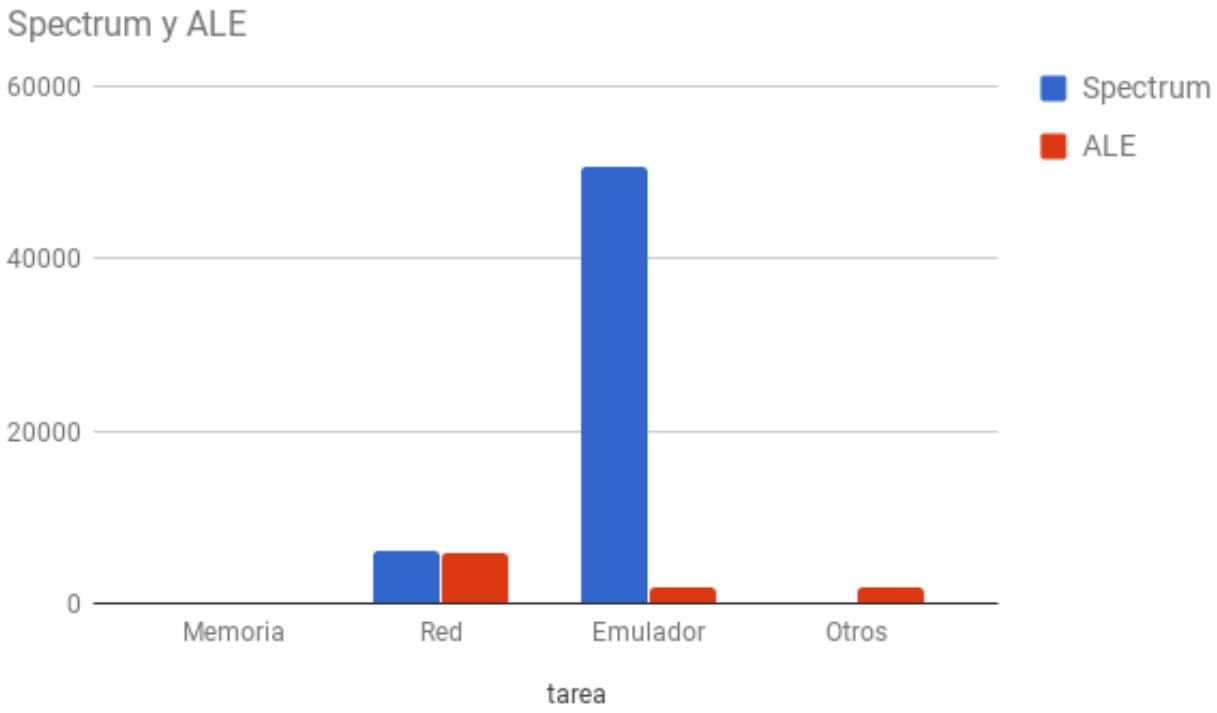


Figura 4.3: Para cada tarea se muestran los tiempos netos comparándolos entre ALE, en rojo, y Spectum, en azul

Tarea	ZX Spectrum	ALE
Memoria	1,79	1,43
Red	66,95	61,86
Emulador	4370,70	182,16

Cuadro 4.2: Tiempo en segundos discriminado por tareas para la fase de calentamiento.

Tarea	ZX Spectrum	ALE
Memoria	232,57	218,77
Red	6056,97	5837,77
Emulador	46334,52	1794,93

Cuadro 4.3: Tiempo en segundos discriminado por tareas para la fase de entrenamiento.

Como conclusión de estas pruebas, se ve que, al contrario de lo asumido, la ejecución en el emulador no es para nada despreciable, e incluso en entornos más eficientes como *Breakout*, toma una parte importante del tiempo. En particular el entorno desarrollado para Manic Miner es sumamente lento.

4.2. Agente

El agente es el responsable de tomar las decisiones sobre cómo actuar en el entrono para maximizar la recompensa a lo largo del tiempo. Un agente basado en DQN utiliza redes neuronales para la aproximación de la función Q. Al momento de tomar una decisión, recibe un estado como entrada y utiliza la red para determinar la acción. La memoria es también un componente central en este tipo de agentes: es donde se almacenan las experiencias a utilizar durante el entrenamiento.

A continuación se presenta un resumen de la biblioteca base de aprendizaje por refuerzos utilizada, haciendo foco en los dos componentes mencionados, incluyendo una sección con las adaptaciones y los nuevos desarrollos.

Tarea	Spectrum	ALE
Memoria	234,36	220,20
Red	6123,92	5899,63
Emulador	50705,22	1977,10
Otros	236,10	1834,33
Total	57299,60	9931,26

Cuadro 4.4: Tiempo en segundos discriminado por tareas para todo el experimento.

4.2.1. Biblioteca base: *Keras_rl*

Luego de explorar varias alternativas, con diferentes grados de éxito, se toma *Keras_rl* como base para los experimentos realizados. Esta biblioteca, desarrollada por Mathias Plappert[6], implementa algunos de los algoritmos de aprendizaje por refuerzos profundo, como *Deep Q Learning* (DQN), *Double DQN* (DDQN), y *Dueling network DQN* (Dueling DQN) además de incluir la Memoria *Experience Replay*. La biblioteca se encuentra escrita enteramente en Python y basa sus interfaces para el entorno en el estándar propuesto por OpenAI Gym, permitiendo integrarlo fácilmente con nuevos entornos que respeten esta interfaz.

Esta biblioteca es extensible ya que se basa en módulos y posee un intrincado sistema de *callbacks* que permite personalizar los *logs* y el proceso de entrenamiento. Como contrapartida, el código de *Keras_rl* es muy extenso, no es conciso y es difícil de interpretar debido a la gran cantidad de funcionalidades soportadas, lo cual complica su adaptación.

Redes

Para la implementación de las redes neuronales se utiliza la biblioteca Keras, una abstracción de alto nivel para especificar redes de aprendizaje profundo, que corre tanto sobre Theano[7] como en Tensorflow[8]. Estas últimas son las bibliotecas que ejecutan efectivamente las operaciones sobre las redes neuronales utilizando cálculo simbólico para hacerlo más eficiente. Se prueban ambas opciones, sin encontrar diferencias entre ellas. El pasaje de una a otra es muy sencillo: una vez que ambas están instaladas, basta con modificar un archivo de configuración. El uso de DQN, DDQN y Dueling está implementado de manera que se pueden emplear en cualquier combinación utilizando siempre la misma arquitectura.

Memoria

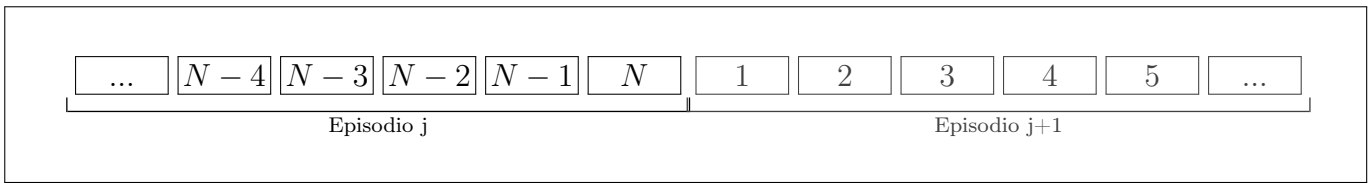
La memoria tiene un rol central en los métodos DQN, ya que almacena el último millón de experiencias que sirven como base del entrenamiento. Se encuentra implementada como una superclase *Memory*, cuyos principales métodos son dos: uno permite añadir una nueva observación, con la acción y recompensa asociada, además de una bandera indicando si es o no terminal. El otro, posibilita obtener un conjunto de experiencias de la memoria de manera aleatoria.

Las experiencias son de la forma:

$$e_i = (s_i, a_i, r_i, s_{i+1})$$

Donde s_i es el estado en el paso i, a_i la acción tomada en el paso i, r_i la recompensa recibida del paso i y s_{i+1} es el estado al que se llega luego de ejecutar la acción. Cada estado contiene cuatro cuadros de 84×84 píxeles cada uno, almacenados como una matriz de enteros de 8 bits, que resultan de preprocesar las observaciones como se indica en la sección 3.1. Para almacenar una experiencia se necesita almacenar dos estados, que totalizan 56 KB. Ignorando el costo de mantener la acción y la recompensa, son algo más de 56 GB la memoria necesaria para guardar

[6] https://github.com/matthiasplappert/keras-rl
[7] https://github.com/Theano/Theano
[8] https://www.tensorflow.org/



Cuadro 4.5: Segmento de memoria en la que se almacenan cuadros para dos episodios consecutivos, donde $\boxed{i}$ contiene al último cuadro del $i-$ésimo episodio

un millón de experiencias, lo cuál es inviable de mantener en RAM con los recursos disponibles en una computadora doméstica.

La optimización lógica es guardar únicamente el estado actual, con lo que se reduce la memoria necesaria a aproximadamente la mitad, 28 GB. La siguiente mejora, utilizada en keras_rl, es almacenar en cada paso únicamente el cuadro actual junto a los demás elementos de la experiencia. Esto hace que el problema sea tratable con los recursos inicialmente disponibles, poco más de 7 GB en cuadros, aunque añade el costo computacional de componer los estados (a partir de los cuadros almacenados), cada vez que se muestrea sobre la memoria. Como todos los cuadros son consecutivos, esta tarea es sencilla: se deben tomar los 3 cuadros anteriores teniendo la precaución de que todos pertenezcan al mismo episodio.

Por ejemplo, para reconstruir la i-ésima experiencia almacenada, se toma el i-ésimo cuadro de la memoria, y se lo concatena con los tres anteriores, generando así el i-ésimo estado. En caso de que alguno de los cuadros anteriores pertenezcan al episodio anterior, se rellena el estado con el primer cuadro del episodio hasta completar la cantidad suficiente. Este mecanismo de relleno no está presente en la biblioteca *Keras_rl*, y es implementado en el contexto del proyecto siguiendo la sugerencia de Zahavy et al. [45]. También se toma el cuadro $i+1$-ésimo, con sus tres cuadros anteriores, para generar el estado $i+1$, con las mismas precauciones. Esto se hace utilizando las marcas de estado terminal, que se almacenan asociadas a la observación, acción y recompensa. Si se encuentra un cuadro que está marcado como terminal, es porque el cuadro que viene a continuación pertenece a otro episodio. Si inicialmente se comienza con un índice i que corresponde con el cuadro final del episodio sencillamente se vuelve a sortear otro índice.

En el ejemplo del cuadro 4.5, no se debe reconstruir la experiencia asociada al índice $i = N$ ya que los estados $\boxed{N-3}\boxed{N-2}\boxed{N-1}\boxed{N}$ y $\boxed{1}\boxed{1}\boxed{1}\boxed{1}$ no corresponden al mismo episodio. Por el contrario, la experiencia formada a partir del índice $i = 2$ es válida, con los estados formados por los cuadros $\boxed{1}\boxed{1}\boxed{1}\boxed{2}$ y $\boxed{1}\boxed{1}\boxed{2}\boxed{3}$, siguiendo la consigna de rellenar los cuadros iniciales faltantes con el primer cuadro del episodio.

El mecanismo descrito implementa una memoria uniforme, ya que los índices i que se sortean para recuperar las experiencias para entrenar, son elegidos aleatoriamente siguiendo una probabilidad uniforme.

4.2.2. Funcionalidades desarrolladas

Con el fin de probar los efectos de diferentes técnicas que no se encuentran en *Keras_rl*, durante el proyecto se desarrollan algunas funcionalidades complementarias a las que provee esta biblioteca: PER y HCR. Junto con ellas se agregan nuevos registros que ayudan a entender

cómo inciden en el proceso de aprendizaje. Se intenta respetar las decisiones de diseño tomadas por su autor para que estas funcionalidades puedan ser incorporadas fácilmente por otros desarrolladores.

Priotirized experience replay: PER

Como su nombre lo indica, PER asigna una distribución diferente a la uniforme para la elección de experiencias, de acuerdo a un cierto criterio de priorización. Las experiencias que parecen más prometedoras en cuanto a lo que tienen por enseñar deberían ser más probables de obtener.

Para implementar esta memoria de forma eficiente se utiliza un *sum-tree*: un árbol binario donde cada nodo es la suma de sus hijos, y cada hoja contiene la probabilidad que tiene de ser elegida, asociada a una experiencia. Esto permite que las operaciones de búsqueda, inserción y actualización sean ejecutadas en $O(\log(n))$

El resto de los datos asociados a la experiencia (acción, recompensa, observaciónes y si es o no terminal) se almacenan tal y como lo hace la implementación de *Keras_rl*. Al muestrear las experiencias sobre la memoria, se utiliza el mecanismo de reconstrucción de experiencias antes mencionado, sólo que en vez de sortear los índices siguiendo una prioridad uniforme, son tomados del árbol según su probabilidad asociada. Además, cada experiencia se debe retornar junto a una referencia al lugar del árbol en que está almacenada, ya que luego de entrenar sobre un conjunto de experiencias, el error asociado a ellas cambia, y por lo tanto se deben actualizar las probabilidades almacenadas en el árbol. Nótese que se necesita exponer un método que permita actualizar las probabilidades mencionadas.

Además se modifica el registro de entrenamiento, agregando el error promedio en PER. Esta métrica indica el promedio de errores reportados por la red luego del entrenamiento, antes de que se actualicen las respectivas probabilidades en la memoria.

Human Checkpoint Replay: HCR

La implementación de HCR es sencilla al tener ya implementada la funcionalidad de guardar y cargar estados internos del emulador. Este estado interno incluye una copia de memoria y registros internos de la CPU, y no debe confundirse con el estado que recibe el agente. Durante el entremaniento se dispone de un conjunto de estados o *checkpoints*. Para cada nuevo episodio se carga un *checkpoint* escogido al azar de este conjunto y la ejecución comienza desde ese punto. La función que reinicia el entorno es la que se utiliza para indicar en qué *checkpoint* se debe comenzar el siguiente episodio. Esto se aparta de los estándares propuestos por OpenAI Gym, pero hace que la implementación de HCR sea directa. Además permite que un mismo agente pueda entrenar en múltiples niveles en una misma ejecución: basta utilizar como *checkpoints* los estados iniciales de cada uno de los niveles.

Para poder comparar algunas métricas del registro de entrenamiento de un agente que entrena con HCR y otro que lo hace sin esta técnica, se deben tomar del registro del primero únicamente aquellos episodios que se corresponden al estado inicial. Esto sucede, por ejemplo, para comparar puntajes o recompensas, ya que en caso contrario se presentan situaciones en las que se compara la recompensa de un episodio que comienza en un estado inicial, con la de otro que comienza en un estado del que es fácil obtener mucha recompensa. Por ejemplo, en el nivel uno, un agente que comienza algún episodio en un *checkpoint* en el que está por tomar las llaves del

nivel superior no es comparable (con estas métricas) con otro que comienza todos sus episodios desde el estado inicial. Por tal motivo, al finalizar cada episodio se guarda en el registro el nombre del *checkpoint* que se utiliza para comenzar, permitiendo luego su identificación.

Capítulo 5

Conclusiones

El presente proyecto se centra en la creación de un jugador artificial utilizando técnicas de Aprendizaje Profundo para el Manic Miner, un videojuego para la ZX Spectrum. Luego de un relevamiento inicial se decide focalizar los esfuerzos en los métodos de la familia DQN. Durante el proyecto se comparan empíricamente diferentes versiones de agentes en conjunto con diversos parámetros y técnicas: DQN, DDQN, Dueling, HCR y PER.

Debido a la inexistencia de un entorno de aprendizaje para ZX Spectrum, se implementa uno siguiendo los estándares de OpenAI Gym, sobre un emulador desarrollado en Python. Se agregan, además, otras funcionalidades, como el comienzo por *checkpoints*, y se construye una interfaz interactiva para modificar la configuración de los niveles de Manic Miner. El entorno resulta ser robusto, mantenible y extensible a otros juegos, aunque en contrapartida no tan eficiente debido a Python que, a pesar de ser muy expresivo y conciso, no compite con la velocidad de ejecución de un lenguaje compilado.

En los resultados se constatan las ventajas de DDQN frente al DQN estándar y la clara ventaja que otorga HCR en la exploración. En cuanto a el uso de Dueling y memoria con priorizado estocástico, la diferencia no se hace tan notoria. Finalmente se consigue un agente capaz de superar el primer nivel en varias oportunidades, utilizando *Double Q-Network* y *Prioritized Experience Replay* en conjunto con *Human Checkpoint Replay*.

El principal problema que se presenta, y que representa un cuello de botella durante el entrenamiento, es la exploración del entorno. Dado los esquemas de tiempo manejados y el pobre desempeño de la política de exploración para este juego, la opción escogida para acelerar el entrenamiento es darle ayuda humana al agente utilizando *Human Checkpoint Replay*. Con esta técnica se supera la meseta de entrenamiento en la que caen los métodos generales, logrando que el agente complete el nivel en más de la mitad de las pruebas, 18 de 30.

Aunque en los trabajos investigados se reportan los resultados referentes al desempeño de sus agentes, resulta llamativo lo poco que se trata el problema del posible sobreajuste en esas soluciones. En este trabajo se realizan pruebas al respecto con dos agentes capaces de completar el primer nivel, concluyendo que, efectivamente, se encuentran sobreajustados. Se considera interesante explorar alternativas para evitar este sobreajuste utilizando intensivamente el editor de niveles desarrollado o entrenando al agente en varios niveles y midiendo su desempeño en uno desconocido.

Por restricciones de tiempo, algunos hiperparámetros de la red o del entrenamiento no son mo-

dificados en ninguna etapa del proyecto: la cantidad, tipo y dimensión de las capas, el algoritmo de descenso por gradiente, etc. Es posible que ajustando estos hiperparámetros específicamente para Manic Miner se logren mejores resultados.

El trabajo de Mnih et al. tomado como punto de partida [29] entrena con cincuenta millones de pasos, Schaul et al., en su propuesta de la técnica PER [35] utilizan doscientos millones, mientras que en el presente trabajo no se superan los diez millones. Queda abierta la pregunta sobre qué sucedería si se utilizan una mayor cantidad de pasos para entrenar. Es posible que el agente logre superar el nivel sin necesidad de utilizar conocimiento de dominio o que modifique su representación interna de los objetos, llevándolo a cambiar su comportamiento en las pruebas de sobreajuste.

Otra posibilidad es mejorar la calidad de las observaciones preprocesadas, lo que implica modificar las dimensiones de la red para recibir una imagen de mayor calidad. Esto no se realiza por falta de recursos: una imagen más grande requiere más memoria para su almacenamiento y lleva más tiempo para el entrenamiento, debido al aumento de parámetros a ajustar. Como punto intermedio se puede modificar simplemente las proporciones de la imagen preprocesada, tratando de mantener la misma cantidad de parámetros.

Finalmente, como futura línea de trabajo se recomienda fuertemente explorar A3C [27, 2] como método alternativo. En caso de escoger seguir por la linea de *Deep Q Networks*, se recomienda profundizar en técnicas de exploración profunda [32] así como considerar incursionar en el uso de agentes preentrenados con conocimiento del mundo, partiendo de un modelo que lo ayude en el proceso de exploración, como en el trabajo de Todd Hester et al. [14].

Bibliografía

[1] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res. (JAIR)*, 47:253–279, 2013.

[2] Marc G. Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Rémi Munos. Unifying count-based exploration and intrinsic motivation. *CoRR*, abs/1606.01868, 2016.

[3] Alex Braylan, Mark Hollenbeck, Elliot Meyerson, and Risto Miikkulainen. Frame skip is a powerful parameter for learning to play atari. In *AAAI-15 Workshop on Learning for General Competency in Video Games*, 2015.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[5] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell: A neural network for large vocabulary conversational speech recognition. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*, pages 4960–4964. IEEE, 2016.

[6] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2017.

[7] Tim de Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. The importance of experience replay database composition in deep reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*, 2015.

[8] Aaron Defazio and Thore Graepel. A comparison of learning algorithms on the Arcade Learning Environment. *CoRR*, abs/1410.8620, 2014.

[9] Richard Dymond. The complete Manic Miner RAM disassembly. `http://skoolkit.ca/disassemblies/manic_miner/`. Accedido: 2017-08-22.

[10] Matthew J. Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general Atari game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(4):355–366, 2014.

[11] Matthew J. Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527, 2015.

[12] Ji He, Jianshu Chen, Xiaodong He, Jianfeng Gao, Lihong Li, Li Deng, and Mari Ostendorf. Deep reinforcement learning with a natural language action space. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August*

7-12, 2016, Berlin, Germany, Volume 1: Long Papers. The Association for Computer Linguistics, 2016.

[13] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. *CoRR*, abs/1603.01121, 2016.

[14] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning, 2017.

[15] Ionel-Alexandru Hosu and Traian Rebedea. Playing Atari games with deep reinforcement learning and human checkpoint replay. *CoRR*, abs/1607.05077, 2016.

[16] Feng-Hsiung Hsu. *Behind Deep Blue: Building the Computer That Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA, 2002.

[17] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition? In *IEEE 12th International Conference on Computer Vision, ICCV 2009, Kyoto, Japan, September 27 - October 4, 2009*, pages 2146–2153. IEEE Computer Society, 2009.

[18] Vadim Kataev. PyZX. `http://www.pygame.org/project/173/1347`. Accedido: 2017-08-22.

[19] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[21] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning, 2016.

[22] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[23] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST database. `http://yann.lecun.com/exdb/mnist/`. Accedido: 2017-08-22.

[24] Yitao Liang, Marlos C. Machado, Erik Talvitie, and Michael Bowling. State of the art control of atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 485–493, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[25] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX93-22750.

[26] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[27] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[29] 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, 02 2015.

[30] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

[31] Michael A. Nielsen. Neural networks and deep learning. http://neuralnetworksanddeeplearning.com/, 2015. Accedido: 2017-08-22.

[32] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped DQN. *CoRR*, abs/1602.04621, 2016.

[33] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[34] Jonathan Schaeffer, Et Al, Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, and Martin Müller Robert Lake. Checkers is solved. *Science*, 2007.

[35] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[36] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[37] Tom Stafford. Fundamentals of learning: the exploration-exploitation trade-off. `http://www.tomstafford.staff.shef.ac.uk/?p=48`. Accedido: 2017-08-22.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press; University of Alberta (Draft), 2012.

[39] Jakub Sygnowski and Henryk Michalewski. Learning from the memory of atari 2600. *CoRR*, abs/1605.01335, 2016.

[40] Erik Talvitie and Michael H. Bowling. Pairwise relative offset features for atari 2600 games. In *AAAI Workshop: Learning for General Competency in Video Games*, volume WS-15-10 of *AAAI Workshops*. AAAI Press, 2015.

[41] Hado van Hasselt, Arthur Guez, Matteo Hessel, and David Silver. Learning functions across many orders of magnitudes. *CoRR*, abs/1602.07714, 2016.

[42] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 2094–2100. AAAI Press, 2016.

[43] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 1995–2003. JMLR.org, 2016.

[44] Hsuan-Tien Lin Wei-Ning Hsu. Active learning by learning. 2015.

[45] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. Graying the black box: Understanding dqns. *CoRR*, abs/1602.02658, 2016.