UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN

TRABAJO FIN DE GRADO

Aplicación de Deep Reinforcement Learning a un juego real - Tetris

Luna Jiménez Fernández

Junio de 2020

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

TECNOLOGÍA ESPECÍFICA DE
COMPUTACIÓN

TRABAJO FIN DE GRADO

Aplicación de Deep Reinforcement Learning a un juego real - Tetris

Autor: Luna Jiménez Fernández

Directores: María Julia Flores Gallego

Junio de 2020

A mi familia, por haberme ayudado a llegar hasta aquí, y a toda la gente que creyó en mí aun cuando yo no fui capaz de hacerlo.

Declaración de Autoría

Yo, Luna Jiménez Fernández con DNI 47092045M, declaro que soy la única autora del trabajo fin de grado titulado *Aplicación de Deep Reinforcement Learning a un juego real - Tetris* y que el citado trabajo no infringe las leyes en vigor sobre propiedad intelectual y que todo el material no original contenido en dicho trabajo está apropiadamente atribuido a sus legítimos autores.

Albacete, a 15 de Junio de 2020.

Fdo.: Luna Jiménez Fernández

Firmado por JIMENEZ FERNANDEZ LUNA - 47092045M el día 15/06/2020 con un certificado emitido por AC FNMT Usuarios

Resumen

El algoritmo de **Deep Q-Learning** es el estado del arte actualmente para problemas de aprendizaje por refuerzo, siendo especialmente notable cuando se aplica a juegos. Por tanto, el objetivo de este trabajo es el análisis e implementación de esta técnica para estudiar su viabilidad aplicada a Tetris.

La memoria comienza con el estudio de la técnica de Deep Q-Learning, que surge a partir de la unión de dos técnicas de aprendizaje automático: **Deep Learning** (el uso de redes neuronales profundas para poder resolver problemas complejos) y **Q-Learning** (uno de los principales algoritmos de aprendizaje por refuerzo sin modelo). Esta técnica tiene además varias posibles mejoras por aplicar, de las cuales se describirá **Prioritized Experience Replay**, una de las mejoras más importantes y notables actualmente.

El juego sobre el que se implementará esta técnica es Tetris. Se comentará el funcionamiento de este juego, junto a las decisiones que se han tomado en su **diseño** (incluyendo el sistema de puntuación, el sistema de dificultad, la generación de piezas o las simplificaciones tomadas para facilitar la aplicación de la técnica) y su **implementación** (realizando una implementación propia para facilitar la labor al algoritmo y permitir la adaptación total a las necesidades de éste).

Para poder aplicar el algoritmo es necesario formalizar varios aspectos del conocimiento. Se describirán las dos aproximaciones que se han hecho a este conocimiento: una primera aproximación basada en imitar las entradas que realizaría un jugador humano en el juego, y una segunda aproximación fundamentada en el razonamiento de un jugador humano durante el juego (centrándose en dónde colocar las piezas en juego para maximizar la puntuación).

De cada una de estas aproximaciones se comentarán los principales elementos a caracterizar para poder resolver el problema (el estado, la acción y las recompensas otorgadas a los agentes). También se describirán los agentes utilizados por cada aproximación (haciendo hincapié en sus arquitecturas internas, su funcionamiento...) y cómo se han implementado estos agentes en el juego.

Finalmente, se estudiará el rendimiento de los agentes durante el aprendizaje y en partidas reales, estudiando la evolución de las líneas completadas, la puntuación obtenida y la longevidad de los agentes (métricas útiles para evaluar la idoneidad de estos agentes como jugadores de Tetris) y comparando estos rendimientos entre ellos, buscando analizar los resultados y extraer conclusiones.

Agradecimientos

En primer lugar, quiero agradecer a mi tutora **María Julia Flores Gallego**. Desde la primera asignatura que cursé con ella ha sido un apoyo constante durante todo mi progreso académico. Además, ha sido una ayuda continua durante la realización de este trabajo, ayudándome a resolver problemas y motivándome para continuar en los momentos más complicados.

También quiero agradecer a **mi familia** por haberme ayudado a llegar hasta aquí, ayudándome y empujándome a seguir mis sueños y a darlo todo.

Finalmente, me gustaría agradecer a todas **mis amistades**, que me han apoyado y creído en mí, aun cuando yo misma no era capaz de hacerlo y quería tirar la toalla.

Índice general

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1

INTRODUCCIÓN Y MOTIVACIÓN

En este capítulo, se realizará una breve introducción a los contenidos que se expondrán a lo largo de este Trabajo Fin de Grado (objetivos, estado de la cuestión, enfoques tomados, experimentación realizada...). Además de esto, se incluirán las principales razones que han motivado a la realización de este Trabajo Fin de Grado. Finalmente, se incluirá una descripción de la estructura que seguirá la memoria.

1.1. Introducción

Actualmente, **Deep Q-Learning** es una de las técnicas más relevantes y punteras de aprendizaje por refuerzo, obteniendo grandes resultados en una amplia gama de problemas y siendo el foco de numerosas investigaciones. El objetivo de este trabajo es la aplicación del algoritmo a una versión simplificada de un videojuego ya existente (Tetris).

El trabajo comienza con un estudio de las técnicas que han precedido a la técnica central: Deep Learning (ahondando también en las redes neuronales) y Q-Learning (ahondando en el aprendizaje por refuerzo). A continuación, se procederá a estudiar el algoritmo de Deep Q-Learning como tal, viendo su evolución y algunas mejoras realizadas sobre este.

Tras esto se describirá en detalle el juego (Tetris), profundizando en las decisiones que se han tomado de cara a la aplicación del algoritmo al juego y en los detalles de la implementación realizada. Después se comentarán las aproximaciones al problema que se han llevado a cabo, detallando para cada una de ellas la caracterización del conocimiento, la estructura de los agentes utilizados y la implementación de estos.

Se procederá después a detallar los experimentos que se han realizado, observando los resultados obtenidos. Finalmente se estudiarán estos resultados por separado y comparándolos entre ellos, analizando los datos obtenidos y extrayendo las conclusiones apropiadas de ellos.

1.2. Motivación

Actualmente, **Deep Learning** es una de las técnicas de aprendizaje automático más relevantes y extendidas. Es el estado del arte en gran cantidad de campos (reconocimiento de imágenes, reconocimiento de voz, genómica...) que hasta hace poco se consideraban inabordables por su gran complejidad y dimensionalidad [1].

Por tanto, es lógico pensar que aplicar esta técnica a problemas de **aprendizaje por refuerzo** daría también buenos resultados. Hoy en día, **Deep Reinforcement Learning** se ha convertido también en una técnica puntera y relevante en el campo del aprendizaje por refuerzo, consiguiendo por ejemplo desarrollar agentes con un rendimiento superior al de jugadores humanos en numerosos videojuegos clásicos de Atari 2600 [2].

Las técnicas de investigación e inteligencia artificial aplicadas a videojuegos siempre son importantes e interesantes por tener resultados inmediatamente más llamativos y motivantes. Ahora bien, también resultan de gran utilidad porque son fácilmente extrapolables a otros problemas de aprendizaje por refuerzo más "tradicionales" (como puede ser, por ejemplo, el control automático de robots), lo que hace interesante su investigación.

Finalmente, otra razón no despreciable para la elección de la temática de este trabajo es el gran interés de la alumna por los campos de la inteligencia artificial y los videojuegos, llegando a ser éstas las razones que la motivaron a decantarse por este grado en primer lugar.

1.3. Estructura de la memoria

Este trabajo se divide en un total de ocho capítulos, que serán descritos brevemente a continuación:

- **Capítulo 1:** Este capítulo introduce el tema a tratar, la motivación y la estructura de la memoria.
- **Capítulo 2:** Este capítulo introduce los objetivos a alcanzar con el Trabajo Fin de Grado, además de la metodología utilizada para alcanzarlos.
- **Capítulo 3:** En este capítulo, se describen los principales antecedentes a la tecnología a tratar: las redes neuronales (junto a las técnicas de Deep Learning) y el aprendizaje por refuerzo (centrándose en las técnicas de Q-Learning). A continuación, se expone la unión de las dos técnicas anteriores: Deep Q-Learning, hablando de sus primeras versiones y de algunas mejoras que se han implementado sobre éste.

- **Capítulo 4:** Este capítulo describe el juego sobre el que se va a aplicar el algoritmo, Tetris. Además, se entra en detalle sobre las decisiones y simplificaciones que se han tomado para realizar el trabajo. Finalmente, se comenta la implementación que se ha realizado del juego.

- **Capítulo 5:** En este capítulo se describen las dos aproximaciones que se han realizado para resolver este problema: en la primera se diseña un agente que imita directamente las entradas que realizaría un jugador humano, mientras que en la segunda el agente se diseña para imitar el proceso de razonamiento que usaría un jugador humano. De cada aproximación se describe la caracterización de su conocimiento, las variaciones del agente utilizadas y sus implementaciones.

- **Capítulo 6:** En este capítulo se describe la experimentación realizada con ambas aproximaciones, exponiendo además los resultados obtenidos por estos experimentos.

- **Capítulo 7** Este capítulo contiene un estudio de los resultados obtenidos por ambas aproximaciones, comparándolos y contrastándolos entre ellos para obtener conclusiones.

- **Capítulo 8:** Finalmente, en este capítulo se describen las conclusiones obtenidas tras la realización del Trabajo Fin de Grado, exponiendo además posibles trabajos futuros a partir de este trabajo y las competencias adquiridas durante su realización.

Además de esto, se incluye al final de la memoria una bibliografía, en la que se encuentra la lista de fuentes y referencias que han sido consultadas durante el desarrollo de este trabajo.

Capítulo 2

OBJETIVOS Y METODOLOGÍA

En este capítulo se describirán los objetivos que se busca alcanzar con el desarrollo de este Trabajo Fin de Grado. Además, se comentará la metodología que se ha utilizado para alcanzar estos objetivos, junto a las tareas que se han realizado para alcanzarlos.

2.1. Objetivos

El principal objetivo de este Trabajo Fin de Grado es el conocimiento y aplicación del algoritmo de Deep Q-Learning a un problema real como es el enseñar a un agente a jugar a un juego ya existente, Tetris, estudiando la viabilidad de esta técnica.

Para alcanzar este objetivo, será necesario alcanzar algunos objetivos parciales previamente:

1. Estudio de bibliografía para entender plenamente la técnica de Deep Q-Learning y las mejoras que se han aplicado a ésta.

2. Búsqueda y análisis de librerías disponibles para realizar la implementación de las redes neuronales necesarias.

3. Estudio, caracterización e implementación del juego (Tetris) para poder trabajar posteriormente sobre él.

4. Caracterización del conocimiento, formalización e implementación de los agentes.

5. Realización de experimentos para observar el comportamiento de los agentes en el entorno real.

6. Estudio y análisis de los resultados obtenidos para extraer observaciones y conclusiones que nos permitan valorar la viabilidad de estas técnicas aplicadas al juego real.

2.2. Metodología

Las tareas que se han llevado a cabo para realizar con éxito los objetivos previamente descritos son las siguientes:

1. Estudio de aplicaciones previas de Deep Q-Learning a otros juegos reales.
2. Estudio y recopilación de bibliografía para comprender mejor el algoritmo.
3. Caracterización y desarrollo del juego (Tetris).
4. Adaptación del algoritmo de Deep Q-Learning al juego mediante la implementación de diversos agentes.
5. Desarrollo de interfaces visuales y ayudas para facilitar y agilizar el uso de los agentes.
6. Experimentación con estos agentes en diversas situaciones controladas.
7. Almacenamiento y procesado de los resultados de los experimentos realizados previamente.
8. Análisis, comparación y estudio de los resultados obtenidos durante la experimentación.

Capítulo 3

ANTECEDENTES Y ESTADO DE LA CUESTIÓN

En este capítulo se describirán las tecnologías y algoritmos que han precedido a la técnica que se usará durante este trabajo: Deep Learning y Q-Learning. Tras esto, se desarrollará el algoritmo de Deep Q-Learning, y finalmente se estudiarán algunas de las mejoras que han sido introducidas sobre este algoritmo.

3.1. Aprendizaje automático: Redes neuronales y Deep Learning

3.1.1. Aprendizaje automático

El **aprendizaje automático** (también conocido como **aprendizaje máquina** por su nombre en inglés, *Machine Learning*) es una rama de la inteligencia artificial que se centra en la creación de algoritmos capaces de aprender de forma automática a partir de la información que se les aporta [3].

También puede ser definido como el conjunto de métodos que permiten detectar automáticamente patrones en los datos y utilizar posteriormente estos patrones para predecir datos futuros o tomar decisiones en situaciones inciertas [4].

Es típico clasificar los métodos de aprendizaje automático en tres grandes grupos, en función del objetivo a cumplir [4]:

- **Aprendizaje supervisado:** El objetivo es aprender una función que, para un conjunto de entradas X, devuelve una salida Y. Para aprender esta función se utiliza un conjunto de pares entrada-salida $D = \{(x_i, y_i)\}_{i=1}^{N}$ ya etiquetados correctamente del que aprender. D es conocido como el conjunto de entrenamiento, siendo N el número de ejemplos en el conjunto de entrenamiento.

Algunos ejemplos típicos de problemas resueltos con aprendizaje supervisado son los problemas de clasificación o de regresión numérica.

- **Aprendizaje no supervisado:** En este caso, el objetivo es aprender una función capaz de extraer información útil de un conjunto de datos $D = \{(x_i)\}_{i=1}^{N}$ del que conocemos únicamente las entradas, pero no ningún tipo de salida o clasificación de antemano.

  Un problema típico resuelto por aprendizaje no supervisado es, por ejemplo, el problema del agrupamiento o *clustering*.

- **Aprendizaje por refuerzo:** El aprendizaje por refuerzo es un tipo distinto de aprendizaje automático, en el que un agente aprende la forma óptima de actuar a partir de la observación del entorno y de la obtención de recompensas y penalizaciones [5]. Este método de aprendizaje es el utilizado por el proyecto descrito en este trabajo, y será explicado con un mayor nivel de detalle en una sección posterior de este capítulo.

Existen diversos algoritmos de aprendizaje automático [5], entre los que se incluyen algunos como máquinas de vector soporte, árboles de decisión, redes Bayesianas... Uno de estos algoritmos son las **redes neuronales**, paradigma utilizado por nuestro trabajo y que será descrito a continuación.

3.1.2. Redes neuronales

En el campo de la inteligencia artificial, una **neurona** o **perceptrón** es la unidad básica lógica que forma una red neuronal artificial [6]. En esencia, una neurona es una función matemática no lineal que, a partir de unas entradas, devuelve una salida. Estas neuronas están conectadas entre ellas a través de conexiones dirigidas, lo que permite propagar la activación (salida) de la neurona a todas las neuronas sucesoras.

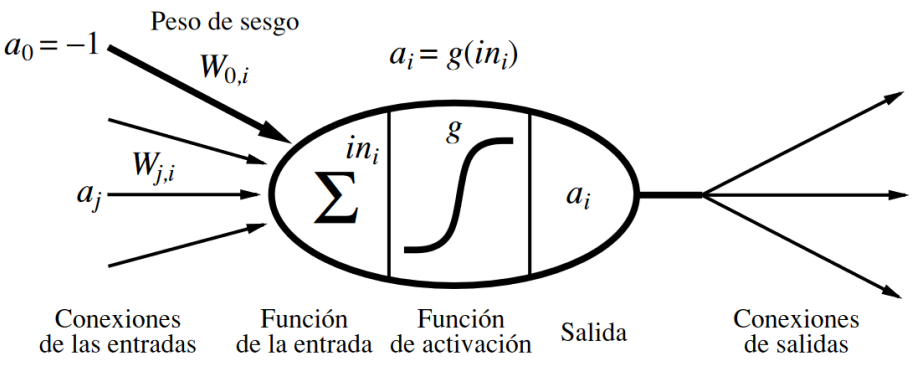


Figura 3.1: Modelo matemático de una neurona.

La estructura típica de una neurona, como se puede observar en la figura 3.1 [6], consta de las siguientes partes:

- **Conexiones de las entradas:** Es el conjunto de entradas que recibe la neurona, pudiendo ser éstas directamente las entradas que recibe la red o las activaciones a_j de una neurona j que sea antecesora de la neurona actual i.
- **Pesos:** Cada entrada de la neurona tiene asociada un peso numérico $W_{j,i}$, que indica la fuerza (a valores más grandes, más peso tiene) y el signo de esa entrada.
- **Función de entrada:** Esta función in_i es simplemente la suma ponderada de todas las entradas, siendo el valor de esta:

$$in_i = \sum_{j=0}^{n} W_{j,i} a_j$$

  donde n es el número total de entradas.

- **Función de activación:** La función de activación es una función g aplicada al resultado de la función de entrada para obtener la salida final de la neurona.

  La función de activación de una neurona se define siempre con dos objetivos principales:

  - Proporcionar salidas adecuadas: Se espera que la neurona esté "activa" (con una salida cercana a $+1$) cuando reciba entradas "correctas", y que esté "inactiva" (con una salida cercana a 0) cuando reciba entradas "incorrectas".
  - Tener una activación no lineal: Se busca siempre que la función de activación sea no lineal, o sería posible colapsar la red neuronal entera en una simple función lineal, limitando su capacidad .

  Algunas funciones de activación utilizadas son las siguientes [7]:

  - Función de activación umbral:

$$g(in_i) = \begin{cases} +1 & \text{si } x > 0. \\ 0 & \text{en otro caso.} \end{cases}$$

  - Función de activación lineal:

$$g(in_i) = c \times in_i$$

  - Función sigmoide/logística:

$$g(in_i) = \frac{1}{1 + e^{-in_i}}$$

La función de activación utilizada con más frecuencia es la sigmoide, ya que es diferenciable (algo que será necesario posteriormente para el algoritmo de aprendizaje de la red neuronal [6]).

- **Salida:** La salida a_i de la neurona no es más que el resultado de la función de activación $g(in_i)$. Esta salida se puede propagar (ser utilizada como entrada) a otras neuronas, o puede ser directamente la salida de la red neuronal.

Como se puede observar en la estructura, todas las neuronas cuentan con lo que se conoce como un **peso de sesgo** $W_{0,i}$, asociado a una entrada de valor fijo $a_0 = +1$ [6]. Este peso supone el umbral real de la neurona, es decir, la neurona se "activará" cuando la suma de los pesos reales $\sum_{j=1}^{n} W_{j,i}a_j$ sea superior al peso de sesgo $W_{0,i}$. Esto nos permite "desplazar" la activación de la neurona en una dirección sin alterar su pendiente, de forma similar al término independiente de una recta.

Una vez definida qué es una neurona, es posible definir propiamente una red neuronal. Una **red neuronal artificial** no es más que una agrupación de neuronas artificiales trabajando en conjunto. Aunque existen varias categorías de redes neuronales, las más importantes y con las que vamos a trabajar son las **redes neuronales con propagación hacia adelante**, y dentro de estas concretamente con los **perceptrones multicapa**.

Una **red neuronal con propagación hacia adelante** es una red neuronal que representa directamente una función de sus propias entradas (es decir, no conserva ningún estado interno más allá de los pesos de sus neuronas [6]). Este tipo de redes neuronales es especialmente importante porque es capaz de aproximar cualquier función continua utilizando los pesos apropiados.

En este tipo de redes neuronales, es típico que las neuronas se organicen en **capas** consecutivas, de forma que una capa de neuronas recibe entradas únicamente de las neuronas de su capa anterior y sus salidas se propagan únicamente a las neuronas de su capa posterior [6]. La razón por la que se conocen como redes neuronales de propagación hacia adelante es por la forma en la que se aprovechan estas capas para obtener las salidas: las salidas de las neuronas de la primera capa sirven como entrada para las neuronas de la segunda capa y así sucesivamente hasta la última capa, cuyas salidas son las salidas de la red neuronal propiamente dicha.

Un **perceptrón multicapa** es la estructura más típica de una red neuronal, siendo ésta una red neuronal con propagación hacia adelante agrupada en múltiples capas con un número variable de neuronas en cada capa. Además, se cumple que las uniones entre neuronas de capas contiguas son todas-con-todas (es decir, una neurona de la capa i recibe entradas de todas las neuronas de la capa $i-1$ y propaga su salida a todas las neuronas de la capa $i+1$). Un ejemplo de esta estructura queda reflejado en la figura 3.2 [8]. Como se puede ver, existen tres tipos de capas:

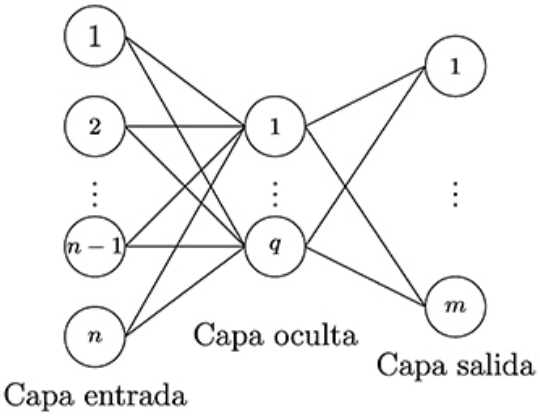


Figura 3.2: Ejemplo de perceptrón multicapa.

- **Capa de entrada:** La primera capa del perceptrón, recibe directamente las entradas a la red neuronal. Es típico que la capa de entrada no tenga ninguna función de activación en sus neuronas [9], propagando simplemente sus valores.
- **Capas ocultas:** Una o más capas de neuronas entre la capa de entrada y la capa de salida.
- **Capa de salida:** La última capa del perceptrón, sus salidas son las salidas del perceptrón multicapa como tal.

El objetivo principal a la hora de desarrollar una red neuronal es encontrar un conjunto de valores para sus pesos tal que la salida de la red neuronal se aproxime lo máximo posible al comportamiento real del problema que se está modelando [9]. Esto se consigue a través de un proceso conocido como **entrenamiento**.

Un conjunto de entrenamiento (formado por pares $(x_k, y_k), k = 1, 2, \ldots, P$ donde x_k es el conjunto de entradas de la red neuronal, y_k el conjunto de salidas esperadas de la red neuronal y P el tamaño total del conjunto de entrenamiento) es procesado por la red neuronal. El rendimiento de la red neuronal es medido por la diferencia entre la salida real obtenida por la red y la salida esperada para cada entrada del conjunto de entrenamiento. Esta diferencia, conocida como **error**, suele ser cuantificada utilizando la **suma de errores cuadráticos**. La idea es ajustar los pesos de la red neuronal para minimizar este error.

El algoritmo utilizado para entrenar una red neuronal se conoce como **retropropagación** (o *backpropagation*, su nombre en inglés). Este algoritmo consiste en ajustar los pesos de la capa de salida para minimizar el error, y posteriormente propagar esta modificación a todas las capas anteriores para ajustar sus pesos usando un gradiente descendiente, reduciendo el error total de la red neuronal [9].

El primer paso del entrenamiento de una red neuronal consiste simplemente en

inicializar aleatoriamente los pesos de la red [9]. Lo más típico es usar una distribución aleatoria uniforme para generar estos pesos, comenzando con valores pequeños. Posteriormente, se procederá a actualizar estos pesos para acercarlos a los pesos esperados.

Para un registro concreto del conjunto de entrenamiento (x_i, y_i) con unos pesos w en la red neuronal, el error para ese registro será:

$$E(x : w) = \frac{1}{2}(y - \hat{y})$$

donde y es la salida esperada para esa entrada e $\hat{y}$ es la salida obtenida realmente por la red neuronal. Asumiendo una función de activación sigmoide (para simplificar los cálculos), la salida de una neurona en la capa de salida sería:

$$\hat{y} = sigmoide(\sum_k w_k h_k)$$

donde h_k son los valores emitidos por las neuronas de la capa oculta anterior, y w_k los pesos asociados a dichas conexiones.

Si se deriva este error respecto a cada peso, obtenemos:

$$\nabla E(w) = \frac{\partial E}{\partial w_k} = -(y - \hat{y})\hat{y}(1 - \hat{y})h_k$$

Este gradiente $\nabla E(w)$ es la dirección que produce el mayor aumento del error desde un determinado punto del espacio. Por tanto, su negación $-\nabla E(w)$ es la dirección que maximiza la disminución del error. De esta forma, la actualización del peso $w_{i,k}$ (peso dirigido desde la neurona i hasta la neurona k) sería:

$$w_{i,k} \leftarrow w_{i,k} + \eta \delta_k h_i$$

donde η es la razón de aprendizaje de la red neuronal (la velocidad a la que aprende), h_i es la salida de la neurona i y con:

$$\delta_k = (y_k - h_k)h_k(1 - h_k)$$

donde y_k es la salida esperada de la neurona k y h_k la salida obtenida. Es típico que el valor de η sea pequeño (alrededor de 0.05).

Esta formula nos es útil para cualquier peso que esté dirigido hacia una neurona de salida, ya que conocemos la salida esperada y por tanto podemos ajustar el error. Ahora bien, el error en las capas ocultas es una incógnita ya que no podemos saber de antemano cuál debería ser la salida correcta de estas neuronas [6].

Por esto, para solucionar este problema, lo que se hace es propagar el error de la capa de salida hacia las capas ocultas. Se puede considerar que cada neurona oculta j es "responsable" de una fracción del error de cada neurona a la que propaga su salida.

Por tanto, lo que se tendrá en cuenta es la influencia que ha tenido esta neurona en el error final de cada neurona sucesora a ésta, δ_j.

La actualización de peso $w_{j,i}$ (dirigido de la neurona j a la neurona i) adaptada para el caso de las neuronas ocultas es la siguiente:

$$w_{j,i} \leftarrow w_{j,i} + \eta \delta_i h_j$$

donde h_j es la salida de la neurona j y con:

$$\delta_i = \left(\sum_{x \in s(i)} w_{i,x} \delta_x \right) h_i (1 - h_i)$$

donde $s(i)$ son los sucesores de la neurona i en la siguiente capa.

Algoritmo 1: Algoritmo de retropropagación

1. Inicializar todos los pesos con valores aleatorios.

2. Mientras no se cumpla la condición de parada:

2.1. Para cada ejemplo de entrada (x, y), hacer:

2.1.1. Calcula la salida h_N que tendrá cada neurona N, utilizando **propagación hacia adelante**.

2.1.2. Para cada neurona N, comenzando por la última capa y retrocediendo capa a capa hasta la primera capa:

Calcula δ_N.

Si N es una neurona de salida:

$$\delta_N = (y_N - h_N) h_N (1 - h_N)$$

Si N es una neurona de una capa oculta:

$$\delta_N = \left(\sum_{x \in s(N)} w_{N,x} \delta_x \right) h_N (1 - h_N)$$

Para todas las neuronas $R \in predecesores(N)$, calcular:

$$\Delta_{RN} = \eta \delta_N h_R$$

2.1.3. Para todos los pesos w_{RN} en la red, calcular:

$$w_{RN} \leftarrow w_{RN} + \Delta_{RN}$$

Figura 3.3: Pseudocódigo del algoritmo de retropropagación.

El pseudocódigo para el algoritmo se puede observar en la figura 3.3 [10]. El algoritmo se ejecuta sobre el conjunto de entrenamiento entero, pudiendo elegir además repetirlo varias veces. Cada vez que el algoritmo se ejecuta sobre el conjunto de entrenamiento entero se conoce como una **época** (generalmente llamado *epoch* por su nombre en inglés) [6].

La **condición de parada** del algoritmo indica cuándo dejará de entrenarse la red. En general, la condición de parada suele ser un umbral para el error o un número concreto de *epochs*.

3.1.3. Deep Learning

El **aprendizaje por representación** es un conjunto de métodos dentro del aprendizaje automático que permiten a una máquina recibir información sin procesar y automáticamente procesar esta información, extrayendo características relevantes para la clasificación o detección.

A partir de esta definición, se puede definir **Deep Learning** como un subconjunto del aprendizaje automático formado por métodos de aprendizaje por representación con múltiples niveles de representación (obtenidos componiendo módulos simples, que van aumentando progresivamente el nivel de abstracción de la representación) [1]. La idea clave de estos módulos es que no son diseñados por humanos, sino que son aprendidos de forma automática por los algoritmos.

Actualmente, Deep Learning se ha convertido en el estado del arte en gran cantidad de problemas antes inabordables como puede ser el reconocimiento de imágenes, el reconocimiento de voz... y en general, cualquier problema con mucha dimensionalidad.

A nivel de implementación, Deep Learning es típicamente implementado utilizando **perceptrones multicapa** con muchas capas [11] (aunque la definición de red neuronal profunda es un perceptrón multicapa con más de una capa oculta), en la que cada capa representa un nivel de abstracción distinto. Ahora bien, existen más estructuras de redes neuronales consideradas parte de Deep Learning (como las **redes neuronales convolucionales** o las **redes neuronales recursivas**).

El notable aumento de tamaño en las redes neuronales (con el consecuente crecimiento del número de pesos) plantea varios problemas:

- Los conjuntos de entrenamiento deben ser mucho más grandes para obtener un resultado adecuado (ya que es necesario ajustar un número de pesos mucho mayor).

- El entrenamiento utilizando retropropagación es mucho más lento (por la misma razón anterior).

- **Problema del gradiente desvaneciente** (*Vanishing gradient*): Al aumentar el numero de capas, el efecto de la retropropagación del error cuando se usan funciones de activación tradicionales se reduce rápidamente al diluirse la propagación del error conforme atraviesa capas (siendo cada vez menor su efecto al ajustar pesos).

La solución más típica a estos problemas es cambiar la función de activación de las neuronas, popularizándose el uso de la función *ReLU* (de sus siglas en inglés *Rectified Linear Unit* [1]):

$$g(z) = max(0, z)$$

Esta función de activación es útil por diversos motivos: es **diferenciable** (y por tanto se puede utilizar con retropropagación), es **rápida de calcular** (lo que acelera notablemente el entrenamiento de las redes) y **evita el problema del gradiente desvaneciente** (manteniendo el efecto de la retropropagación aunque el número de capas sea muy elevado). Por todo esto, *ReLU* es actualmente la función de activación más típica en redes neuronales profundas (término usado para referirse a las redes neuronales utilizadas por Deep Learning).

3.2. Aprendizaje por refuerzo: Q-Learning

3.2.1. Aprendizaje por refuerzo

El **aprendizaje por refuerzo** es un conjunto de métodos de aprendizaje automático que consisten en enseñar a un agente a actuar de forma óptima en cada situación para maximizar una recompensa numérica [12].

Existen dos características principales que distinguen al aprendizaje por refuerzo de otros tipos de aprendizaje (como el supervisado o no supervisado):

- El agente no conoce de antemano qué acción debe realizar en cada situación, sino que debe probar acciones para evaluar su utilidad.

- Las acciones, además de tener una recompensa inmediata, pueden afectar al entorno y a las recompensas futuras que se obtengan.

Otra diferencia clave es que no se definen métodos de aprendizaje automático, sino **problemas de aprendizaje automático** [12]. Cualquier método que sea capaz de resolver uno de estos problemas será considerado un método de aprendizaje automático.

Los dos componentes principales de un problema de aprendizaje automático son el **agente** (la parte que aprende y toma decisiones) y el **entorno** (todo lo que no es el agente, la parte con la que interactúa el agente y sobre la que no tiene control directo).

Además, hay tres elementos necesarios para poder definir el problema, cuya correcta definición es importante para poder resolver el problema adecuadamente [12]:

- **Estado:** Una representación del estado actual del entorno conforme lo interpreta el agente. Esta representación, s_t, pertenece a un conjunto total de estados S.
- **Acción:** La actuación que realiza el agente, tomada dependiendo del estado actual. Una acción a_t pertenece a un conjunto de acciones en un estado concreto, $A(s_t)$.
- **Recompensa:** Una recompensa o penalización otorgada por el entorno tras la acción del agente. La recompensa es un simple número, y el objetivo final del agente es maximizar la recompensa obtenida.

  La razón para usar recompensas es poder **formalizar la meta** que tiene el agente. Por tanto, las recompensas que el agente reciba deben servir para que lleven al agente a alcanzar su meta si maximiza las recompensas.

  La recompensa obtenida viene del entorno, en vez de ser procesada directamente por el agente como podría ser intuitivo. Esto se debe a que la meta del agente siempre debería ser algo sobre lo que tiene control **incompleto** (por ejemplo, el agente no debería ser capaz de determinar por sí mismo que ha recibido una recompensa de un valor concreto tras una acción arbitraria), y no debe depender del agente.

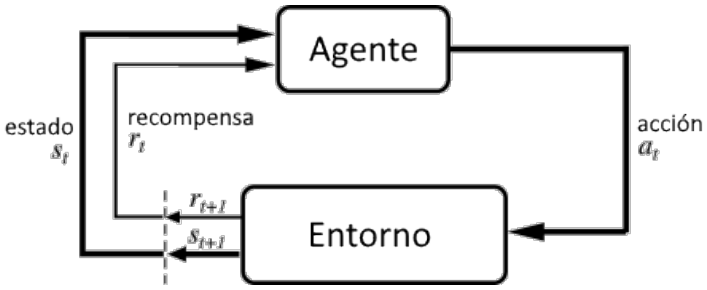


Figura 3.4: Interacción entre agente y entorno en el aprendizaje por refuerzo.

El bucle de actuación general de un agente se puede observar en la figura 3.4 [12], siendo un bucle continuo en el que, en cada paso t:

- El agente recibe un estado s_t del entorno.
- El agente interpreta el entorno y elige una acción a_t, que ejecuta.
- El entorno es modificado por la acción ejecutada y devuelve al agente el nuevo estado s_{t+1} junto a la recompensa obtenida r_{t+1}.

Una propiedad que tiene que cumplir la definición del estado para que éste sea válido en el marco de un problema de aprendizaje por refuerzo es la **Propiedad de Markov:** el estado debe ser capaz de representar toda la información relevante [12]. Traducido al bucle que acabamos de ver, un problema se dice que es de Markov si el estado alcanzado s_{t+1} depende únicamente del estado previo s_t y de la acción tomada a_t, y no de ningún estado o acción anterior.

Además del agente y el entorno, existen otros sub-elementos relevantes para los sistemas de aprendizaje automático, siendo estos [12]:

- **Política (π):** La política define la forma de actuar del agente en un momento concreto, siendo ésta una función $\pi(s) = a$ que toma como entrada el estado actual y devuelve la acción que debe realizarse. La política es un punto clave del aprendizaje por refuerzo, ya que es el objetivo a optimizar y por sí misma es capaz de determinar la actuación de un agente.

- **Modelo de recompensas ($R(s, a)$):** El modelo de recompensas es una función que, para cada estado (o par estado-acción) del entorno devuelve la recompensa apropiada al agente, asignando por tanto la "deseabilidad" a cada estado (los estados deseables tendrán mayores recompensas que los estados no deseables).

- **Modelo:** El modelo es una representación del entorno y de su comportamiento, siendo capaz de (por ejemplo) predecir el estado que se alcanzará y la recompensa que se obtendrá si se aplica una acción en un estado concreto. El modelo es un elemento opcional (no todos los algoritmos de aprendizaje automático utilizan un modelo).

Podemos definir la **utilidad** de un conjunto de estados $U_h(s_0, \ldots, sn)$ como la suma de recompensas obtenidas en cada estado [6]. Es típico que la utilidad utilice **recompensas depreciativas**, utilizando un factor de descuento $\gamma \epsilon [0, 1]$ que va devaluando la recompensa de los estados futuros. Esto permite representar una preferencia por las recompensas recientes frente a las recompensas del futuro lejano.

Es posible medir la calidad de una política aprendida por un agente midiéndola como la utilidad esperada de la secuencia de estados que generaría esa política, siendo esta:

$$U^{\pi} = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi\right]$$

Esta función se puede extrapolar a la utilidad esperada cuando se empieza desde un estado concreto s:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi, s_0 = s\right]$$

El objetivo de cualquier algoritmo de aprendizaje por refuerzo es aprender la **política óptima** π^*, siendo esta la política que maximiza la utilidad esperada.

3.2.2. Q-Learning

Q-Learning es un algoritmo de aprendizaje por refuerzo sin modelo que permite a los agentes aprender a actuar de forma óptima en entornos Markovianos (que cumplen la propiedad de Markov) sin necesidad de construir un modelo del entorno [13].

El funcionamiento del algoritmo es simple: el agente prueba una acción concreta en un estado y evalúa las consecuencias de esa acción en base a la recompensa obtenida y a la estimación del valor del estado que se ha alcanzado. Probando acciones en todos los estados, el agente será capaz de aprender la mejor acción para cada estado [13].

En este algoritmo, la utilidad pasa a denotarse como $Q(s, a)$, representando ésta la utilidad esperada de ejecutar la acción a en el estado s siguiendo la política π del agente.

Conociendo s' (el estado alcanzado tras aplicar la acción a en el estado s) y r (la recompensa obtenida por haber aplicado la acción a en el estado s), se puede calcular la utilidad de la siguiente manera:

$$Q(s, a) = r \quad \text{si } s' \text{ es un estado final}$$

$$Q(s, a) = r + \gamma U^*(s') \quad \text{en cualquier otro caso}$$

siendo $U^*(s')$ la máxima utilidad esperada en el estado s'.

A partir de estas funciones de utilidad $Q(s, a)$ es posible aprender la política óptima. La acción óptima para cada estado se obtendría de la siguiente manera:

$$\pi^*(s) = argmax_a Q(s, a)$$

es decir, la acción que maximice la utilidad de ese estado. Además, es posible vincular la máxima utilidad esperada de un estado con la función de utilidad de la siguiente forma:

$$U^*(s) = max_a Q(s, a)$$

Por tanto, las funciones anteriores se pueden reescribir de la siguiente manera:

$$Q(s, a) = r \quad \text{si } s' \text{ es un estado final}$$

$$Q(s, a) = r + \gamma \cdot max_{a'} Q(s', a') \quad \text{en cualquier otro caso}$$

Esta ecuación es recurrente y será la base para obtener un algoritmo iterativo.

Nuestro objetivo con el algoritmo de Q-Learning es obtener una estimación del valor real de Q, denominada como $\hat{Q}$. $\hat{Q}$ es una tabla (teniendo por ejes los posibles estados y acciones) donde en cada celda se almacena la utilidad estimada de ejecutar la acción a en el estado s [13].

Teniendo esto en cuenta, las ecuaciones anteriores adaptadas para estimar $\hat{Q}$ son las siguientes:

$$\hat{Q}(s,a) = r \quad \text{si } s' \text{ es un estado final}$$

$$\hat{Q}(s,a) = r + \gamma max_{a'}\hat{Q}(s',a') \quad \text{en cualquier otro caso}$$

Esta ecuación tiene un problema, y es que es válida únicamente para entornos deterministas. En caso de entornos no deterministas puede provocar problemas de oscilación y no convergencia.

Esto se puede solucionar adaptando la ecuación a entornos no deterministas, siendo la ecuación resultante [12]:

$$\hat{Q}(s,a) = \hat{Q}(s,a) + \alpha \left[r + \gamma max_{a'}\hat{Q}(s',a') - \hat{Q}(s,a) \right]$$

que, reescrita para mayor claridad, sería:

$$\hat{Q}(s,a) = (1-\alpha)\hat{Q}(s,a) + \alpha \left[r + \gamma max_{a'}\hat{Q}(s',a') \right]$$

El término α representa el peso que se da a la nueva información recibida. Como se puede observar, el nuevo valor de $\hat{Q}(s,a)$ es la suma ponderada de dos términos: el término de $(1-\alpha)$ - el valor actual de $\hat{Q}(s,a)$ - y el término de α - el nuevo valor de $\hat{Q}(s,a)$ calculado. De esta forma, se evita que el no determinismo provoque grandes oscilaciones en los valores estimados de $\hat{Q}$.

El algoritmo general de Q-Learning utilizado queda reflejado en la figura 3.5 [12]:

Un problema que tiene este algoritmo es que es **voraz** (es decir, siempre que el agente está en el estado s elegirá la acción a que maximice $\hat{Q}(s,a)$). Esto puede provocar que el algoritmo converja a un máximo local (ya que no será capaz de explorar otras posibilidades que quizás sean mejores).

Por esto, es necesario alcanzar un equilibrio entre **exploración** (maximizar el buen comportamiento a largo plazo buscando posibilidades mejores) y **explotación** (maximizar la utilidad realizando las mejores acciones en cada estado) [6].

Esto se suele conseguir utilizando un parámetro adicional, **epsilon** (ϵ). Epsilon es una variable que se fija antes del entrenamiento ($\epsilon \in [0,1]$) y que sirve para generar acciones aleatorias. En el paso 2.2.1. del algoritmo (visto en la figura 3.5) se generaría un número aleatorio entre 0 y 1. Si este número es mayor o igual que ϵ se realizará la acción que maximice la utilidad (el comportamiento actual del algoritmo), pero si es menor se realizará una acción aleatoria. De esta forma se obtendrá un equilibrio entre

Algoritmo 2: Algoritmo de Q-Learning

1. Inicializar $\hat{Q}$ con valores aleatorios

2. Mientras no se cumpla la condición de parada:

2.1. Obtener estado inicial s

2.2. Repetir:

2.2.1. Seleccionar la acción a según la política óptima π^*, calculada a partir de $\hat{Q}$.

2.2.2. Ejecutar la acción a en el estado s

2.2.3. Obtener del entorno el nuevo estado s' y la recompensa r.

2.2.4. Actualizar $\hat{Q}(s,a)$:

$\hat{Q}(s,a) = (1-\alpha)\hat{Q}(s,a) + \alpha\left[r + \gamma max_{a'}\hat{Q}(s',a')\right]$

2.2.5. $s \leftarrow s'$

hasta que s **sea un estado final.**

3. Generar la política óptima π^* a partir de $\hat{Q}$.

Figura 3.5: Pseudocódigo del algoritmo de Q-Learning.

exploración (las acciones aleatorias permitirán explorar estados nuevos) y explotación (ya que se sigue realizando la mejor acción).

3.3. Integración de Deep Learning y Q-Learning: Deep Q-Learning y antecedentes

El algoritmo de **Q-Learning** tiene una serie de limitaciones a la hora de garantizar que acabe convergiendo [14]:

- Todos los pares de estado-acción tienen que poder ser representados de forma discreta.
- Todas las acciones tienen que ser probadas en todos los estados repetidamente (tiene que haber suficiente exploración como para que no sea necesario un modelo del entorno)

Esto supone que el algoritmo es incapaz de funcionar en entornos cuyos estados o acciones no puedan ser representadas de forma continua. Además, los problemas que contengan un gran número de estados u acciones posibles tampoco pueden ser resueltos con Q-Learning en la práctica, ya que el tamaño de la tabla $\hat{Q}$ es excesivo como para almacenarla en memoria [15].

Por tanto, es típico usar un aproximador de funciones genérico para aprender la función de valor (tomando estados como entrada). Al ser las **redes neuronales pro-**

fundas aproximadores de funciones no lineales muy potentes, son una buena posibilidad para cumplir esta función. De esta forma surge **Deep Q-Learning** [2], una familia de algoritmos que juntan el algoritmo de Q-Learning con las capacidades de las redes neuronales profundas para poder trabajar con problemas de dimensionalidad mayor y características distintas.

3.3.1. Antecedentes: Fitted Q-Learning y Neural Fitted Q-Learning

Antes de llegar al algoritmo de Deep Q-Learning como tal, hubo varios intentos de aplicar aproximadores de funciones a Q-Learning para paliar sus problemas.

Uno de estos intentos fue **Fitted Q-Learning** [14]. En esta versión de Q-Learning, las experiencias D se almacenaban en forma de tuplas $< s, a, r, s' >$, donde s es el estado, a la acción realizada en ese estado, r la recompensa obtenida por ese par estado-acción y s' el estado alcanzado.

El algoritmo utilizado para aproximar la utilidad Q de un par estado-acción (sea cual sea este algoritmo) comienza con una inicialización aleatoria de los valores de Q, $Q(s, a; \theta_0)$, donde θ_0 son los parámetros iniciales del algoritmo. Tras esto es posible actualizar la aproximación de los valores de Q en una iteración k concreta, $Q(s, a; \theta_k)$ hacia su valor esperado:

$$Y_k^Q = r + \gamma \underset{a' \in A}{max} Q(s', a'; \theta_k)$$

donde Y_k^Q es el valor que debería devolver la función usada para aproximar los valores de Q, y θ_k los parámetros del algoritmo en la iteración k.

Cuando se aplica Fitted Q-Learning utilizando una red neuronal como función aproximadora, el algoritmo pasa a conocerse como **Neural Fitted Q-Learning** [2]. En este caso, la red neuronal tomará como entrada el estado actual s y devolverá como salida el valor Q, $Q(s, a)$ para cada posible acción.

Los valores de Q, como ya se ha indicado antes, serán aproximados utilizando una red neuronal con una función $Q(s, a; \theta_k)$, siendo los parámetros θ_k (los pesos de la red neuronal) actualizados utilizando **gradiente descendente estocástico (retropropagación)** o cualquier variante de este algoritmo, buscando minimizar el error cuadrático:

$$L_{DQN} = \left(Q(s, a; \theta_k) - Y_k^Q \right)^2$$

Por tanto, la actualización de los valores de Q-Learning es simplemente una actualización de los pesos de la red utilizando retropropagación, usando la fórmula:

$$\theta_{k+1} = \theta k + \gamma \left(Y_k^Q - Q(s, a; \theta_k) \right) \nabla_{\theta_k} Q(s, a; \theta_k)$$

donde γ es el factor de aprendizaje y ∇_{θ_k} es el gradiente que maximiza la disminución del error en los pesos en la iteración k.

La elección del error cuadrático no es una decisión arbitraria, ya que asegura que la aproximación $Q(s, a; \theta_k)$ tenderá al valor real $Q^*(s, a)$ con suficientes iteraciones y suficiente experiencia en el conjunto D [14].

Ahora bien, esta propuesta tiene también algunos problemas:

- A la vez que se actualizan los pesos de la red neuronal, las salidas esperadas de la red neuronal (los valores devueltos para el estado siguiente s') también van cambiando. Teniendo en cuenta la capacidad de generalización y extrapolación de una red neuronal, esto puede provocar que se acumulen errores y que no se garantice la convergencia final de la función, llegando incluso a divergir.

- Los valores de Q devueltos por la función y aprendidos suelen sobreestimarse, al utilizar el operador max para obtener la mejor salida esperada de la red neuronal.

Por todo esto, esta propuesta resulta muy inestable y hay que diseñarla con especial cuidado para conseguir un aprendizaje adecuado.

3.3.2. Deep Q-Learning

El algoritmo de **Deep Q-Learning** [2] se puede entender como una mejora respecto al Neural Fitted Q-Learning, aplicando algunas restricciones adicionales para evitar la inestabilidad:

- **Valores esperados Q fijos:** Se utiliza una segunda red neuronal adicional, la **red de objetivo** (*Target Q-Network*) [2], utilizada a la hora de calcular la salida esperada usando la formula $Y_k^Q = r + \gamma \underset{a' \in A}{max} Q(s', a'; \theta_k)$, pasando esta a ser:

  $$Y_k^Q = r + \gamma \underset{a' \in A}{max} Q(s', a'; \theta_k^-)$$

  donde θ_k^- son los pesos de la red de objetivo. Esta red neuronal se actualiza únicamente cada C iteraciones del algoritmo (siendo típico actualizarla, por ejemplo, tras alcanzar un estado final), siendo su actualización simplemente copiar los pesos de la red neuronal usada para calcular los valores de Q.

  De esta forma, se evita que las inestabilidades se propaguen rápidamente y se reduce la posibilidad de que el algoritmo diverja, ya que los valores esperados Y_k^Q se mantendrán fijos [14].

- ***Experience Replay***: El algoritmo almacena en una estructura de datos (conocida como el *Replay Memory*) toda la experiencia obtenida de las últimas N iteraciones del algoritmo, obteniendo esta experiencia utilizando una política de exploración-explotación [14].

  Las experiencias se almacenan con una estructura de $< s, a, r, s' >$ (estado s, acción ejecutada a, recompensa r y estado alcanzado s'). Los pesos de la red neuronal se van actualizando a partir de muestras aleatorias pequeñas del *Replay Memory*.

  De esta forma, las actualizaciones de pesos pueden abarcar un mayor número de estados y acciones (ya que se almacenan experiencias viejas, se muestrea al azar y todas las experiencias tienen la misma posibilidad de ser elegidas) [2]. Además evita las correlaciones temporales (al mezclar experiencias viejas y nuevas por el muestreo), permite que las experiencias poco frecuentes se aprendan varias veces y permite optimizar el rendimiento del algoritmo paralelizando el aprendizaje de experiencias con técnicas modernas como el uso de GPUs.

- **Beneficios de Deep Learning**: Es posible aprovechar diversas ventajas de Deep Learning [2] como puede ser el pre-procesado de la entrada para reducir su dimensionalidad, la utilización de capas de convolución para poder trabajar con imágenes o el uso de variantes de gradiente descendiente estocástico más avanzadas como **RMSprop** o **Adam** [16].

A la hora de la implementación, resulta muy similar a la de Neural Fitted Q-Learning: la red neuronal toma como entrada un estado y devuelve como salidas los valores $Q(s, a; \theta_k)$ para cada combinación de estado y acción. Ahora bien, es típico que la red neuronal tome una secuencia de estados preprocesados ϕ (el estado actual y sus predecesores, siendo esta una función $\phi(s)$) como entrada, representando la historia que ha llevado hasta el estado actual, para permitir a la red neuronal tener en cuenta el contexto del estado actual [17].

El pseudocódigo del algoritmo queda reflejado en la figura 3.6 [17].

3.3.3. Mejoras: Prioritized Experience Replay

Como ya se ha visto antes, el *Experience Replay* permite al agente almacenar experiencias anteriores y aprender a partir de ellas en cualquier momento, posibilitando aprender varias veces de ellas y rompiendo la correlación temporal. Ahora bien, el muestreo que se hace del *Replay Memory* sigue una distribución de probabilidad uniforme, lo cual significa que todas las experiencias tienen la misma posibilidad de ser elegidas. Esto supone una limitación, ya que existen experiencias más útiles que otras (ya sea

Algoritmo 3: Algoritmo de Deep Q-Learning

1. Inicializar el *Replay Memory* D con un tamaño máximo N.

2. Inicializar la función aproximadora Q (la red neuronal) con pesos aleatorios.

3. Para cada *epoch* desde 1 hasta M:

3.1. Obtener estado inicial $s_1 = \{x_1\}$ (donde x_1 es la imagen del estado sin procesar) y preprocesarlo $\phi_1 = \phi(s_1)$

3.1. Para cada momento t desde 1 hasta el final del *epoch*:

3.1.1. Elegir acción:

Con probabilidad ϵ, elegir acción aleatoria a_t.

Si no, selecciona acción $a_t = max_a Q^*(\phi(s_t), a; \theta)$

3.1.2. Ejecutar la acción a_t, obtener recompensa r_t e imagen del nuevo estado x_{t+1}.

3.1.3. Fijar estado actual $s_{t+1} = s_t, a_t, x_{t+1}$ y preprocesar $\phi_{t+1} = \phi(s_{t+1})$.

3.1.4. Almacenar experiencia $(\phi_t, a_t, r_t, \phi_{t+1})$ en el *Replay Memory* D.

3.1.5. Tomar muestra aleatoria de experiencias $(\phi_j, a_j, r_j, \phi_{j+1})$ del *Replay Memory* D.

3.1.6. Fijar el valor de y_j $\begin{cases} r_j & \text{si } \phi_{j+1} \text{ es terminal.} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{en otro caso.} \end{cases}$

3.1.7. Realizar gradiente descendiente usando $(y_j - Q(\phi_j, a_j; \theta))^2$ como error.

Figura 3.6: Pseudocódigo del algoritmo de Deep Q-Learning.

por ser más relevantes, ser más raras...) y sería interesante poder aprender más esas experiencias que otras experiencias irrelevantes [18].

Una primera idea que se podría tener para solucionar esta limitación sería muestrear el *Replay Memory* de forma **voraz**: las experiencias se ordenarían por su error cuadrático

$$\delta = \left(Q(s, a; \theta) - Y^Q\right)^2$$

y se elegirían de forma determinista en orden de mayor a menor error. La primera vez que una experiencia es insertada, al no conocer su error, sería insertada con un error infinito (para garantizar que será elegida). Si bien esta idea reduce el tiempo de aprendizaje, tiene algunos problemas [18]:

- Para evitar tener que recalcular los errores con cada experiencia insertada, el error se recalcula únicamente cuando la experiencia ha sido elegida. Por tanto, las experiencias con un error bajo tardarán mucho tiempo en ser elegidas de nuevo (y, a efectos prácticos, nunca serán elegidas).

- Este método es muy sensible a los picos de ruido (por ejemplo, cuando se utilizan recompensas aleatorias o cuando se realizan aproximaciones).

- Al elegir únicamente las experiencias con mayor error, el agente se enfocará únicamente en una pequeña parte de las experiencias. Esta falta de diversidad puede provocar que la red neuronal acabe sobreajustando.

Para evitar estos problemas surge la técnica del ***Prioritized Experience Replay*** [18], un método de muestreo estocástico que resulta un punto medio entre el muestro uniforme y la selección voraz de experiencias. Este método garantiza que la probabilidad de que una experiencia sea elegida sea monótona respecto a la importancia de la experiencia, asegurando a la vez que ninguna experiencia tiene una probabilidad de ser elegida de cero (aunque sea la menos importante).

La probabilidad de elegir una experiencia i se define como:

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

donde p_i ($p_i > 0$) es la prioridad que tiene la transición i, y el exponente α es el grado de priorización a utilizar (siendo $\alpha = 0$ equivalente al muestreo aleatorio simple).

Existen dos variantes para el valor de la prioridad p_i:

- $p_i = |\delta_i| + \epsilon$, donde ϵ es una constante pequeña positiva, que garantiza que ninguna de las prioridades valdrá cero aunque el error valga cero.

- $p_i = \frac{1}{rango(i)}$, donde $rango(i)$ es el rango de la experiencia i cuando se ordena el *Replay Memory* de mayor a menor error.

Ambos casos garantizan una función monótona respecto al error, pero la segunda opción es más robusta al no ser sensible a casos extremos [18]. Aun así, ambas variantes consiguen grandes reducciones en el tiempo necesario para el aprendizaje del agente.

La utilización de *Prioritized Experience Replay* supone un problema: se añade un sesgo (ya que la distribución de la muestra de experiencias ha dejado de ser uniforme, provocando que la solución a la que converge la red neuronal sea distinta). Es posible solucionar ese sesgo utilizando pesos basados en la importancia de cada experiencia:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^\beta$$

(donde N es el número de experiencias almacenadas en el *Replay Memory*) que son capaces de compensar totalmente el sesgo cuando $\beta = 1$. Estos pesos se utilizan directamente en el algoritmo de Deep Q-Learning, sustituyendo la actualización del error δ_i por $\delta_i w_i$. Además, por razones de estabilidad, los pesos se normalizan usando un

factor de $\frac{1}{max_i w_i}$, garantizando así que la normalización siempre hará que los pesos sean iguales o menores a su valor original [18].

El efecto del sesgo es únicamente relevante al final del aprendizaje (cuando los cambios son menores), ya que el comienzo del aprendizaje es altamente inestable por cambios de pesos, política... Por tanto, el valor de β se va alterando linealmente durante el entrenamiento del agente, empezando con un valor pequeño ($\beta \in [0{,}4, 0{,}6]$) y llegando a $\beta = 1$ únicamente al final del entrenamiento [18].

Un beneficio de utilizar pesos basados en la importancia en conjunto con el *Prioritized Experience Replay* es que las experiencias con un error grande serán muestreadas muchas veces, pero su peso basado en la importancia será bajo, provocando que el error se modifique lentamente y evitando grandes oscilaciones.

Capítulo 4

JUEGO UTILIZADO: TETRIS

En este capítulo se describirá el juego sobre el que se pretende aplicar el algoritmo de Deep Q-Learning: Tetris. Se empezará con una descripción general del juego, su funcionamiento y sus objetivos. Tras esto, se describirá el desarrollo realizado del juego, remarcando las decisiones que se han tomado para su diseño (de cara a la implementación y al uso posterior de agentes) y la implementación en sí.

4.1. Descripción general del juego

Tetris es un videojuego de puzle diseñado originalmente por Alexey Pajitnov en 1984 con un gran éxito. Desde entonces, numerosas versiones y variaciones del juego han sido lanzadas para ordenador y toda clase de videoconsolas, convirtiéndose en uno de los juegos más importantes y reconocibles a nivel mundial en la actualidad.

Una partida de Tetris [19] se juega en una zona de juego que consta de veinte filas y diez columnas. Desde este momento en adelante, a la intersección de una fila y una columna se le llamará una **celda**.

De la parte superior de la zona de juego caen *tetraminos* (desde ahora referidos como **piezas** por simplicidad y recogidos todos los posibles en la figura 4.1). Estas piezas son elegidas al azar, ocupan todas cuatro celdas y caen de una en una (no es posible que haya dos piezas en juego a la vez).

Figura 4.1: Tetraminos (piezas) utilizados en Tetris.

El jugador tiene control sobre la pieza que está actualmente cayendo, pudiendo realizar las siguientes acciones:

- **Desplazar** la pieza hacia la izquierda o la derecha. Este movimiento se realiza en incrementos de una celda cada vez que se pulsa la tecla apropiada, y únicamente si hay hueco para realizar el desplazamiento.
- **Rotar** la pieza en incrementos de 90^{o}. Esta rotación solo se puede realizar si la pieza tiene suficiente hueco como para hacerlo (aunque algunas versiones del juego desplazan la pieza si no hay hueco, permitiendo el giro)
- **Dejar caer más rápido** la pieza hacia abajo.
- **Soltar** la pieza, dejando que caiga hasta el fondo instantáneamente.
- **Guardar** la pieza actualmente en juego. Esta pieza será almacenada y será sustituida por la pieza que hubiese guardada (si ya se había guardado alguna pieza antes) o por una pieza nueva (si es la primera vez que se guarda la pieza). Una vez se guarda una pieza, no se puede volver a guardar la pieza hasta que la nueva pieza en juego se haya bloqueado.

Las acciones del jugador no pueden hacer que la pieza en juego atraviese otras piezas ya bloqueadas en la zona de juego, ni que atraviese las paredes que marcan los límites de la zona de juego.

Además de los movimientos del jugador, las piezas van cayendo lentamente de forma automática, desde la parte de arriba de la zona de juego hasta la parte de abajo. Una vez la pieza se apoya sobre el suelo o sobre otra pieza ya bloqueada, se quedará bloqueada en su posición actual, impidiendo que el jugador la desplace.

Tras bloquear la pieza se comprueba si se han hecho una o más **líneas** (siendo una línea una fila de la zona de juego cuyas celdas están todas ocupadas por una pieza bloqueada). Si se han hecho, se eliminan esas filas (aumentando la puntuación y la dificultad de la partida en función del número de líneas) y se desplaza la posición de todas las piezas que estén bloqueadas sobre las líneas eliminadas, bajando tantas celdas como líneas se hayan eliminado.

Finalmente, se comprueba que la zona de juego no esté llena (si hay alguna celda ocupada por encima del limite superior de la zona de juego). Si está llena se acaba la partida, y si no se continua haciendo aparecer una nueva pieza y repitiendo el bucle.

La dificultad de la partida únicamente modifica la velocidad a la que cae automáticamente la pieza en juego. Cuanto mayor es la dificultad, más rápido cae la pieza (y por tanto, menos tiempo tiene el jugador para reaccionar).

Se puede observar un ejemplo de una partida de Tetris (sin la capacidad de almacenar piezas) en la figura 4.2.

El principal **objetivo** del usuario es aguantar el máximo tiempo posible sin que acabe la partida (ya que, por defecto, una partida de Tetris no acaba hasta que el

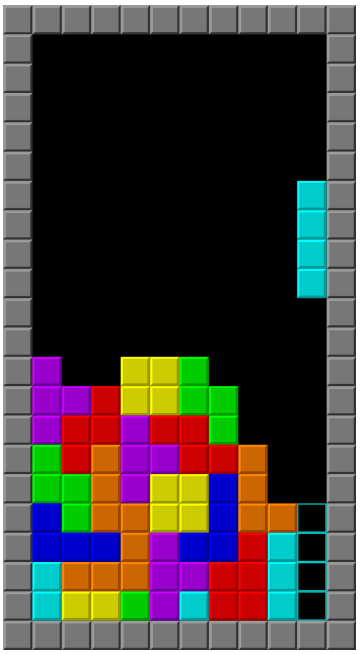


Figura 4.2: Ejemplo de partida típica de Tetris.

jugador pierda). Ahora bien, existen otros objetivos secundarios: **maximizar** el número de líneas eliminadas y la puntuación del jugador. En general, ambos objetivos están estrechamente relacionados:

- Eliminar líneas es la forma de adquirir puntuación.
- Cuanto mayor es la dificultad, más puntos se adquiere por eliminar una línea. Para aumentar la dificultad, es necesario eliminar un mínimo de líneas.
- Eliminar más de una línea multiplica exponencialmente la puntuación obtenida (hasta un máximo de cuatro líneas eliminadas a la vez, conocido como un **tetris**)

4.2. Desarrollo del juego

En vez de utilizar alguna versión de Tetris ya existente, se ha decidido desarrollar una versión propia especifica para este proyecto. De esta forma, se tiene control total sobre el desarrollo, permitiendo realizar simplificaciones de cara al entrenamiento posterior de los agentes y facilitando el propio uso de los agentes dentro del juego.

4.2.1. Decisiones de diseño

En esta sección se detallan todas las decisiones de diseño que se han tomado para el desarrollo del juego, incluyendo tanto las funcionalidades que se han decidido añadir como las simplificaciones que se han tomado para facilitar posteriormente la tarea de entrenar a los agentes.

Es importante destacar que existe un reglamento de diseño oficial [20] que establece todos los detalles (acciones, puntuación, velocidades...) que debe seguir el juego, pero este reglamento no ha sido seguido durante el desarrollo, optando por realizar una versión más simple.

Acciones del jugador

El jugador humano es capaz de realizar las siguientes acciones sobre la pieza en juego:

- **Desplazar** la pieza a izquierda o a derecha.
- **Rotar** la pieza.
- **Dejar caer más rápido** la pieza (en concreto, la pieza cae una celda hacia abajo).
- **Soltar** la pieza para que caiga instantáneamente.

Se ha optado por **no permitir guardar piezas** al jugador. Esto se debe a que se ha considerado que permitir al agente la posibilidad de cambiar la pieza complicaría notablemente la implementación y su aprendizaje. Por tanto, se ha optado por simplificar.

Las acciones concretas que cada agente es capaz de realizar serán discutidas en el próximo capítulo, durante la caracterización del conocimiento.

Dificultad y velocidad de juego

La dificultad del juego depende del número total de líneas que se han eliminado durante el transcurso de la partida, e influye tanto en la velocidad de caída automática de la pieza en juego como en la puntuación obtenida por eliminar líneas.

La dificultad comienza en **nivel 0**. En este nivel, la pieza descenderá una celda automáticamente cada 0.5 segundos. Tras esto, el nivel de dificultad aumentará cada **diez** líneas eliminadas, disminuyendo el tiempo en 0.025 segundos. La dificultad máxima alcanzada es el **nivel 9**, tras eliminar 90 líneas, llegando a una velocidad de 0.275 segundos. Formalizado, la velocidad sería $velocidad = 0{,}5 - (min(9, floor(lineas/10)) * 0{,}025$.

Hay que destacar que, cuando el jugador es un agente entrenado y durante el entrenamiento de los agentes (aunque también es posible activar esta opción para un jugador humano), **la velocidad del juego está fija**, cayendo la pieza una celda cada 0.25 segundos independientemente de las líneas totales eliminadas. La razón de esto es poder sincronizar la caída automática con las actuaciones del agente (que, como se verá después, actúa cuatro veces por segundo).

De esta forma, se garantiza que la pieza caerá a la vez que actúa el agente siempre, haciendo las transiciones entre estados **deterministas**, simplificando el aprendizaje del agente. De la otra forma, existe la posibilidad de que cuando actúe el agente (por ejemplo, desplazando la pieza hacia la derecha), la pieza descienda a la vez que se desplaza o que no descienda, provocando transiciones no deterministas.

Sistema de puntuación

El sistema de puntuación del juego es el siguiente, ofreciendo la siguiente puntuación cada vez que se bloquea una pieza dependiendo de la dificultad:

- Bloquear una pieza sin llenar una línea: $profundidad * (dificultad + 1)$ puntos.

  Se otorga una cantidad de puntos igual a la fila más profunda que ha alcanzado la pieza (siendo ésta, a nivel de dificultad 0, 20 puntos si la pieza toca el fondo, 19 si se queda una línea encima y así sucesivamente)

- Bloquear una pieza llenando líneas: $100 * 2^{(lineas-1)} * (dificultad + 1)$ puntos.

  Se otorga una mayor cantidad de puntos cuantas más líneas se hacen a la vez (siendo la máxima puntuación 800 veces la dificultad cuando se eliminan cuatro líneas a la vez).

Como se puede ver, la puntuación incentiva al jugador a intentar llenar el máximo numero de líneas posibles (recompensando especialmente eliminar varias líneas a la vez y bloquear las piezas en las posiciones más hondas posibles, llenando huecos).

Ahora bien, existe un sistema de puntuación simplificado para agentes (aunque también es posible activarlo para jugadores humanos), ya que la dificultad no es relevante para los agentes:

- Bloquear una pieza sin llenar una línea: 1 punto.

- Bloquear una pieza llenando líneas: $10 \cdot 2^{lineas-1}$ puntos.

De nuevo, se incentiva llenar líneas ante todo. Ahora bien, esta vez las puntuaciones son mucho más simples (haciendo las puntuaciones finales más legibles, al ser números más pequeños).

Generación aleatoria de piezas

En las versiones originales de Tetris, la generación de piezas era **uniformemente distribuida** (es decir, todas las piezas tenían la misma probabilidad de ser generadas como la próxima pieza).

Ahora bien, las versiones modernas de Tetris, y nuestra versión, utilizan lo que se conoce como el **generador aleatorio de bolsa** [21]. Esta generación consiste en crear una permutación de las siete posibles piezas (conocida esa permutación como una "bolsa"). Una vez creada, se generarán las piezas necesarias siguiendo el orden de la permutación hasta que se hayan usado todas las piezas en ella. Tras esto, se generará una nueva "bolsa" y se repetirá el proceso.

La razón para utilizar este generador es garantizar tanto al jugador como a los agentes que la generación de piezas sea **más justa**. Con una generación uniformemente distribuida, no hay ninguna garantía de que a corto plazo se generen todas las piezas de forma equitativa, sesgando el resultado. En cambio, el generador de bolsa garantiza que todas las piezas se utilizarán y evita que una pieza concreta no se genere durante un tiempo excesivo (el máximo número de piezas generadas entre dos piezas del mismo tipo es 12).

Ayudas visuales al jugador humano

Para facilitar el juego a los jugadores humanos, se incluyen dos tipos de ayudas visuales. Este tipo de ayudas son irrelevantes para los agentes, ya que no serán consideradas por éstos de ninguna forma. Las ayudas son las siguientes:

- **Pieza fantasma:** En todo momento, habrá en pantalla una representación de la posición en la que se bloquearía la pieza actualmente si se dejase caer sin acción del jugador (o haciendo el jugador que caiga la pieza instantáneamente).
- **Indicador de próxima pieza:** El jugador puede ver la próxima pieza que se generará cuando se bloquee la pieza actual en juego, permitiéndole tenerlo en cuenta a la hora de decidir la posición en la que bloqueará a la pieza actual.

4.2.2. Implementación

En esta sección se detallan los aspectos más relevantes de la implementación del juego. Solo se detallarán las partes del código relacionadas con el funcionamiento interno del juego y con el uso por parte de jugadores humanos, describiendo todos los detalles de la integración de los agentes en el juego en el capítulo siguiente.

El desarrollo se ha realizado utilizando **Python 3.7.6** como lenguaje de programación, usando además la librería **pygame** [22] para gestionar todos los aspectos relacionados con gráficos, sonidos, interacción con el jugador... Un listado completo de los requisitos necesarios para ejecutar el código se incluye en el anexo "Manual de Usuario".

La implementación original del juego se basó en una serie de tutoriales [23] y posteriormente fue adaptada y modificada para ampliar la funcionalidad, adaptarla a las necesidades del proyecto y solucionar los errores que se encontraron.

Todo el código del juego se encuentra en el fichero *tetris.py*, incluyendo este fichero además la documentación completa de su estructura y funcionamiento (en inglés). Debido a que una cantidad considerable del código son métodos auxiliares (encargados de la interfaz de usuario, del sonido y de cálculos internos), se describirán únicamente las partes más relevantes para el proyecto.

Representación interna de la pieza en juego

Las piezas en juego se representan mediante objetos de una clase *Piece*, que contiene todos los valores relevantes de la pieza: sus **coordenadas actuales** (en formato (x, y)), la **pieza** que representa el objeto (una de las siete posibles), el **color** de la pieza (utilizado para representarla en pantalla) y la **rotación actual**.

Esta información se almacena en un objeto principalmente para **encapsular** la pieza, facilitando trabajar con ella (pudiendo simplemente modificar sus valores internos y que los métodos auxiliares apropiados se encarguen de todos los cálculos de su posición actual en la zona de juego).

Representación interna de la zona de juego

La zona de juego se representa internamente mediante dos estructuras de datos:

- Un **diccionario** que almacena las celdas que están ocupadas por piezas actualmente. La clave del diccionario son las coordenadas de la pieza en una tupla $(columna, fila)$, mientras que el valor asociado a la clave es el color de la pieza que ocupa esa celda, en formato RGB.

- Una **matriz** de tamaño $20x10$ (20 filas, 10 columnas). En cada posición $(fila, columna)$ se almacena un color en RGB, pudiendo ser este el color del fondo de la zona de juego (en cuyo caso se considera una celda "vacía") u otro color (considerándose una celda "llena").

El diccionario se mantiene constante toda la partida, y durante su transcurso se irán añadiendo a éste las piezas que se bloqueen en la zona de juego (añadiendo entradas), y eliminando las que desaparezcan cuando se completan líneas (borrando las entradas). En cambio, la matriz es generada en cada iteración del bucle principal de juego a partir del diccionario y de la pieza actualmente en juego.

La razón de existencia de la matriz (en vez de utilizar únicamente el diccionario) es que nos facilita algunos cálculos (como la legalidad de los movimientos de la pieza o las líneas completadas) y la representación gráfica del juego en pantalla, ya que es más fácil realizar esos cálculos iterando sobre una matriz estructurada que sobre las claves desordenadas de un diccionario.

Bucle principal del juego

En el bucle principal es donde se controla toda la lógica de juego, incluyendo la creación de piezas, la caída automática de las piezas, el procesamiento de las entradas del jugador... Es importante destacar este bucle ya que será la base sobre la que posteriormente se implementarán los agentes, modificándolo en los puntos que sean necesarios.

Es interesante recalcar también que el juego utiliza un **reloj físico** para controlar la caída automática de las piezas. Este reloj se actualiza al principio de cada bucle, para tener un control del tiempo real independiente de la velocidad a la que se ejecute el juego. El reloj se para durante los efectos visuales que paran el juego, para evitar que sea injusto para el jugador.

El pseudocódigo del bucle del juego se encuentra en la figura 4.3. Se pueden observar capturas de la implementación del juego en el anexo "Manual de usuario" [A].

Algoritmo 4: Bucle principal del juego

1. Inicializar el reloj físico C, el diccionario de piezas bloqueadas D, las piezas *actual* y *siguiente* y los contadores *tiempo_caida*, *lineas* y *puntuacion*.

2. Mientras el jugador no haya perdido la partida:

- **2.1.** Genera la matriz *matriz* a partir de D.
- **2.2.** $caida \leftarrow caida + \Delta C$
- **2.3.** Procesa las entradas del usuario y mover la posición de la pieza *actual* apropiadamente (bloqueando si es necesario).
- **2.4.** Si *caida* es mayor que el tiempo de caída actual (variable global), desplaza *actual* hacia abajo.
  - **2.4.1.** Si *actual* se apoya sobre una pieza en D, bloquea *actual*.
- **2.5.** Añade *actual* a *matriz*.
- **2.6.** Si *actual* está bloqueado:
  - **2.6.1.** Añade *actual* a D.
  - **2.6.2.** $actual \leftarrow siguiente$, genera una nueva pieza para *siguiente*.
  - **2.6.3.** Comprueba si se ha completado alguna línea, eliminando las piezas en las líneas eliminadas de D.
  - **2.6.4.** Calcula $puntuacion\_obtenida$ a partir de las líneas eliminadas.
  - **2.6.5.** $lineas \leftarrow lineas + (lineas\_eliminadas)$; $puntuacion \leftarrow puntuacion + puntuacion\_obtenida$.
  - **2.6.6.** Actualiza la dificultad dependiendo de *lineas*.
- **2.7.** Comprueba si se ha perdido la partida a partir de las posiciones de D.

Figura 4.3: Pseudocódigo del bucle principal del juego.

Capítulo 5

APLICACIÓN DE DEEP Q-LEARNING A TETRIS

En este capítulo se describirá la implementación del algoritmo de Deep Q-Learning en el juego descrito en el capítulo anterior, centrándose en las dos aproximaciones distintas que se han seguido: una aproximación que pretende imitar las entradas del jugador (el desplazamiento de la pieza) y otra que busca imitar el razonamiento del jugador (la posición final en la que colocará la pieza).

Para cada aproximación se detallará la formalización del conocimiento (estado, acciones y recompensas). Tras esto, se describirá el agente estándar y su implementación, junto a las variantes que se han realizado de susodicho agente. Finalmente, se hablará del funcionamiento de los agentes dentro del juego desarrollado.

5.1. Primera aproximación: Imitación de los inputs de un jugador humano

En esta aproximación, se busca realizar una implementación lo más cercana posible al algoritmo puro de Deep Q-Learning. Por esto, se decidió que el agente imitase directamente las entradas que realizaría el jugador (las acciones que el jugador realizaría para desplazar la pieza).

5.1.1. Caracterización del conocimiento

Estado

El estado, conforme lo observará el agente, se representa mediante una **matriz** de tamaño $20x10$ (20 filas, 10 columnas). El estado representa directamente la zona de juego, habiendo una equivalencia directa entre las celdas de la zona de juego y las celdas del estado.

Las celdas del estado pueden tomar dos valores únicamente: 0 si la celda está vacía (no hay ningúna pieza bloqueada ni en juego en esa celda) o 1 si la celda está ocupada (hay alguna pieza bloqueada o en juego en esa celda). Se puede observar un ejemplo en la figura 5.1 (donde negro significa 0 y blanco significa 1).

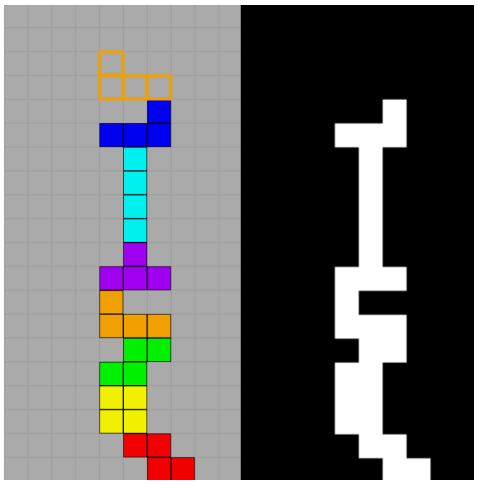


Figura 5.1: Equivalencia entre la zona de juego (izquierda) y el estado (derecha).

La razón para representar el estado de esta forma (en vez de, por ejemplo, utilizar una captura de la zona de juego directamente) es simplificar la estructura del agente. Este estado se puede considerar como una representación ya preprocesada y simplificada de la zona de juego, por lo que no será necesario que el agente aprenda a interpretar capturas de la zona de juego, garantizando que ésto no influirá en el rendimiento de los agentes.

Consideraremos además como estado final cualquier estado en el que la pieza en juego acaba bloqueada. Se ha decidido de esta forma ya que tras bloquear la pieza, se empieza a jugar con una pieza nueva (y, por tanto, supone un punto y aparte respecto a la pieza anterior).

Acciones

Las acciones que puede realizar el agente están basadas directamente en las entradas que puede realizar un jugador humano, siendo éstas:

- **Desplazar** la pieza a izquierda o a derecha.
- **Rotar** la pieza.
- **Soltar** la pieza para que caiga instantáneamente.

A diferencia de los jugadores humanos, el agente no tiene la capacidad de hacer bajar a la pieza más rápido. Se ha decidido no dar ésta capacidad al agente para limitar su complejidad, considerando que la habilidad adicional que tendría el agente

con esta acción extra no compensaría la dificultad añadida al entrenamiento (pudiendo llegar incluso a perjudicar el resultado final).

Recompensas

Se han diseñado dos sistemas de recompensas para el agente, cada uno enfocando las recompensas otorgadas a un aspecto distinto del juego:

El primer sistema (conocido como **recompensas de juego**) otorga recompensas al agente de forma parecida a cómo la puntuación recompensa a un jugador humano. Las recompensas otorgadas por este sistema son:

- Si la acción ha provocado que el agente pierda la partida: -2.

  Queremos evitar que el agente pierda la partida, por lo que una acción que le provoque perder la partida tiene una penalización considerable (mayor que cualquier otra posible).

- Si la acción no ha provocado que la pieza se bloquee: $-0{,}1$.

  Penalizando las acciones que no bloquean piezas, se incita al agente a realizar acciones útiles (bloquear piezas o llenar líneas). También se busca evitar que el agente realice acciones inútiles (como rotar continuamente la pieza u oscilar la pieza entre dos columnas)

- Si la acción ha provocado que la pieza se bloquee:

  - Si no se ha completado ninguna línea:

    $$(fila/10) - 1$$

    donde $fila$ indica la fila de la zona de juego en la que se ha colocado la ficha, ordenada ascendentemente de arriba a abajo (siendo la primera fila 0 y la última 19)

    Esto se traduce en un valor en un rango de $+0{,}9$ (si se coloca la pieza en la fila más baja) a -1 (si se coloca la pieza en la fila más alta). Esta recompensa incentiva al agente a intentar colocar las piezas en la posición más baja posible (llegando incluso a penalizar por bloquear piezas en posiciones elevadas). De esta forma, se espera que el agente empiece a llenar los huecos para buscar posiciones más bajas y, de ahí, empezar a llenar líneas.

  - Si se ha completado una o más líneas: 2^{lineas}.

    Esta recompensa oscila entre $+2$ (eliminando una sola línea) y $+16$ (eliminando cuatro líneas a la vez). Siempre es una recompensa mayor que la de bloquear la pieza sin llenar líneas (pudiendo llegar a ser mucho mayor), por

lo que el agente buscará llenar todas las líneas que pueda, y cuando sea capaz intentará eliminar varias líneas a la vez.

El segundo sistema (conocido como **recompensas heurísticas**) otorga recompensas al agente en base a una evaluación heurística de los estados antes y después de aplicar la acción [24]. Esta evaluación tiene en cuenta los siguientes cuatro parámetros del estado:

- **Altura total**: Suma de la altura de cada columna (la posición bloqueada más alta en cada columna). Esto mide cómo de "elevado" es el estado actual, y por lo tanto se busca minimizar este valor.
- **Líneas completas**: Número total de líneas completas en el estado (filas llenas en el estado). Al ser el objetivo del agente completar líneas, se busca maximizar este valor.
- **Huecos**: Número total de huecos en el estado (siendo un hueco una o más celdas vacías con una celda ocupada encima de ellas en la misma columna). Los huecos son más complicados de llenar, por lo que se busca minimizar este valor.
- **Grado de desigualdad**: Suma de diferencias en la altura entre columnas contiguas. Los estados "desiguales" (por ejemplo, un estado con un gran agujero en un lateral o un estado con muchos picos) no son deseables porque es complicado realizar líneas en ellos, por lo que se busca minimizar este valor.

La evaluación de un estado se obtiene con la siguiente fórmula [24]:

$$puntuacion = -0{,}51 \cdot altura + 0{,}76 \cdot lineas - 0{,}36 \cdot huecos - 0{,}18 \cdot desigualdad$$

La recompensa se calcula, a partir de esta evaluación, de la siguiente forma:

- Si la acción ha provocado que el agente pierda la partida: -2.

  De nuevo, se busca que el agente no realice acciones que le hagan perder la partida.

- Si la acción ha completado una o más líneas: 2^{lineas}.

  La recompensa oscila entre $+2$ (una sola línea) y $+16$ (cuatro líneas). Igual que antes, se busca otorgar una gran recompensa al agente por completar líneas, incentivándolo a llenar líneas siempre que le sea posible.

- En cualquier otro caso:

  1. Se obtiene, antes de realizar la acción, cuál sería el estado si se dejase caer la pieza hasta el fondo.

2. *evaluacion_antes* = evaluación de ese estado hipotético antes de realizar la acción.
3. Tras realizar la acción, se obtiene cuál sería el estado si se dejase caer la pieza hasta el fondo.
4. *evaluacion_despues* = evaluación de ese estado hipotético después de realizar la acción.
5. $recompensa = evaluacion\_despues - evaluacion\_antes$.

La idea que se busca es comparar el potencial del estado tras aplicar la acción con el del estado antes de aplicar la acción, usando la evaluación heurística para estimar ese potencial. Si el nuevo estado es mejor que el anterior, la recompensa será positiva. En cambio, si el nuevo estado es peor que el anterior, la recompensa será negativa. Así se pretende otorgar al agente unas recompensas intermedias más informadas, permitiéndole evaluar si las acciones que está tomando llevan a estados buenos o malos.

5.1.2. Agentes e implementación

Se han desarrollado tres tipos de agente para esta aproximación. Se describirá en detalle el primer tipo de agente (agente **estándar**) al ser el agente principal, comentando las estructuras de datos y métodos más relevantes que contiene. Para el resto de agentes, únicamente se explicarán las diferencias que presentan respecto al agente estándar.

Agente estándar

El agente estándar es el agente básico de Deep Q-Learning desarrollado para esta primera aproximación. Representa una implementación estándar del algoritmo, sin ninguna clase de mejora adicional.

La implementación de este agente. junto a su documentación, se puede observar en el fichero *agents/old/dql_agent_old.py*, en la clase **DQLAgentOld**.

Arquitectura de la red neuronal: La red neuronal del agente ha sido implementada utilizando **Keras** [25], una librería típica para el desarrollo de redes neuronales en Python.

La estructura de la red neuronal del agente (usando la nomenclatura de las capas de Keras) es la siguiente:

- **Capa de entrada:** La capa de entrada utiliza una capa de **Flatten**. Esta capa recibe la entrada de la red (en este caso, una matriz de tamaño $20x10$) y la

"aplana", transformando una representación multidimensional en una unidimensional (un *array*). De esta forma, las capas posteriores pueden trabajar con esa representación de la información. Esta capa no tiene función de activación ni de inicialización de pesos.

- **Capas ocultas:** La red tiene dos capas ocultas de tipo Dense (capas de neuronas totalmente conectadas). Cada una de estas capas contiene 64 neuronas, utilizando *ReLU* como su función de activación y *Glorot y Bengio* [26] como función de inicialización de pesos. Ambas funciones son estándar y típicas en redes neuronales profundas.

  Tras experimentar con varios valores, se ha considerado que dos capas con 64 neuronas cada una es una cantidad suficiente como para que la red neuronal tenga suficiente potencia, pero no demasiada como para que se sobreajuste en exceso.

- **Capa de salida:** La capa de salida es una capa de tipo Dense con cuatro neuronas. Cada una de estas neuronas se corresponderá con una de las posibles acciones que puede realizar el agente, siendo la salida de cada una el valor Q de realizar esa acción en el estado pasado como entrada. La función de activación utilizada es *lineal*, y la función de inicialización de pesos es *Glorot y Bengio* de nuevo.

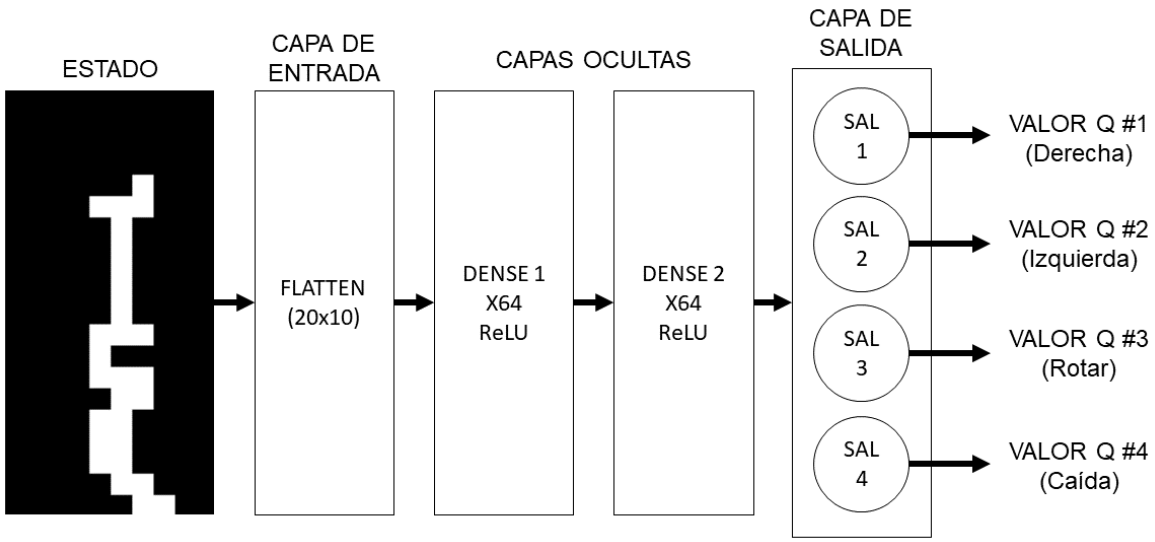


Figura 5.2: Arquitectura de la red neuronal del agente en la primera aproximación.

Todas las funciones de inicialización de pesos utilizan una semilla *seed* para garantizar la reproducibilidad, pasada como parámetro.

Se puede observar una representación de la arquitectura de la red neuronal en la figura 5.2.

Aunque es típico que las redes neuronales de agentes de Deep Q-Learning incluyan al comienzo una serie de capas de **convolución**, en este caso no es necesario por la representación del estado que se ha utilizado. Las capas de convolución sirven para extraer características de imágenes, y son típicas para preprocesar estados que son capturas del juego. Ahora bien, al ser nuestro estado una matriz ya preprocesada que contiene la información más relevante de la zona de juego, no es necesario realizar ninguna convolución con esa información.

De cara al entrenamiento de la red neuronal, la medida de error que se considera es el **error cuadrático medio** (la medida de error utilizada por definición en Deep Q-Learning), y el algoritmo de optimización es **Adam** [16] (una variante del gradiente descendiente estocástico que es estado del arte en redes neuronales profundas).

El agente contiene dos redes neuronales con esta arquitectura, como es típico del algoritmo de Deep Q-Learning: la **red Q** (actualizada continuamente y utilizada para actuar) y la **red objetivo** (actualizada únicamente al acabar los *epochs* y usada para tener una estimación constante del valor esperado del estado alcanzado durante el algoritmo de retropropagación).

***Replay Memory*:** La información almacenada en el *Replay Memory* del agente se almacena con la estructura $< s, a, r, s', f >$, donde:

- s: Estado inicial de la experiencia.
- a: Acción aplicada en el estado s.
- r: Recompensa obtenida por aplicar la acción a en el estado s.
- s': Estado alcanzado tras aplicar la acción a en el estado s.
- f: Indica si el estado s' es un estado final (la pieza se ha bloqueado en la zona de juego) o no.

La estructura interna del *Replay Memory* es una **cola** de tamaño máximo N (pasado al agente como parámetro), en la que se van insertando las experiencias. Una vez la cola está llena, las experiencias nuevas irán sustituyendo a las experiencias más viejas en orden FIFO. De esta forma, se garantiza que el agente irá renovando su memoria conforme actúe y obtenga nuevas experiencias.

Actuación del agente: Este método se utiliza cada vez que es necesario que el agente realice una acción. El algoritmo que se utiliza para actuar (decidir la acción a realizar) es el siguiente:

1. El agente recibe un estado s que procesa a través de la red Q, obteniendo los valores Q para cada acción.

2. Con probabilidad ϵ, selecciona aleatoriamente (distribución uniforme) una acción a de entre todas las posibles.

   Si no, selecciona la acción a con valor máximo de Q.

El parámetro ϵ representa la probabilidad de realizar una acción aleatoria como parte del proceso de exploración-explotación. Este valor de *epsilon* va decreciendo linealmente cada *epoch* hasta alcanzar un valor mínimo $epsilon_{min}$ una vez se ha cumplido el $E\,\%$ de los *epochs*. El objetivo de esta reducción es conseguir una gran **exploración** al principio cuando el agente tiene poco conocimiento del problema, pero permitiendo **explotar** el conocimiento adquirido al final del entrenamiento (cuando la política del agente ha mejorado y es más estable).

Los valores de ϵ, ϵ_{min} y E son pasados al agente como parámetros.

Memorización de experiencia: Tras cada acción realizada, el agente almacena en memoria la experiencia relacionada con esa acción en forma de tupla $< s, a, r, s', f >$ como se ha descrito previamente.

Es relevante indicar que, tras cada memorización, el agente hará un proceso de *Experience Replay*, entrenando su red neuronal a partir de una pequeña muestra del *Replay Memory*.

Aprendizaje a partir de la memoria: El proceso de entrenamiento de la red neuronal es muy similar al estudiado en Deep Q-Learning, aunque cuenta con algunas pequeñas diferencias que es interesante destacar:

- Se toma una muestra de tamaño M del *Replay Memory* para entrenar la red neuronal. Ahora bien, si el tamaño del *Replay Memory* es menor que M se toma directamente el *Replay Memory* como muestra.

  De esta forma, es posible entrenar la red neuronal aunque haya pocas experiencias en la memoria (por ejemplo, al principio del entrenamiento).

- En vez de actualizar la red neuronal de forma estocástica (es decir, se procesan las experiencias de la muestra una a una, actualizando los pesos de la red para cada una) se actualiza en *batch* con la muestra entera (se procesan todas las experiencias a la vez en paralelo).

  El orden concreto en el que se realiza esto es el siguiente:

  1. Se procesan los valores Q para los estados actuales s de la muestra en *batch*, pasando la lista de estados s a través de la red Q.
  2. Se procesan los valores esperados Q para los próximos estados s' de la muestra en *batch*, pasando la lista de próximos estados s' a través de la red objetivo.

3. Calcula cuáles serán los nuevos valores Q para la lista de estados actuales s uno a uno, utilizando las recompensas r asociadas a la experiencia y los valores esperados calculados previamente.
4. Entrena la red neuronal Q en *batch*, utilizando los nuevos valores Q actualizados para la retropropagación.

Utilizar la red neuronal (tanto para predicción como para entrenamiento) en *batch* es un cambio importante, ya que supone un incremento notable del rendimiento del agente (agilizando los cálculos), permitiendo al agente actuar en tiempo real durante la partida.

Ahora bien, esto supone también un problema, y es que la paralelización proporcionada por Keras no garantiza la reproducibilidad aunque se utilicen semillas a la hora de inicializar los pesos (por la paralelización que se realiza). Sin embargo, la mejora de rendimiento obtenida compensa este problema.

Los parámetros γ (valor que se otorga a la recompensa durante el aprendizaje) y M, usados durante el entrenamiento de la red, son pasados al agente como parámetros.

Finalización del *epoch*: Se considera como *epoch* para el agente una partida entera de Tetris. En este caso, supondremos que una partida se ha completado si el agente pierde o si completa 100 líneas (para poner un límite a la duración de las partidas).

Al finalizar el *epoch*, el agente realiza una serie de acciones para preparar la siguiente iteración:

1. Entrena la red Q como se ha descrito previamente.
2. Actualiza la red objetivo, copiando los pesos de la red Q en la red objetivo. Este es el momento en el que se actualizan los valores Q esperados para el siguiente *epoch*.
3. Actualiza el valor de ϵ, decrementandolo linealmente como se ha indicado previamente.
4. Almacena en un fichero *CSV* información relevante sobre el *epoch* (líneas completadas, puntuación obtenida y cantidad de acciones realizadas).

   Esta información será utilizada posteriormente para evaluar el rendimiento de los agentes.
5. Cada diez *epochs*, almacena los pesos de la red neuronal Q como *checkpoint*. De esta forma, se va almacenando el progreso del agente, pudiendo usar los pesos del *epoch* seleccionado para comprobar el rendimiento del agente en ese punto del entrenamiento y tener **trazabilidad**.

Tanto el fichero *CSV* como los pesos se almacenan en una carpeta *results*. El fichero *CSV* será utilizado posteriormente para la generación de gráficas, como será descrito en el anexo "Script auxiliar para generación de gráficas" [B].

Agente con acciones no uniformes

El agente con acciones no uniformes es una variación del agente estándar que tiene la peculiaridad de que, cuando elige una acción a realizar al azar, no utiliza una distribución uniforme. En vez de eso, utiliza la siguiente distribución de probabilidades:

- Moverse a derecha: 0,25.
- Moverse a izquierda: 0,25.
- Rotar: 0,4.
- Soltar la pieza: 0,1.

La razón de esta distribución es tratar de subsanar un problema que se observó viendo el rendimiento del agente estándar durante el entrenamiento: el agente estándar entrenado se vuelve propenso a soltar inmediatamente la pieza, pero reacio a rotarla. Este problema se debe a que todas las acciones tienen la misma probabilidad de ser elegidas, pero su impacto inmediato sobre el juego es distinto (soltar una pieza es mucho más impactante que rotar una pieza, ya que cuando se suelta una pieza se bloquea inmediatamente en la zona de juego, impidiendo ya moverla o rotarla).

Por tanto, los estados explorados finalmente por el agente estaban sesgados (experimentando poco con piezas en los laterales de la zona de juego o rotadas). Con esta distribución, se busca forzar artificialmente al agente a que rote más y suelte menos las piezas, con el objetivo de que explore una mayor cantidad de pares estado-acción.

Por lo demás, la implementación de este agente es idéntica a la del agente estándar, y se puede encontrar (junto a su documentación) en el fichero *agents/old/weighted_agent_old.py*.

Agente aleatorio

El agente aleatorio es una variación especial del agente estándar, que no incluye ninguna capacidad de aprendizaje. En vez de eso, siempre que sea necesario que actúe el agente, devolverá una acción aleatoria.

Este agente está pensado únicamente para servir como *baseline* para el resto de agentes (buscando comprobar si su rendimiento es mejor al de un jugador totalmente aleatorio).

La implementación de este agente se encuentra (junto a su documentación) en el fichero *agents/old/random_agent_old.py*.

5.1.3. Uso de los agentes en el juego

La integración de los agentes en el juego se realiza de forma separada al bucle principal de juego descrito en el capítulo anterior, a pesar de ser modificaciones de éste. Esta separación se debe a las necesidades específicas que tienen los agentes durante el juego, que no podrían ser satisfechas de forma clara y simple a través de estructuras condicionales en un único bucle de juego, por lo que se ha optado por separarlo en tres bucles por claridad.

Existen dos modos específicos implementados para los agentes: el modo **juego** (para agentes ya entrenados) y el modo **aprendizaje** (para entrenar agentes). Hay una serie de diferencias que comparten ambos modos respecto al bucle principal de juego para jugadores humanos:

- Es posible seleccionar como parámetro el tipo de agente que se utilizará de la lista descrita previamente, con la particularidad de que el agente aleatorio sólo se puede utilizar en modo juego (al no ser capaz de ser entrenado).
- El procesamiento de las entradas del usuario es sustituido por la generación del estado actual (a partir del diccionario de piezas bloqueadas y de la pieza actualmente en juego). Este estado será pasado al agente, que devolverá la acción apropiada a realizar.
- El sistema de puntuación utilizado es el sistema simple, y la dificultad del juego (velocidad de caída automática de las piezas) está fija, como se describió en las decisiones de diseño del juego.
- Las partidas se acaban automáticamente si el agente completa 100 líneas. Añadiendo este punto de parada, se evita la posibilidad de agentes que nunca paren de jugar, provocando bucles infinitos.

Se procede ahora a comentar los detalles concretos de cada modo.

Modo juego

El **modo juego** consiste en utilizar a un agente ya entrenado como sustituto directo del jugador humano, siendo el agente el encargado de introducir las acciones en el juego.

En esencia, el modo juego es muy similar al bucle de juego de un jugador humano, incluyendo las diferencias descritas previamente junto a algunas características propias de este modo:

- Es posible cargar pesos ya entrenados para el agente (obtenidos en el modo entrenamiento). De esta forma, el agente utilizará una política ya entrenada, pudiendo jugar con el conocimiento adquirido. Si no se carga ningún peso, utilizará los pesos generados aleatoriamente (comportándose en esencia como el agente aleatorio).

- Todos los valores de ϵ (la posibilidad de acciones aleatorias) son fijados a 0. En este modo, se supone que se va a utilizar a un agente ya entrenado con una política bien definida, por lo que no tiene sentido que el agente realice acciones de forma aleatoria.

- Al acabar la partida, se imprime información relevante sobre el rendimiento del agente en la consola: líneas completadas, puntuación alcanzada y acciones totales realizadas.

  De esta forma, se podrá evaluar posteriormente el rendimiento de los agentes ya entrenados para compararlos entre ellos (frente a medir su rendimiento durante el entrenamiento).

- Durante el transcurso de la partida, se muestra en pantalla información adicional en un lateral que permite analizar el comportamiento del agente durante la partida.

  Concretamente se muestra una representación gráfica de cómo percibe el estado el agente, los valores Q obtenidos para susodicho estado (recalcando el mejor valor Q), la acción realizada y el número total de acciones que lleva el agente.

Se puede observar una captura del modo juego en la figura 5.3.

Modo aprendizaje

El **modo aprendizaje** permite entrenar a un agente durante un número determinado de *epochs*, establecido como parámetro.

Este entrenamiento consiste en ejecutar una variante del bucle de juego tantas veces como *epochs* haya, entrenando al agente durante la ejecución de estos bucles. El aprendizaje del agente se realizará utilizando los métodos descritos previamente.

Este modo tiene algunas peculiaridades comparado con el resto de modos, que son importante destacar:

- Se puede elegir como parámetro el tipo de recompensas que se utilizarán (estando disponibles los dos sistemas de recompensas descritos previamente).

- Las experiencias se almacenan en el *Replay Memory* del agente al final del bucle de juego (tras comprobar si la partida se ha acabado), almacenándose experiencias únicamente en las iteraciones del bucle en las que se ha realizado alguna acción.

  Al utilizarse un reloj físico, no se realizan acciones en todas las iteraciones del bucle (solo en las iteraciones en las que los contadores de tiempo hayan superado el valor). Por tanto, hay que tener esa consideración en cuenta para evitar insertar experiencias vacías.

- Durante el transcurso de la partida, se muestra en la pantalla información adicional que permite comprobar el comportamiento del agente durante la partida. Concretamente se muestra:
  - El *epoch* actual (para ver el progreso en el entrenamiento).
  - Una representación gráfica del ultimo estado s y próximo estado s' insertado en el *Replay Memory*.
  - La última acción a realizada y su recompensa asociada r.
  - La cantidad total de acciones que se han realizado en este *epoch*.
  - El valor actual de ϵ, para ver la probabilidad actual de realizar una acción aleatoria.
  - Información sobre el mejor *epoch* hasta el momento: qué *epoch* era, la puntuación alcanzada, las líneas completadas y las acciones realizadas.

Este modo ofrece además lo que se conoce como el *entrenamiento rápido*: una variación de este modo pensada para agilizar lo máximo posible el entrenamiento del agente, simulando las partidas.

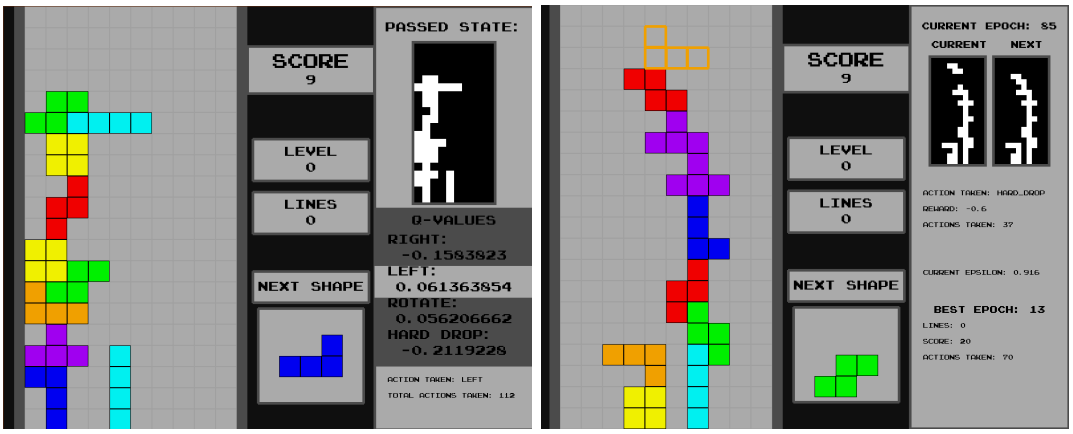
- Se elimina totalmente la representación gráfica (imprimiendo únicamente información en consola), evitando el coste computacional requerido para mostrar el juego en pantalla.
- Se sustituye el reloj físico por un reloj lógico. Este reloj aumenta en un tic cada iteración del bucle, y hace que se simule una acción del agente en cada ejecución del bucle (en vez de tener la limitación física de cuatro acciones por segundo).

Por lo demás, el funcionamiento es exactamente igual. De esta forma, se garantiza que las simulaciones son fiables (y por tanto, tienen la misma validez que entrenar al agente sin usar entrenamiento rápido). Esta forma de entrenamiento es notablemente más rápida al no estar limitada por las restricciones del reloj físico (pudiendo llegar a simular varios *epochs* por segundo).

Se puede observar una captura del modo entrenamiento en la figura 5.3.

5.2. Segunda aproximación: Imitación del proceso de razonamiento de un jugador humano

Como se observará en el siguiente capítulo, la primera aproximación no presenta buenos resultados, generando agentes que no son capaces de jugar de forma útil a Tetris a pesar de realizar correctamente el proceso de aprendizaje.



(a) Modo juego (b) Modo entrenamiento

Figura 5.3: Ejemplos del uso de los agentes en la primera aproximación.

En respuesta a esto se realiza una nueva aproximación basada en un artículo ya existente que ofrece mejores resultados [27]. La diferencia respecto a la aproximación original es el enfoque que se hace de las acciones: en este caso, el agente entenderá como acción la **posición y rotación** final en la que pretende bloquear la pieza. Esto se asemeja más al razonamiento real de un jugador humano (que piensa la mejor posición para bloquear la pieza, desplazando la pieza únicamente para poder alcanzar esa posición).

5.2.1. Caracterización del conocimiento

Estado

La definición de estado no cambia respecto a la aproximación original: se representa el estado como una **matriz** de tamaño $20x10$, siendo este un reflejo directo de la zona de juego.

Cada celda de la matriz tomará un valor entre 0 (si no hay una pieza bloqueada en esa celda en la zona de juego) y 1 (si hay una pieza bloqueada). El ejemplo de la figura 5.1 sigue siendo relevante.

En este caso, se considerarán como estados finales aquellos estados que provocan que la partida se acabe (ya sea por derrota del agente o porque el agente ha sido capaz de completar 100 líneas). Esta definición es la más lógica, ya que el final de la partida supone también el final del *epoch* durante el entrenamiento.

Acciones

La principal diferencia respecto a la aproximación anterior se encuentra en la caracterización de las acciones. En vez de tener una lista definida de cuatro posibles acciones, aquí las acciones posibles para el estado son todas las posibles **posiciones y**

rotaciones finales (es decir, en las que la pieza acaba bloqueada) de la pieza actual en juego.

Estas acciones se representan como una tupla $a = (x, rot)$ donde x es la columna en la que se bloqueará la pieza y rot el valor de rotación de la pieza (donde $0 = 0°$, $1 = 90°$ y así sucesivamente en incrementos de $90°$).

El conjunto total de acciones A es variable dependiendo de la pieza actualmente en juego (ya que no todas las piezas tienen la misma cantidad de posibles rotaciones únicas).

Estas acciones, por sí solas, no tienen sentido en el contexto del juego, ya que la pieza tiene que desplazarse hasta la posición indicada por la acción. Por esto, a partir de las acciones se generará una serie de **pasos**, que indican uno a uno los desplazamientos y rotaciones que deberá realizar el agente sobre la pieza para llevarla a la posición indicada por la acción.

Recompensas

De nuevo, se tienen dos sistemas de recompensas posibles, uno enfocado en la puntuación obtenida por realizar los movimientos (**recompensas de juego**) y otro enfocado en una evaluación heurística de los estados alcanzados (**recompensas heurísticas**).

El sistema de **recompensas de juego** es idéntico al utilizado en la primera aproximación, pero adaptándolo a la nueva definición de acción. Las recompensas otorgadas son las siguientes:

- Si la acción ha provocado que el agente pierda la partida: -2.

  Perder una partida va directamente en contra del objetivo del agente (sobrevivir el máximo tiempo posible para maximizar su puntuación), por lo que las acciones que provoquen su derrota tienen una penalización considerable.

- Si la acción no ha completado ninguna línea:

  $$(fila/10) - 1$$

  donde $fila$ es la fila de la zona de juego en la que se bloquea la pieza (siendo 0 la fila más alta y 19 la más baja)

  La recompensa obtenida, por tanto, oscila entre $+0{,}9$ (si se bloquea la pieza en la última fila) y -1 (si se bloquea en la parte más alta de la zona de juego). De esta forma, se incentiva al agente a bloquear las piezas lo más hondo posible (al tener mejores recompensas), con la idea de que esto lo lleve a completar líneas.

- Si la acción ha completado una o más líneas: 2^{lineas}.

Esta recompensa oscila entre +2 (una única línea) y +16 (cuatro líneas a la vez). Es una recompensa mayor que cualquier otra recompensa que se pueda obtener, por lo que el agente buscará maximizar el número de líneas (prefiriendo llenar varias a la vez, al obtener mejores recompensas)

El sistema de **recompensas heurísticas** se basa en el sistema homónimo de la aproximación original. Como recordatorio, la evaluación heurística de un estado s es la siguiente [24]:

$$puntuacion = -0{,}51 \cdot altura + 0{,}76 \cdot lineas - 0{,}36 \cdot huecos - 0{,}18 \cdot desigualdad$$

Las recompensas son las siguientes:

- Si la acción ha provocado que el agente pierda la partida: -2.

  De nuevo, se busca evitar que el agente pierda la partida, por lo que las acciones que provoquen esto tendrán una penalización grande.

- Si la acción ha completado una o más líneas: 2^{lineas}.

  La recompensa oscila entre +2 (una sola línea) y +16 (cuatro líneas). La idea es incentivar al agente a realizar acciones que llenen líneas (ya que es la mejor forma de mejorar el rendimiento del agente). Además, la bonificación crece exponencialmente cuantas más líneas se eliminan a la vez, para motivar al agente a intentar llenar varias líneas con una sola acción.

- En cualquier otro caso:

  1. Se calcula la evaluación del estado actual, $evaluacion\_antes$.
  2. Tras realizar la acción, se calcula la evaluación del nuevo estado, $evaluacion\_después$.
  3. $recompensa = evaluacion\_después - evaluacion\_antes$.

  La idea con este sistema de recompensas es permitir evaluar de forma más honesta el resultado de las acciones. Al evaluar el estado antes y después de la acción, es posible comprobar el impacto que ha tenido la acción y recompensarlo adecuadamente: si la acción ha llevado a un mejor estado se da una recompensa positiva, pero si la acción ha empeorado el estado se da una penalización negativa.

5.2.2. Agentes e implementación

Se han desarrollado tres tipos de agentes para esta aproximación. Se describirá en detalle el primer tipo de agente (agente **estándar**) al ser el agente principal del que

derivan los demás. De este agente se comentarán las estructuras de datos y métodos más relevantes que contiene.

Para el resto de agentes se comentarán únicamente las diferencias respecto al agente estándar.

Agente estándar

El agente estándar es el agente más básico desarrollado para esta aproximación. En este caso representa una implementación de las ideas de la aproximación, sin ninguna clase de mejora adicional (más allá del cambio de enfoque).

A diferencia de la anterior aproximación, este agente no sigue tan estrictamente el algoritmo de Deep Q-Learning, cambiando algunos detalles como la arquitectura de la red o la estructura del *Replay Memory*.

La implementación de este agente. junto a su documentación, se puede observar en el fichero *agents/new/dql_ agent_ new.py*, en la clase **DQLAgentNew**.

Arquitectura de la red neuronal: La red neuronal del agente ha sido implementada de nuevo usando **Keras** [25]. La estructura de esta red (usando la nomenclatura de las capas de Keras) es la siguiente:

- **Capa de entrada:** La capa de entrada utiliza una capa de **Flatten**, recibiendo la entrada de la red (en este caso, una matriz de tamaño $20x10$) y "aplanando-la", transformando una representación multidimensional en una unidimensional (un *array*). De esta forma, las capas posteriores pueden trabajar con esa representación de la información. Esta capa no tiene función de activación ni de inicialización de pesos.

- **Capas ocultas:** La red tiene dos capas ocultas de tipo Dense (capas de neuronas totalmente conectadas). Cada una de estas capas contiene 64 neuronas, utilizando *ReLU* como su función de activación y *Glorot y Bengio* como función de inicialización de pesos. Ambas funciones son estándar y típicas en redes neuronales profundas.

  Igual que con la aproximación original, se ha considerado que dos capas de 64 neuronas cada una es una cantidad adecuada como para tener suficiente capacidad de aprendizaje sin llegar a sobreajustar.

- **Capa de salida:** La capa de salida es una capa de tipo Dense con una única neurona. La salida de esta neurona es el valor Q del estado (la valoración que realiza la red de la calidad del estado). La función de activación utilizada es *lineal*, y la función de inicialización de pesos es *Glorot y Bengio* de nuevo.

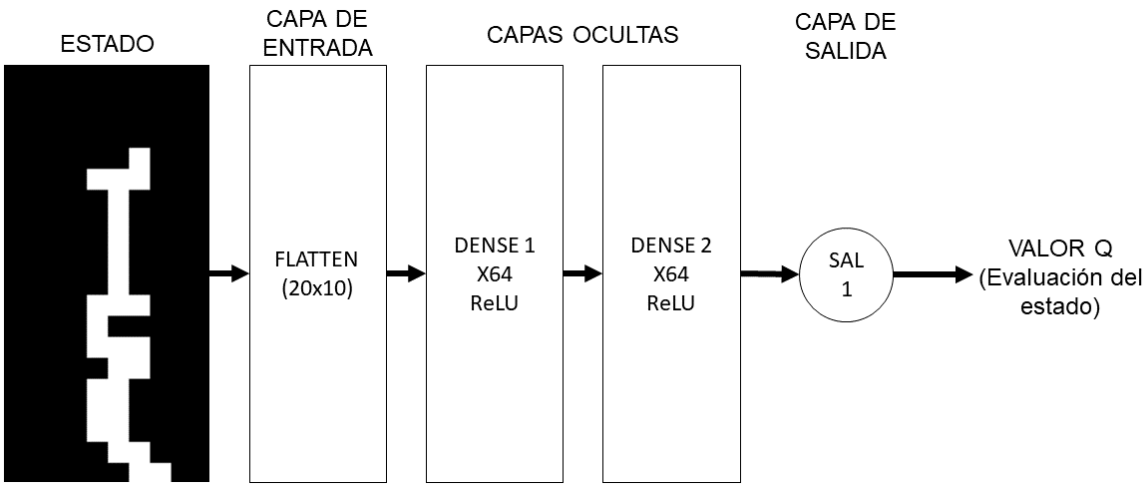


Figura 5.4: Arquitectura de la red neuronal del agente en la segunda aproximación.

Todas las funciones de inicialización de pesos utilizan una semilla *seed* para garantizar la reproducibilidad, pasada como parámetro.

Se puede observar una representación de la arquitectura de la red neuronal en la figura 5.4.

La diferencia respecto a la arquitectura de la primera aproximación es la capa de salida, que ha pasado de cuatro neuronas a una única neurona. Al ser el número de acciones variable (depende de la pieza en juego), no es posible tener una neurona de salida para cada par estado-acción. Por tanto, lo que se hace es utilizar una única neurona de salida que devolverá la valoración que realiza el agente para ese estado. La forma en la que se usará la red neuronal se describirá posteriormente.

Igual que en la primera aproximación, no es necesario utilizar capas de convolución al comienzo de la red (ya que el estado ya está preprocesado). De cara al entrenamiento de la red neuronal la medida de error utilizada es el **error cuadrático medio** y el algoritmo de optimización es **Adam**.

El agente contiene dos redes neuronales con esta arquitectura, igual que en la aproximación anterior: la **red Q** (actualizada continuamente y utilizada para actuar) y la **red objetivo** (actualizada únicamente al acabar los *epochs* y usada para tener una estimación constante del valor esperado del estado alcanzado durante el algoritmo de retropropagación).

***Replay Memory*:** Las experiencias almacenadas en el *Replay Memory* del agente se insertan con la estructura $< s, r, s', f >$, donde:

- s: Estado inicial de la experiencia.
- r: Recompensa obtenida por haber aplicado la acción elegida en el estado s.
- s': Estado alcanzado tras aplicar la acción elegida en el estado s.

- f: Indica si el estado s' es un estado final (la acción ha provocado que se acabe la partida) o no.

  Al cambiar el enfoque de las acciones, no tiene sentido considerar que un estado es final cuando la pieza se ha bloqueado (pues todos los estados s' tienen la pieza en juego bloqueada, por la definición de las acciones). Por tanto, se ha pasado a considerar que un estado es final cuando provoca que se acabe la partida (ya sea por derrota o por completar 100 líneas).

A diferencia de la primera aproximación, el agente no almacena la acción que se ha realizado en el *Replay Memory*. Esto se debe a que no es necesario almacenarla: la acción queda recogida implícitamente en s' (ya que este estado es alcanzado desde s realizando la acción), y posteriormente no será necesario conocer la acción que se ha realizado para elegir el valor Q apropiado de la red neuronal (ya que hay una única salida).

La estructura interna del *Replay Memory* es una **cola** FIFO de tamaño máximo N (pasado al agente como parámetro) en la que se irán insertando las experiencias.

Generación de las posibles acciones: Como se ha comentado ya varias veces, las acciones posibles para un estado s son variables, dependiendo del propio estado s y de la pieza actual en juego (ya que no todas las piezas tienen las mismas posibles rotaciones). Por tanto, es necesario generar una lista de acciones A para el estado s, que será la que utilice el agente posteriormente para decidir su actuación.

Para generar estas acciones, se sigue el algoritmo descrito en la figura 5.5.

Aunque sería idóneo que la generación de posibles acciones la realizase el agente, por comodidad en nuestra implementación el código encargado de esta tarea se encuentra junto a la implementación del juego (para poder acceder a los métodos auxiliares).

Actuación del agente: La forma de actuar del agente en esta aproximación difiere notablemente respeto a la primera aproximación, siendo ésta:

1. El agente recibe una lista A con las acciones posibles. Cada elemento de esta lista tiene la estructura (x, rot, s') donde x es la columna en la que se colocará la pieza, rot es la rotación de la pieza y s' el estado alcanzado finalmente.

2. Se procesan todos los estados s' por la red Q, obteniendo el valor Q (la valoración del agente) de cada estado.

3. Con probabilidad ϵ (probabilidad de realizar una acción aleatoria) se elige aleatoriamente una de las acciones de A.

   Si no, se elige la acción de A con máximo valor de Q.

Algoritmo 5: Generación de las posibles acciones para un estado s

Entrada: Estado actual s, pieza actual en juego *pieza*.

1. Preparar una lista A para almacenar las posibles acciones.

2. Obtener el posible número de rotaciones de la pieza *pieza*, *rotacion*.

3. Para cada posible rotación de la pieza, *rot*, entre 0 y *rotacion*:

- **3.1.** Para cada posible columna en la que puede estar la pieza, x, de 0 a 9:
  - **3.1.1.** Simula el movimiento de *pieza* hasta la columna x, con valor de rotación *rot*.
  - **3.1.2.** Si *pieza* está ya en una posición ilegal o ha encontrado una posición ilegal durante el camino, se continua con la siguiente iteración del bucle (descartando esta combinación de x y *rot*).
  - **3.1.3.** Mientras *pieza* esté en una posición válida (no esté solapando con ninguna pieza ya bloqueada):
    - **3.1.3.1.** Desplaza *pieza* una fila hacia abajo.
  - **3.1.4.** Una vez *pieza* alcanza la primera posición no válida, sube *pieza* una fila hacia arriba (para volver a la última posición legal).
  - **3.1.5.** Genera el estado s' a partir del estado s y *pieza*.
  - **3.1.6.** Inserta el nuevo estado en la lista A, con estructura (x, rot, s').

4. Devuelve A.

Figura 5.5: Pseudocódigo de la generación de las posibles acciones para un estado s.

Como se puede observar, lo que hace el agente es comprobar todos los posibles estados resultantes s' (el resultado de aplicar la acción a en el estado original s) y evaluarlos todos. Entonces, elegirá el mejor de esos estados (o uno al azar si se realiza exploración) como la acción a ejecutar. Esta valoración de los estados se realizará en *batch* (paralelamente) para agilizar el proceso.

Realmente la acción elegida, representada en s', no es directamente útil para el agente, ya que solo le indica qué estado pretende alcanzar pero no cómo llegar hasta él. Por esto, el agente además generará la **secuencia de pasos** que tiene que realizar en el juego para pasar del estado s al estado s'. Los pasos serán generados en el siguiente orden:

1. Rota la pieza hasta alcanzar la rotación deseada.
2. Desplaza la pieza a izquierda o derecha hasta alcanzar la columna esperada.
3. Suelta instantáneamente la pieza, dejándola caer.

El agente finalmente devolverá la acción elegida (con formato $(columna, rotacion, s')$ y la secuencia de pasos reales que deberá ejecutar para alcanzar s'.

Los valores de ϵ, junto a ϵ_{min} y E (como se vio en la aproximación original) funcionan de forma idéntica y son pasados como parámetros.

Finalización del *epoch*: A la hora de finalizar el *epoch* solo hay una pequeña diferencia, referente a la información que se almacenará en el fichero CSV.

No tendría sentido almacenar la cantidad total de acciones realizadas, ya que no serían comparables con las de la primera aproximación (y, por tanto, no se podrían analizar en conjunto). En cambio, el número total de **pasos** (desplazamientos de la pieza realizados por el agente) sí que podría ser almacenado, al ser equivalente a la definición previa de las acciones. Por tanto, se sustituirán acciones por pasos a la hora de almacenar la información del *epoch*.

Por lo demás, todo el proceso de finalización del *epoch* es idéntico.

Otros aspectos: El resto de aspectos comentados en la primera aproximación (**memorización de las experiencias** y **aprendizaje a partir de la memoria**) no han tenido ningún cambio respecto a ésta, siendo su implementación idéntica. Por tanto, no es relevante volver a comentar estos aspectos.

Agente con *Prioritized Experience Replay*

El agente con *Prioritized Experience Replay* es una variación del agente estándar que modifica la forma en la que utiliza *Experience Replay* para entrenar a su red neuronal, pasando a usar la técnica del *Prioritized Experience Replay* (dando mayor importancia a las experiencias que tienen un mayor error entre el valor esperado y el obtenido actualmente). Concretamente, se utilizará *Prioritized Experience Replay* con el método **basado en el rango**, donde la prioridad de un elemento i es $p_i = \frac{1}{rango(i)}$.

Los aspectos en los que ha sufrido cambios respecto al agente estándar son los siguientes:

***Replay Memory*:** Se cambiará la estructura del *Replay Memory*. Ahora, las experiencias del agente se almacenan en tuplas de la forma $(error, insercion, < s, r, s', f >)$, siendo $< s, r, s', f >$ la experiencia, *insercion* el número de la experiencia que se ha insertado (el agente cuenta el número total de experiencias que se han insertado) y *error* el error cuadrático como se vio en la definición.

Esta nueva estructura para las tuplas se debe al uso que se hará posteriormente de ellas: el *error* es necesario para priorizar las experiencias y poder ordenarlas (quedando antes las experiencias con un mayor error), e *insercion* se usa para llevar control del orden en el que se insertan las experiencias y para garantizar que no se puedan repetir (ya que el valor de *insercion* irá creciendo).

La cola sigue teniendo un tamaño máximo fijo, N, pasado por parámetro.

Memorización de las experiencias: La memorización de experiencias es bastante parecida, con la diferencia de que la experiencia insertada ahora tiene la forma $(error, insercion, < s, r, s', f >)$ (como ya se ha visto).

Como se comentó durante la explicación de esta mejora, la primera vez que una experiencia es introducida en la cola, su valor de *error* será fijado como $error = \infty$, para hacer más probable que todas las experiencias sean muestreadas al menos una vez (las nuevas experiencias tendrán la máxima prioridad).

Tras insertar cada experiencia, se creará una lista ordenada de los contenidos del *Replay Memory* (ordenada por *error* e *insercion*, de mayor a menor). Esta lista se utilizará durante el aprendizaje a partir de la memoria, y se almacena en una estructura de datos separada para no interferir con el orden de los elementos en la cola. Finalmente, se entrenará al agente usando *Experience Replay*.

Aprendizaje a partir de la memoria: A la hora de aprender a partir de la memoria, habrá algunas diferencias en el proceso. Concretamente cambiará la forma en la que se muestrea y maneja el *Replay Memory*. El proceso de aprendizaje ahora es el siguiente:

1. Se calcula la probabilidad de elegir cada elemento del *Replay Memory* con la fórmula apropiada, utilizando como **rango** de cada experiencia su posición en la lista ordenada generada previamente.

2. Se muestrea el *Replay Memory* utilizando las probabilidades calculadas anteriormente. Este muestreo se realizará sin reemplazo, para evitar aprender varias veces la misma experiencia en un mismo muestreo y para abarcar un mayor número de experiencias.

3. Se realiza el algoritmo de Deep Q-Learning estándar con la muestra elegida, con la diferencia de que se cambia la actualización del error ϕ_i por $w_i\phi_i$. Además, se conserva el nuevo error cuadrático de cada experiencia, para la actualización posterior del *Replay Memory*.

4. Se buscan los elementos muestreados en el *Replay Memory* y se actualizan sus errores cuadráticos, de cara a futuros muestreos.

Los valores necesarios para el uso de *Prioritized Experience Replay* (α y β) son fijos en vez de ser pasados por parámetros. Concretamente tomarán los valores obtenidos experimentalmente por el artículo original en el que se presenta esta propuesta [18], siendo estos:

- $\alpha = 0{,}5$.
- $\beta = 0{,}5$.

Agente aleatorio

El agente aleatorio es una variación del agente estándar que no incluye ninguna capacidad de aprendizaje. En vez de eso, siempre que sea necesario que actúe el agente devolverá una acción aleatoria.

Igual que su equivalente en la primera aproximación, este agente está pensado únicamente para servir como *baseline* para el resto de agentes (buscando comprobar si su rendimiento una vez entrenados es mejor al de un jugador totalmente aleatorio). La diferencia en este caso es el tipo de acciones procesadas por este agente (usando posiciones finales de las piezas como acciones).

La implementación de este agente se encuentra (junto a su documentación) en el fichero *agents/new/random_agent_new.py*.

5.2.3. Uso de los agentes en el juego

La integración de los agentes en el juego se realiza de nuevo de forma separada al bucle principal de juego que se describió en el capítulo anterior. Igual que en la primera aproximación, se distinguen dos modos distintos para los agentes: el modo **juego** (para agentes ya entrenados) y el modo **aprendizaje** (para entrenar a los agentes).

La implementación de este enlace entre agente y juego es similar a la realizada en la aproximación anterior, pero hay una diferencia clave que es necesario desarrollar.

Antes del bucle de juego como tal (en el que se controla la lógica del juego y el agente desplaza a las piezas como es necesario) se añade una parte de **pre-bucle**. Durante el pre-bucle, se realizan las siguientes acciones:

1. A partir de las piezas bloqueadas actualmente (sin tener en cuenta la pieza en juego) se obtiene el estado s. Este será el estado antes de aplicar ninguna acción.

2. Se calculan todas las posibles acciones A que se pueden realizar a partir del estado s y la pieza actualmente en juego *pieza*. Esta pieza realmente aún no está visible para el jugador (acaba de ser creada), pero el agente puede tener acceso a ella igualmente.

3. El agente procesa el conjunto de acciones A, devolviendo una acción a a realizar (conteniendo al estado s' que se alcanza si se aplica la acción a en s).

4. A partir de la acción a (que, recordamos, tiene el aspecto $(columna, rotacion, s')$) se genera una serie de **pasos** (una cola de desplazamientos que se deben realizar a la pieza actual para alcanzar el estado objetivo s' a partir del estado actual s.

El bucle de juego como tal es prácticamente idéntico, con la diferencia de que ahora se sustituyen las llamadas al agente (usadas previamente para decidir la acción

a realizar a partir del estado actual s) por la cola de pasos. Actuando cuatro veces por segundo (de forma consistente con la primera aproximación), se extraerá el primer paso de la cola y se realizará ese movimiento a la pieza actual. Estas iteraciones del bucle de juego se harán hasta que la cola de pasos esté vacía (se ha alcanzado el estado s' con éxito) o la pieza se haya bloqueado antes de tiempo (por alguna razón no se ha podido alcanzar el estado s').

Tras esta condición, se volverá al pre-bucle para calcular la próxima acción, y todo el proceso se repetirá hasta que acabe la partida (ya sea por derrota del agente o porque ha completado 100 líneas).

Hay también una serie de diferencias específicas a comentar para cada modo, que se proceden a desarrollar.

Modo juego

La única diferencia en el modo juego que no se ha mencionado ya es que cambia la información adicional que se muestra en pantalla, siendo la nueva información:

- El agente específico que está actualmente jugando.
- Una representación gráfica del estado s' a alcanzar (resultado de aplicar la acción a al estado actual s).
- El valor Q de la acción que se ha realizado.
- El último paso que se ha dado.
- El número total de acciones y pasos que ha realizado el agente.

Se puede observar una captura del modo juego en la figura 5.6.

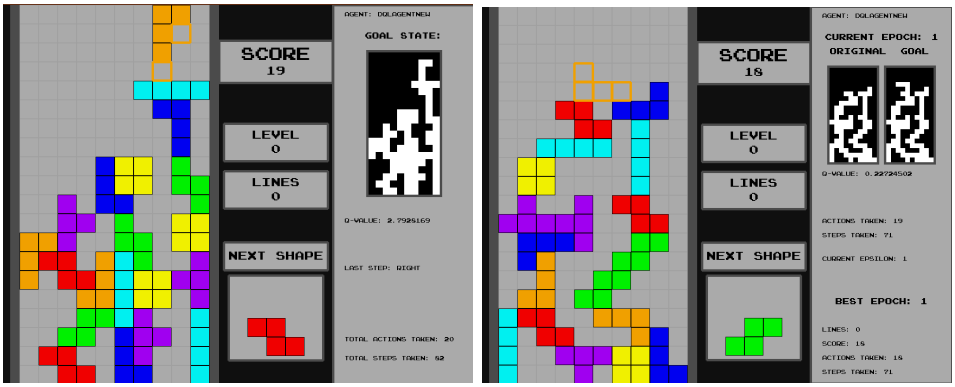
Modo aprendizaje

En el modo aprendizaje hay unas particularidades respecto al modo juego y a la implementación en la aproximación anterior que es necesario remarcar:

- Además del pre-bucle y el bucle, se añade un **post-bucle**, ejecutado tras finalizar el bucle (ya sea porque se han seguido correctamente todos los pasos o se ha bloqueado prematuramente la pieza). En el post-bucle se almacena la experiencia en el *Replay Memory* (calculando en el momento la recompensa asignada a la acción realizada).
- Se modifica la información adicional mostrada en pantalla para adaptarla a la nueva aproximación, siendo la nueva información:

- El agente específico que se está entrenando.
- El *epoch* actual.
- El estado actual s y el estado objetivo s' (alcanzado tras aplicar la acción a en el estado s).
- El último valor Q obtenido.
- La cantidad total de acciones y pasos que se han realizado en el *epoch* actual.
- El valor actual de ϵ, para medir el grado de exploración-explotación del *epoch* actual.
- Información sobre el mejor *epoch* hasta el momento: qué *epoch* era, la puntuación alcanzada, las líneas completadas y el número total de acciones y pasos realizados.

Se puede observar una captura del modo aprendizaje en la figura 5.6.



(a) Modo juego (b) Modo entrenamiento

Figura 5.6: Ejemplos del uso de los agentes en la segunda aproximación.

Capítulo 6

EXPERIMENTACIÓN Y RESULTADOS

En este capítulo se describirán los experimentos realizados con los agentes descritos previamente. Concretamente se detallarán tanto los experimentos que que se han realizado como las condiciones en los que han hecho. Posteriormente se presentarán los resultados obtenidos, ofreciendo un breve comentario sobre éstos.

6.1. Experimentos y condiciones

Los experimentos consistirán en el entrenamiento de cada par agente-heurística durante un número determinado de *epochs* para posteriormente comparar sus rendimientos (tanto durante el entrenamiento como durante el juego), estudiando la idoneidad de estos. Este rendimiento se controlará midiendo tres variables que deben **maximizarse**:

- Número de líneas completadas.
- Puntuación total obtenida.
- Número de acciones realizadas.

Se ha decidido controlar estas variables en vez de otras más típicas de algoritmos de aprendizaje por refuerzo (como el **valor Q medio por *epoch***) ya que el objetivo de los experimentos es ver la viabilidad e idoneidad de los agentes en el juego. Por esto, nos interesa más ver su rendimiento real como jugadores que como agentes de Deep Q-Learning.

Para garantizar que todos los experimentos son comparables entre sí, se han fijado los parámetros de los agentes a los siguientes valores:

- *Epochs*: 2000

- Semilla (para reproducibilidad): 0.
- Gamma (γ, depreciación de recompensas futuras): 0,99.
- Epsilon (ϵ, probabilidad de realizar una acción aleatoria): 1,0 en el primer *epoch*, reduciendose linealmente hasta 0,05 tras el 75 % de los *epochs* (1500 *epochs* en este caso).
- Tamaño máximo del *Replay Memory*: 20000.
- Tamaño de las muestras del *Replay Memory*: 32.
- Razón de aprendizaje de la red neuronal: 0,001.

Estos valores fueron inicialmente tomados de otros experimentos con éxito realizados en juegos de Atari [2] para tener una base, y fueron ajustados manualmente de forma experimental hasta los parámetros actuales.

6.2. Resultados

Los resultados obtenidos serán separados por aproximaciones, para poder establecer una comparación entre los agentes de cada aproximación. Estos resultados serán expuestos en forma de gráficas, para una mayor claridad. Además, se ha aplicado un filtro a las gráficas para obtener resultados más suavizados y legibles, como se describe en el anexo "Script auxiliar para generación de gráficas" [B].

En general, los agentes no han sido capaces de completar **líneas** de una forma consistente, por lo que una gráfica mostrando las líneas por *epoch* no sería informativa. En su lugar, se ha decidido utilizar un diagrama mostrando las **líneas totales** que se han eliminado durante todo el proceso de entrenamiento, pudiendo mostrar de esa forma una aproximación mejor al rendimiento del agente en este aspecto.

Las gráficas utilizadas (tanto las originales como las suavizadas) son incluidas junto a la memoria en una resolución mayor, como se describe en el anexo "Contenidos del CD" [C].

6.2.1. Primera aproximación

Los agentes entrenados para la primera aproximación han sido los agentes **estándar** y **con acciones no uniformes**, ambos siendo entrenados por separado con los sistemas de recompensas **de juego** y **heurístico**.

Comentando las **líneas totales** realizadas por cada agente, recogidas en la figura 6.1, vemos que el mejor rendimiento es el del agente con **pesos no uniformes** cuando usa **recompensas de juego**. El siguiente mejor agente es el **estándar**, obteniendo

resultados similares independientemente de las recompensas utilizadas. Finalmente, en contra de lo que podría esperarse (al usar recompensas más informadas), el peor rendimiento lo ofrece el agente de pesos no uniformes cuando se utilizan **recompensas heurísticas**, apenas pasando de 6 líneas.

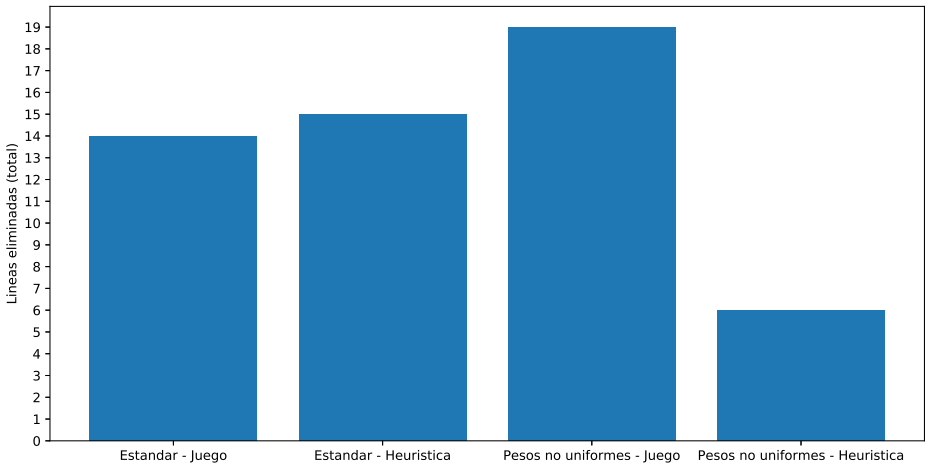


Figura 6.1: Líneas totales realizadas por cada agente - Primera aproximación.

Respecto a la evolución de la **puntuación** durante el entrenamiento, recogida en la figura 6.2, vemos que al final los mejores resultados los ofrecen ambos agentes cuando se utilizan las **recompensas de juego**, teniendo el agente de **pesos no uniformes** una mejora continua durante el entrenamiento y el agente **estándar** un valle hasta aproximadamente el *epoch* 1200, tras lo cual mejora rápidamente.

Contrastando, los agentes que utilizan la **recompensa heurística** obtienen puntuaciones peores, manteniéndose la puntuación del agente **estándar** estable e incluso empeorando la puntuación del agente de **pesos no uniformes** con el tiempo.

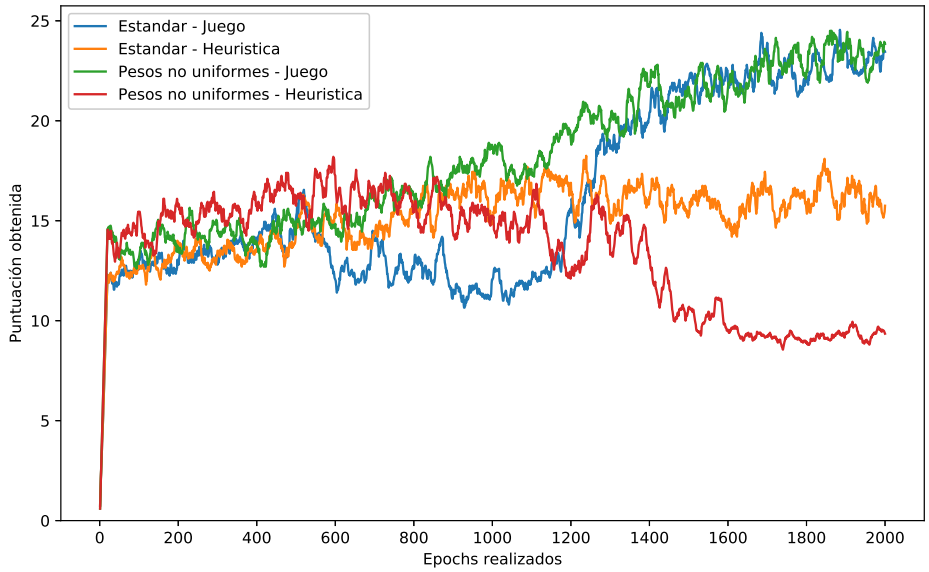


Figura 6.2: Puntuación obtenida en cada *epoch* por los agentes - Primera aproximación.

Hablando del número de **acciones** realizadas, recogidas en la figura 6.3, el agente con más acciones (y, por tanto, el más longevo) es el agente **estándar** usando **recompensas heurísticas**, aumentando las acciones con el tiempo. Contrastando, el resto de agentes realiza un número notablemente menor de acciones. Se puede observar cómo los dos agentes que utilizan **recompensas de juego** ofrecen resultados muy similares.

De nuevo, el agente **de pesos no uniformes** con **recompensas heurísticas** obtiene los peores resultados, empezando a empeorar notablemente a mitad del entrenamiento. Esto sugiere que este agente empezó a divergir en algún punto, empeorando su resultado en todos los aspectos.

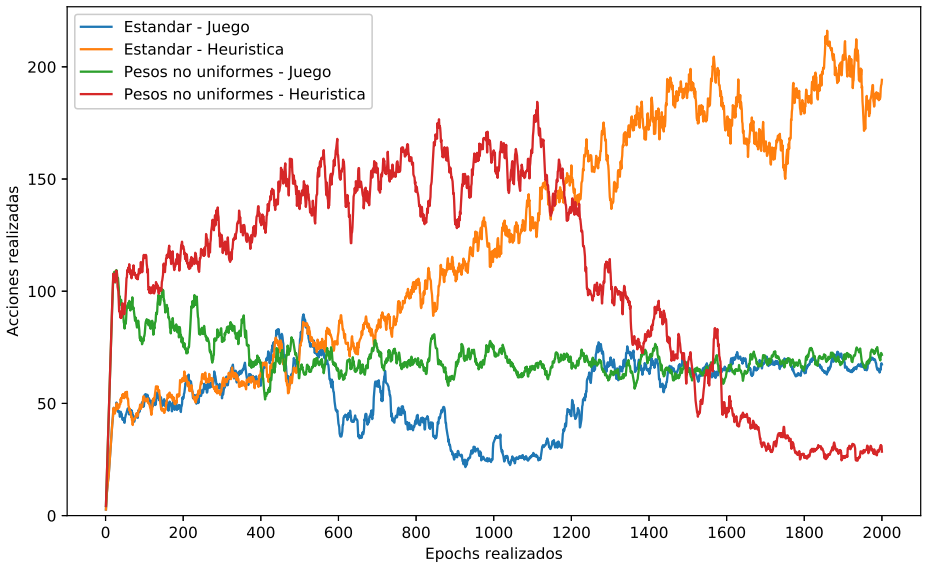


Figura 6.3: Acciones realizadas en cada *epoch* por los agentes - Primera aproximación.

6.2.2. Segunda aproximación

Los agentes entrenados para la segunda aproximación han sido los agentes **estándar** (en la versión adaptada para esta aproximación) y **con *Prioritized Experience Replay*** (desde este momento referido como **agente priorizado**), ambos siendo entrenados por separado con los sistemas de recompensas **de juego** y **heurístico**.

Empezando por las **líneas totales** completadas por los agentes, recogidas en la figura 6.4, se observa a simple vista que el mejor rendimiento lo ofrece el **agente priorizado** con recompensas de **juego**, completando sobre 140 líneas durante todo el entrenamiento, el doble que el siguiente mejor agente, el **agente estándar** con recompensas de **juego** (que completa aproximadamente 70 líneas).

Los agentes con recompensas **heurísticas** ofrecen los peores resultados, completando unas 20 líneas cada uno. Aun así, es importante destacar que estos resultados siguen siendo iguales o mejores a los mejores resultados de la primera aproximación.

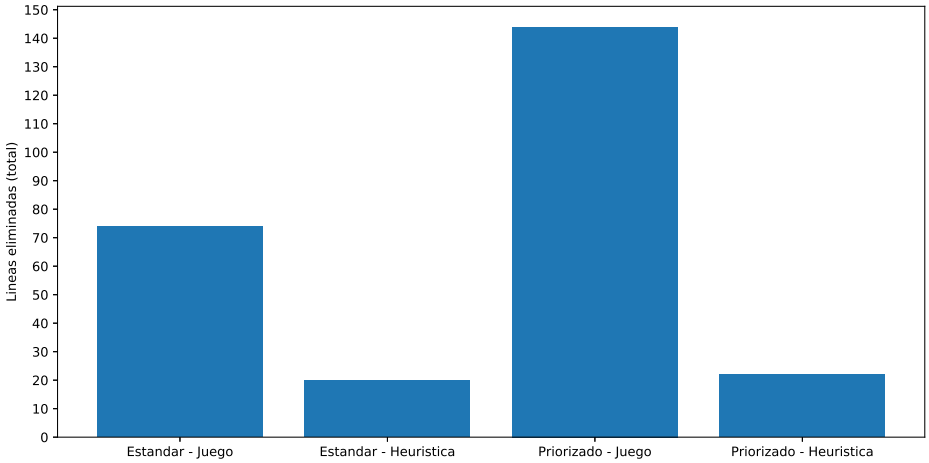


Figura 6.4: Líneas totales realizadas por cada agente - Segunda aproximación.

Hablando de la evolución de la **puntuación** durante el entrenamiento, como se puede ver en la figura 6.5, vemos que de nuevo el mejor agente es el **priorizado** con recompensas de **juego**, seguido por el agente **estándar** con recompensas **heurísticas** (aunque el rendimiento de ambos es muy similar, incrementando linealmente con el tiempo).

El rendimiento de los otros agentes (**estándar** con recompensas **de juego** y **priorizado** con recompensas **heurísticas**) es peor, manteniendo su rendimiento constante (o empeorando incluso un poco) con el tiempo.

Como se puede ver también, el rendimiento de los dos mejores agentes es muy similar en puntuación, a pesar de que el mejor (**priorizado** con recompensas **de juego**) completa el doble de líneas que el siguiente mejor agente.

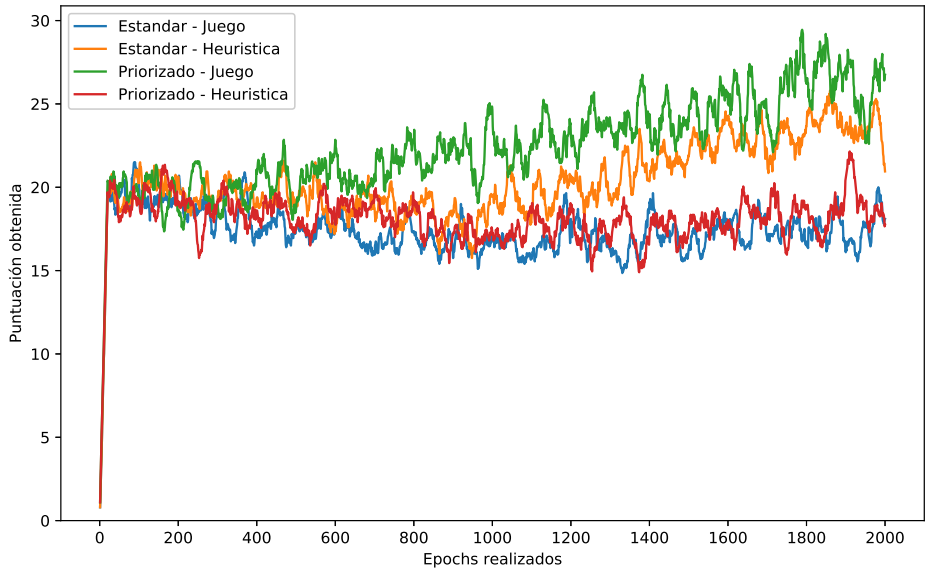


Figura 6.5: Puntuación obtenida en cada *epoch* por los agentes - Segunda aproximación.

Finalmente, estudiando el número de acciones (la longevidad) de cada agente, recogido en la figura 6.6, vemos un comportamiento muy parecido al de la gráfica anterior: los dos mejores agentes son los agentes **priorizado** con recompensas **de juego** y **estándar** con recompensas **heurísticas**, aumentando el número de acciones realizadas linealmente con el tiempo.

Los otros dos agentes vuelven a estancarse, no mejorando su rendimiento en esta faceta con el paso del tiempo, manteniendo la cantidad de acciones realizadas por *epoch* alrededor de 80 de forma estable.

Esta relación entre **puntuación** y **acciones** es de esperar, ya que ambas variables están relacionadas: cuanto más tiempo sobrevive el agente en la partida, más piezas puede colocar y más líneas puede llenar (y, por tanto, más puntuación puede conseguir).

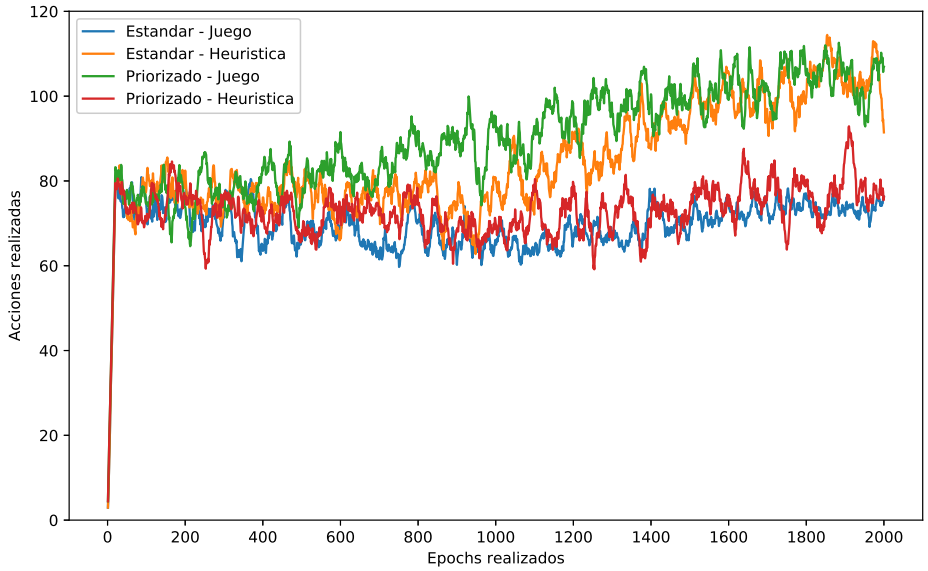


Figura 6.6: Acciones realizadas en cada *epoch* por los agentes - Segunda aproximación.

Capítulo 7

ANÁLISIS DE RESULTADOS

En este capítulo se estudiarán los resultados obtenidos en la experimentación, analizándolos y extrayendo conclusiones. Específicamente se compararán los resultados de las dos aproximaciones entre ellos, viendo el resultado de los agentes durante el entrenamiento y el de los agentes entrenados en partidas reales. Finalmente, se comentarán los puntos claves aprendidos a partir de este análisis.

7.1. Comparativa de los resultados en el aprendizaje

Para realizar la comparación de las dos aproximaciones se elegirá la mejor variante de cada agente estudiado como representante. De esta forma se conseguirán gráficas representativas pero legibles. La mejor variación de cada agente se ha elegido en base a la **puntuación** (al no ser las líneas consistentes, se ha considerado un mejor estimador de la habilidad del agente).

Los agentes elegidos son:

- Agente estándar (primera aproximación) con recompensas de juego.
- Agente de pesos no uniformes con recompensas de juego.
- Agente estándar (segunda aproximación) con recompensas heurísticas.
- Agente priorizado con recompensas de juego.

Como se puede ver en la figura 7.1, los agentes de la segunda aproximación obtienen un mejor resultado completando más líneas que los de la primera. Concretamente, el agente **priorizado** es notablemente mejor que el resto de agentes, y el agente **estándar de la segunda aproximación** es también mejor que todos los demás agentes de la primera aproximación (a pesar de ser el agente de la segunda aproximación que menos líneas totales hizo).

Pese a esto, hay que recordar que estos resultados son las **líneas totales** durante todo el entrenamiento (2000 *epochs*). Esto significa que el mejor agente (el que ha completado más líneas) de media solo completaría una línea cada 15 partidas. Este resultado indica que los agentes entrenados no son buenos jugadores de Tetris.

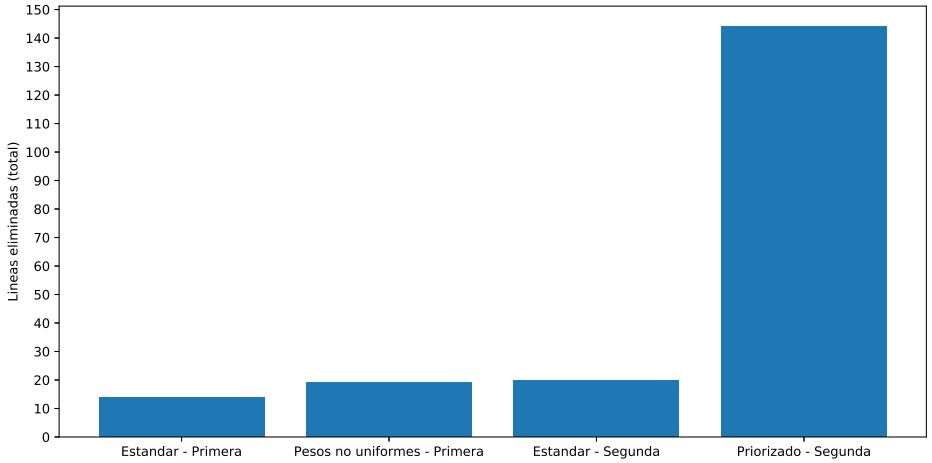


Figura 7.1: Líneas totales realizadas por cada agente - Comparativa.

Si nos centramos en la **puntuación**, recogida en la figura 7.2, la diferencia es más ajustada. El agente **priorizado** ofrece de nuevo los mejores resultados (creciendo lentamente con el tiempo), mientras que el resto de agentes acaban obteniendo una puntuación similar y ligeramente inferior.

Es interesante destacar que el aprendizaje de los agentes de la primera aproximación es más pronunciado en el tiempo, por lo que (suponiendo un crecimiento constante) podrían obtener mejores resultados con un entrenamiento más largo.

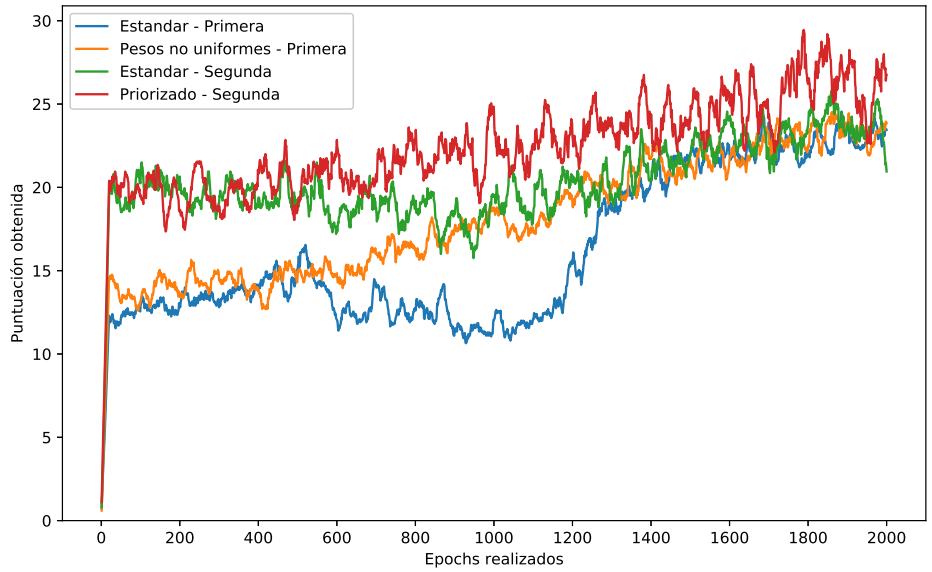


Figura 7.2: Puntuación obtenida en cada *epoch* por los agentes - Comparativa.

Finalmente, comparando las **acciones** realizadas, recogidas en la figura 7.3, se observa claramente que los agentes de la segunda aproximación realizan un mayor número de acciones, ambos teniendo resultados similares y creciendo con el tiempo. En cambio, los resultados de la primera aproximación se mantienen constantes, también siendo parecidos entre ellos.

Esta brecha se puede explicar teniendo en cuenta la diferencia en las acciones de cada aproximación: la primera aproximación tiene la posibilidad de dejar caer la pieza instantáneamente en cualquier momento (en vez de seguir una secuencia fija de pasos como en la segunda aproximación), haciendo más probable que se bloqueen las piezas más rápido y, por tanto, que acaben antes la partida (al no ser capaces de completar líneas para hacer hueco).

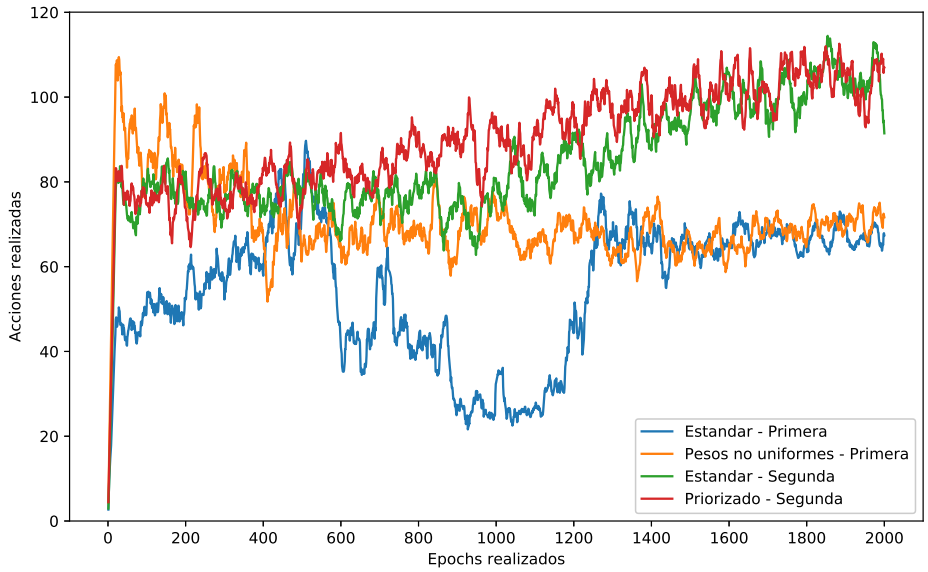


Figura 7.3: Acciones realizadas en cada *epoch* por los agentes - Comparativa.

7.2. Comparativa de los resultados en el juego

De cara a obtener una valoración más honesta del rendimiento de los agentes entrenados, se comparará su rendimiento en partidas reales tras el proceso de aprendizaje.

Para esto, se jugarán **diez** partidas consecutivas (usando una semilla inicial 0) con cada combinación de **agente** y **heurística** entrenada, calculando el valor medio de las variables estudiadas previamente (líneas completadas, puntuación obtenida y acciones realizadas). Tras esto, se compararán los resultados obtenidos.

Se incluyen en la tabla los rendimientos de tres agentes adicionales a los ya entrenados: dos agentes de *baseline* (los agentes aleatorios de ambas aproximaciones) y un agente *objetivo* (lo que se buscaría alcanzar). Este agente (llamado **agente El-Tetris**) es un agente algorítmico (sin ninguna clase de aprendizaje) que utiliza el algoritmo

El-Tetris [28], basado en evaluar los posibles estados alcanzables con una función heurística y elegir el estado con una mejor valoración.

Los resultados están contenidos en la tabla 7.1, estando ésta dividida a su vez en tres grupos de agentes: los **referentes** (baselines y objetivo), los de la **primera aproximación** y los de la **segunda aproximación**. El mejor agente en cada caso se encuentra marcado en negrita.

	Líneas	**Puntuación**	**Acciones**
El-Tetris (Algorítmico)	**90.3**	**1103.7**	**1048.5**
Aleatorio (Primera)	0	13.2	45
Aleatorio (Segunda)	0.1	21.9	81.2
Estándar - Juego (Primera)	0	22.9	62.1
Estándar - Heurística (Primera)	0	16.7	187.9
Pesos no uniformes - Juego (Primera)	**0**	**22.9**	**63.9**
Pesos no uniformes - Heurística (Primera)	0	9.2	21.6
Estándar - Juego (Segunda)	0	18.7	79.6
Estándar - Heurística (Segunda)	0	24.4	105.6
Priorizado - Juego (Segunda)	**0.2**	**25**	**96.9**
Priorizado - Heurística (Segunda)	0	17.7	77.8

Tabla 7.1: Rendimiento medio de los agentes en 10 partidas.

Se observa que los resultados concuerdan con los resultados finales estudiados durante el entrenamiento (obteniendo los agentes valores de líneas, puntuación y acciones similares a los presentes durante el final del entrenamiento).

También podemos ver que los agentes han sido capaces de mejorar su rendimiento, siendo todos ellos (salvo una excepción) mejores que el primer agente de *baseline* (el agente aleatorio de la primera aproximación) y teniendo los mejores agentes (los elegidos en la sección anterior) un rendimiento superior al segundo agente de *baseline* (el agente aleatorio de la segunda aproximación).

Ahora bien, también es fácil ver un problema: pese a que los agentes han mejorado su rendimiento, **no son buenos jugadores de Tetris**. Sólo el agente priorizado ha sido capaz de completar líneas en las partidas jugadas, y ninguno de los agentes se acerca siquiera al rendimiento del agente *objetivo*, pese a implementar éste un algoritmo sencillo utilizando conocimiento inyectado sobre Tetris.

7.3. Conclusiones del análisis

De estos dos análisis, se pueden obtener algunas conclusiones sobre los experimentos y el proyecto en general:

- Los agentes son capaces de aprender, por lo que **el algoritmo de Deep Q-Learning es aplicable a Tetris**. Como se ha visto, el rendimiento aumenta durante el entrenamiento de los agentes y son capaces de superar el rendimiento de un agente totalmente aleatorio (un *benchmark* básico).

- Las **recompensas heurísticas** propuestas no funcionan bien en este caso concreto. Salvo un agente (el **agente estándar de la segunda aproximación**), el rendimiento de todos los agentes es notablemente peor cuando se utilizan las recompensas heurísticas frente a las recompensas basadas directamente en la puntuación del juego.

- Como se esperaba, **la segunda aproximación ofrece un mejor rendimiento comparada con la primera**. Los dos mejores agentes (priorizado con recompensas de juego y estándar con recompensas heurísticas) pertenecen a la segunda aproximación, y un agente aleatorio utilizando la segunda aproximación ofrece un mejor rendimiento que su equivalente en la primera aproximación.

- **Prioritized Experience Replay** mejora el resultado de Deep Q-Learning. Se puede ver que el mejor agente es el agente priorizado, que implementa esta técnica. Esto indica que este añadido al algoritmo ofrece una mejora considerable en el rendimiento de los agentes, pudiendo aprender mejor (al tener más acceso a las experiencias importantes).

- Pese a ser capaces de aprender, los agentes entrenados **no son útiles** cuando se aplican a partidas reales de Tetris. Ninguno de los agentes es capaz de jugar de forma consistente (limitándose únicamente a intentar llenar el tablero colocando todas las piezas posibles antes de perder).

Como comparación, el agente **El-Tetris** ofrece un rendimiento muy superior sin necesidad de realizar ninguna clase de aprendizaje, utilizando únicamente conocimiento inyectado en forma de una heurística que evalua la calidad de las posibles zonas de juego tras colocar la pieza en juego.

Capítulo 8

CONCLUSIONES FINALES

En este capítulo se detallan las conclusiones del trabajo realizado, desarrollándolas. Tras esto, se comentarán futuras líneas de trabajo para este proyecto y se hablará de las competencias adquiridas con la realización de este proyecto.

8.1. Conclusiones

Los principales objetivos de este proyecto eran el **estudio** y la **aplicación** del algoritmo de **Deep Q-Learning** al juego de Tetris, con el fin de estudiar su viabilidad.

A lo largo de esta memoria se ha **estudiado** la técnica de Deep Q-Learning desde un punto de vista teórico (qué es la técnica y cómo se ha llegado hasta ella), además del estudio del juego en sí y cómo se caracterizaría todo el conocimiento necesario para aplicar la técnica al juego (incluyendo el conocimiento del problema de aprendizaje por refuerzo y la definición de los agentes).

Las principales conclusiones que se pueden obtener de este trabajo son:

- **La técnica de Deep Q-Learning no es útil cuando se aplica a Tetris**. Como se observó en el capítulo anterior, los agentes entrenados presentan un rendimiento muy malo: ningún agente es capaz de completar líneas de forma consistente ni de acercarse al rendimiento de **agentes algorítmicos** sin aprendizaje basados en heurísticas ya existentes.

  Ahora bien, esto **entra dentro de lo esperado** por dos motivos:

  - En el estudio original sobre Deep Q-Learning aplicado a videojuegos [2] se observó que la técnica funcionaba notablemente mejor en juegos en los que el agente tenía que reaccionar a su entorno frente a juegos estratégicos (en los que hay una gran diferencia temporal entre la toma de acciones y la obtención de recompensas por parte del agente), entre los que se incluye Tetris.

  - Aunque se ha intentado en otras ocasiones aplicar técnicas de aprendizaje por refuerzo a Tetris [29], en general siempre se ha observado que su rendimiento era malo comparado con otras técnicas como programación dinámica o algoritmos genéticos [30], teniendo resultados malos incluso cuando se comparan con jugadores humanos. Por tanto, se puede esperar que Deep Q-Learning (una técnica más moderna de aprendizaje por refuerzo) también presente este problema.

- Pese a no ser útil, **los agentes son capaces de aprender utilizando Deep Q-Learning**. Como se ha observado en el análisis de resultados, el rendimiento de los agentes ha mejorado durante el proceso de aprendizaje, y los agentes ya entrenados ofrecen un rendimiento superior a agentes aleatorios utilizados como *baseline*.

- **La representación del conocimiento influye de forma notable en el rendimiento de Deep Q-Learning**. Se ha podido ver en los resultados cómo cambia el rendimiento entre agentes a partir de cambios en la caracterización del conocimiento, como puede ser el cambio de la definición de las **acciones** (ofreciendo mejores resultados el uso de acciones más parecidas al razonamiento de jugadores humanos) o de las **recompensas** (teniendo mejor resultado los agentes que reciben recompensas parecidas a la puntuación del juego).

- **Los añadidos al algoritmo de Deep Q-Learning mejoran su rendimiento**. El agente que mejor rendimiento ha presentado es el que implementa *Prioritized Experience Replay*, un añadido al algoritmo que permite aprender de forma más eficiente de las experiencias.

  Deep Q-Learning es un algoritmo novedoso que sigue en continua investigación y evolución, por lo que existen numerosas mejoras actualmente que pueden aumentar su rendimiento.

8.2. Trabajo futuro

Existen varias formas de continuar este proyecto, con la idea de mejorar su rendimiento y utilidad:

- **Entrenar a los agentes en versiones distintas de Tetris:** Los agentes de este proyecto han sido entrenados en una versión simplificada de Tetris, con algunas limitaciones.

  Una posibilidad sería entrenar a los agentes con versiones de Tetris ajustadas a los reglamentos de diseño actuales (incluyendo por ejemplo la acción de guar-

dar pieza) y sin algunas de las simplificaciones tomadas (volviendo a añadir la dificultad variable, por ejemplo).

- **Continuar añadiendo mejoras a Deep Q-Learning:** Se han ido proponiendo diversos añadidos al algoritmo de Deep Q-Learning, con la idea de aumentar su rendimiento. Entre estas mejoras se incluye el *Doble DQL*, el *uso de redes neuronales enfrentadas*, el *aprendizaje por refuerzo distribuido*...

  La combinación de todas estas técnicas se conoce como el algoritmo **Rainbow** [31] (o, simplemente, *rainbow Deep Q-Learning*), ofreciendo los mejores resultados y siendo el estado del arte actualmente en este campo.

- **Probar una aproximación híbrida entre agente y jugador humano:** Es posible utilizar una aproximación híbrida entre Deep Q-Learning y aprendizaje automático supervisado estándar.

  Esta aproximación consiste en ir alternando el control entre el agente (utilizando alguna variante del algoritmo de Deep Q-Learning) y un jugador humano (que jugará y almacenará experiencias en el *Replay Memory* del agente, a partir de las cuales aprenderá).

- **Independizar al agente del juego:** Si bien esto no supondría una mejora de rendimiento, sí supondría una mejora de usabilidad el permitir a los agentes jugar a cualquier implementación de Tetris (no solo esta versión especifica) una vez han sido entrenados.

  En este caso, el agente no tendría acceso a las variables internas del juego, por lo que será necesario dotarlo de capacidad de interpretar capturas del juego y extraer la información necesaria de ahí.

8.3. Competencias

Se han desarrollado varias competencias con el trabajo en este proyecto. Las principales competencias que se han desarrollado en el ámbito de la **Computación** (en el que se enmarca este proyecto) son las siguientes:

- **[CM1]** Capacidad para tener un conocimiento profundo de los principios fundamentales y modelos de la computación y saberlos aplicar para interpretar, seleccionar, valorar, modelar, y crear nuevos conceptos, teorías, usos y desarrollos tecnológicos relacionados con la informática.

  Esta competencia se ha visto principalmente desarrollada en el estudio que se ha realizado tanto de Deep Q-Learning como de las técnicas en las que se fundamenta

(Deep Learning y Q-Learning). Gracias a este estudio y aprendizaje, ha sido posible la aplicación y adaptación apropiada del algoritmo al juego.

- **[CM5]** Capacidad para adquirir, obtener, formalizar y representar el conocimiento humano en una forma computable para la resolución de problemas mediante un sistema informático en cualquier ámbito de aplicación, particularmente los relacionados con aspectos de computación, percepción y actuación en ambientes o entornos inteligentes.

  Una parte importante de este proyecto ha sido la formalización del conocimiento que iba a ser utilizado, principalmente en la definición de los elementos del problema de aprendizaje por refuerzo (estado, acción y recompensas) y en la definición de los agentes (sus características, arquitecturas...)

- **[CM7]** Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.

  La técnica utilizada en este proyecto (Deep Q-Learning) es una de las técnicas más vigentes actualmente en el campo del aprendizaje por refuerzo. Por tanto, su implementación supone sin duda el uso de técnicas de aprendizaje (siendo precisamente el aprendizaje automático el objetivo principal de este proyecto).

Junto a estas competencias principales se han tratado otras competencias secundarias de diversa índole, incluyendo algunas **competencias específicas** (como el desarrollo de proyectos de software) y **transversales** (como la capacidad de redacción y comunicación o la capacidad de trabajo autónomo).

Bibliografía

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, no. 521, pp. 436–444, May 2015. 2, 14, 15

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb 2015. [Online]. Available: https://doi.org/10.1038/nature14236 2, 21, 22, 23, 62, 73

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org. 7

[4] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012. 7

[5] T. O. Ayodele, "Types of Machine Learning Algorithms," in *New Advances in Machine Learning*, Y. Zhang, Ed., 2010. [Online]. Available: http://www.intechopen.com/books/new-advances-in-machine-learning/types-of-machine-learning-algorithms 8

[6] S. Russell and P. Norvig, *Inteligencia Artificial: Un enfoque moderno*, 2nd ed. USA: Prentice Hall Press, 2003. 8, 9, 10, 12, 14, 17, 19

[7] "7 Types of Activation Functions in Neural Networks: How to Choose?" accedido el 02/06/2020. [Online]. Available: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ 9

[8] H. Vivas, H. J. Martínez, and R. Pérez, "Método secante estructurado para el entrenamiento del perceptrón multicapa," *Revista de Ciencias*, vol. 18, pp. 131 – 150, 12 2014. [Online]. Available: http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S0121-19352014000200010&nrm=iso 10

[9] Q. J. Zhang and K. C. Gupta, *Neural Networks for RF and Microwave Design (Book + Neuromodeler Disk)*, 1st ed. USA: Artech House, Inc., 2000. 11, 12

[10] L. De La Ossa and J. A. Gámez, "Apuntes de la asignatura Minería de Datos, Curso 2019/2020," 2019. 14

[11] J. Schmidhuber, "Deep Learning in Neural Networks: An Overview," *Neural Networks*, vol. 61, 04 2014. 14

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Mass.: MIT Press, 1998. [Online]. Available: http://incompleteideas.net/sutton/book/ebook/the-book.html 15, 16, 17, 19

[13] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992698 18, 19

[14] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An Introduction to Deep Reinforcement Learning," *CoRR*, vol. abs/1811.12560, 2018. [Online]. Available: http://arxiv.org/abs/1811.12560 20, 21, 22, 23

[15] R. Heberer, "Why Going from Implementing Q-learning to Deep Q-learning Can Be Difficult," 2019, accedido el 04/06/2020. [Online]. Available: https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-be-difficult-36e7ea1648af 20

[16] S. Ruder, "An overview of gradient descent optimization algorithms," 2016, accedido el 04/06/2020. [Online]. Available: https://ruder.io/optimizing-gradient-descent/ 23, 41

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with Deep Reinforcement Learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602 23

[18] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," 2016. [Online]. Available: https://arxiv.org/abs/1511.05952 24, 25, 26, 56

[19] HardDrop, "Gameplay overview," 2017, accedido el 07/06/2020. [Online]. Available: https://harddrop.com/wiki/Gameplay_overview 27

[20] ——, "Tetris Guideline," 2017, accedido el 07/06/2020. [Online]. Available: https://harddrop.com/wiki/Tetris_Guideline 30

[21] ——, "Random Generator," 2017, accedido el 08/06/2020. [Online]. Available: https://harddrop.com/wiki/Random_Generator 32

[22] P. Shinners *et al.*, "pygame," 2011, accedido el 14/06/2020. [Online]. Available: http://pygame.org/ 32

[23] techwithtim, "PyGame Tutorial - Creating Tetris," 2018, accedido el 08/06/2020. [Online]. Available: https://techwithtim.net/tutorials/game-development-with-python/tetris-pygame/tutorial-1/ 32

[24] Y. Lee, "Tetris AI - The (Near) Perfect Bot," 2013, accedido el 09/06/2020. [Online]. Available: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/ 38, 50

[25] F. Chollet *et al.*, "Keras," 2015, accedido el 14/06/2020. [Online]. Available: https://keras.io 39, 51

[26] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html 40

[27] M. Stevens and S. Pradhan, "Playing Tetris with Deep Reinforcement Learning," 2016. [Online]. Available: http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf 48

[28] I. El-Ashi, "El-Tetris - An Improvement on Pierre Dellacherie's Algorithm," 2011, accedido el 14/06/2020. [Online]. Available: https://imake.ninja/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/ 70

[29] D. Carr, "Applying reinforcement learning to Tetris," 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.580.1421&rep=rep1&type=pdf 74

[30] S. Algorta and Ö. Simsek, "The game of tetris in machine learning," *CoRR*, vol. abs/1905.01652, 2019. [Online]. Available: http://arxiv.org/abs/1905.01652 74

[31] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *CoRR*, vol. abs/1710.02298, 2017. [Online]. Available: http://arxiv.org/abs/1710.02298 75

[32] M. Abadi *et al.*, "Install TensorFlow 2," 2020, accedido el 14/06/2020. [Online]. Available: https://www.tensorflow.org/install 84

[33] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007, accedido el 14/06/2020. [Online]. Available: https://matplotlib.org/3.1.1/index.html 88

Apéndice A

MANUAL DE USUARIO

Este anexo contiene información relevante para el uso del proyecto. Concretamente se especifica el funcionamiento del proyecto para un jugador humano (indicando las teclas usadas para controlarlo y capturas del proceso), además de los requisitos necesarios para su uso y las opciones de lanzamiento disponibles para acceder a los diversos modos y ajustar el funcionamiento.

A.1. Uso del juego

El proyecto se lanza ejecutando el fichero *tetris.py* a través de la consola de comandos. Una vez se ha lanzado el juego, se muestra al jugador la pantalla de título, como se puede ver en la figura A.1. Desde esta pantalla es posible cerrar el proyecto pulsando la tecla **ESC** o continuar al juego en sí pulsando cualquier otra tecla.

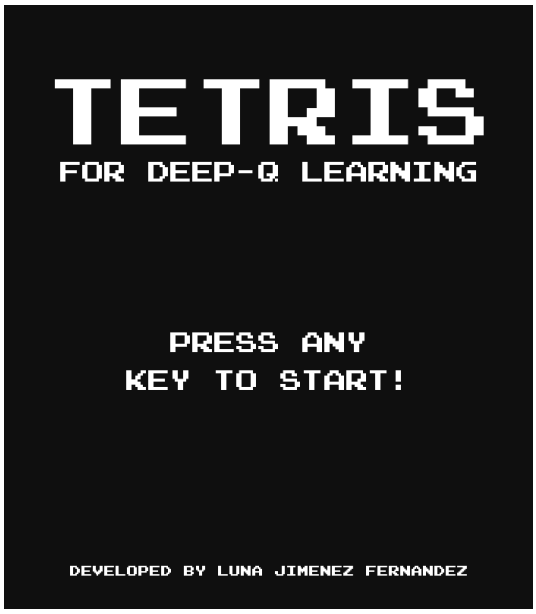


Figura A.1: Pantalla de título del proyecto.

Una vez dentro del juego, se muestra en pantalla al jugador la zona de juego típica de Tetris. En esta zona se observan las piezas ya bloqueadas, al fondo, junto a la pieza actualmente en juego cayendo (indicando la posición en la que caerá con una pieza fantasma, como se describió en el cuerpo de la memoria).

Es posible controlar la pieza en juego con las siguientes teclas:

- **Flecha derecha** y **Flecha izquierda**: Desplazar la pieza hacia la derecha o la izquierda, respectivamente.

  No es posible mantener pulsada la tecla para continuar moviendo la pieza, por lo que es necesario pulsarla una vez por cada desplazamiento.

- **Tecla R**: Rotar la pieza 90 grados hacia la derecha.

- **Flecha abajo**: Hacer caer la pieza más rápido. Específicamente, pulsar la tecla hace que la pieza baje una fila automáticamente.

- **Flecha arriba**: Dejar caer instantáneamente la pieza, bloqueándola en la posición indicada por la pieza fantasma.

Además, es posible pulsar la tecla **ESC** en cualquier momento para acabar la partida y volver al menú principal.

A la derecha de esta zona se muestra información relevante para el jugador: la **puntuación** actual, las **líneas** completadas durante la partida, la **dificultad** actual y un indicador de la próxima pieza que aparecerá. Se puede observar una captura del proyecto durante una partida normal en la figura A.2.

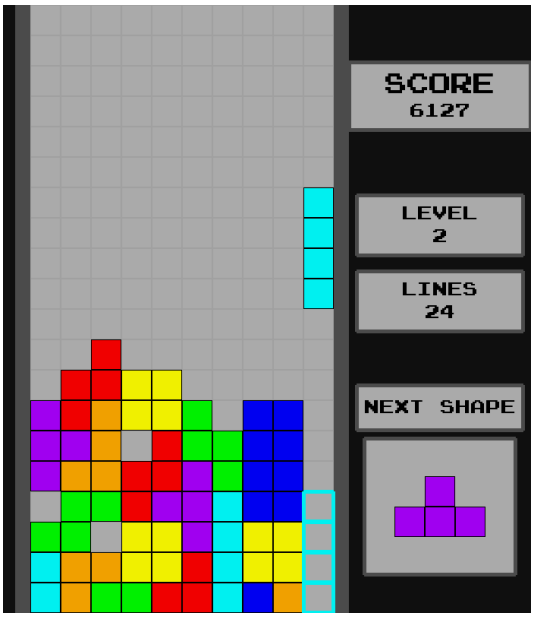


Figura A.2: Captura de una partida estándar.

La partida continua hasta que una pieza se bloquea en el extremo superior de la zona de juego, tras lo cual se mostrará un efecto visual (la pantalla llenándose de bloques grises, de abajo a arriba) y tras una breve pausa volverá automáticamente a la pantalla del título. Se puede observar el final de la partida en la figura A.3.

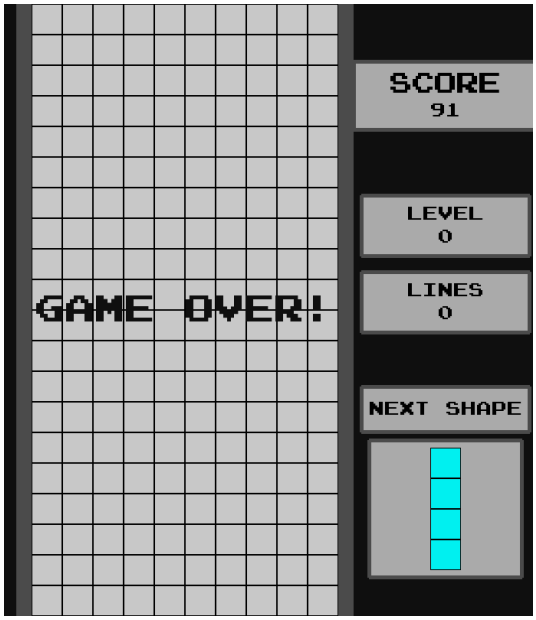


Figura A.3: Final de la partida.

A.2. Requisitos y dependencias

Los requisitos para el uso del proyecto desarrollado (junto a las versiones que fueron utilizadas durante su desarrollo) son los siguientes:

- **Python v3.7.6.** con las siguientes librerías:
  - TensorFlow v2.1.0.
  - Keras v2.3.1.
  - h5py v2.10.0.
  - Numpy v1.18.1.
  - pygame v1.9.6.
- **Windows 10** como sistema operativo.

Además, para poder utilizar una GPU (de cara a agilizar el uso de las redes neuronales mediante paralelización) se han utilizado las siguientes dependencias adicionales:

- Una GPU **nVidia GTX 1660 Ti** compatible con CUDA.

- Drivers actualizados para la GPU (v441.87).
- **CUDA** Toolkit v10.1.243.
- **cuDNN** v6.14.11.10010.

La instalación de TensorFlow puede resultar complicada, especialmente si se quiere usar con GPUs. Por esto, es recomendable seguir la guía ofrecida [32] donde se indican los pasos y versiones a instalar de todos los componentes.

Si bien es posible que otra combinación de versiones del software, hardware o librerías diferente a las indicadas funcione, ésta es la única que ha sido probada y de la que se puede garantizar un funcionamiento correcto.

A.3. Opciones de lanzamiento

Como ya se ha indicado, el proyecto es lanzado a través de una consola de comandos estándar con el comando *python tetris.py*. Ahora bien, existe una serie de argumentos opcionales que se pueden añadir al lanzamiento (escribiéndolos detrás del comando) para modificar el comportamiento del juego.

Las opciones de lanzamiento son las siguientes:

- *-sil/--silent*: Modo silencioso. Desactiva todos los sonidos del juego.
- *-fs/--fixedspeed*: Modo de velocidad fija. Cambia el funcionamiento de la puntuación y la dificultad del juego al modo utilizado por los agentes (con puntuación simplificada y velocidad constante)

  Esta opción solo es aplicable cuando no está activo ningún modo con agentes (modo juego o modo aprendizaje), ya que en esos casos está activa por defecto y no es posible desactivarla.
- *-ai <modo>/--ai <modo>*: Activa el modo con agentes especificado por el usuario. *modo* puede tomar dos valores:
  - *play*: Activa el modo juego, diseñado para que agentes ya entrenados jueguen.
  - *learn*: Activa el modo aprendizaje, diseñado para entrenar a los agentes durante un número determinado de partidas.
- *-s <semilla>/--seed <semilla>*: Especifica la semilla que se utilizará para todos los eventos aleatorios (incluyendo la generación de piezas y las decisiones de exploración-explotación durante el entrenamiento de los agentes).

  Si no se fija una semilla, se utilizará una semilla aleatoria por defecto.

Además de estas opciones, existen unas opciones específicas para los modos juego y aprendizaje, disponibles únicamente cuando está activo alguno de estos modos:

- *-at <agente>/--agenttype <agente>*: Especifica el tipo de agente que se utilizará. Los tipos de agentes disponibles son los siguientes:
  - *standard_new*: Agente estándar de la segunda aproximación.
  - *prioritized_new*: Agente con *Prioritized Experience Replay* de la segunda aproximación.
  - *random_new*: Agente aleatorio de la segunda aproximación.
  - *standard_old*: Agente estándar de la primera aproximación.
  - *weighted_old*: Agente con pesos no uniformes de la primera aproximación.
  - *random_old*: Agente aleatorio de la primera aproximación.
  - *el-tetris*: Agente algorítmico implementando **El-Tetris** para tomar las decisiones.

  El valor por defecto es *standard_new* (por defecto se utiliza un agente estándar de la segunda aproximación).

- *-w <pesos>/--weights <pesos>*: Carga unos pesos pre-entrenados en el agente. *pesos* es la ruta al fichero H5 donde están almacenados estos pesos. Esta opción está solo disponible en modo juego (el modo entrenamiento siempre parte de pesos aleatorios).

  Si no se especifican unos pesos a cargar, el agente utilizará los pesos aleatorios iniciales para su red neuronal.

- *-f/--fast*: Activa el modo rápido. Este modo desactiva la interfaz gráfica y cambia el reloj físico del juego por un reloj lógico para poder simular las partidas más rápido. Esto agiliza notablemente el entrenamiento de los agentes, pudiendo simular partidas en segundos.

  Además, si se activa en modo juego, se ejecutan diez partidas consecutivas, devolviendo los valores medios de líneas, puntuación y acciones.

Finalmente, hay una serie de opciones solo aplicables en modo entrenamiento, usadas para ajustar los parámetros del algoritmo de Deep Q-Learning:

- *-er <tamaño>/--experiencereplay <tamaño>*: Especifica el tamaño máximo del *Replay Memory*. Por defecto, el tamaño máximo es **20000** experiencias.

- *-b <tamaño>/--batchsize <tamaño>*: Especifica el tamaño de las muestras del *Replay Memory* usadas para entrenar la red neuronal. Por defecto, el tamaño de la muestra es de **32** experiencias.

- *-rw <sistema>/--reward <sistema>*: Indica el sistema de recompensas que se utilizará. *sistema* puede tomar dos valores:

  - *game*: Se utilizará el sistema de recompensas de juego.
  - *heuristic*: Se utilizará el sistema de recompensas heurísticas.

  Por defecto, se utiliza el **sistema de recompensas de juego**.

- *-g <gamma>/--gamma <gamma>*: Fija el valor de gamma (γ), indicando la importancia dada a las recompensas futuras. El valor por defecto de *gamma* es **0.99**.

- *-eps <epsilon>/--epsilon <epsilon>*: Fija el valor inicial de epsilon (ϵ), indicando la probabilidad inicial de realizar una acción aleatoria durante el proceso de exploración-explotación. El valor por defecto de *epsilon* es **1**.

- *-epp <porcentaje>/--epsilonpercentage <porcentaje>*: Fija el porcentaje de *epochs* que pasan entre el valor inicial y el valor final de epsilon (cuando se hayan realizado *porcentaje* % *epochs*, el valor de epsilon será mínimo). Por defecto, su valor es de **75** %.

- *-mep <minimo>/--minimumepsilon <minimo>*: Indica el valor mínimo que alcanzará epsilon. Su valor por defecto es de **0.05**.

- *-lr <razon>/--learningrate <razon>*: Indica la razón de aprendizaje de la red neuronal (el valor que se da a las actualizaciones de peso durante la retropropagación del error). Por defecto, su valor es de **0.001**.

- *-epo <epochs>/--epochs <epochs>*: Indica el número de *epochs* (partidas) sobre las que se entrenará el agente. Por defecto, el agente se entrena jugando **2000** partidas.

Apéndice B

SCRIPT AUXILIAR PARA GENERACIÓN DE GRÁFICAS

Este anexo detalla el script auxiliar que se ha utilizado para generar las gráficas utilizadas en los capítulos "Experimentación y resultados" y "Análisis de resultados". Se describirá la estructura de los ficheros CSV generados por los agentes y el uso del script para procesar estos ficheros, generando las figuras pertinentes.

B.1. Estructura del fichero CSV

Durante el entrenamiento, todos los agentes generan un fichero CSV en el que se va almacenando información relevante del proceso. Tras cada *epoch*, el agente escribe la siguiente información en el fichero (en el orden indicado):

- El *epoch* actual.
- La puntuación obtenida en este *epoch*.
- Las líneas completadas en este *epoch*.
- Las acciones realizadas (primera aproximación) o los desplazamientos realizados (segunda aproximación) durante este *epoch*.

B.2. Uso del script

El script está contenido en el fichero *plot_ generator.py*. Para poder utilizar el script, es necesario además indicarle por argumentos los ficheros CSV sobre los que se quieren generar gráficas. Esto se indica con la opción de lanzamiento

-f/--file <ruta al fichero>[<nombre del agente>]

donde los argumentos son:

- **Ruta al fichero**: La ruta al fichero CSV donde se almacena la información a procesar. Esta ruta debe contener la extensión *.csv*.
- **Nombre del agente**: Opcional. Nombre que se dará a este conjunto de datos. El nombre se utiliza para identificar al conjunto de datos en la leyenda de los diagramas de puntos, y para nombrar la barra apropiada en el diagrama de barras.

Este argumento puede aparecer una o más veces. Si se utiliza varias veces se generarán las gráficas conteniendo todos los conjuntos de datos a la vez, permitiendo así generar diagramas para comparar la información.

El script generará gráficas para las siguientes variables:

- **Líneas**: se generará un diagrama de puntos (donde el eje X son los *epochs* y el eje Y las líneas completadas en ese *epoch*) y un diagrama de barras conteniendo las **líneas totales** completadas por cada conjunto de datos.
- **Puntuación**: se generarán dos diagramas de puntos (donde el eje X son los *epochs* y el eje Y la puntuación obtenida en ese *epoch*). Uno de los diagramas contendrá la información original, y el otro tendrá la información suavizada.
- **Acciones totales**: se generarán dos diagramas de puntos (donde el eje X son los *epochs* y el eje Y las acciones realizadas en ese *epoch*). Uno de los diagramas contendrá la información original, y el otro tendrá la información suavizada.

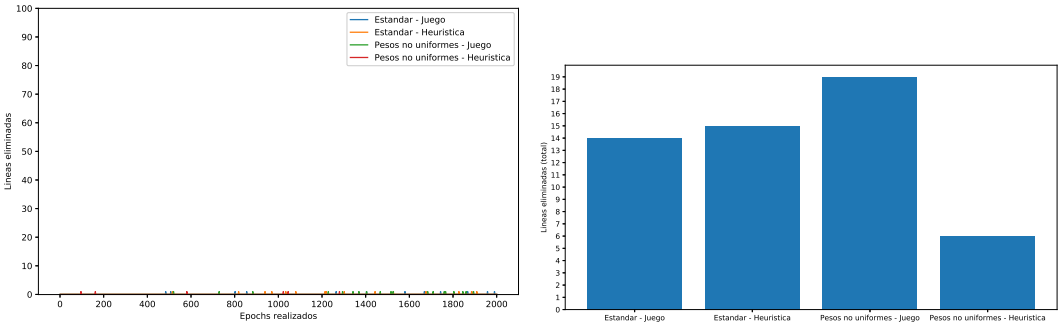
El script utiliza la librería **Matplotlib** [33] para generar todas las figuras (siendo esta una librería típica de Python utilizada para la creación de figuras y gráficos). Las gráficas se crearán en dos formatos: PNG (para tener gráficas en alta resolución que puedan ser observadas por humanos) y EPS (para poder utilizarlas y escalarlas en la memoria sin perder resolución).

La generación de los gráficos de puntos es simple y se realiza como se podría esperar (utilizando directamente los datos del fichero CSV para crear las figuras, tomando la columna de *epochs* como el eje X y la columna apropiada para el eje Y). Ahora bien, es importante comentar los otros dos tipos de gráficas que se han mencionado:

- **Diagrama de barras**: Al no ser capaces los agentes de completar líneas de forma consistente, las gráficas de puntos generadas a partir de estos datos son muy poco útiles (ya que consisten en líneas rectas con pequeños bultos cada vez que se llena una línea, y si hay varios conjuntos de datos a la vez se suelen solapar entre ellas).

  Por esto, se optó por generar adicionalmente un diagrama de barras a partir de los datos. De esta forma, es posible ver de forma más clara y directa el rendimiento de los agentes durante todo el entrenamiento.

Se puede observar una comparación entre el diagrama de puntos original y el diagrama de barras asociado generado en la figura B.1.



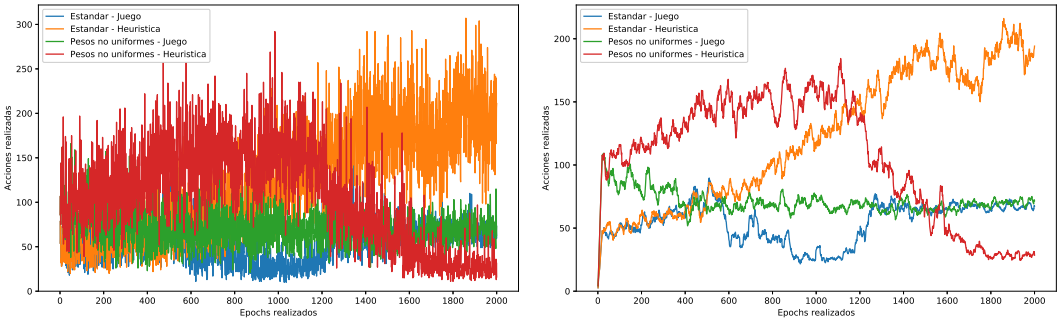
(a) Diagrama de puntos. (b) Diagrama de barras.

Figura B.1: Comparación de los diagramas de líneas generados.

- **Diagramas suavizados**: En el caso de la puntuación y las acciones realizadas, los resultados que obtienen los agentes son muy inestables, sin llegar a converger. Por tanto, cuando esto se representa en una gráfica se obtienen figuras excesivamente ruidosas y difíciles de entender.

  Para evitar esto, se aplica un **suavizado** a los datos, utilizando un filtro **lfilter** de la librería **scipy** (librería estándar de computación para Python). Este filtro suaviza el ruido en los datos, generando figuras con la misma forma pero mucho más legibles.

  Se puede observar una comparación entre un diagrama de puntos original y uno suavizado en la figura B.2. Es importante destacar que, aunque parezca que el segundo gráfico tiene valores distintos, los valores son iguales: lo que ha cambiado es la escala del eje Y (al perder los puntos más extremos por el suavizado).



(a) Sin aplicar filtro. (b) Aplicando filtro.

Figura B.2: Comparación de diagramas sin filtro y filtrados generados.

Apéndice C

CONTENIDOS DEL CD

En el contenido del CD que acompaña a la memoria se pueden encontrar los siguientes recursos:

- Memoria del trabajo en formato PDF en el directorio "Memoria".
- Código fuente del trabajo dentro del directorio "Código fuente".
- Ficheros CSV con la información sobre el entrenamiento de los agentes dentro del directorio "Ficheros CSV".
- Pesos entrenados de los agentes, en formato H5, dentro del directorio "Pesos".
- Gráficas usadas para el estudio de resultados, en mayor resolución, dentro del directorio "Gráficas".

Todo el contenido descrito en este anexo se encontrará disponible además, con una estructura similar, en un repositorio público online (hospedado en *GitHub*), disponible en el siguiente enlace: https://github.com/MoonDollLuna/dqlearning-tetris.