2048 con Deep Q-Learning

Inteligencia Artificial de Inspiración Biológica
Máster en Ciencia y Tecnología Informática

Cosmin Petre - 100428943

Índice

Introducción

En este trabajo se presenta la aplicación de la técnica de aprendizaje por refuerzo profundo Deep Q-Learning o DQN para la resolución del juego 2048.

En las secciones restantes se encuentra una descripción del juego 2048 en sí; descripción de la técnica Q-Learning original así como de la variante DQN empleada en este trabajo; cómo se aplicó esta técnica al juego 2048; una evaluación del agente DQN entrenado; y por último conclusiones.

Descripción del juego 2048

2048 es un juego sencillo que consta de un tablero de 4x4. Cada casilla puede contener una baldosa o estar vacía. Los valores de las baldosas son las potencias de 2 (2, 4, 8, 16, 32, …, 2048). Cuando dos baldosas del mismo valor se encuentran adyacentes se pueden combinar en una única baldosa cuyo valor será la suma de ambas.

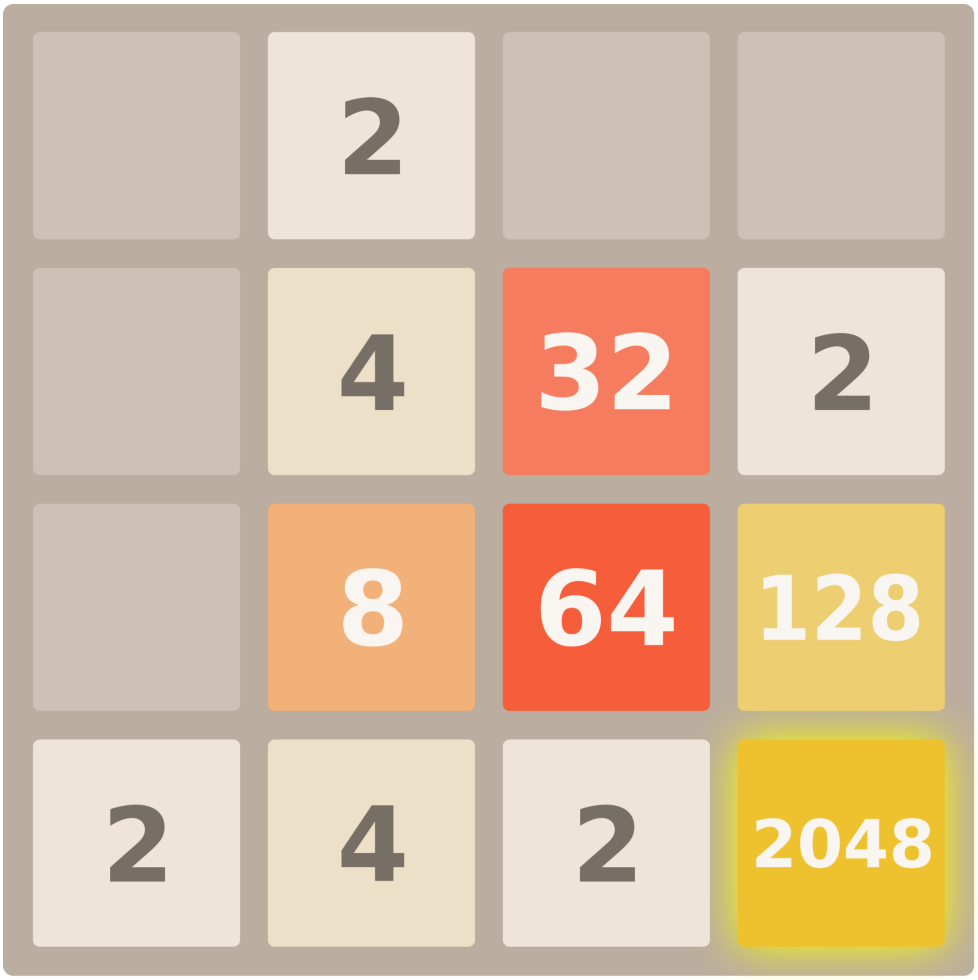

Figura 1: Juego 2048

El jugador tiene 4 posibles acciones: mover todas las baldosas hacia arriba, abajo, derecha o izquierda. Para mover las baldosas debe haber al menos una casilla vacía o dos baldosas iguales que se puedan combinar.

Cada baldosa alcanzada da puntos: cuanto más grande el valor de la baldosa mayor la puntuación obtenida. El objetivo del juego es, por un lado, alcanzar la baldosa de valor 2048, aunque también puede haber baldosas de valor más grande (4096, 8192, 16384…); y por otro lado obtener la puntuación más alta posible. El juego termina cuando no queda espacio libre ni baldosas iguales adyacentes, dado que no se pueden mover las baldosas.

Deep Q-Learning (DQN)

Para resolver el juego del 2048 se ha empleado un agente de aprendizaje por refuerzo profundo basado en Deep Q-Learning (DQN). Ahora, antes de describir la técnica DQN como tal se debe hablar sobre su base, el Q-Learning.

Q-Learning es una técnica de aprendizaje por refuerzo, y como tal hay dos elementos principales: un entorno (el tablero 4x4 del 2048 en este caso) y un agente que opera en ese entorno (el jugador).

El agente debe aprender a operar en el entorno probando diferentes acciones y aprendiendo de estas gracias al refuerzo que proporcionan, ya sea positivo (recompensa) o negativo (castigo), para así tomar la acción óptima en cada estado.

En Q-Learning se trabaja con Q-values, que se pueden entender como la calidad de tomar una acción dado un estado. Habría una tabla con todas las combinaciones de estados y acciones posibles y los Q-values correspondientes, como esta:

Estado/Acción	Q-value
Estado 1/Acción 1	0
Estado 1/Acción 2	1
Estado 2/Acción 1	-2
Estado n/Acción n	9

Tabla 1: Tabla de Q-values

Todos los Q-values se inicializan en 0, y la idea es actualizarlos entrenando al agente (realizar acciones) siguiendo una ecuación de Bellman:

$$Q^{nuevo}(S_t, A_t) = (1 - \alpha) * Q(S_t, A_t) + \alpha * (r + \gamma * max_a Q(S_{t+1}, a))$$

Donde:

- Q es el Q-value actual
- α es el ratio de aprendizaje: sirve para ajustar cuánto explorará el modelo para aprender nuevos caminos y cuánto explotará sus conocimientos ya adquiridos para maximizar la recompensa
- r es el refuerzo obtenido tras cada acción
- γ es el factor de descuento: limita el valor de estados futuros

Al principio el agente carece de conocimiento del entorno, por lo que deberá explorar la mayoría del tiempo. Sin embargo, conforme va entrenando empieza a explotar conocimientos adquiridos y maximizar la recompensa. Cuando el agente ya está muy entrenado se dedica mayoritariamente a explotar sus conocimientos, aunque siempre realiza un poco de exploración.

Uno de los problemas principales del Q-Learning es que el tamaño de la tabla de Q-values crece de manera cuadrática con el número de estados. Por ello en problemas con un espacio de estados muy alto no se puede aplicar esta técnica, sino que se deben aproximar los Q-values de otra manera, por ejemplo con una red neuronal.

Precisamente esa es la idea del aprendizaje por refuerzo profundo: una red neuronal que tiene un estado inicial como entrada, pasa por varias capas ocultas y genera los Q-values correspondientes a cada acción que puede tomarse desde dicho estado.

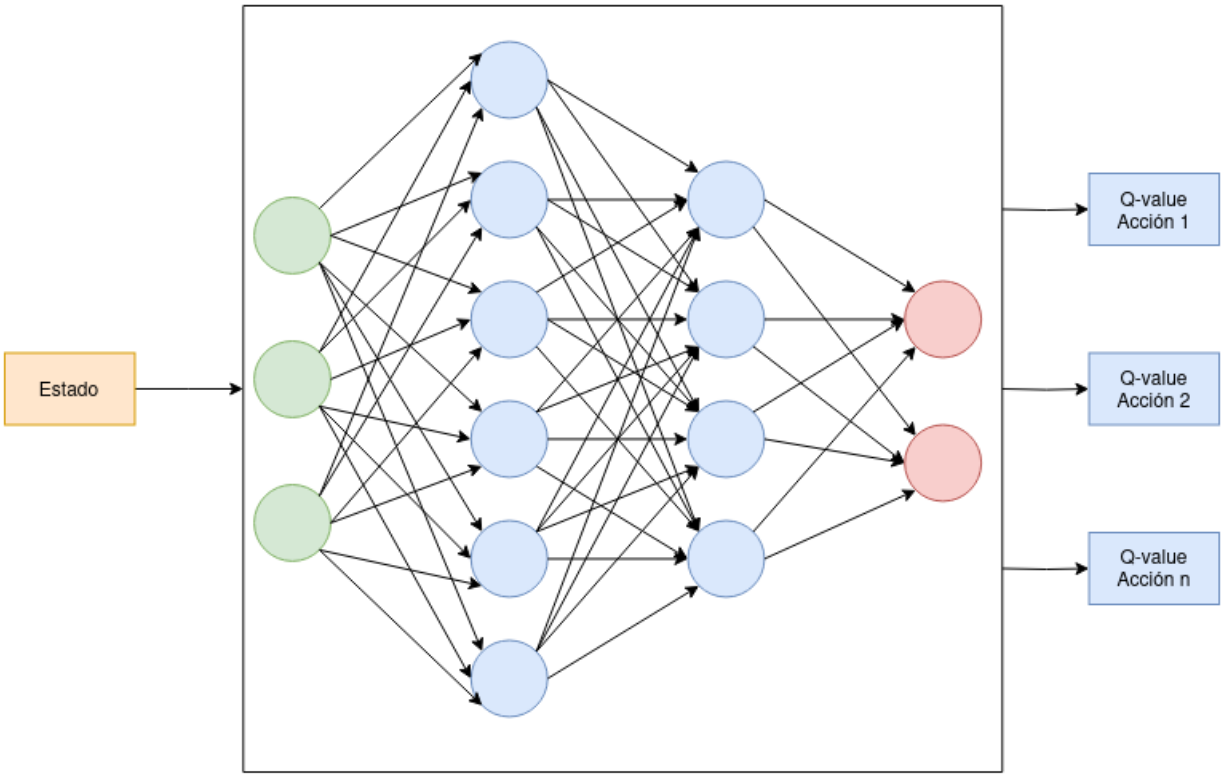

Figura 2: Deep Q-Learning

La ventaja de la técnica DQN es que, al ser una red neuronal, no requiere tanta memoria como la tabla de Q-values tradicional. Por ello se emplea esta técnica para resolver el juego del 2048.

Aplicación de DQN al juego 2048

Representación de estados

Se representa un estado como el tablero del 2048, es decir, una matriz 4x4 donde el valor de cada celda es una potencia de 2 (2^0 sería la celda vacía, 2^{11} sería 2048, etc.). El valor máximo de una baldosa es 2^{16}, por tanto hay 17 posibles valores para cada celda, y 16 celdas totales. De esta manera el espacio de estados sería $16^{17} \sim 10^{27}$. Dado que hay demasiados estados posibles, no se puede aplicar Q-Learning tradicional, y por ello se optó por Deep Q-Learning.

Red Neuronal Profunda

En primer lugar cabe mencionar la entrada de la red neuronal. No se ha empleado simplemente el estado actual, sino que se ha utilizado un tensor que contiene todos los estados posibles en los próximos dos pasos (el primer paso generará 4 estados y el segundo 16, en total 20 estados posibles). De esta manera se espera que la red neuronal logre aproximar de manera más precisa los Q-values.

Los estados de entrada se codifican con *one-hot encoding*, por lo que cada estado se compone de 16 matrices, porque $2^{16} = 65536$ sería la baldosa más grande que puede haber en el juego. Cada matriz representa con un 0 si la cuadrícula está vacía y con un 1 si hay una baldosa presente. Hay una matriz para cada posible valor de baldosa, por esto son 16 matrices en total. Las dimensiones de cada matriz son 4x4, ya que representan el tablero del 2048, y estas son sus dimensiones.

Por tanto, las dimensiones del tensor de entrada serían las siguientes:

- Número de estados posibles en los 2 siguientes pasos: $4 + 16 = 20$
- Número de matrices para la codificación *one-hot*: 16
- Dimensiones de cada tablero: $4 * 4 = 16$
- Entradas totales: $20 * 16 * 16 = 5120$

En segundo lugar cabe hablar de la arquitectura de la red neuronal. Esta red se compone de una capa inicial que aplana el tensor de entrada, dos capas profundas con activación ReLU: una primera capa oculta de 512 unidades lineales rectificadas y una segunda de 256 unidades; y finalmente una capa de salida con activación lineal de 4 neuronas, una para cada acción.

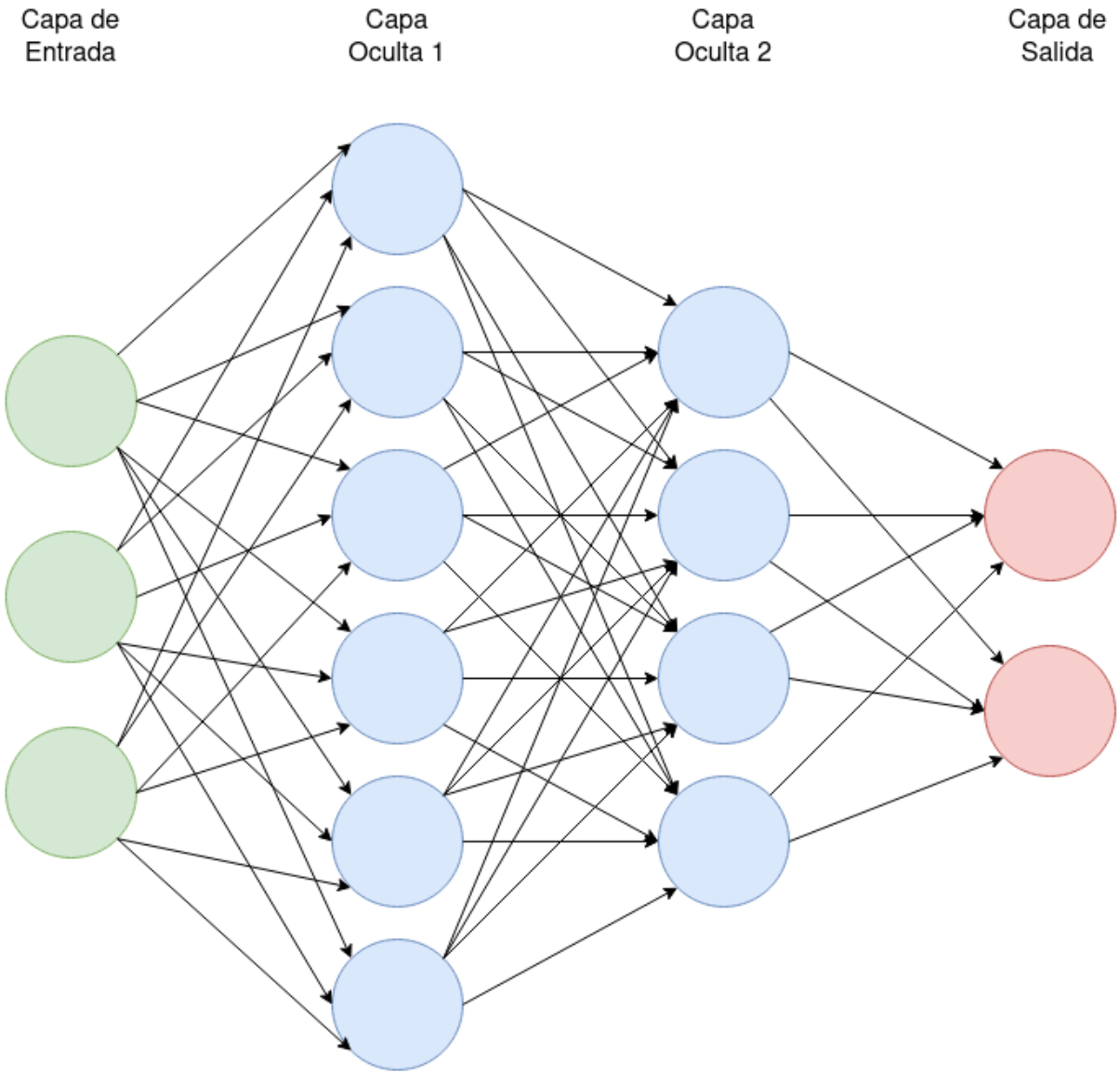

Figura 3: Arquitectura de la Red Neuronal Profunda

Agente DQN

Cabe destacar algunos aspectos del agente DQN empleado. Por un lado se ha utilizado un factor de descuento $\gamma = 0.99$, una política de entrenamiento ε-greedy con linear annealing para reducir la exploración a lo largo del entrenamiento, y una política ε-greedy para testing.

Estrategia de optimización

Para optimizar el modelo se ha empleado el algoritmo de optimización Adam con un ratio de aprendizaje de 0.00025 y la métrica a optimizar ha sido el error cuadrático medio (MSE).

Entrenamiento

El agente DQN descrito se ha entrenado durante más de 20.000 episodios, cada episodio correspondiendo a una partida completa del juego. En la **Figura 4** se puede observar la evolución del agente a lo largo de la fase de entrenamiento.

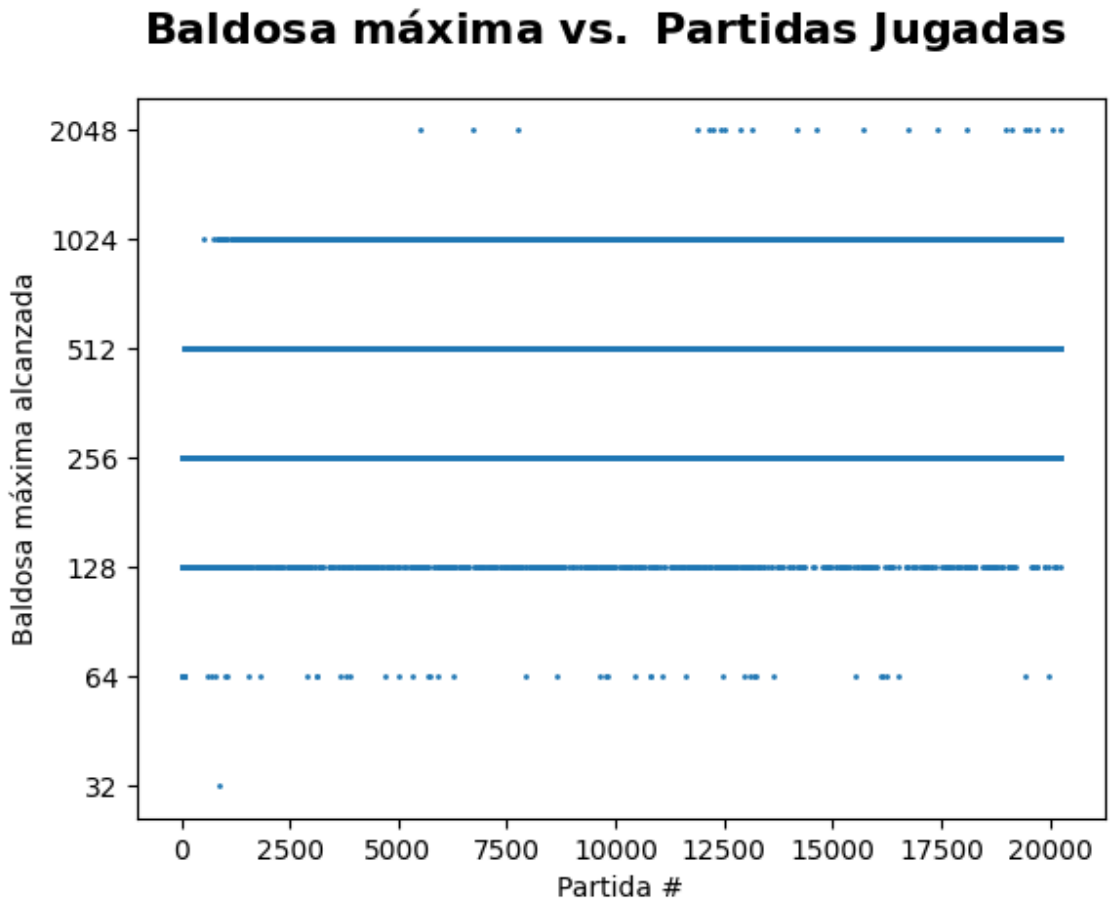

Figura 4: Baldosa máxima alcanzada a lo largo del entrenamiento

Se puede ver que en los primeros 1000 episodios empezó a alcanzar la baldosa 512, y a partir del episodio 2000 alcanzaba la baldosa 1024 regularmente. No obstante, hasta el episodio 6000 no logra alcanzar la baldosa 2048, y aún así solo la encuentra tres veces antes del episodio 12.000. A partir de entonces consiguió alcanzar la baldosa 2048 solo unas 20 veces.

Evaluación

Para evaluar el rendimiento del agente entrenado se ha llevado a cabo el siguiente experimento: 10.000 partidas jugadas por el agente DQN entrenado.

Entorno experimental

Cabe mencionar que para entrenar y evaluar el agente de aprendizaje por refuerzo profundo se ha empleado una máquina equipada con una CPU Intel Core i7 12700H de 14 cores (6P+8e), 32GB de RAM y una GPU Nvidia GeForce RTX 3070 de 8GB de VRAM. El modelo se entrenó con la GPU mediante el framework TensorFlow.

Análisis de resultados

En primer lugar cabe hablar de la baldosa máxima alcanzada en partida. En la **Figura 5** se puede observar que el agente alcanza la baldosa 1024 regularmente, aunque muchas veces se queda en 512 o 256. En la **Figura 6** se puede ver que, de hecho, la baldosa 512 es la más común, seguida de la 1024.

Además se observan algunos picos en los que el agente alcanza la baldosa 2048, pero esto no ocurre con mucha regularidad. De hecho, en la **Figura 6** se puede ver que la frecuencia con la que el agente alcanza la baldosa 2048 es similar a alcanzar la baldosa 64.

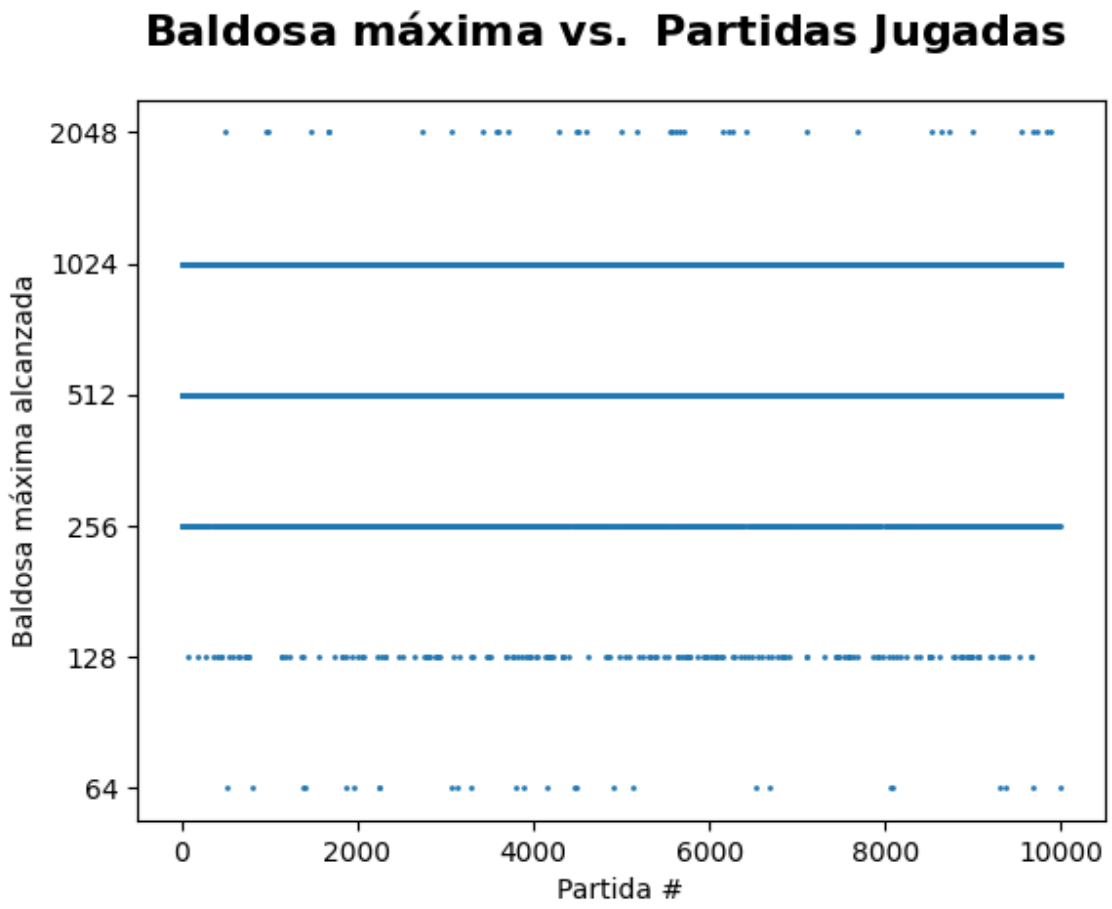

Figura 5: Baldosa máxima alcanzada en testing

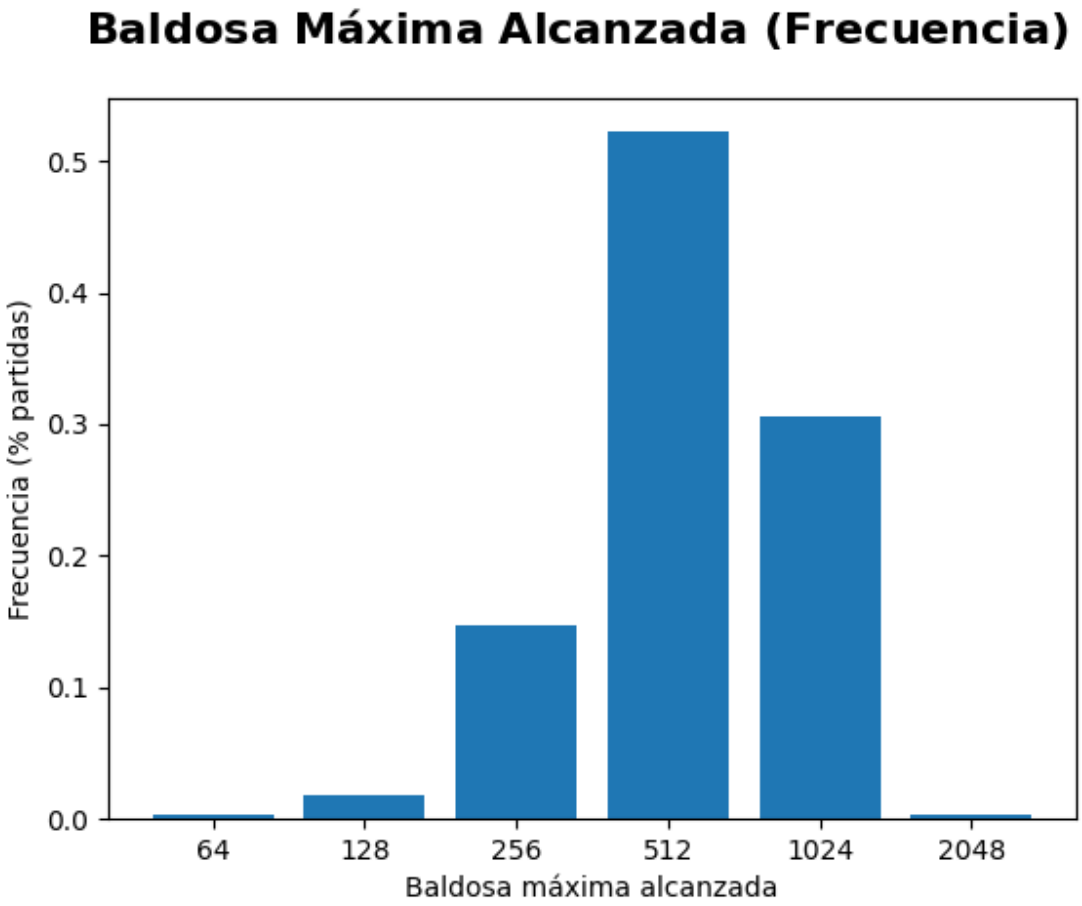

Figura 6: Frecuencia de baldosa máxima alcanzada en testing

Con estos resultados se puede inferir que el agente requiere más entrenamiento, puesto que no logra alcanzar la baldosa 2048 con regularidad ni tampoco alcanza baldosas superiores. Sin embargo, se podría decir que el agente juega al 2048 mejor que la mayoría de jugadores humanos, ya que alcanzar baldosas grandes como 1024 o 2048 es una tarea difícil.

Conclusiones

En este trabajo se ha podido ver que las técnicas de aprendizaje por refuerzo profundo como Deep Q-Learning son muy útiles para resolver juegos como el 2048: con una red neuronal sencilla se ha podido entrenar en poco tiempo un agente capaz de jugar al 2048 de manera competente, seguramente mejor que el jugador humano promedio.

Como conclusiones personales quiero mencionar que no había entrenado una red neuronal antes de realizar este trabajo, este modelo ha sido el primero para mí: no había pensado que entrenar un modelo como este era tan sencillo como programarlo con unas pocas líneas de código, dejarlo entrenando e ir viendo cómo mejoraba con el tiempo o cómo se quedaba atascado a veces antes de alcanzar un pico con una baldosa más alta.

Además, el hecho de no necesitar datos para entrenarlo me parece fascinante: el agente va probando acciones y aprende por prueba y error lo que funciona bien en el entorno y lo que no, aprende poco a poco que en un estado una acción puede ser mejor a largo plazo que otra que a priori parece más útil pero en realidad lleva a perder la partida antes.