Resolución de problemas de OpenAI Gym con algoritmos de *Deep Q-Learning* y Neuroevolutivos

Jaime Ferrando Huertas, Javier Martínez Bernia

Mayo 2021 @ MIARFID, UPV

Contents

1 Introducción

El Aprendizaje por Refuerzo es un campo dentro del Aprendizaje Automático en el que se pretende aprender el comportamiento de los agentes para maximizar una recompensa recibida tras tomar acciones. Los sistemas aprenden a lo largo del tiempo interactuando con el entorno mediante ensayo y error. Este problema se estudia en otras disciplinas, como por ejemplo la teoría de juegos o la investigación operativa. En este trabajo, se pretende desarrollar un sistema de Aprendizaje por Refuerzo utilizando dos técnicas: el *Deep Q-Learning* y una combinación de Algoritmos Evolutivos con Redes Neuronales (Neuroevolutivos). Se utilizarán estas técnicas para resolver distintos problemas de *OpenAI Gym*, una plataforma que nos proporcionará un entorno de simulación.

En esta memoria se presenta el proyecto realizado para la asignatura Herramientas y Aplicaciones de la Inteligencia Artificial. En las siguientes secciones se trata con detalle el problema resuelto en este proyecto.

2 *Deep Q-Learning*

La primera técnica que se va a aplicar para resolver los problemas es el *Deep Q-Learning*. Esta técnica es una técnica de Aprendizaje por Refuerzo en la que se pretende aprender una serie de normas de comportamiento que guíen a un agente a tomar ciertas decisiones, maximizando la recompensa total recibida por realizar ciertas acciones. La letra 'Q' del nombre viene de la función 'Q', la función encargada de calcular una estimación de la recompensa que se obtendría dado un estado del agente y una acción a realizar. En los siguientes subapartados se encuentra una breve descripción de los puntos clave de esta técnica.

2.1 *Q-Learning vs. Deep Q-Learning*

La principal diferencia entre el *Q-Learning* tradicional y el *Deep Q-Learning* es que en el tradicional, se intentaba aprender una especie de tabla con las probabilidades de tomar cada acción en cada estado del agente. Para ello se utilizaban distintos algoritmos, como algunos basados en programación dinámica o el Método de Montecarlo. Lo que ocurre con el *Deep Q-Learning* es que se sustituye esa tabla por una red neuronal que recibe como entrada un estado y tiene como salida las acciones que se pueden tomar. Los valores asociados a las salidas serán la estimacion de recompensa al tomar cada acción.

2.2 *Experience replay* y *Replay Memory*

Con las redes *Deep-Q-Learning* a menudo utilizamos una técnica llamada **Experience replay** durante el entrenamiento. Con Experience replay, almacenamos las experiencias del agente en cada paso en un conjunto de datos llamado **Replay Memory**. En un instante de tiempo t, la experiencia del agente se define como esta tupla:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1}) \tag{1}$$

Esta tupla contiene el estado del entorno, la acción tomada desde el estado, la recompensa otorgada al agente en ese momento como resultado del par estado-acción anterior y el siguiente estado del entorno. De hecho, esta tupla nos da un resumen de la experiencia del agente en ese momento.

Todas las experiencias del agente en cada instante de tiempo sobre todos los episodios reproducidos por el agente se almacenan en la Replay Memory. Normalmente se pone un límite al número de experiencias para almacenar y se actualiza cada cierto tiempo con nuevos datos.

Este conjunto de datos de Replay Memory es lo que muestrearemos al azar para entrenar la red. El acto de adquirir experiencia y tomar muestras de la Replay Memory que almacena estas experiencias es lo que se denomina Experience Replay.

2.3 Red neuronal Objetivo *vs* Red neuronal Predictiva

Cuando entrenamos un sistema de *Deep Q-Learning* intentamos minimizar el error cuadrático entre la salida que nos da la red y la salida esperada. La función objetivo sería la siguiente,

$$(r_{t+1} + \lambda \max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2 \tag{2}$$

donde r_{t+1} es la recompensa obtenida en el instante $t+1$, λ es un factor de atenuación, $\max_a Q(s_{t+1}, a)$ es la salida de la red dado el estado s_{t+1} y la acción a que nos maximice la recompensa según la red (el *target*), y $Q(s_t, a_t)$ es la salida de la red en el instante actual s_t tomando la acción a_t.

Como la función Q es la que nos proporciona la red, si intentamos modificar los pesos de la misma para que las salidas se parezcan a los *targets*, es posible que los *targets* también cambien, y estemos ante un problema que se conoce como *chasing a moving target*, es decir, estamos persiguiendo un objetivo que se mueve. Para que esto no ocurra, se utilizan dos redes, una para cada función Q de la ecuación del error cuadrático. De esta manera, tendríamos la red objetivo y la red predictiva (*target network* y *prediction network*), por lo que al tocar los pesos de la red predictiva no modificaremos los de la red objetivo y las salidas esperadas serán fijas. Cada cierto tiempo, se actualizan los pesos de la red objetivo con los nuevos pesos de la red predictiva. En la siguiente figura podemos ver un esquema de cómo estas dos redes se utilizan en el cálculo de la función objetivo.

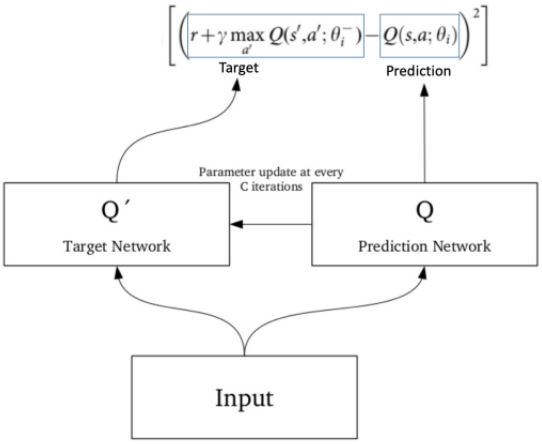

Figure 1: Esquema de funcionamiento de un sistema de *Deep Q-Learning* con dos redes neuronales para evitar el problema de perseguir un objetivo que se mueve. Cada red aporta a una parte de la función objetivo, y cada cierto tiempo se actualizan los pesos de la red objetivo con los calculados en la red predictiva.

3 Algoritmos Neuroevolutivos

Un algoritmo genético es una técnica metaheurística inspirada en el proceso de selección natural. Los algoritmos genéticos se utilizan comúnmente para generar soluciones de alta calidad para problemas de optimización y búsqueda basándose en operadores inspirados biológicamente como mutación, cruce y selección.

En este apartado exploraremos un caso de algoritmos genéticos o evolutivos aplicado a las redes neuronales. Concretamente, estas técnicas reciben el nombre de Neuroevolución o Algoritmos Neuroevolutivos.

3.1 NEAT

NEAT (*NeuroEvolution of Augmenting Topologies* es un algoritmo evolutivo que crea redes neuronales. Para ello, va modificando tanto los pesos como la estructura, añadiendo o quitando neuronas y/o conexiones entre las mismas. Se tiene una población con distintas especies de redes y se va evolucionando intentando maximizar la función *fitness*.

Este algoritmo se presenta por primera vez en [6], y se puede encontrar implementado en diferentes librerías. Para este proyecto, se hará uso de NEAT-Python [2], una librería popular que implementa NEAT y con la cual hemos experimentado en la última sesión de la asignatura.

3.2 Codificación de la red

Para utilizar la red como individuo en un algoritmo evolutivo necesitamos codificar la misma en un genoma. Cada genoma será una lista de genes conectados, donde cada uno indica la conexión entre dos nodos. En cada posición de la lista hay almacenados cinco valores: el nodo de entrada, el nodo de salida, el peso de la conexión, un número de innovación y un bit que expresa si el enlace está activo o no.

3.3 Operadores genéticos

Respecto a los operadores genéticos, en cuanto a la parte estructural (sin tener en cuenta la modificación de pesos), tendríamos la operación de cruzamiento y la mutación:

- Cruzamiento estructural: Se hace una mezcla de dos redes. La parte común entre ambas sigue igual, y se hace una combinación de los nodos de la parte disjunta de los genes.
- Mutación estructural: Se añaden o eliminan (desactivan) nodos y/o conexiones entre ellos.

Esto sería un ejemplo de cómo se harían los cruces y mutaciones cuando se utiliza una estructura de red basada en un grafo. Para hacer las operaciones, existen distintos tipos y maneras.

4 OpenAI Gym

OpenAI Gym es una plataforma que proporciona un entorno de simulación donde se pueden resolver distintos problemas de Aprendizaje por Refuerzo. Los problemas escogidos de esta web para la evaluación de estas técnicas han sido Cartpole-V1 [4] LunarLander-V2 [5] y Breakout-V1 [3]. OpenAI proporciona código Python para interactuar con el problema, permitiéndonos generar tantas instancias del mismo como sea necesario y ejecutar simulaciones. Se hará uso de redes neuronales basadas en *Deep Q-Learning* o NEAT para resolver estos problemas.

4.1 Cartpole

Este problema consiste en darle mover una plataforma (carro) horizontalmente para que el palo que se encuentra encima de ella se mantenga equilibrado y no caiga a ninguno de los lados. El sistema tiene como entrada la posición de la plataforma, la velocidad de la misma, el ángulo del palo y la velocidad angular del palo. Como acciones posibles tiene dos: moverse hacia la derecha o moverse hacia la izquierda.

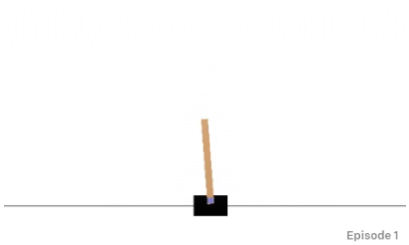

Figure 2: Problema Cartpole-v1

Se considera resuelto si consigues una puntuación de 500 en la simulación.

4.2 Lunar Lander

Para el segundo problema se ha querido aumentar la dificultad afrontando una tarea un poco más compleja. Este problema consiste en aterrizar una nave en una plataforma. Esta nave tiene tres motores, uno central y dos laterales, con los que tiene que ayudarse para aterrizar correctamente en una plataforma. El sistema recibe como entrada 8 valores, correspondientes a coordenadas, velocidad, ángulo y sensores de las patas de la nave. Existen cuatro acciones posibles: no hacer nada, encender el motor derecho, encender el motor izquierdo y encender el motor central.

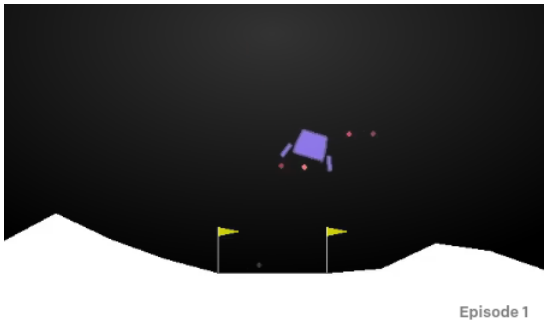

Figure 3: Problema LunarLander-v1

Este problema se considera resuelto si se consigue una score de mas de 200 puntos.

4.3 Atari Breakout

Este problema emula el famoso juego *Breakout* de la consola retro Atari. En este juego se debe mover una plataforma para hacer rebotar una bola que se

mueve por la pantalla con el objetivo de hacer que la bola golpee a todos los bloques y los remueva. El problema tiene dos variantes: utilizando los bytes de la memoria RAM en cada instante como entrada al sistema, y utilizando una imagen RGB de los píxeles de la pantalla (210, 160 px) como entrada.

Figure 4: Problema Breakout-v1

Este problema no

5 Experimentación con *Deep Q-Learning*

Para nuestra primera técnica hemos realizado implementaciones de Deep Q-Learning para resolver los tres problemas. Se han usado distintas implementaciones de DQN para cada problema variando el tamaño de entrada y la arquitectura para la entrada de cada problema (imagen o vector 1D).

5.1 Cartpole

Empezamos con el problema Cartpole-V1 porque era el más sencillo y el código de OpenAI permitía acceder directamente al ángulo en el que se encontraba el palo. Esta posición del palo nos viene codificada en un vector de una dimensión y 4 valores que serán la entrada a nuestra red. Se ha utilizado una red *fully connected* con tres capas ocultas. Los resultados de la experimentación han sido positivos y se ha conseguido entrenar una red capaz de conseguir alrededor de 200 puntos durante cien episodios seguidos a partir del episodio 850. A continuación podemos ver la puntuación obtenida en cada episodio de entrenamiento.

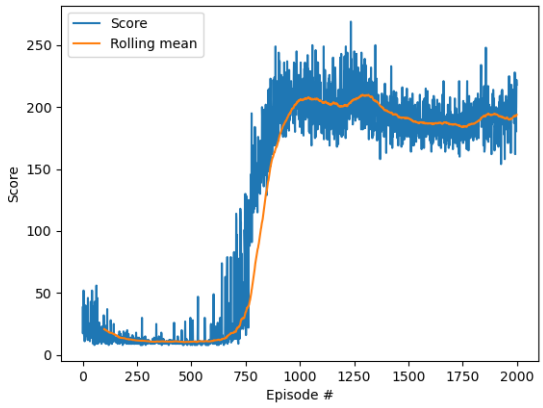

Figure 5: Puntuaciones para el problema CartPole

La red no ha sido capaz de llegar a los 500 puntos de puntuación máxima del problema, pero se mantiene en unos 200 puntos estables.

5.2 LunarLander

En el segundo problema LunarLander hemos podido usar la red utilizada en CartPole con pocas modificaciones (cambiar tamaños de entrada y salida) y hemos conseguido una puntuación media de 200 (puntuación necesaria para ser considerado resuelto) durante 100 episodios. Al conseguir esta puntuación paramos el entrenamiento y consideramos el problema resuelto. A continuación podemos ver la puntuación obtenida en cada episodio de entrenamiento.

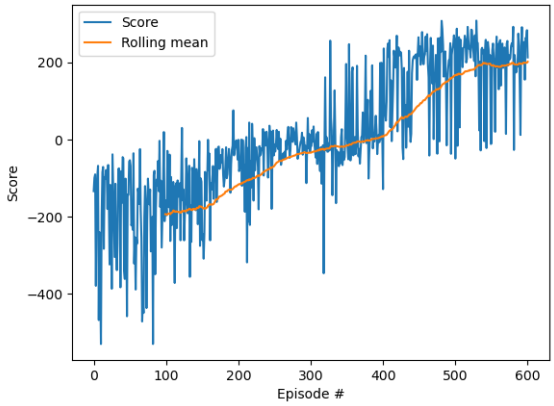

Figure 6: Puntuaciones para el problema LunarLander

La red ha sido capaz de llegar a los 200 puntos necesarios para considerar el problema resuelto.

5.3 Breakout

Para el problema de Breakout-V1 se han hecho modificaciones a nuestra implementación. El problema ya no proporciona un vector que representa el estado del problema como el vector 1D de Cartpole o LunarLander. En vez de eso solo proporciona la imagen generada en cada paso. Se ha modificado la red con capas convolucionales para aprender mejor a partir de la imagen y poder generar predicciones en el siguiente paso del agente. En la siguiente figura podemos ver la evolución de las simulaciones.

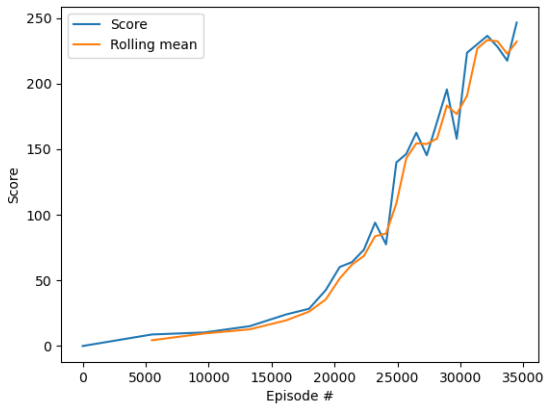

Figure 7: Puntuaciones para el problema Breakout

No hemos encontrado una puntuación necesaria para que este problema sea considerado resuelto, pero hemos conseguido llegar una media de 240 puntos.

A continuación, podemos ver una tabla con los mejores resultados obtenidos para cada problema contra la puntuación necesaria para ser considerado resuelto.

Problema	Puntuación conseguida	Puntuación necesaria
CartPole-v1	270	500
LunarLander-v1	263	200
Breakout-v1	246	-

6 Experimentación con Algoritmos Neuroevolutivos

Para esta segunda técnica, se ha implementado un script en Python para lanzar los distintos problemas. En cada uno, simplemente hay que modificar el nombre del entorno del Gym que se quiere cargar y modificar el número de neuronas de entrada y de salida en el fichero de configuración. A partir de ahí, ya se pueden ajustar los parámetros de la configuración para resolver el problema. Se han añadido argumentos para ajustar el experimento desde la línea de comandos, como por ejemplo indicar el número de pasos de simulación máximos por individuo, el número de generaciones, indicar si se quiere renderizar para visualizar la simulación, indicar un número de núcleos para trabajar en paralelo, etc.

6.1 Cartpole

En primer lugar, se ha lanzado un experimento con el problema Cartpole, y se ha conseguido resolverlo de forma sencilla. En la siguiente figura podemos ver la evolución de la función *fitness* a lo largo de 10 generaciones de simulación.

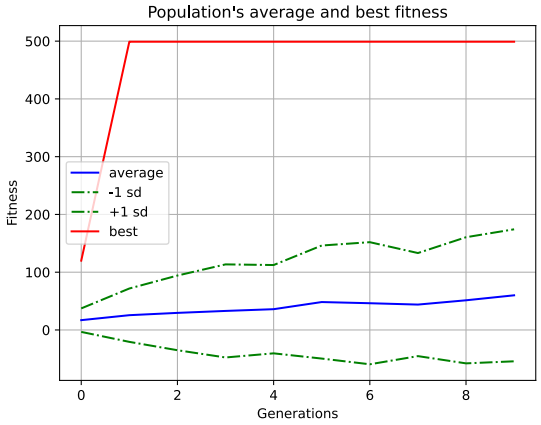

Figure 8: Evolución de la función *fitness* en la simulación de Cartpole.

Como podemos observar en la figura anterior, el promedio de la función *fitness* se mantiene más o menos estable conforme avanzan las generaciones, aunque se puede ver que el mejor *fitness* consigue el máximo (500) en la segunda generación. Respecto a las especies, se han creado varias especies durante la simulación, aunque la especie predominante y que se ha mantenido hasta el final ha sido la inicial. En la siguiente figura podemos ver la especiación durante la simulación:

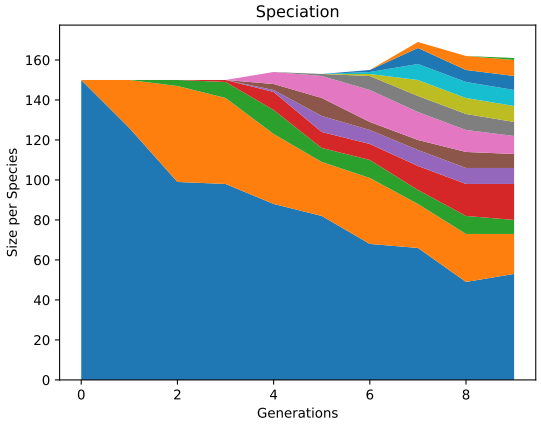

Figure 9: Especiación en la simulación Cartpole.

Finalmente, respecto a la red neuronal, en la siguiente figura podemos ver un esquema de la red que ha resuelto el problema:

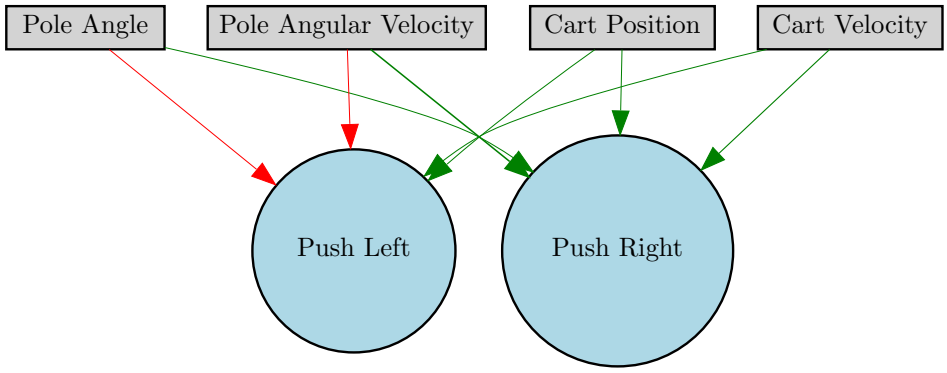

Figure 10: Red neuronal resultante en la simulación Cartpole.

Como podemos ver, se ha creado una red que asocia pesos negativos a la acción de mover a izquierda para las entradas del ángulo y la velocidad angular del palo, mientras que asocia pesos positivos para la otra acción. Respecto a la posición y velocidad de la plataforma (cart), se puede ver que influyen positivamente en ambas acciones.

6.2 Lunar Lander

Para el segundo problema, se ha adaptado la entrada y salida del fichero de configuración, y se han cambiado algunos parámetros:

- Probabilidad de añadir una conexión: $0.5 \rightarrow 0.95$
- Probabilidad de eliminar una conexión: $0.5 \rightarrow 0.15$
- Probabilidad de añadir un nodo: $0.2 \rightarrow 0.25$
- Probabilidad de eliminar un nodo: $0.2 \rightarrow 0.10$

Esto se ha hecho para que la red se haga más grande de manera más rápida, ya que al ser un problema más complejo se necesita una red con más conexiones y nodos.

Tras varias pruebas, se ha logrado una puntuación máxima de 120.36 tras 95 generaciones. En la siguiente figura podemos ver la evolución de la función *fitness*.

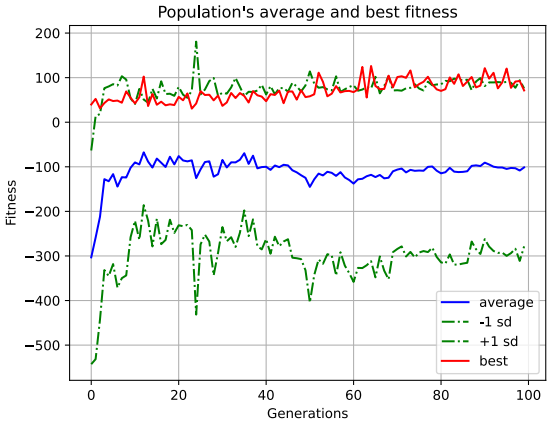

Figure 11: Evolución de la función *fitness* en la simulación de Cartpole.

Adicionalmente, se ha lanzado otro experimento haciendo unas cuantas generaciones más partiendo de la mejor puntuación, y se ha llegado a 160.08 con una red con 10 nodos y 46 conexiones.

6.3 Breakout

En el caso del tercer problema, se ha intentado abordar mediante el algoritmo NEAT, pero no se ha podido ejecutar debido al gran tamaño de la entrada. Tanto usando la imagen (haciendo una transformación a escala de grises y aplanando el vector) como la memoria RAM en la entrada de las redes, el tamaño era demasiado grande para poder ejecutar el algoritmo y no se disponía de suficiente memoria.

7 Comparativa de técnicas

En este apartado se comparan ambas técnicas utilizadas en este proyecto desde distintos puntos de vista.

En primer lugar, respecto a la implementación de modelos, en el caso de los Algoritmos Neuroevolutivos ha sido bastante sencilla y rápida, ya que no cambiaba demasiado respecto a los ejemplos vistos en la última sesión teórica de la asignatura. En el caso de la aproximación por *Deep Q-Learning* ha sido un poco más costoso de implementar ya que hay algunos conceptos como las dos redes o la memoria (Experience Replay) que nunca habíamos implementado. Una vez la implementación base estaba hecha, lanzar distintos problemas era tan sencillo como cambiar el nombre del entorno de Gym que se quería usar y unos pocos parámetros.

En segundo lugar, respecto a la potencia para resolver los problemas, hemos visto que la técnica de *Deep Q-Learning* ha conseguido mejor resultado en problemas más grandes, mientras que el algoritmo de NEAT no ha llegado a los máximos y por tanto a resolver el problema de Lunar Lander y el de Breakout. Esto es debido a que los experimentos requerían muchas generaciones, y el tiempo de cómputo que requiere conforme crece la población y se van creando nuevas especies es cada vez mayor. A pesar de ello, la solución basada en NEAT nos funciona mejor en el problema del Cartpole, ya que consigue una red muy sencilla, pero que funciona perfectamente.

Respecto al tiempo de ejecución, no se pueden comparar ambos algoritmos con el número de pasos, ya que son algoritmos distintos, pero si que se ha visto que cuando el problema es mayor, los algoritmos de *Deep Q-Learning* nos han dado un mejor rendimiento en cuanto a tiempo requerido frente a calidad de la solución.

8 Conclusiones y futuros trabajos

En este proyecto se han visto dos técnicas del campo de la Inteligencia Artificial, que están dentro del campo del Aprendizaje por Refuerzo. Se han utilizado ambas técnicas para resolver tres problemas disponibles en el Gym de OpenAI. Tras haber aplicado ambas técnicas, la principal conclusión a la que hemos llegado es que los Algoritmos Neuroevolutivos se comportan muy bien en problemas sencillos, pero cuando crece el tamaño de la red y del problema el tiempo requerido de ejecución para conseguir un resultado bueno es muy alto en comparación a *Deep Q-Learning*. Se ha explorado la opción de utilizar una mejora del algorimo NEAT conocida como HyperNEAT, en la cual se utiliza una malla (o matriz) en la entrada de la red, y un módulo CPPN (Compositional Pattern Producing Network) es el encargado de extraer las relaciones entre los puntos cercanos de la malla. Se ha intentado probar con el problema de Breakout, pero tampoco

se ha podido ejecutar debido a la memoria requerida.

Por todo esto, vemos con más potencia a los algoritmos de *Deep Q-Learning*, ya que con suficiente memoria para el Experience Replay y con redes más complejas creemos que se pueden resolver problemas más complejos y grandes.

Como futuros trabajos, estarían hacer experimentos con otros entornos de Gym, ya que hay una gran variedad de problemas creados por la comunidad y de uso libre. También experimentar con una computadora con bastantes núcleos de CPU para lanzar más experimentos con el Lunar Lander, ya que la ejecución era lenta llegado a cierto número de generaciones. Adicionalmente, probar a resolver problemas de Atari con HyperNEAT, basándonos en el artículo "A Neuroevolution Approach to General Atari Game Playing" [1].

Finalmente, este proyecto ha resultado bastante enriquecedor, ya que se han probado dos técnicas de Inteligencia Artificial para resolver problemas de Aprendizaje por Refuerzo. Hemos aprendido bastante sobre Aprendizaje por Refuerzo y sobre estas dos técnicas, las cuales nunca habíamos utilizado. Nos han resultado muy interesantes ambas aproximaciones y nos han dado una base para afrontar problemas similares en el futuro.

References

[1] Matthew Hausknecht, Joel Lehman, Risto Miikkulainen, and Peter Stone. A neuroevolution approach to general atari game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6:355–366, 12 2014.

[2] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. neat-python. https://github.com/CodeReclaimers/neat-python.

[3] OpenAI. Breakout: Maximize score in the game breakout.

[4] OpenAI. Cartpole: Balance a pole on a cart.

[5] OpenAI. Lunar lander: Navigate a lander to its landing pad.

[6] Kenneth O. Stanley and Risto Miikkulainen. Efficient evolution of neural network topologies. In William B. Langdon, Erick Cantu-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant Honavar, G"unter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, and Alan C. Schultz, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1757–1762, Piscataway, NJ. San Francisco, CA: Morgan Kaufmann.