UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación de un algoritmo mediante aprendizaje por refuerzo para la resolución de un juego de plataformas (Ghosts n' goblins) y extrapolación del algoritmo a juegos similares

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Daniel Medina Vázquez

Tutor: Carlos David Martínez Hinarejos

2018-2019

Resumen

El objetivo de este TFG es desarrollar un algoritmo mediante aprendizaje por refuerzo, utilizando la biblioteca para Python `gym-retro` de OpenAI. En este caso el algoritmo utilizado es el de Deep Q-Learning, que es el más adecuado de aprendizaje por refuerzo para las características del juego en cuestión. La utilización de aprendizaje por refuerzo ha resultado ser prometedora en la resolución de juegos sencillos, por lo que es interesante aplicarlo a un juego de mayor complejidad como es el de 'Ghost n' goblins'. Además, este TFG se introduce en un problema abierto actualmente como es la extrapolación de algoritmos entrenados en un juego para resolver otro distinto de características similares.

Palabras clave: aprendizaje por refuerzo, videojuego, openAI, redes neuronales, Tensorflow

Abstract

This thesis' main objective is to implement a reinforcement learning algorithm, using OpenAI's `gym-retro` library for Python. In this case, the chosen algorithm is deep Q-Learning, which is the most suitable for the characteristics of the game. The usage of reinforcement learning has been proven to be successful for solving simple games, which is why it is interesting to apply it to more complex games such as 'Ghosts n' goblins'. Furthermore, this thesis explores a still open problem that is extrapolating algorithms trained on one game to solve another of similar general characteristics.

Keywords: reinforcement learning, videogame, openAI, neural network, Tensorflow

Resum

L'objectiu d'aquest TFG és desenvolupar un algorisme mitjançant aprenentatge per reforç, emprant la biblioteca per a Python `gym-retro` d'OpenAI. En aquest cas l'algoritme emprat és el de Deep Q-Learning, que és el més adequat per a l'aprenentatge per reforç per a les característiques del joc considerat. L'emprament de l'aprenentatge per reforç ha resultat prometedora a la resolució de jocs més senzills, per la qual cosa és interessant aplicar-ho a un joc de major complexitat com és el de 'Ghost n' goblins'. A més, aquest TFG s'introdueix en un problema obert actualmente com és l'extrapolació d'algorismes entrenats en un joc per a resoldre un altre distint de característiques similars.

Paraules clau: aprenentatge per reforç, videojoc, openAI, xarxes neuronals, Tensorflow

Tabla de contenidos

1. Introducción

El cerebro es el órgano más increíble del cuerpo humano. Dicta la manera en la que percibimos cada vista, sonido, olor, tacto y gusto. Nos permite almacenar recuerdos, experimentar emociones, e incluso soñar. Sin él, seríamos organismos primitivos, incapaces de nada salvo los más simples de los reflejos. El cerebro es, lo que nos hace inteligentes.

El cerebro de un niño pesa apenas medio kilo, pero de alguna manera resuelve problemas que nuestros más potentes superordenadores encuentran imposibles. En apenas unos meses desde el nacimiento, los niños pueden reconocer las caras de sus padres, distinguir objetos concretos, e incluso diferenciar voces. En menos de un año, ya han desarrollado una intuición para la física natural, pueden seguir objetos aún cuando están parcial o totalmente bloqueados y pueden asociar sonidos con significados concretos. Y para su infancia temprana, ya tienen un entendimiento sofisticado de la gramática y miles de palabras en su vocabulario. [1].

No resulta raro teniendo esta información en cuenta que se pretenda intentar conseguir replicar la capacidad del cerebro humano para determinadas tareas que resultan imposibles de otro modo para un ordenador. La capacidad de asociación y de reconocimiento de patrones, y la intuición sobre cómo actuar ante ellos es, sin duda, una capacidad envidiable para cualquier ente que quiera actuar de manera autónoma, es decir, es imprescindible este reconocimiento de patrones para la toma de decisiones.

La informática se dirige pues, de manera inexorable, hacia la resolución de un número cada vez mayor de problemas usando redes neuronales, debido a su creciente capacidad de emular el cerebro humano.

De manera breve, un modelo de red neuronal (también llamada red conexionista) consiste en una serie de unidades computacionales (también llamadas células) y un conjunto de conexiones unidireccionales que unen estas unidades. En ciertos momentos, una unidad examina sus *inputs* y computa un número real llamado activación como su *output*. La nueva activación pasa por sus conexiones hacia otras unidades. Cada conexión posee un número real, llamado peso, que determina si una activación que viaja a través suya influye a la célula receptora para producir una activación similar o una activación diferente según el signo (+ o -) del peso. El tamaño del peso determina la magnitud de la influencia de la activación de una célula emisora sobre una célula receptora; en consecuencia, un peso negativo o positivo grande proporciona al emisor un efecto mayor sobre el receptor que un emisor con menor peso. [2].

En el presente trabajo se pretende conseguir entrenar una red neuronal que, mediante los algoritmos y topología elegidos, aprenda patrones de comportamiento para lograr superar el juego elegido para el entrenamiento. Una vez entrenada esta red neuronal, se intentará aplicar esos conocimientos en un juego similar, es decir, se va a intentar extrapolar lo que ha aprendido en un juego a otro.

Como es bien sabido, las redes neuronales tienen gran cantidad de aplicaciones (de reconocimiento de formas, predictivas, etc. [3],[4]). Para todas estas aplicaciones es esencial que la red neuronal reconozca patrones.

Esta primera parte del reconocimiento de patrones la lograremos con las capas convolucionales de nuestra red neuronal[5], lo que descrito de una forma sencilla significa trocear la imagen en una suerte de cuadrícula y aplicar las operaciones matemáticas pertinentes para que la red pueda determinar lo que está viendo.

Una vez pasadas estas capas convolucionales [5] le siguen las capas completamente conectadas, cuya función es la de encontrar los patrones que proceden de realizar una determinada acción para una pantalla del juego. Es aquí donde el presente trabajo se encuentra con su mayor dificultad: conseguir que la red sepa qué acciones debe tomar en cada momento para cumplir los objetivos del juego. Para ello nos ayudamos de la función de recompensa, que mide lo bien que la red ha actuado ante una determinada situación. Somos nosotros los que, con ayuda de nuestros conocimientos previos del juego, asignamos cierto valor a determinados movimientos; por ejemplo, para un juego de lucha es recomendable que esquive los golpes enemigos, por lo que en nuestra función recompensaríamos de forma negativa el ser golpeado por el enemigo.

La función de recompensa toma dos roles simultáneos en el aprendizaje por refuerzo. El primer rol es el de la evaluación en tanto que la función de recompensa especificadora de tareas es usada por el agente diseñador para evaluar el comportamiento del agente. La segunda es la de guía en tanto que la función de recompensa es también usada por el algoritmo de aprendizaje por refuerzo usado por el agente para determinar su comportamiento. [6]

Es relevante destacar que, al ser el objetivo de la red neuronal maximizar la recompensa obtenida de cada acción, no importa si nuestra función de recompensa penaliza por los fallos, premia los aciertos, o una mezcla de ambos, pues la red neuronal acabará convergiendo en las acciones que le generen una mayor recompensa, aunque esta recompensa esté por debajo de cero.

Motivación

Este trabajo está motivado por una parte de manera personal, en tanto que uno de los primeros juegos altamente demandantes a los que jugué fue la adaptación del Ghosts and Goblins, el llamado Ultimate Ghosts and Goblins para la PSP. Cuando lo jugué por primera vez pasé incontables horas en las primeras fases del juego, que aun siendo las más fáciles del juego, no dejan pasar ni un fallo.

Incluso así, esta versión del juego es más fácil que el juego que nos ocupa, pues en esta reedición el progreso no se pierde cuando las vidas del jugador llegan a cero, por lo que quería poner a prueba mi capacidad para implementar un modelo de red neuronal capaz de llegar todo lo lejos posible en este videojuego.

Finalmente, está motivado de manera profesional, ya que realizar un trabajo de estas características puede ser el primer pilar para una carrera profesional basada en la inteligencia artificial. Como ya hemos comentado anteriormente, la inteligencia artificial es un campo en expansión, y este trabajo puede ser una carta de presentación excelente para un puesto como desarrollador de inteligencia artificial para videojuegos, que es el objetivo que aspiro a conseguir algún día.

Objetivos

Con este trabajo se pretende ahondar en las técnicas de inteligencia artificial necesarias para que las redes neuronales puedan desenvolverse de manera fluida en el entorno de un videojuego en concreto, e intentar que apliquen esa experiencia obtenida en otros videojuegos. Se plantean los siguientes objetivos prioritarios:

- Investigación de los algoritmos necesarios
- Selección de un algoritmo y una topología para la red neuronal
- Preparación del programa para que pueda interactuar con el videojuego
- Entrenamiento de la red neuronal
- Extrapolación de los valores aprendidos al segundo videojuego

Impacto esperado

El valor de este trabajo no radica únicamente en jugar a videojuegos *per se*, sino que pretende investigar cómo se puede utilizar la información obtenida en un entorno para actuar en otro de características similares, dado que los videojuegos son un entorno idóneo para el aprendizaje de las redes neuronales. Realizar este cambio de entorno de un videojuego a otro puede ser un precursor de un futuro cambio de entorno donde estos conocimientos aprendidos en los videojuegos puedan ser utilizados en la vida real. Pudiera darse el caso entonces de que usando procedimientos similares a los realizados en el presente trabajo se entrenen redes neuronales para, por ejemplo, la conducción automática en videojuegos y simuladores, y que más tarde esta información sirva para la conducción automática en la vida real sin necesitar una fase de entrenamiento que potencialmente pueda poner en peligro al resto de los usuarios de las vías públicas.

Espero pues seguir una serie de pasos para la resolución de un problema mediante aprendizaje por refuerzo de manera que sean replicables para otros ámbitos más "serios", sirviendo así este proyecto de base para lograr un aprendizaje sobre un entorno seguro de manera que ese aprendizaje pueda más tarde utilizarse en entornos en los que empezar un aprendizaje partiendo desde cero pueda ser arriesgado.

2. Estado del arte

En cuanto al estado del arte de la inteligencia artificial en videojuegos, el logro más mediático es sin duda el logrado por el equipo de OpenAI Five (desarrolladores de la librería `gym-retro` con la que trabajaremos más tarde), quienes han conseguido que un equipo formado por cinco 'bots' haya derrotado en el videojuego "Dota 2" a los entonces campeones del mundo de la escena competitiva de dicho juego.

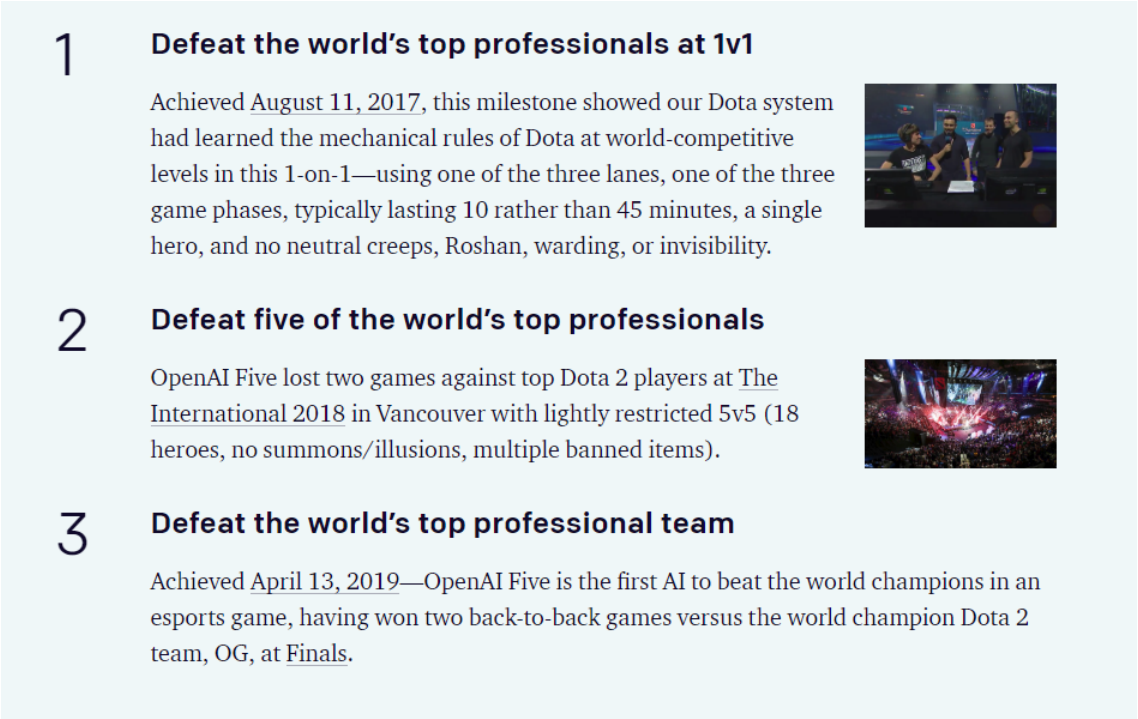

Figura 1 Logros principales del equipo de OpenAI Five [7]

Como podemos ver en la figura 1 este es un logro que se ha conseguido tan solo recientemente, y que lleva detrás casi 4 años de desarrollo, en estos años de manera constante se han ido haciendo avances, como podemos ver tanto en la figura 1 como en la figura 2. Es particularmente interesante ver que no solo se han centrado en ganar a jugadores cada vez más competentes, sino que cuanto más entrena la IA más mecánicas del juego han ido aprovechando:

- En primer lugar se usaron con partidas en los que los personajes estaban predefinidos [en este videojuego tenemos una serie de personajes (llamados campeones) que están disponibles para utilizar en las partidas, cada uno de estos campeones tiene unas habilidades, fortalezas y debilidades distintas, lo que implica que para cada uno de ellos el estilo de juego será ampliamente distinto del resto de campeones]

- Se continuó con partidas que elegían de manera aleatoria los personajes

- Por último, se tomaron partidas que elegían los personajes en lo que se conoce como un 'draft', es decir, cada equipo bloquea cierto número de personajes, haciendo que no puedan ser seleccionados por ninguno de los dos equipos, y más tarde se van eligiendo de manera alternativa personajes; en esta selección de campeones entran en juego los campeones con los que un determinado jugador se siente cómodo jugando, la posición que estos ocupan dentro del equipo, y los campeones que tienen ventaja enfrentándose a los que el equipo rival ha seleccionado

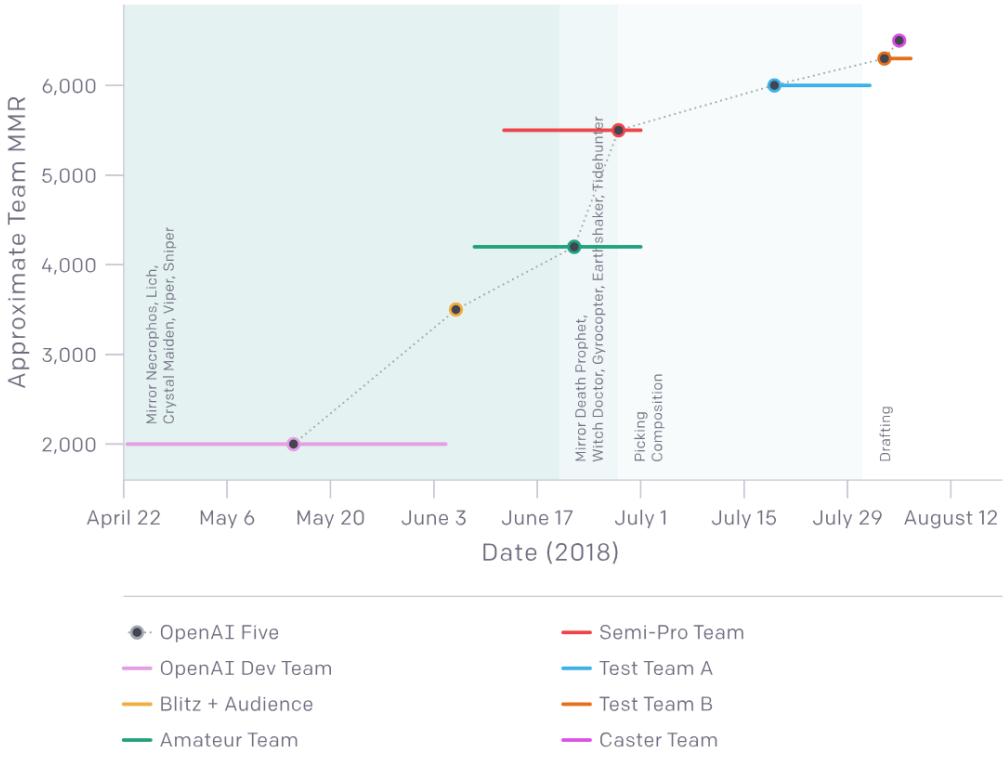

Figura 2 Progreso de las actividades de OpenAI Five [7]

Juntar todas estas destrezas es, sin duda, un logro considerable, pero lo más impresionante todavía es que hayan conseguido sobreponerse a la incertidumbre del juego, pues en "Dota 2" cada equipo tiene información únicamente sobre lo que los personajes de su equipo están viendo en ese momento; el resto del mapa está oculto por lo que se llama la 'niebla de guerra', que hace que los 'bots' de OpenAI tengan que predecir los movimientos de los rivales cuando no tienen información acerca de dónde se encuentran ni de lo que están haciendo para poder responder a sus movimientos, ya que de lo contrario la derrota estará asegurada.

3. Los juegos

La biblioteca `gym-retro` [6] pone a nuestro alcance integraciones con cientos de juegos de aproximadamente una decena de consolas distintas. De entre estos juegos se ha optado por dos en concreto cuyas características particulares explicaremos a continuación; estos dos juegos corresponden al género de los juegos de plataformas de acción, los cuales tienen una sencillez relativa si los comparamos con juegos de rol en los que se toma gran cantidad de pequeñas decisiones que afectan a la partida, y gran parte del tiempo se pasa en menús eligiendo los objetos y el equipamiento correcto.

Sin embargo, comparados con estilos de juego conceptualmente más sencillos como los juegos de carreras, los de lucha o los juegos arcade, que tienen muy poca variabilidad de acciones a tomar, resultan bastante más complejos.

En los juegos de plataformas tenemos que resolver los puzles que nos plantea el terreno, evitar recibir daño por parte de los enemigos, y que acabar con los enemigos de la forma que se nos proporcione. El juego de plataformas más conocido es quizá el 'Super Mario Bros.', los juegos elegidos tienen un concepto muy similar a éste, con la diferencia de que para acabar con los enemigos tenemos que dispararles, y no saltar sobre ellos.

3.1 Ghosts and Goblins (NES)

Ghosts and goblins (en adelante GnG)[1] es un juego de plataformas de *scroll* lateral; el objetivo del juego es avanzar por los distintos niveles manejando a Arthur el caballero hasta que logramos rescatar a la princesa en el último nivel.

Para el presente trabajo solo tendremos en consideración el primero de estos niveles, que consta de dos pequeñas fases:

La primera es el cementerio lleno de enemigos zombis y flores que lanzan proyectiles con un *mini-boss* al final, como podemos ver de forma simplificada en la figura 3.

Figura 3 Primera fase del primer nivel de Ghosts n' Goblins

[1] https://en.wikipedia.org/wiki/Ghosts_%27n_Goblins_(video_game)

La segunda fase es un bosque encantado con huecos en el suelo en los que podemos caer hacia la muerte de nuestro personaje, fantasmas y un *boss* final, como podemos ver de forma simplificada en la figura 4.

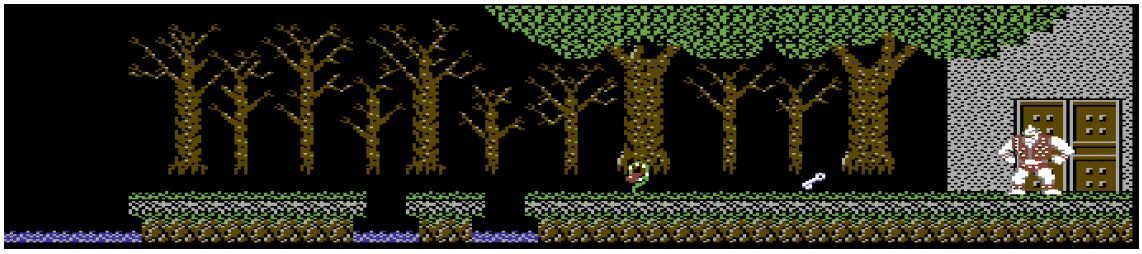

Figura 4 Segunda fase del primer nivel de Ghosts n' Goblins

Este juego tiene una alta complejidad para un jugador humano, ya que requiere de unos reflejos agudos para poder sortear todos los enemigos y proyectiles. Nuestro algoritmo va a jugar *frame* por *frame*, por lo que los reflejos no van a ser un problema; el reto para nuestra inteligencia artificial es conseguir que avance hacia la dirección que corresponde, y que llegue hasta el *boss* final con un arma que le permita acabar con el (en este juego hay varias armas y cada *boss* es inmune a un arma en concreto).

3.2 Mega-Man 2 (NES)

Mega-Man 2[2] es también un juego de plataformas en el que manejamos al titular cíborg Mega-Man a lo largo de una serie de niveles en los que tiene que ir derrotando a una serie de jefes en cada fase y a un jefe final. La premisa del juego es, como vemos, muy similar a GnG.

Al igual que con Ghosts and Goblins, la dificultad de este juego está presente de manera primaria en las acciones precisas que el jugador debe tomar, lo que no debería ser un problema para nuestra inteligencia artificial (IA) , reduciendo el problema a que la IA consiga entender el propósito del juego y llevarlo a cabo para recibir la mayor recompensa posible.

No ahondaremos más en este juego ya que el componente principal de este proyecto es el aprendizaje del modelo de red neuronal, el cual se realiza en el juego mencionado anteriormente.

[2] https://en.wikipedia.org/wiki/Mega_Man_2

4. OpenAI `gym-retro`

`gym-retro` es una biblioteca [6] desarrollada por OpenAI que nos permite emular el funcionamiento de distintas consolas; en este caso estamos interesados exclusivamente en la Nintendo Entertainment System (NES).

Esta biblioteca nos proporciona el siguiente flujo de trabajo para conseguir que nuestro *script* Python se comporte como una consola emulada:

1. Definición del entorno, el cual inicializamos con la función *make*(), que toma como argumentos el nombre del juego, una variable booleana que indica si empezar el juego desde el punto 0 del archivo de configuración o desde que se enciende la consola, el número de jugadores, un booleano que indica si queremos que se guarden las acciones tomadas para reconstruir la partida en un vídeo, y el tipo de observación que queremos.
   Estas observaciones pueden ser de pantalla, de modo que tendremos acceso a un *array* tridimensional en el que tendremos los valores RGB de la pantalla para cada píxel. También pueden ser observaciones de la memoria RAM, donde tendremos acceso a los valores que el juego almacena en la RAM (estos valores definen por ejemplo el número de vidas restantes del jugador, su posición en la pantalla, el tiempo restante, etc.).
   En adelante nos referiremos a este entorno como la variable 'env'.

2. Una vez definido el entorno debemos reiniciarlo para asegurarnos de que empezamos la partida desde cero cada vez que vayamos a jugar; para ello utilizamos el método *env.reset()*
   Este método nos devuelve la observación inicial, que dependiendo del tipo de observación que hayamos elegido para el entorno será los valores de la memoria RAM o la pantalla del juego.
   En adelante nos referiremos a esta observación como 'obs'.

3. Finalmente, mientras que la partida no termine, debemos tomar acciones utilizando el método *env.step(action);* este método tiene como entrada una acción, lo que en una consola equivaldría a los botones del mando que el jugador está presionando en cada *frame*. En el presente trabajo hemos reducido el número de acciones que nuestra IA puede seleccionar para tratar de simplificar el aprendizaje todo lo posible; esta reducción de posibles acciones no implica una pérdida de funcionalidad, pues se ha tenido en cuenta la naturaleza del juego a la hora de seleccionar las acciones posibles (por ejemplo, se ha eliminado la posibilidad de que se presionen a la vez los botones izquierdo y derecho del mando, ya que produciría el mismo resultado que no presionar ninguno de los dos).

En concreto, para la NES las acciones tienen la forma siguiente:

[B, NULL, SELECT, START, UP, DOWN, LEFT, RIGHT, A]

Teniendo esto en cuenta, hemos reducido el número total de acciones que puede tomar la máquina de 8! (40.320) a 13:

1: [0,0,0,0,0,0,0,0,0]
2: [1,0,0,0,0,0,0,0,0]
3: [0,0,0,0,0,0,0,0,1]
4: [0,0,1,0,0,0,0,0,0]
5: [0,0,0,1,0,0,0,0,0]
6: [0,0,0,0,1,0,0,0,0]
7: [0,0,0,0,0,1,0,0,0]
8: [0,0,0,0,0,0,1,0,0]
9: [0,0,0,0,0,0,0,1,0]
10: [1,0,0,0,0,0,0,1,0]
11: [1,0,0,0,0,0,1,0,0]
12: [0,0,0,0,0,0,0,1,1]
13: [0,0,0,0,0,0,1,0,1]

Este método *step* nos devuelve como salida:

- La nueva observación tras aplicar la acción seleccionada en el entorno.
- El valor calculado por la función de recompensa (más tarde explicaremos con detalle esta función).
- Un booleano que indica si la partida ha terminado (por ejemplo, cuando el jugador se queda sin tiempo o se le acaban las vidas).
- Un valor con información que desestimaremos para el presente proyecto ,pues no nos aporta utilidad alguna

Con estos valores de salida podemos entrenar a nuestra red neuronal para que aprenda que acciones suponen una mayor recompensa.

Otra función de `gym-retro` que vamos a utilizar es la función *env.render(),* que muestra en pantalla la imagen que estaría viendo el jugador si estuviera jugando al juego de manera normal en vez de simplemente introduciendo los presionados de botón en un emulador. Esta función nos sirve para poder visualizar el movimiento de nuestra IA, ya que es imposible para una persona entender una imagen a partir de un *array* de valores RGB.

5 Introducción al aprendizaje por refuerzo

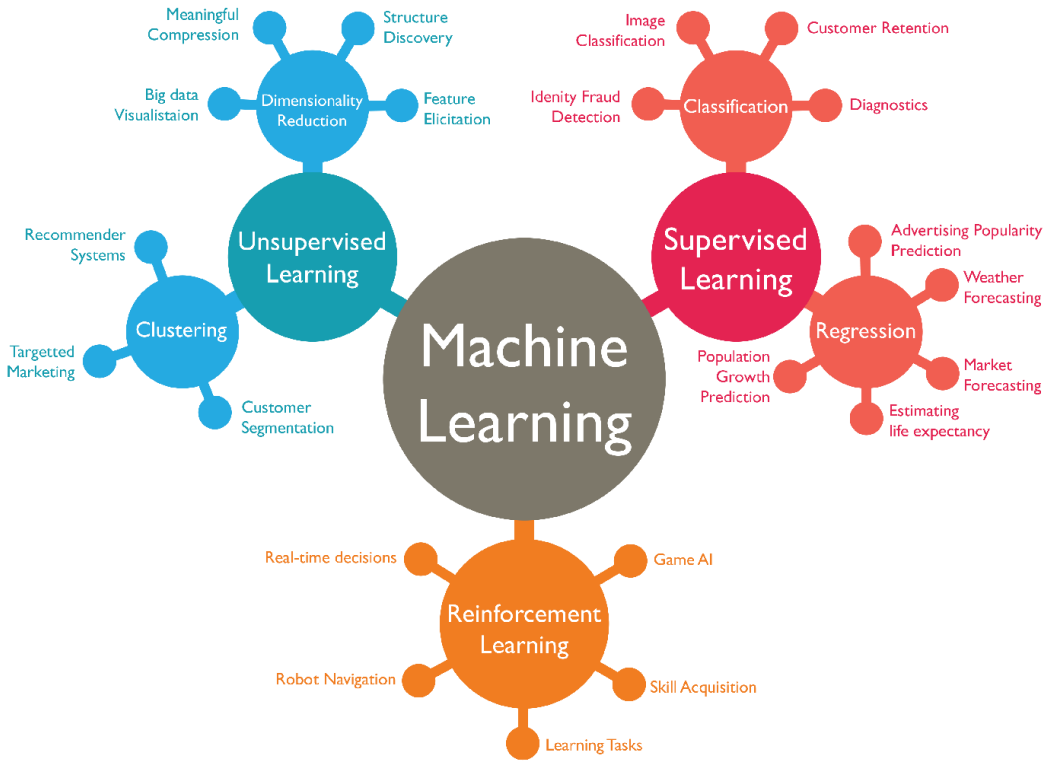

Figura 5 Representación de la disciplina de inteligencia artificial (aprendizaje automático) y sus diversas ramas [8]

El aprendizaje por refuerzo, en el contexto de la inteligencia artificial mostrado en la figura 5, es un tipo de programación dinámica que entrena algoritmos usando un sistema de castigo y recompensa.

Un algoritmo de aprendizaje por refuerzo, o agente, aprende interactuando con su entorno. El agente recibe recompensas completando tareas correctamente y castigos cuando completa tareas incorrectamente. El agente aprende sin intervención humana, maximizando su recompensa y minimizando su castigo.

El aprendizaje por refuerzo es una aproximación al *machine learning* que está inspirado por la psicología del comportamiento. Es similar a como un niño aprende a realizar una nueva tarea. El aprendizaje por refuerzo contrasta con otras aproximaciones al *machine learning* basadas en reglas en tanto que al algoritmo no se le dice explícitamente cómo realizar la tarea, sino que intenta resolver el problema por sí mismo [9].

Antes de profundizar en técnicas concretas de aprendizaje por refuerzo debemos definir los siguientes términos:

- Agente:
  - Programas de software que toman decisiones inteligentes. Son los aprendices en el aprendizaje por refuerzo. Estos agentes interactúan con el entorno mediante acciones y reciben recompensas basadas en dichas acciones.
- Entorno:
  - Es la demostración del problema a ser resuelto. Podemos tener entornos reales o simulados con los que nuestro agente va a interactuar (en el caso que nos ocupa, nuestro agente va a interactuar con el entorno simulado del juego con ayuda de la biblioteca `gym-retro` de la que ya hemos hablado antes).
- Estado:
  - Es la posición del agente en un momento específico del tiempo. De este modo, cuando el agente lleva a cabo una acción el entorno le da al agente una recompensa y un nuevo estado alcanzado tomando esa acción (y todas las anteriores en consecuencia).
  - En nuestro caso particular el estado inicial del juego será siempre el mismo; además, durante las primeras oleadas de enemigos, el juego es determinista, es decir, estos enemigos aparecen en posiciones concretas del mapa y se moverán de una manera concreta. A partir de cierto número de estas oleadas los enemigos ya comienzan a comportarse de manera aleatoria.
- Transición:
  - El movimiento de un estado a otro se llama transición.
- Probabilidad de transición:
  - La probabilidad de que un agente se mueva de un estado a otro.

Una vez definidos los términos se puede aplicar la `Propiedad de Markov`, que dice que: "El futuro es independiente del pasado dado el presente".

Lo que matemáticamente se expresa como:

$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_1, \ldots, S_t]$

Donde S_t denota el estado actual del agente y S_{t+1} denota el siguiente estado y S_k, k=1,...t-1, todos los estados tomados por el agente en los tiempos anteriores al actual. [10]

Lo que esta propiedad significa para nosotros, es que, para tomar la decisión de qué acción tomar para llegar al siguiente estado, no necesitamos una lista de todos los estados anteriores, tan solo necesitamos el estado actual. Teniendo en cuenta el tamaño y número de los estados anteriores esto reduce enormemente el peso computacional del problema.

5.1 Introducción al *Q-Learning*

Una vez dejados claros estos conceptos, podemos pasar a explicar el tipo en concreto de acercamiento al aprendizaje por refuerzo (AR) que vamos a utilizar.

Dentro de las posibles técnicas del aprendizaje por refuerzo, una de las más utilizadas es el *Q-learning*. Este tipo de aprendizaje por refuerzo se basa en la existencia de una tabla (llamada Q-tabla) donde se almacena la recompensa esperada para determinada acción en determinado estado.

Los pasos que nuestro agente seguirá para cada estado que encuentre son:

1. Buscar en la Q-tabla la fila correspondiente al estado encontrado. Aquí tenemos dos opciones; si ya hemos estado en este estado anteriormente habrá una entrada en la tabla para este estado, junto con el valor esperado para cada una de las posibles acciones a tomar, y nuestro agente elegirá la acción con la mayor recompensa esperada.
   Esta acción será elegida de manera aleatoria con una probabilidad épsilon para evitar tomar siempre las mismas acciones. Esta técnica de elección de acciones se conoce como '*epsilon greedy*' y nos permite explorar finalmente todas las posibles acciones. La importancia de este tipo de elección de acción radica en que al inicio de la ejecución del programa no sabemos el valor que vamos a obtener de cada acción, por lo que todos los valores de la Q-tabla son el mismo, lo que hace que todas las acciones tengan la misma probabilidad de ser escogidas. Esto puede dar lugar a que una acción escogida que devuelva un mal resultado no vuelva a ser elegida, aunque en el resto de los casos fuera la mejor acción; es por esto que elegimos de manera aleatoria cierto número de acciones.
   Si no hay una entrada en la tabla para el estado que encontramos se elegirá una acción al azar y se creará una entrada en la tabla

2. Una vez elegida la acción que se va a tomar, se ejecuta en el entorno, y actualizamos la Q-tabla según la ecuación 1.

$$Q'(s,a)=Q(s,a)+ \alpha[R(s,a) + \gamma \max Q'(s', a') - Q(s,a)] \qquad (1)$$

Donde:

- *s* representa el estado en el momento de tomar la acción
- *a* representa la acción tomada
- *s*' representa el estado tras tomar la acción a
- *a*' representa la futura acción a elegir
- Q'(*s,a*) es el nuevo valor que tendrá la Q-tabla para la fila s y la columna a, es decir, la recompensa esperada al tomar la acción a en el estado s teniendo en cuenta la ecuación
- Q(*s,a*) es el valor actual de la Q-tabla para el estado s y la acción a

- α es el ratio de aprendizaje. Este valor, como podemos ver en la ecuación 1, se multiplica por la recompensa obtenida y la recompensa esperada, y el resultado de esta multiplicación es sumado al valor actual de la entrada de la Q-tabla, de modo que este valor es el que determina lo rápido que los valores de la Q-tabla fluctúan con cada ejecución; generalmente suele estar en un rango de entre 0.1 y 0.3; valores más pequeños hacen que la Q-tabla se actualice de manera muy lenta, y valores mayores provocan un cambio demasiado rápido que puede dar lugar a una inestabilidad que provoque que los valores nunca confluyan hacia sus valores finales.
- R(s,a) es la recompensa de realizar la acción a en el estado s.
- γ es el factor de descuento; un valor más alto de γ significa dar más importancia a las acciones futuras y un valor bajo indica más importancia para las acciones inmediatas. Este valor generalmente está en el rango de 0.8 y 0.99
- maxQ'(s',a') es el máximo esperado de la futura recompensa dado el estado s' (al que vamos a pasar desde el estado s con la acción a) para todas las acciones posibles en este nuevo estado

El problema de esta Q-tabla es que crece muy rápidamente con el número de acciones y el número de posibles estados; en concreto, el tamaño de la tabla que contenga todos los posibles estados será:

$$S \cdot a$$

Donde S es el número de posibles estados y a el número de posibles acciones que podemos tomar en cada estado.

Este valor, *a priori*, no parece que vaya a ser demasiado elevado hasta que nos planteamos qué significa S exactamente; en nuestro caso en concreto a es un valor conocido, las 13 posibles acciones que comentamos en el capítulo anterior.

El problema de intentar aplicar esta aproximación a nuestro problema viene cuando ejecutamos el programa y se queda sin espacio en memoria, pero. ¿Qué ha ocurrido?

Empecemos definiendo S en nuestro problema; S es el conjunto de todos los posibles s, o estados que nos podemos encontrar en nuestro problema.

Para nuestro problema, queremos que nuestro modelo de red neuronal aprenda a jugar al juego con la información que recibe por pantalla, es decir, el array de todos los valores RGB para cada píxel de la pantalla, por lo que la dimensión de este valor es de:

$$220 \cdot 224 \cdot 3 = 147840$$

Es decir, que por cada imagen que vemos en pantalla tenemos cerca de ciento cincuenta mil números que definen nuestro estado; además, como con una sola imagen no podemos apreciar la dirección ni velocidad de los objetos en movimiento, por lo que por cada estado de nuestro problema almacenamos 4 imágenes (o *frames*) consecutivas, con lo que llegamos a 600 mil números por estado.

Para reducir este número se han utilizado los siguientes recursos:

- Comprimir la imagen para que tenga un tamaño más reducido. Esta compresión hace que pasemos de tener una imagen con 224 pixeles de alto por 220 pixeles de ancho a una imagen de 200x200 píxeles
- Utilizar las imágenes en blanco y negro para tener una sola dimensión de color en vez de tres.

Con estos cambios tenemos que cada uno de nuestros estados tiene la forma siguiente:

$$4 \cdot 200 \cdot 200 = 160.000$$

Como podemos ver, de esta manera representamos un estado en aproximadamente el mismo espacio que usábamos para almacenar uno solo de los cuatro *frames* que componen cada estado, lo que supone una mejora notable.

Sin embargo, la Q-tabla no almacena un único estado, sino que almacena todos los posibles estados del juego, lo que implica que para los 255 posibles valores de la escala de grises el número de posibles estados de nuestro problema resulta ser:

$$255^{160.000}$$

Teniendo en cuenta que el número total de átomos que se estima existen en todo el universo es de aproximadamente 10^{80} podemos ver que es imposible que resolvamos este problema utilizando una Q-tabla.

Es imprescindible, pues, investigar una aproximación factible.

5.2 Introducción al *Deep Q-Learning*

El *Deep Q-learning* aplica los mismos principios que el *Q-learning* pero utilizando redes neuronales en vez de q-tablas, como se ve en la figura 6; de esta forma podemos resolver problemas para los que no habría suficiente memoria si se implementasen utilizando tablas.

Para esta implementación, que es la elegida para el presente trabajo, se ha optado por una red neuronal con tres capas convolucionales y una capa completamente conectada, como datos de entrada para la red neuronal tenemos, como hemos comentado anteriormente, el estado que construimos a partir del *frame* actual y los 3 anteriores, comprimidos y en escala de grises

Es necesario también para el aprendizaje una función de recompensa que, como hemos visto anteriormente, utilizaremos tanto para evaluar la evolución de nuestro agente, como para que el propio agente pueda aprender la efectividad de sus acciones.

La configuración inicial de la biblioteca para este juego en concreto daba 200 puntos de recompensa por matar a un enemigo; como queremos que nuestra IA avance hacia la derecha tenemos que cambiar la función de recompensa. Para esto tenemos que observar de la memoria RAM del juego el valor de la posición x del personaje; se ha utilizado para ello la herramienta de integración de la librería, la cual nos permite explorar valores de la RAM mientras jugamos, para poder deducir qué valores cambian al moverse a derecha e izquierda, y poder asignarles un valor en la función de recompensa.

Una vez hemos hecho esto, la función de recompensa de la biblioteca nos devuelve 200 puntos al matar a un enemigo, y 3 puntos al movernos. Como el valor que nos devuelve al movernos son 3 puntos independientemente de si vamos a izquierda o derecha, debemos en nuestro programa de entrenamiento de la red neuronal implementar una función que comprueba si estaba presionado el botón derecho o izquierdo a la hora de realizar esa acción en concreto. Con esto hacemos que la función de recompensa nos devuelva 200 puntos por acabar con un enemigo, 20 puntos por moverse a la derecha, y -20 puntos por moverse hacia la izquierda.

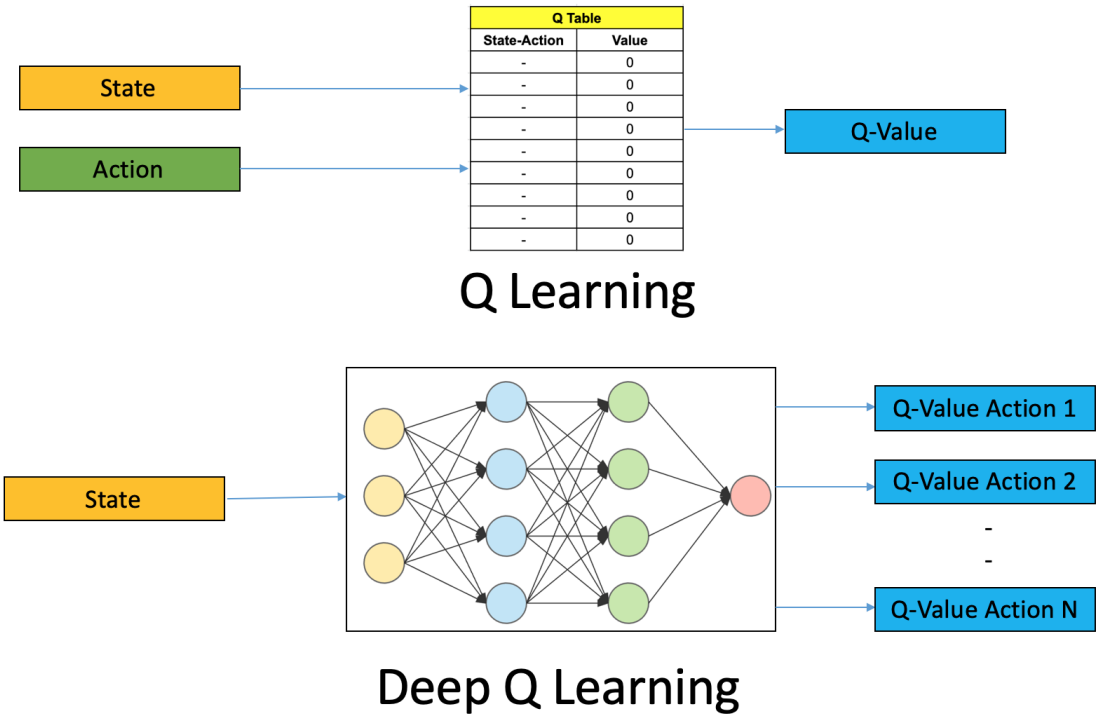

Figura 6 [7]

6 Explicación del algoritmo

Para el presente trabajo utilizaremos un par de estrategias que nos ayudarán a conseguir mejores resultados. La primera de ellas es '*experience replay*'; lo que significa es que en vez de actualizar la red neuronal a cada paso, guardamos los pares estado-acción en un *buffer*, llamado *replay buffer* cuando hemos llenado el *buffer* hasta un nivel predefinido la máquina deja de simular y empieza a aprender, seleccionando de manera aleatoria pares de ese *buffer*. De esta manera evitamos sobreentrenar la red para ciertas partes del videojuego; en particular evita que si una parte del videojuego se repite siempre al principio sea este aprendizaje el que se lleva todas las veces a cabo, lo que hace que la red no sea tan efectiva en otras partes del videojuego.

La otra estrategia es utilizar una '*target network*'; lo que esto quiere decir es que no utilizamos una única red neuronal, sino que utilizaremos dos. La causa de esto es que al no tener un conjunto de datos sobre el que entrenar, sino que el programa va aprendiendo conforme juega, este aprendizaje puede ser inestable. Esta '*target network*' tiene la misma configuración que la red neuronal principal, pero se actualiza únicamente cada cierto número de episodios (que equivaldrían a los '*epochs*' si estuviéramos trabajando sobre un conjunto de datos tradicional); de esta manera el aprendizaje se estabiliza pues la '*target netw*ork' es la encargada de predecir las acciones a llevar a cabo para cada estado, y la red principal aprende de cada una de estas decisiones. Una vez este número de episodios ha pasado, los datos de la red principal son copiados a la '*target network*'; en este momento ambas redes son idénticas tanto en topología como en los pesos de los tensores, es decir, que la '*target network*' se va periódicamente quedando atrás con respecto a la red principal y poniéndose al día periódicamente.

Teniendo en cuenta estas consideraciones, el algoritmo comienza inicializando los parámetros y las dos redes neuronales, así como reservando espacio en memoria para los *arrays* que utilizaremos tanto en el 'replay buffer' como para los resultados.

```
env = retro.make('GhostsnGoblins-Nes')
epsilon = 1.0
epsilon_min = 0.1
epsilon_change = (epsilon - epsilon_min) / 500000
model = DQN(
    K=K,
    conv_layer_sizes=conv_layer_sizes,
    hidden_layer_sizes=hidden_layer_sizes,
    scope="model"
)
target_model = DQN(
    K=K,
    conv_layer_sizes=conv_layer_sizes,
    hidden_layer_sizes=hidden_layer_sizes,
    scope="target_model"
)
```

Código 1

Como podemos ver en el fragmento de código 1, inicializamos las redes neuronales como objetos de la clase DQN, la cual, aparte de varias funciones auxiliares que nos permiten guardar los pesos de todos los tensores de la red neuronal, implementa el método *learn* que es el que efectúa las operaciones de la ecuación 1. Podemos ver este método en el fragmento de código 2

```
def learn(model, target_model, experience_replay_buffer, gamma, batch_size):
  # Sample experiences
  states, actions, rewards, next_states, dones =
experience_replay_buffer.get_minibatch()

  # Calculate targets
  next_Qs = target_model.predict(next_states)
  next_Q = np.amax(next_Qs, axis=1)
  targets = rewards + np.invert(dones).astype(np.float32) * gamma * next_Q

  # Update model
  loss = model.update(states, actions, targets)
  return loss
```

Código 2

Los métodos *predict* y *update* que se pueden observar en el fragmento de código 2 son los de la biblioteca `Tensorflow`, en los cuales no entraremos en detalle, pero sí que es importante destacar que el equivalente de la ecuación 1 es la línea 9 donde se calcula *targets*.

Después se llena el 'replay buffer' hasta un número mínimo de datos almacenados utilizando acciones aleatorias; esto asegura que cuando la red empiece a entrenar tenga suficientes datos de entre los que elegir para entrenar de manera aleatoria.

```
  for i in range(MIN_EXPERIENCES):

    action = np.random.choice(K)
    action_buttons = number2buttons(action)
    obs, reward, done, _ = env.step(action_buttons)
    reward = process_reward(action_buttons, reward)
    obs_small = image_transformer.transform(obs, sess) # not used anymore
    experience_replay_buffer.add_experience(action, obs_small, reward, done)

    if done:
      obs = env.reset()
```

Código 3

En este fragmento de código 3 podemos ver funciones de la biblioteca *gym-retro* de las que hemos hablado anteriormente, como *env.step* y *env.reset*.

A partir de aquí empieza el aprendizaje propiamente dicho, para lo cual realizaremos los siguientes pasos mostrados en el código 4 tantas veces como número de episodios pretendamos ejecutar.

```
# Inicializamos el entorno del juego
  obs = env.reset()
  obs_small = image_transformer.transform(obs, sess)
  state = np.stack([obs_small] * 4, axis=2)
  loss = None

  # inicializamos las variables de control
  total_time_training = 0
  num_steps_in_episode = 0
  episode_reward = 0
```

Código 4

A continuación, empezamos el bucle, mostrado en el código 5.

```
  done = False
  while not done: #Cada iteración de este bucle es un paso del juego
```

Código 5

Dentro del bucle del código 5 ejecutamos los fragmentos de código 6, 7, 8 y 9.

```
    # Actualizamos la target network cada TARGET_UPDATE_PERIOD pasos
    if total_t % TARGET_UPDATE_PERIOD == 0:
      target_model.copy_from(model)
      print("Copied model parameters to target network. total_t = %s, period = %s" %
(total_t, TARGET_UPDATE_PERIOD))
```

Código 6

```
# La red neuronal principal predice una accion para el estado
action = model.sample_action(state, epsilon)
# convertimos la acción elegida en los botones presionados
action_buttons=number2buttons(action)
# realizamos el paso del juego y recogemos los datos que nos devuelve
obs, reward, done, _ = env.step(action_buttons)
# procesamos la recompensa
reward = process_reward(action_buttons, reward)
# transformamos la imagen al tamaño elegido y escala de grises
obs_small = image_transformer.transform(obs, sess)
# creamos el estado nuevo para la iteración siguiente
next_state = update_state(state, obs_small)
```

Código 7

En el fragmento de código 7 podemos ver la llamada a la ya comentada función *env.step*, y cómo procesamos la recompensa que nos devuelve la biblioteca por defecto para ajustarla a los valores que hemos elegido para entrenar nuestra red neuronal

```
# Calculamos la recompensa total
episode_reward += reward

# guardamos los resultados en el replay buffer
experience_replay_buffer.add_experience(action, obs_small, reward, done)
```

Código 8

En el código 8 vemos como trata la experiencia y la añade al *replay buffer*

```
# Entrenamos la red utilizando las funciones de tensorflow
t0_2 = datetime.now()
loss = learn(model, target_model, experience_replay_buffer, gamma, batch_size)
dt = datetime.now() - t0_2
```

Código 9

En los códigos 9 y 10 vemos como la red utiliza las funciones de Tensorflow para entrenar y actualiza los valores que utilizamos para controlar la ejecución del programa

```
# actualizamos los valores de debug
total_time_training += dt.total_seconds()
num_steps_in_episode += 1


state = next_state
total_t += 1

epsilon = max(epsilon - epsilon_change, epsilon_min) #actualizamos epsilon

#devolvemos los resultados
return total_t, episode_reward, (datetime.now() - t0), num_steps_in_episode,
total_time_training/num_steps_in_episode, epsilon
```

Código 10

7 Análisis de resultados de la primera fase

Una vez se han realizado todas las iteraciones del código mostrado en el capítulo 6 la red neuronal está entrenada y lista para jugar. En este caso el entrenamiento realizó 3500 pasos y tardó algo más de 12 días entrenando en una gpu Nvidia GTX970. Los resultados se muestran en las figuras 7, 8, 9 y 10. En todas ellas el eje X es el número de iteraciones y el eje Y el valor obtenido por la función de recompensa para esa iteración en concreto.

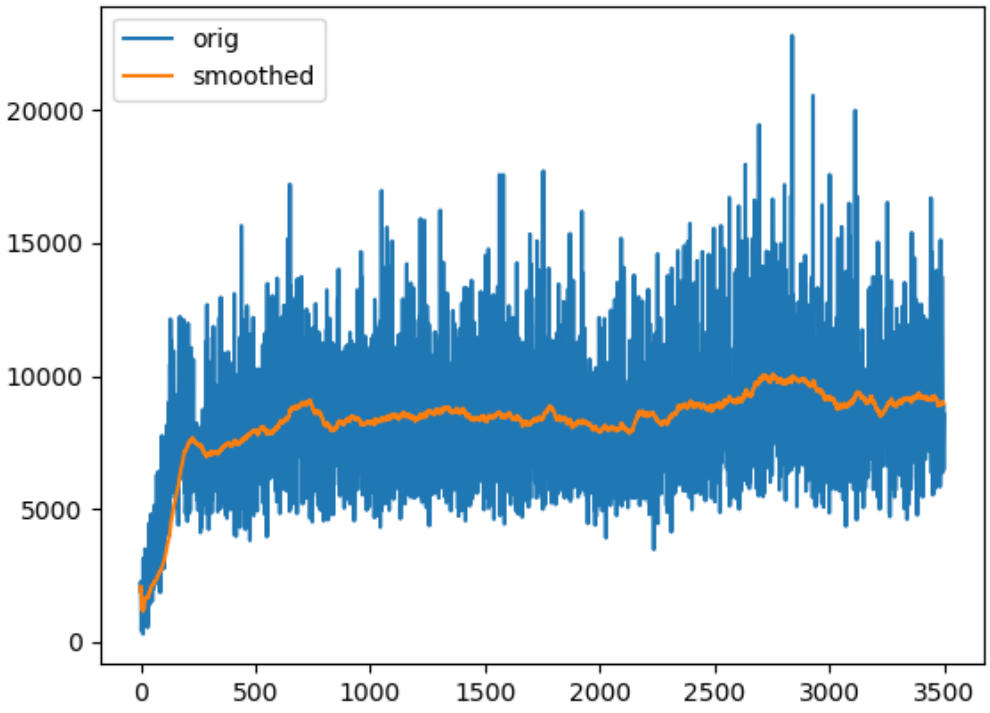

Figura 7 Variación global(azul) y promedio(naranja) de la función de recompense con el número de iteraciones

Para facilitar la visualización se ha suavizado los datos, de manera que lo que la línea naranja representa es la media de los últimos 100 episodios.

Como podemos observar en la figura 7, la recompensa obtenida por nuestra red neuronal crece rápidamente hasta aproximadamente la iteración número 500; a partir de ahí se mantiene un crecimiento, pero con una rapidez marginal en comparación con el conocimiento previo, lo cual nos lleva a concluir que se puede replicar este experimento reduciendo el número de iteraciones de entrenamiento, esto agilizaráa todo el proceso de entrenamiento y nos permitiría probar un mayor número de metaparámetros.

En cuanto a la causa de este parón repentino en el crecimiento de la recompensa que nuestra red neuronal consigue, es muy probablemente debido a que abandona la parte determinista del juego y se adentra en la parte aleatoria. Como hemos comentado anteriormente, los primeros enemigos aparecen en sitios fijos del mapa y se mueven utilizando siempre el mismo patrón, pero una vez que avanzamos un poco tanto la aparición de enemigos como su movimiento se vuelve aleatorio.

Evaluar visualmente el desempeño de nuestro agente confirma nuestra teoría, viendo que supera con facilidad los primeros enemigos pero que empieza a tener dificultades cuando el juego deja de ser determinista.

En la figura 8 podemos observar con más detalle como crece la recompensa durante los primeros 100 episodios. Este rápido crecimiento inicial, que ya hemos comentado, se corresponde con nuestro agente aprendiendo a desenvolverse en la parte determinista del juego, de ahí que cada vez la recompensa sea mayor.

Es importante destacar que la recompensa es resultado tanto del número de enemigos que nuestro agente elimina, como de la distancia que consigue viajar hacia la derecha antes de perder el juego. Por tanto, por lo general, cuanta mayor recompensa, mayor distancia ha viajado nuestro agente

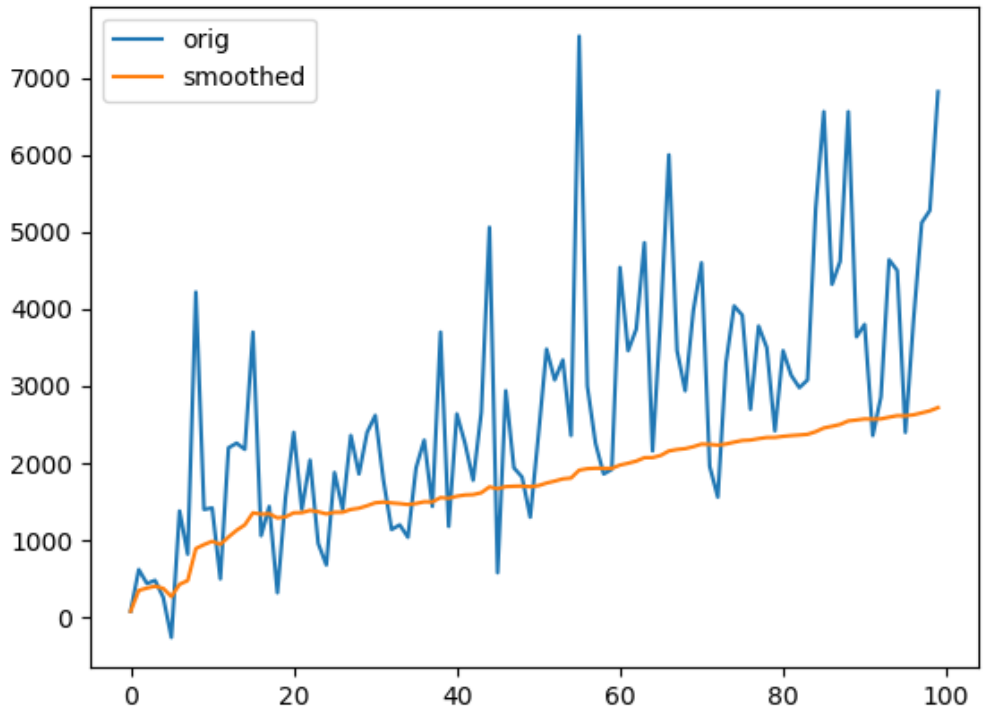

Figura 8 Detalle de las 100 primeras iteraciones de la evolución de la recompense global (azul) y promedio (naranja).

Utilizando esta red entrenada, comparamos el rendimiento de predecir la acción para cada paso frente a elegir la acción de forma aleatoria para 100 episodios; estos resultados los podemos ver en la figura 9

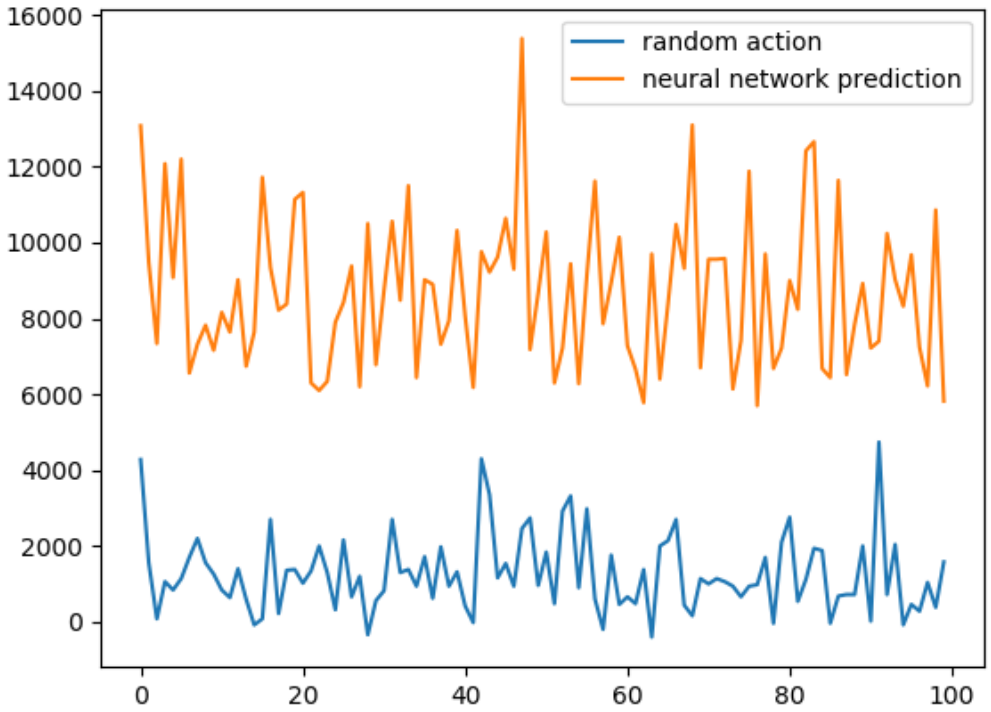

Figura 9 Valor de la función de recompense con toma de decision aleatoria (azul) o predicha por la red neuronal (naranja).

Como vemos, las predicciones de la red neuronal nos devuelven un resultado significativamente mejor que las acciones aleatorias. Es motivo de atención acerca del gráfico anterior que la variabilidad entre los resultados obtenidos por la red neuronal se debe a que, aun estando entrenada, se ha dejado un valor de épsilon residual para forzar algunas acciones aleatorias y poder comprobar su actuación de mejor forma. Si dejásemos que todas las acciones fueran predichas por la red obtendríamos unos resultados como los que podemos observar en la figura 10.

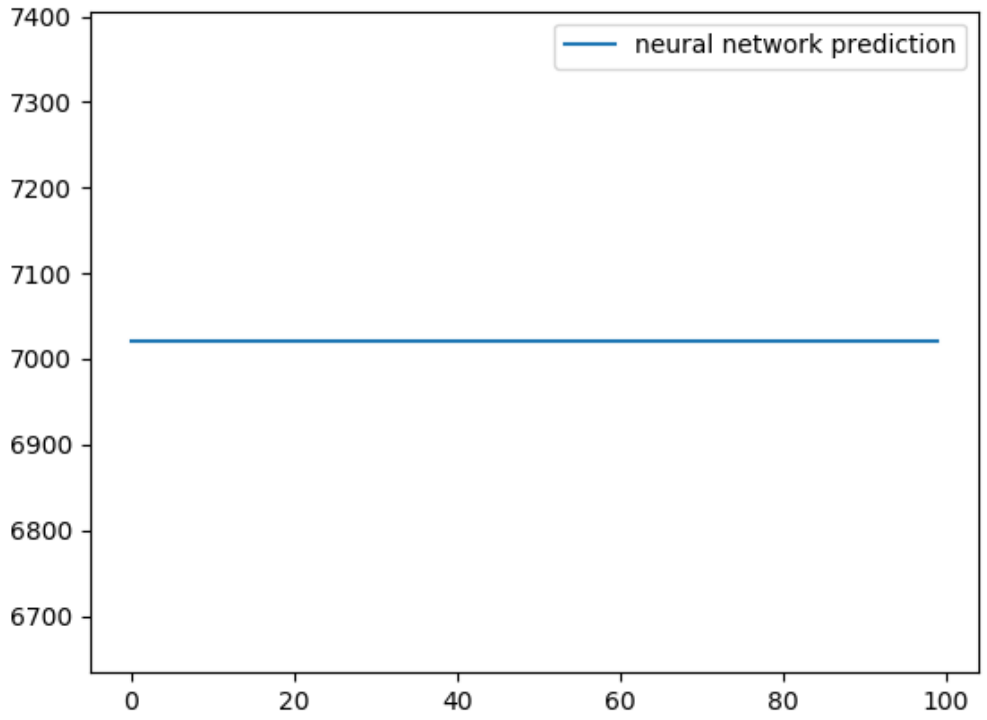

Figura 10 Valor de la función de recompense con predicción determinista

Aquí podemos ver lo que anteriormente comentábamos de que el juego es al principio determinista, y que si dejamos que la red neuronal siga siempre sus predicciones llega a un resultado idéntico cada vez, lo que nos indica que nuestro algoritmo puede estar sobreentrenado para el tramo inicial, pero no ser efectivo para el siguiente tramo. Es por esto que añadiendo variabilidad aleatoria a las acciones que toma nuestro agente (en el caso de la gráfica 9 la probabilidad residual de elegir una acción aleatoria es de 0.05, o lo que es lo mismo, 1 de cada 20 acciones se toma de manera aleatoria) conseguimos unos resultados mejores.

Tampoco podemos dejar de fijarnos en que, aunque muy superiores, los resultados obtenidos por nuestro agente son demasiado cercanos a los que consigue un agente que únicamente toma acciones aleatorias. Esto podría haber sido subsanado probando un rango mayor de meta parámetros de aprendizaje que, por el tiempo de ejecución que requieren, han sido imposibles de probar.

8 Extrapolación a un juego similar

El presente trabajo no se centra solo en un videojuego, sino que pretende extrapolar los resultados de entrenar la red en un juego e intentar jugar en otro distinto.

El segundo videojuego elegido es el 'Mega-Man 2', también para la *Nintendo Entertainment System*. Este juego ha sido elegido por ser el más similar al juego en el que hemos entrenado la red dentro de los soportados por la biblioteca `gym-retro` para la consola *NES;* también se trata de un juego de *scroll* lateral de plataformas en los que los botones realizan acciones idénticas a las que realizan en el 'Ghosts and Goblins'.

Esta extrapolación se ha realizado sin dejar a la red neuronal entrenar en el segundo juego, es decir, se ha intentado que toda la información necesaria para pasarse este segundo juego haya sido obtenida en el primero.

Por desgracia, los resultados obtenidos con este videojuego son tanto fallidos como irrepresentables en una gráfica por dos motivos.

El primero es que, a diferencia del 'Ghosts and Goblins', donde al inicio ya nos aparecen enemigos contra los que pelear, en 'Mega Man 2' empezamos en un pasillo vacío, con una pared a la izquierda y un tramo vacío que recorrer hacia la derecha hasta que llegamos a los primeros enemigos y obstáculos del mapa; esto, combinado con la falta de un temporizador que termine la partida si nos demoramos demasiado, hace que las partidas jugadas con *inputs* aleatorios no terminen nunca, pues el juego no llega hasta la zona donde hay enemigos y puede morir.

Tampoco tenemos mucha más suerte jugando al juego con la red neuronal, pues a pesar de ser el juego disponible más similar al juego en el que la red ha entrenado, la red decide ir hacia la izquierda y chocarse con la pared de manera infinita, por lo que el juego tampoco acaba nunca.

Para evitar esta situación se podría intentar no despreciar los colores de la imagen a la hora del entrenamiento, ya que muchas veces los colores ayudan a reconocer a los enemigos más que las formas geométricas que los componen. También se podría intentar entrenar en un mayor número de juegos antes de realizar la extrapolación, teniendo en cuenta que habitualmente el conjunto de entrenamiento es mucho mayor que el conjunto de predicción.

9 Conclusiones

Como hemos podido observar en las figuras 7, 8, 9 y 10, nuestro algoritmo ha aprendido a jugar al juego con una eficiencia considerablemente mayor que la que obtenemos al ejecutar acciones aleatorias. También hemos podido ver al visualizar la máquina jugando en tiempo real que este progreso ha sido real, y que no se queda simplemente quieto esperando a que aparezcan nuevos enemigos. Esto se conoce como *'reward farming'*, y es un problema al que se enfrentan todos los agentes que intentan jugar de manera automática a un juego.

Aún así, como hemos comentado en el capítulo 7, el entrenamiento de nuestro agente es efectivo, pero no a un nivel satisfactorio, lo que hace que sea necesario replantear los meta parámetros con los que entrenamos a nuestro agente.

No obstante, no se ha conseguido que supere la primera fase del juego; esto es mayormente debido a la gran dificultad que tiene el juego debido a los movimientos precisos que se deben realizar para conseguir avanzar sin recibir daño. Tampoco se ha conseguido extrapolar el algoritmo a un juego similar sin dejar que aprendiera de este segundo juego, entrando esta segunda parte dentro de lo esperable, ya que es un tema de actual investigación la extrapolación de algoritmos para diferentes juegos.

10 Trabajo futuro

Si se desease seguir investigando el sujeto de este proyecto, se podría intentar entrenar la red con metaparámetros distintos, como pueden ser un tamaño distinto del *batch*, cambiar épsilon y la forma en la que épsilon decrece, cambiar el factor de descuento γ, es decir, la importancia que se le da a las acciones futuras con respecto a las acciones inmediatas, o incluso se podría probar con otra topología de la red neuronal que pueda dar mejores resultados.

Queda pendiente además conseguir que el algoritmo sea extrapolado a un juego similar. Se ha intentado realizar esta extrapolación utilizando un único juego como entrenamiento y un único juego para probar esta extrapolación; el actual estado del arte para esta extrapolación está en manos de OpenAI, que están en este momento desarrollando un algoritmo que juegue de manera autónoma a toda la gama de juegos de ATARI disponibles para la librería `gym-retro` que hemos utilizado en este trabajo. Sus resultados son todavía insatisfactorios, de modo que, aunque en este trabajo no se haya conseguido realizar dicha extrapolación, los resultados entran dentro de lo esperado, ya que para empezar a ver resultados necesitaríamos una gran cantidad de juegos para realizar el entrenamiento, lo que implica ajustar los parámetros para esos juegos y destinar una gran cantidad de recursos físicos (tiempo de ejecución en GPU) que quedan completamente fuera del alcance del presente trabajo.

11 Relación con el grado

El presente trabajo bebe de conceptos de las asignaturas de Percepción, Aprendizaje Automático y Agentes Inteligentes.

De Agentes Inteligentes extraemos los conceptos de agente, entorno, juego, etc. Que nos permiten poner contexto a lo que nuestro modelo de red neuronal está intentando conseguir.

Relativo al propio modelo de red neuronal, tomamos conocimiento sobre éstas en las asignaturas de Percepción y Aprendizaje Automático, ya que, aunque el aprendizaje por refuerzo en específico es un tema que no se ve con apenas profundidad, estas dos asignaturas enseñan los conceptos básicos acerca de las redes neuronales, y es en estas clases donde surgió mi idea de realizar el presente trabajo de fin de grado.

Referencias

1. Gallant, Stephen I. Neural Network Learning and expert systems, The MIT press, 1995
2. Guo, Xiaoxiao Deep learning and reward Design for Reinforcement Learning, University of Michigan, 2017
3. Güera, David and Delp, Edward j. Deepfake video detection using recurrent neural networks
4. Silver, David et al. Mastering the game of Fo with Deep neural networks and tree search
5. Goodfellow et al, Deep Learning, MIT press, 2016
6. https://github.com/openai/retro
7. https://openai.com/five/
8. https://i.imgur.com/rY1jjwx.png
9. https://www.techopedia.com/definition/32055/reinforcement-learning
10. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/