Integración y aplicación de técnicas de aprendizaje por refuerzo al robot IRB120 en el entorno virtual de MuJoCo

Lixiang Dong
Universidad Pontificia Comillas ICAI
(lixiang.dl@alu.comillas.edu)

Resumen—**En este proyecto se diseña e implementa el algoritmo de aprendizaje por refuerzo A3C con la finalidad de entrenar a un agente en el entorno virtual de MuJoCo capaz de acercar el punto terminal del robot IRB120 de ABB a un volumen objetivo situado de forma aleatoria en el espacio de trabajo. El entrenamiento se realizará a partir de las imágenes que se obtienen del entorno y de los elementos que ahí en él realizando así una aproximación al aprendizaje por refuerzo profundo. Además se analizará el comportamiento del agente según distintas funciones de recompensa.**

I. INTRODUCCIÓN

EL aprendizaje por refuerzo es una técnica de aprendizaje en el que un agente aprende media prueba y error a cómo actuar en un entorno con el objetivo de maximizar una recompensa [1].

El aprendizaje por refuerzo, junto con el aprendizaje supervisado y el aprendizaje no supervisado, se consideran los tres paradigmas de Machine Learning.

En el aprendizaje supervisado, el modelo aprende utilizando ejemplos de entrenamiento que fueron clasificados por un experto a priori. Los ejemplos relacionan diferentes situaciones con la respuesta correcta o clase, enseñando al modelo a determinar las decisiones a tomar en situaciones similares y extrapolando o generalizando unos comportamientos adecuados en situaciones no presentes en los ejemplos de entrenamiento. Es complicado implementar este tipo de aprendizaje cuando se quiere aprender a partir de la interacción con el entono dado lo poco práctico que es obtener el correcto comportamiento para las diferentes situaciones que puedan surgir.

El aprendizaje no supervisado es una técnica de aprendizaje que se utiliza para encontrar el patrón o estructura en una serie de datos sin conocimiento previo del tipo de dato que se trata. Es posible relacionar el aprendiza no supervisado con el aprendizaje por refuerzo ya que son técnicas que no dependen de un conocimiento a priori del comportamiento óptimo. El aprendizaje no supervisado puede ayudar a descubrir el patrón de la experiencia de un agente, pero no puede resolver el problema de maximizar la recompensa. Es por ello que el aprendizaje no supervisado y el aprendizaje por refuerzo se dividen en dos categorías distintas.

En la Figura 1 se muestra la estructura del aprendizaje por refuerzo.

Gracias al éxito del algoritmo de aprendizaje por refuerzo profundo DQN desarrollado por DeepMind en 2015 [2], se ha motivado la investigación de técnicas de aprendizaje por refuerzo en otros campos como la robótica. No obstante, aunque los algoritmos de aprendizaje por refuerzo profundo ofrecen un gran potencial, es complicado implementar estas técnicas en un robot física debido a los largos tiempos que se requieren de entrenamiento. Los numerosos episodios necesarios para que un agente aprenda mediante prueba y error puede resultar en posibles daños tanto al robot como a su entorno. Sin embargo, entrenando los agentes en un entorno simulado para la posterior transferencia al robot físico, se evitarían los problemas asociados a movimientos del robot al librarse de las limitaciones físicas, además de poder acelerar el aprendizaje.

En esta tesis se implementará un algoritmo de aprendizaje por refuerzo para entrenar un agente en la tarea de alcanzar un objetivo usando un modelo 3D del brazo robótico IRB120.

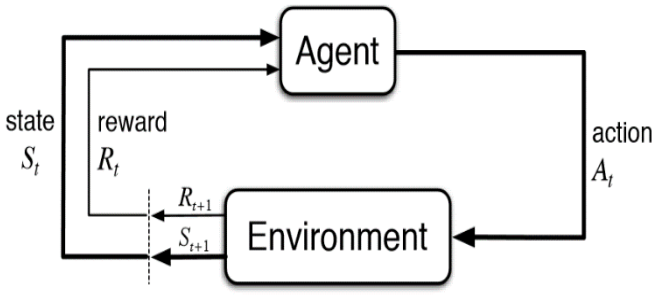

Figura 1. Esquema del aprendizaje por refuerzo

II. METODOLOGÍA

Se ha realizado el modelado 3D del robot utilizando el motor de física MuJoCo.

Se ha utilizado como lenguaje de programación Python al ser el lenguaje más utilizado en temas de aprendizaje automático, además de contar con la librería de mujoco_py, permitiendo la interacción entre Python y MuJoCo.

El algoritmo de aprendizaje por refuerzo profundo a implementar es A3C (Asynchronous Advantage Actor-Critic), utilizando como referencia la red neuronal convolucional expuesta en [3], donde recibe como entrada una imagen RGB de 64x64 para determinar los movimientos de las articulaciones hasta alcanzar el objetivo.

El agente entrenado debe cumplir con los siguientes objetivos planteados:

1. El robot debe ser capaz de acercar su punto terminal a un objeto colocado aleatoriamente dentro de su área de trabajo desde una posición inicial constante o aleatoria.
2. Debe obtener una probabilidad de éxito superior al 95 % si el robot empieza siempre desde la misma posición inicial, y una probabilidad de éxito superior al 85 % si comienza con posiciones aleatorias.
3. Se considera episodio exitoso si la distancia euclídea entre el punto terminal del robot y el objetivo es menor de 5 cm.

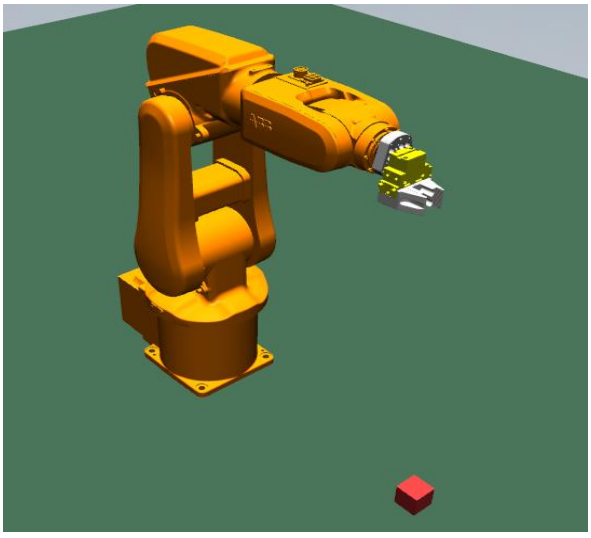

Figura 2. Modelo MuJoCo de IRB120 renderizado

Para lograr estos objetivos se ha dividido el proyecto en las siguientes tareas a completar:

1. Estudio y comprensión de los fundamentos teóricos necesarios para la implementación del algoritmo de aprendizaje por refuerzo del proyecto.
2. Familiarización con el motor de física MuJoCo con el fin de crear un modelo virtual del robot IRB120 de ABB.
3. Integración del modelo MuJoCo en un proyecto Python para inducir y medir cambios en la simulación. Para ello, se estudiará y descargará la versión de Python y librerías necesarias para crear el interfaz MuJoCo-Python.
4. Análisis y elección de las librerías de aprendizaje automático para la implementación de aprendizaje por refuerzo: TensorFlow, Pytorch, etc.
5. Creación de un entorno de aprendizaje por refuerzo profundo con el modelo IRB120.
6. Implementación de un algoritmo de aprendizaje por refuerzo profundo.

III. HERRAMIENTAS

I. MuJoCo

MuJoCo es un motor de física que permite simular sistemas mecánicos, robóticos y otras áreas donde se requiera de simulaciones precisas y rápidas.

Se trata de un producto comercial. Por lo que se requiere de licencia para su utilización. Existe la posibilidad de obtener de forma gratuita una licencia personal al presentar una solicitud como estudiante. Aprobada la solicitud, se recibirá un correo con la llave de activación.

Tras obtener la llave, se escogerá la versión de MuJoCo Pro a descargar teniendo en cuenta las características de la máquina utilizada. Para este proyecto se ha utilizado un ordenador Windows de 64-bits por lo que se utilizará mujoco200 de 64 bits. Se recomienda extraer el zip en la carpeta del usuario con nombre ".mujoco" como se muestra a continuación. Ej. C:/Users/username/.mujoco/. Se podrá utilizar MuJoCo una vez pegado el archivo txt en la carpeta ".mujoco" y otra vez en "bin".

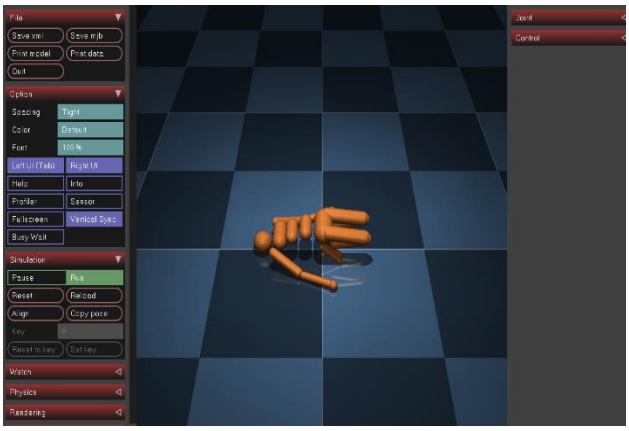

Figura 3. Renderizado del modelo Humanoid

La carpeta MuJoCo viene con unos modelos y códigos ejemplo que se pueden usar para tener una idea general de cómo funciona MuJoCo. Para ejecutar una simulación es necesario situarse en la carpeta bin con el terminal y correr la línea de código: simulate /path-to-xml/archivo.xml. En la Figura 3 se puede observar el interfaz del simulador Mujoco 2.0. Se pueden realizar simples interacciones con el modelo usando el ratón, pero para realizar controles más complejos sería necesario modificar y recompilar el archivo fuente *simulation*, pero para este proyecto se utilizará librerías Python compatibles con MuJoCo 2.0 para implementar el entorno y algoritmos de aprendizaje por refuerzo profundo.

MuJoCo puede cargar archivos XML en su formato nativo MJCF, así como en formato URDF.

Los modelos MJCF pueden representar dinámicas complejas con un amplio rango de características y elementos.

La parte principal del archivo MJCF es un árbol XML creado con elementos "cuerpo" (*body*) anidados. Los cuerpos son elementos usados para construir un árbol cinemático. Un modelo MuJoCo puede consistir de uno o varios árboles cinemáticos, que pueden tener bases flotantes y objetos aislados. Los cuerpos cuentan con matrices de masa e inercia, pero no propiedades de geometría. Son sólo sistemas de coordenadas con propiedades de inercia. Internamente, cada cuerpo tiene un sistema de coordenadas local con centro en el centro de masa y alineado con los ejes principales de inercia. Siempre hay que definir un cuerpo "*world*" que engloba todos los demás cuerpos del árbol cinemático. Dentro del cuerpo se puede definir los siguientes elementos más relevantes:

- Las geometrías (*geom*) se adjuntan de manera rígida al cuerpo dentro del cual se ha definido. Múltiples geometrías pueden adjuntarse al mismo cuerpo. Durante la simulación, las geometrías determinan las propiedades de colisión y apariencia del cuerpo.
- Los elementos *site* son unos tipos de geometrías simplificados. Pueden definirse para representar posiciones de interés relativos al cuerpo.
- Las articulaciones (*joint*) se usan para crean grados de libertad entre un cuerpo y sus elementos padre. En caso de que no se definan las articulaciones entre cuerpos padre e hijo, estos cuerpos aparecerán soldados. MuJoCo cuenta con cuatro tipos de articulaciones: *slide, hinge, ball y free* que hacen referencia a articulaciones deslizante, de bisagra, esférica y libre respectivamente. Una articulación libre se utiliza para definir una base flotante.
- La inercia (*inertia*) especifica las propiedades de masa e inercia del cuerpo. Si no se define este elemento, las propiedades se infieren a partir de la geometría.

MuJoCo proporciona un modelo de actuadores flexible. Todos los actuadores son SISO (*single-input-single-output*). La entrada al actuador *i* es un control escalar u_i especificado por el usuario. La salida es una fuerza escalar p_i. Los componentes que determinan el funcionamiento del actuador son: Transmisión (*transmission*), dinámicas de activación (*activation dynamics*), y generación de fuerza (*force generation*). Se pueden fijar de manera independiente para una mayor flexibilidad o usar métodos simplificados para instanciar actuadores comunes. Los tipos de actuadores más comunes: *motor, velocity* y *positio*n, cuyas señales de control son par, velocidad y posición respectivamente.

II. Python

Durante los últimos años, Python se ha convertido en uno de los lenguajes de programación por defecto a la hora de desarrollar proyectos de aprendizaje automático (ML) e inteligencia artificial (AI) gracias a 1) su simplicidad y consistencia, 2) acceso a una gran librería y marcos de trabajo para ML y AI, 3) flexibilidad, 4) plataforma independiente, y 5) una amplia comunidad de usuarios.

- Simplicidad y consistencia: Python es un lenguaje de programación fácil de aprender y entender. Mientras que los algoritmos detrás del ML y AI son complejos, la simplicidad de Python permite que los desarrolladores centren sus esfuerzos en resolver problemas de ML en vez de matices técnicos del lenguaje.
- Extensa selección de librerías y marcos de trabajo: La implementación de algoritmos de

ML y AI puede ser complicado y requiere de una gran cantidad de tiempo. Python, con su gran extensión de librerías y marcos de trabajo, ayuda a reducir el tiempo de desarrollo. Algunas de las librerías más utilizadas son:

- Tensorflow, Pytorch, y Keras para ML.
- NumPy para el análisis de dato y computación científica de alto rendimiento.
- OpenCV para visión artificial.

- Plataforma independiente: La independencia de plataforma permite sistemas desarrollados en una máquina pueda ser transferidas a otras máquinas sin necesidad de modificaciones. Python cuenta con suporte en plataformas Linux, Windows, y macOS, permitiendo que su software pueda ser fácilmente distribuido y utilizado en esos sistemas operativos.
- Una gran comunidad y popularidad: La comunidad de AI de Python ha crecido en todo el mundo. Existen multitud de foros de Python con intercambios activos relacionados con soluciones de ML. Para cualquier problema que uno pueda tener, hay una posibilidad muy alta de que alguien ya lo haya intentado o resulto. Por lo que uno puede encontrar consejos y orientación de otros desarrolladores fácilmente.

Para el desarrollo de este proyecto se ha utilizado Python 3.6 con las siguientes librerías:

- Mujoco-py: Permite la interacción con MuJoCo utilizando Python.
- Pytorch: Pytorch junto con TensorFlow son dos de las librerías más utilizadas para ML. Pytorch es una librería de ML gratis y de código abierto. Pytorch es simple y fácil de usar. Comparado con TensorFlow, se requiere menos tiempo para ajustarse a Pytorch ya que se siente como una extensión del marco de trabajo de Python. Es fácil de depurar dado que se tiene la opción de acceder al dato actual en vez de referencias simbólicas. Además, Pytorch ofrece una gran comunidad de apoyo. Su homepage cuenta con documentación detallada y actualizaciones de sus librerías bien explicadas. Gracias a esto, existe una gran contribución por parte de la comunidad de usuarios de Pytorch, dando lugar a numerosos proyectos disponibles que uno puede acceder. También es el marco de trabajo que los investigares más utiliza gracias a su simplicidad y buen API[4].
- Gym: Es un interfaz de código abierto para proyectos de aprendizaje por refuerzo. Cuenta con una colección de entornos que uno puede utilizar para implementar algoritmos de aprendizaje por refuerzo.
- Plotly: Es una librería gratis y de código abierto utilizado para hacer gráficas interactivas.
- NumPy: Es la librería fundamental para la computación científica en Python.
- OpenCV-Python: Es una librería de código abierto dedicado a visión artificial.

IV. IMPLEMENTACIÓN

La mayor parte de los algoritmos se pueden clasificar en dos grupos: 1) basados en política (policy-based) como es el caso de la política de gradientes o 2) basados en valor (value-based) como el Q-learning. El Actor-Critic trata de combinar las ventajas de los dos tipos de algoritmos anteriores. La idea del Actor-Critic es de dividir la red neuronal en dos partes, una para calcular las acciones en base al estado y la otra para computar la función de valor.

Mientras que el Q-learning cuenta con una tabla de Q-valores en base a pares de estado-acción $Q(s,a)$, en el A3C se obtiene lo que se conoce como Actor-Critic. El Actor $pi(a|s)$ representa la probabilidad de realizar una acción para un determinado estado y el Critic $V(s)$ representa la función de valor, que indica la "calidad" de encontrarse en un determinado estado.

Por otro, a diferencia de la política de gradientes, que utiliza la recompensa descontada (Discounted Reward) $R = \gamma(r)$ para determinar si una acción ha sido buena o mala, el A3C emplea el concepto de Advantage para estimar, no sólo si una acción ha sido satisfactoria, sino también cuanto mejor ha salido de lo esperado. La función Advantage $A(s,a)$ se obtiene como la diferencia entre $Q(s,a)$ y $V(s)$:

$$A(s,a) = Q(s,a) - V(s)$$

Como en el Actor-Critic no se obtiene el Q-valor, se utiliza la recompensa descontada R como una estimación de $Q(s,a)$, lo que resulta en:

$$A(s) = R - V(s)$$

Por último, la parte asíncrona (Asynchronous) del A3C se refiere que algoritmo puede hacer uso de múltiples agentes entrenando en paralelo. Cada uno de estos agentes trabaja con una copia del entorno independiente de los demás y se comparte la experiencia en una red global. Esto no solo acelera el proceso de entrenamiento dado que hay una mayor exploración del entorno, sino que mejora los

resultados obtenidos ya que los agentes se pueden ayudar mutuamente a salir de posibles estados subóptimos.

Para implementar un agente A3C es necesario definir lo siguiente:

- Modelo de la red neuronal que relacione cada estado con un valor V (Critic) y política π (Actor).
- Función de pérdidas para el Actor y el Critic.
- Función de optimización para optimizar las pérdidas totales y obtener los parámetros θ de la red neuronal.

La arquitectura de la red neuronal se muestra en la Figura 4. Se trata de una red convolucional que acepta como entrada una imagen RGB de 64x64 y devuelve como salidas las acciones de las articulaciones. El modelo está compuesto por dos capas convolucionales, una capa completamente conectada y una capa recurrente LSTM, seguido de una última capa completamente conectada del cual se predice la política y la función de valor. La primera capa convolucional cuenta con 16 kernels de 8x8 y un stride de 4x4, resultando en un mapa de características de dimensión 15x15x16. La segunda capa convolucional usa 32 kernels de 5x5 y stride de 2x2, dando una salida de 6x6x32. La capa completamente conectada está formada por tanto de 1152 unidades (6x6x32). La capa LSTM cuenta con 128 estados ocultos.

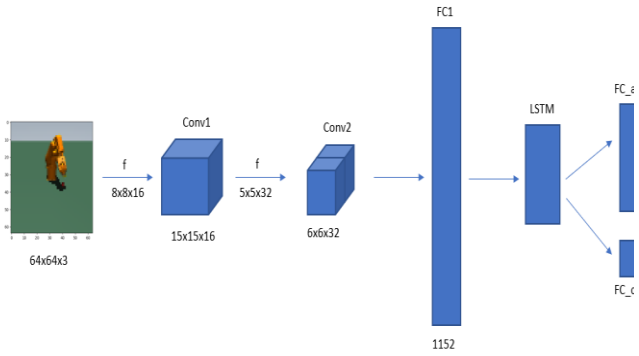

Figura 4. Arquitectura de la red neuronal del agente A3C

Las pérdidas se obtienen como la suma de las pérdidas del Actor y las pérdidas del Critic.

Las pérdidas del Critic y Actor se determinan según:

$$L_{critic} = \left(R - V(s)\right)^2 = A(s)^2$$
$$L_{actor} = -log(\pi(a|s)A(s)$$

En [5] incluye un término de entropía $H(\pi) = \sum log(p(x))p(x))$ en el cálculo de pérdidas del Actor para favorecer la exploración, resultando en:

$$L_{actor} = -log(\pi(a|s)A(s) - \beta H(\pi))$$

, donde β es el peso de la entropía.

Existe una variante de estimación del Advantage denominado GAE (Generalized Advantage Estimation) que reduce la varianza de Advantage estimado[6].

$$A_{gae} \leftarrow A_{gae}\lambda\gamma + TD_{residual}$$
$$TD_{residual} = r + \gamma V(s') - V(s)$$

, donde λ ajusta la compensación varianza-bias, y γ es el factor de descuento.

Teniendo en cuenta las pérdidas del Actor y del Critic, se tiene que las pérdidas totales $L(\theta)$ son:

$$L(\theta) = A(s)^2 - log(\pi(a|s)A_{gae}(s) - \beta H(\pi)$$

Para un método Actor-Critic convencional, se distinguen una red neuronal para el Actor con parámetros θ y otra red neuronal para el Critic con parámetros w. Usando el método de descenso por gradiente, los parámetros w y θ se actualizarían según:

$$\theta \leftarrow \theta + \alpha A(s)\nabla_\theta log(\pi(a|s) + \beta\nabla_\theta H(\pi)$$
$$w \leftarrow w - \alpha\nabla_w A(s)^2$$

En este proyecto, se emplea el método de optimización RMSprop sobre las pérdidas totales:

$$g = \nabla_\theta L(\theta)$$
$$sq = \alpha\, sq + (1-\alpha)g^2$$
$$\theta \leftarrow \theta - \eta \frac{g}{\sqrt{sq + \epsilon}}$$

, donde α es la decadencia, η el ratio de aprendizaje, y el factor ϵ para evitar inestabilidades por divisiones con valores próximos a 0.

V. RESULTADOS

Se han entrenado varios agentes utilizando diferentes funciones de recompensa para observa el efecto que tienen en el aprendizaje. Los agentes se han entrenado durante 5 millones de pasos. Cada 50 mil pasos, se ha evaluado el modelo durante 30 episodios de 50 pasos cada episodio para observar la evolución del aprendizaje. Las funciones de recompensa utilizadas son:

1.a) El agente recibe una recompensa positiva (+1) si *d*, distancia entre las pinzas y el objetivo, está por debajo de un umbral y 0 si está por encima de este límite.

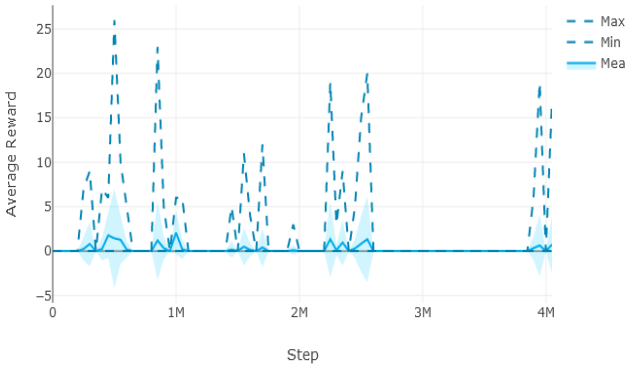

Figura 5. Evaluación recompense 1.a

1.b) El agente es penalizado (-1) si la distancia es mayor del umbral y 0 si está dentro del límite establecido.

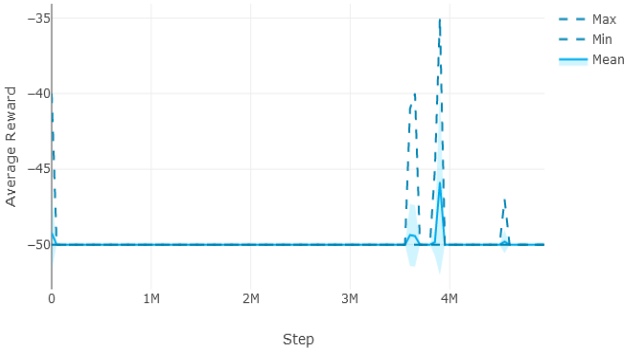

Figura 6. Evaluación recompense 1.b

2.a) La recompensa es positiva y es inversamente proporcional a la distancia entre objetivo y pinza:

$$recompensa = \left(\frac{1}{d + 0.01}\right) \cdot 0.1$$

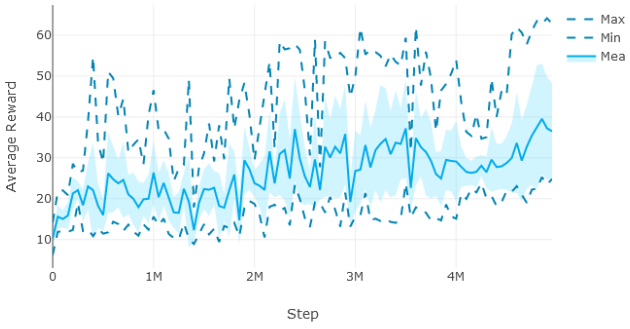

Figura 7. Evaluación recompense 2.a

2.b) La recompensa es negativa y es proporcional a la distancia al cuadrado:

$$recompensa = -(2d)^2$$

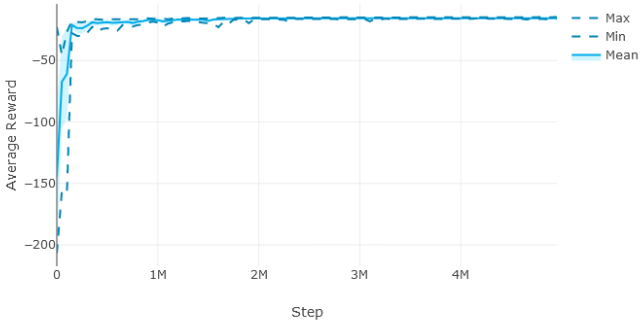

Figura 8. Evaluación recompense 2.b

Se pueden observar en las Figura 5 y 6 que el aprendizaje es despreciable tras los 5 millones de pasos. Aunque hay episodios puntuales donde se alcanzan unas recompensas notables, la recompensa media obtenida por periodo de evaluación (30 episodios) es próximo a los mínimos posibles.

En la Figura 7 y 8 se muestra el aprendizaje con las funciones de recompuestas continuas positiva (2.a) y negativa (2.b) respectivamente. Se puede observar cómo las recompensas medias han ido creciendo a lo largo del tiempo. Para el caso 2.a, los máximos y mínimos obtenidos en una sesión de evaluación de 30 episodios presentan una diferencia notable, mientras que para el caso 2.b se puede observar que la recompensa converge en un punto.

Función recompensa	Posición inicial del robot durante entrenamiento	Posición inicial del robot durante evaluación	Nº episodios exitosos (Total 30)
1.a	Constante	Constante	0
1.b	Constante	Constante	0
2.a	Constante	Constante	24
2.b	Constante	Constante	29
2.b	Constante	Aleatorio	15
2.b	Aleatorio	Constante	29
2.b	Aleatorio	Aleatorio	27

Figura 9. Tabla comparativa

Utilizando el modelo obtenido de entrenar el agente con la función recompensa del caso 2.b, se ha renderizado la simulación para observar su funcionamiento. Considerando como episodio exitoso si el robot es capaz de acercar la pinza a una distancia menor de 5cm del objetivo, se han obtenido unas probabilidades de éxito del 96.67 % asumiendo que el robot empieza desde la misma posición inicial. No obstante, la probabilidad de éxito de este modelo baja a los 50 % si se aleatoriza la posición inicial del robot en cada episodio. Se ha entrenado otro modelo con la misma función de recompensa, pero con posiciones iniciales del robot aleatorios. El resultado obtenido es satisfactorio, con unas probabilidades de éxito del 90 % con posiciones iniciales aleatorios, y del 96.67 %

empezando desde la misma posición inicial. Se ha podido obtener unas precisiones similares cambiando la forma y color del objetivo considerando que solo hay un objeto en el entorno. Al colocar más de un objeto en el entorno, el robot tiende a acercarse al objetivo de color rojo, dado que es el color del objeto utilizado en el entrenamiento.

VI. CONCLUSIÓN

Desde alcanzar rendimientos sobrehumanos en los videojuegos de Atari hasta conseguir que un modelo humano 3D aprenda desde cero a correr mientras esquiva obstáculos, el aprendizaje por refuerzo ha demostrado la multitud de posibilidades que ofrece como método de aprendizaje automático. En esta tesis se ha podido comprobar el potencial de estos algoritmos mediante la implementación de un algoritmo A3C donde un modelo 3D del brazo robótico IRB120 ha sido capaz de aprender la tarea de alcanzar un objetivo en su área de trabajo a partir de imágenes RGB de 64 x 64 del entorno y una función de recompensa.

Aunque es cierto el gran potencial que ofrece el aprendizaje por refuerzo, se requiere demasiado tiempo y recursos computacionales en entrenar un agente. En este proyecto se han entrenado varios agentes con las diferentes funciones de recompensa mencionadas. Cada episodio de entrenamiento ha requerido de 5 millones de pasos, que se traduce en 330 minutos aproximadamente con la capacidad computacional de la máquina que se ha utilizado para el desarrollo de este proyecto. Se ha comprobado la importancia de la función de recompensa en cuanto al tiempo de entrenamiento requerido y convergencia. Es necesario que la función de recompensa aporte una realimentación gradual para que el agente sepa que se está mejorando y acercándose al objetivo, acelerando así el aprendizaje.

Aunque se ha conseguido entrenar un agente en la realización de la tarea especificada anteriormente con una alta precisión, se aprecian ciertos comportamientos indeseados: 1) El robot oscila en torno al objetivo y 2) termina empujando la mesa si falla en acercarse al objetivo.

Es necesario resolver en especial el segundo problema si se procediese a transferir el agente entrenado al robot físico para evitar daños. Para ello, es necesario definir nuevos términos en la función de recompensa que se utilizarán como referencia para penalizar los comportamientos indeseados. Una posible solución consiste en monitorizar la coordenada z del punto terminal del robot en bases del robot y penalizar los estados donde dicha coordenada queda por debajo de un valor límite. En cuanto al primer problema, se puede monitorizar la velocidad de las articulaciones y penalizar cuando la velocidad es mayor que cero si se ha cumplido la condición de distancia. Una vez resuelto estos problemas, se procedería a transferir el aprendizaje al robot físico.

Aunque los resultados obtenidos en el entorno simulado son satisfactorios, no significa que se consiga los mismos resultados en el mundo real debido a las discrepancias entre el modelo simulado y el modelo real. Este problema se conoce como reality gap. Por ello, es necesario realizar fine-tuning, que consiste en utilizar el modelo pre-entrenado en simulación como punto de partida para adaptarlo al sistema real. No obstante, aunque la mayor parte del entrenamiento se ha realizado en el entorno simulado, fine-tuning en entorno real puede aun así requerir de un tiempo considerable al utilizarse la misma estructura de red neuronal. Además, fine-tuning puede conllevar al olvido catastrófico que ocurre cuando parámetros importantes de la red neuronal se cambien para adaptarse a nuevos datos, comprometiendo su habilidad para encargarse de datos antiguos. Por ello, sería interesante implementar una red neuronal progresiva [7], que permite la transferencia de aprendizaje mediante conexiones laterales entre redes neuronales como se muestra en Figura 10,evitando así el olvido catastrófico, ya que cada columna se actualiza de forma local al realizar fine-tuning. Además, cada columna es independiente de las demás, por lo que se puede reducir la red neuronal para acelerar el entrenamiento.

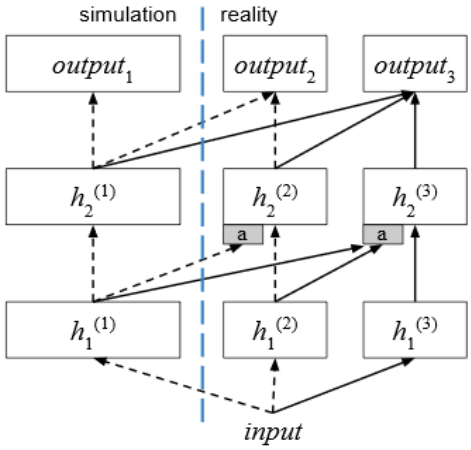

Figura 10. Red neuronal progresiva

VII. BIBLIOGRAFÍA

[1] Richard S. Sutton and Andrew G. Barto.Reinforcement Learning: An Introduction.The MIT Press, second edition, 2018.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Anto-noglou, Daan

Wierstra, and Martin A. Riedmiller. Playing atari with deep rein-forcement learning.CoRR, abs/1312.5602, 2013.

[3] Andrei A. Rusu, Matej Vecerík, Thomas Rothörl, Nicolas Heess, Razvan Pascanu,and Raia Hadsell. Sim-to-real robot learning from pixels with progressive nets.CoRR, abs/1610.04286, 2016.

[4] Horace He. The state of machine learning frameworks in 2019.The Gradient, 2019.

[5] Yuandong Tian Yuxin Wu. Training agent for first-person shooter game with actor-critic curriculum learning.ICLR, 2017.

[6] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In Proceedings of the International Conference on Learning Representations (ICLR),2016.

[7] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. CoRR, abs/1606.04671, 2016.