ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

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

Autor: Lixiang Dong

Directores: Lucía Güitta López

Álvaro Jesús López López

Madrid, Julio 2020

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título **Integración y aplicación de técnicas de aprendizaje por refuerzo al robot IRB120 en el entorno virtual de MuJoCo**

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2019/20 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Lixiang Dong Fecha: 14 / 07 / 20

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Lucía Güitta López Fecha: 20 / 07 / 20

Fdo.: Álvaro Jesús López López Fecha: 2 / 7 / 20

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

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

Autor: Lixiang Dong

Directores: Lucía Güitta López

Álvaro Jesús López López

Madrid, Julio 2020

INTEGRACIÓN Y APLICACIÓN DE TÉCNICAS DE APRENDIZAJE POR REFUERZO AL ROBOT IRB120 EN EL ENTORNO VIRTUAL DE MUJOCO

Autor: Dong, Lixiang.
Director: Güita López, Lucía;
López López, Álvaro Jesús.
Entidad Colaboradora: ICAI -Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

1. Introducción

El aprendizaje por refuerzo se considera el tercer paradigma del aprendizaje automático junto con el aprendizaje supervisado y el aprendizaje no supervisado. Es una clase de algoritmos en el campo del aprendizaje automático que permite a un agente aprender a cómo comportarse en un entorno donde la única realimentación consta de una señal de recompensa escalar, la cual indica cómo de bien lo está haciendo en el momento inmediato. El objetivo del agente consiste en ejecutar acciones que maximice la recompensa en el largo plazo o retorno.

Si bien las técnicas de aprendizaje por refuerzo están siendo impulsadas por diversos grupos investigadores en varios ámbitos, sobre todo en los juegos de Atari y la robótica, la complejidad del movimiento de los brazos robóticos puede parecer a priori un hándicap para aplicar este proceso de aprendizaje que requiere de numerosos episodios para que el agente explore y aprenda a partir de prueba y error. Sin embargo, mediante el entrenamiento en entornos simulados y su posterior transferencia al mundo real se evitan los riesgos asociados a movimientos del robot que puedan resultar en posiciones singulares o en daños al medio y se favorece un aprendizaje más rápido ya que se infieren los parámetros desde el modelo virtual y no se está limitado por restricciones físicas.

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

2. Metodología

El modelado 3D del robot IRB120 se realizará utilizando MuJoCo, una herramienta de simulación de física diseñado para facilitar la investigación y desarrollo en áreas donde sistemas de compleja dinámica requieren de simulaciones rápidas y precisas, como es el caso de la robótica.

El lenguaje de programación utilizado es Python. Gracias a la librería de mujoco_py, se permite la interacción entre Python y MuJoCo.

El algoritmo de aprendizaje por refuerzo que se va a implementar es A3C (Asynchronous Advantage Actor-Critic), utilizando como referencia la red neuronal convolucional expuesta en [1], donde recibe como entrada una imagen RGB de 64x64 para determinar los movimientos de las articulaciones hasta alcanzar el objetivo.

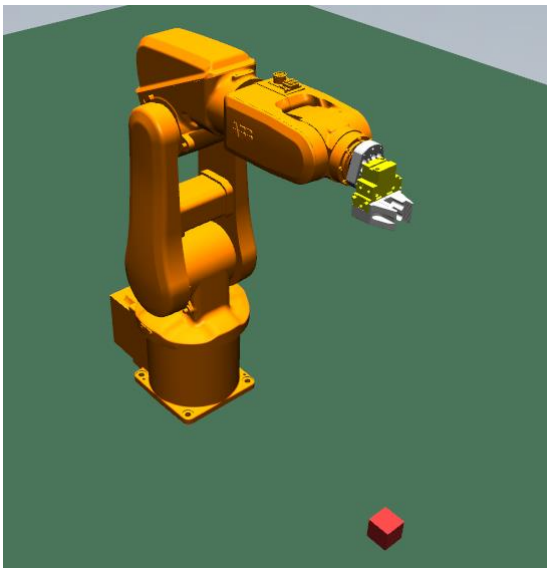


Figura 1. Modelo MuJoCo de IRB120 renderizado

3. 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 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.

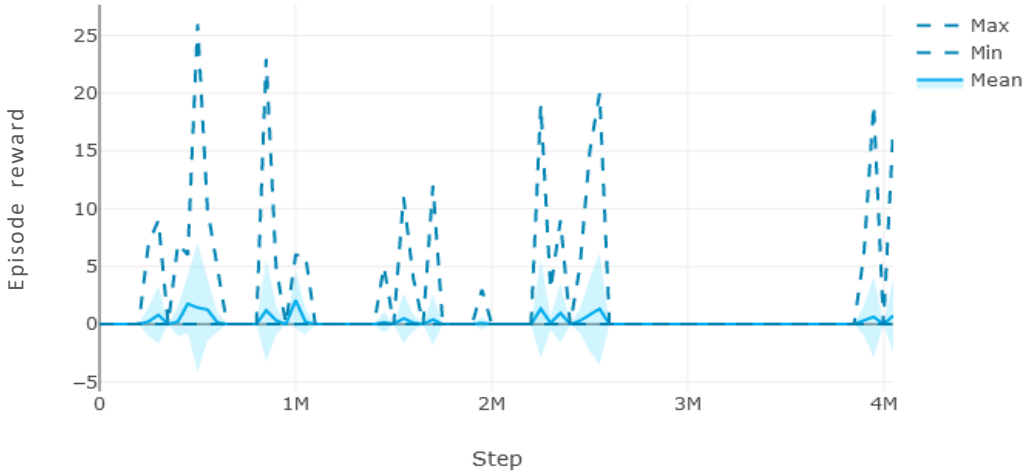


Figura 2. Evaluación recompensa 1.a

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

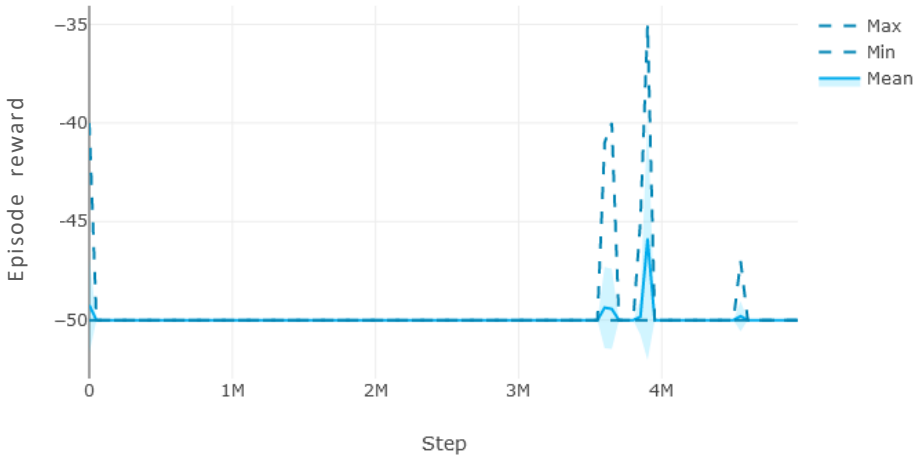


Figura 3. Evaluación recompensa 1.b.

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

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

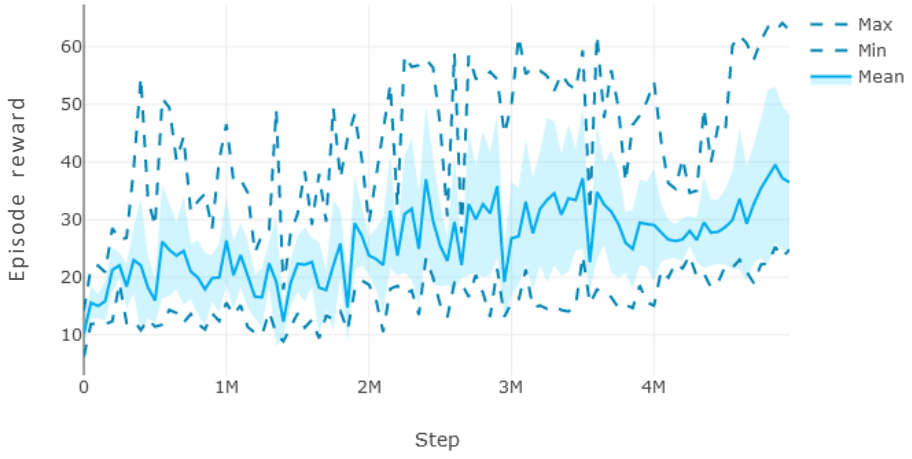


Figura 4. Evaluación recompensa 2.a

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

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

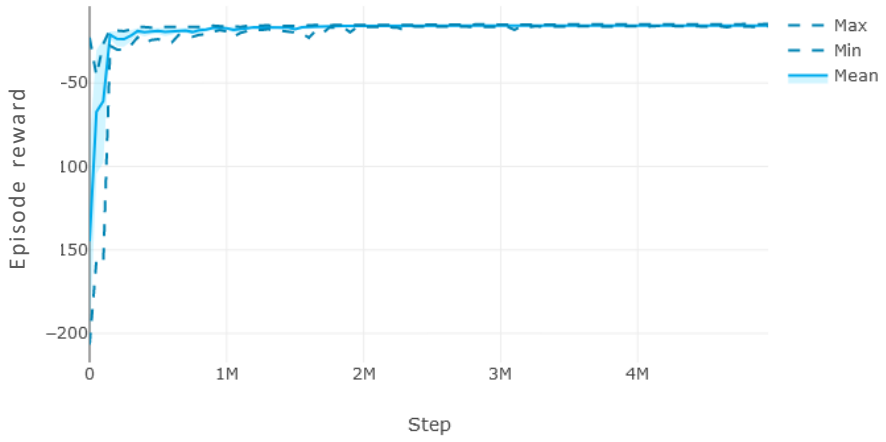


Figura 5. Evaluación recompensa 2.b

Se pueden observar en las Figuras 2 y 3 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 4 y 5 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.

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.

4. Conclusiones

En este proyecto se ha podido comprobar el potencial del aprendizaje por refuerzo mediante la implementación de un algoritmo de aprendizaje por refuerzo (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 simples imágenes RGB de 64 x 64 y una función de recompensa.

Aunque es cierto el gran potencial que ofrece el aprendizaje por refuerzo, se requiere demasiado tiempo en entrenar un agente. En este proyecto se han entrenado varios agentes con las diferentes funciones de recompensa. Cada periodo de entrenamiento ha requerido de 5 millones de pasos, que se traduce en unos 330 minutos 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.

5. Referencias

[1] 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.

INTEGRATION AND IMPLEMENTATION OF REINFORCEMENT LEARNING TECHNIQUES TO THE ROBOT IRB120 IN A VIRTUAL ENVIRONMENT USING MUJOCO AS THE PHYSICS ENGINE

Author: Dong, Lixiang.
Supervisors: Güita López, Lucía;
López López, Álvaro Jesús.
Collaborating Entity: ICAI -Universidad Pontificia Comillas.

ABSTRACT

1. Introduction

Reinforcement learning is considered the third paradigm of machine learning along with supervised learning and unsupervised learning. It is a class of algorithms in the field of machine learning that allows an agent to learn how to behave in an environment with a scalar reward as feedback. The objective of the agent is to execute actions that maximize the long-term reward or return.

Although reinforcement learning techniques are pushed forward by many research groups in many fields, such as Atari games and robotics, the movement complexity of robotics arms seems to be a problem to implement learning techniques that require numerous episodes for the agent to explore and learn from trial and error. Nonetheless, by training in a simulated environment and its later transfer to the real world, the risks associated with the movement of physical robot can be avoided. Using a simulated environment also increases the learning speed since it is not limited by physical constraints and the parameters can be inferred from the virtual model.

In this thesis a reinforcement learning algorithm will be implemented using a 3D model of the IRB120 robot manipulator to carry out a reach target task.

2. Methodology

The 3D modelling of the IRB120 robot will be done using MuJoCo, a physics engine designed to facilitate research and development in fields where complex dynamic systems require of fast and accurate simulation, such as robotics.

The programming language used is Python. Thanks to the mujoco_py library, interfacing between Python and MuJoCo is allowed.

The reinforcement learning algorithm that is going to be implemented is A3C (Asynchronous Advantage Actor-Critic), using the convolutional neural network presented in [1] as reference, where 64x64 RGB image is used as input to determine the joint movements to reach the target.

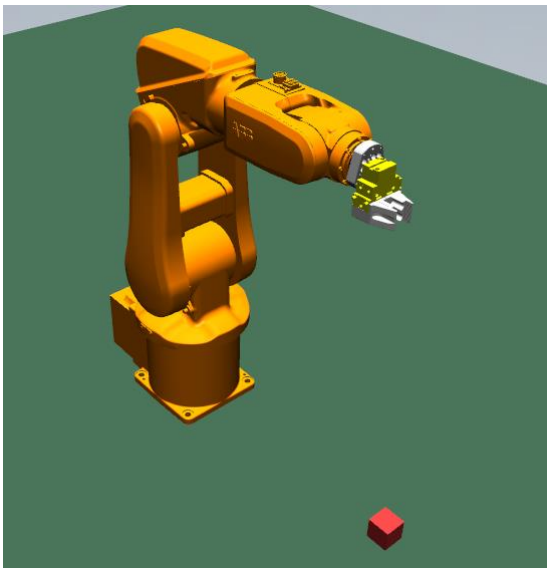
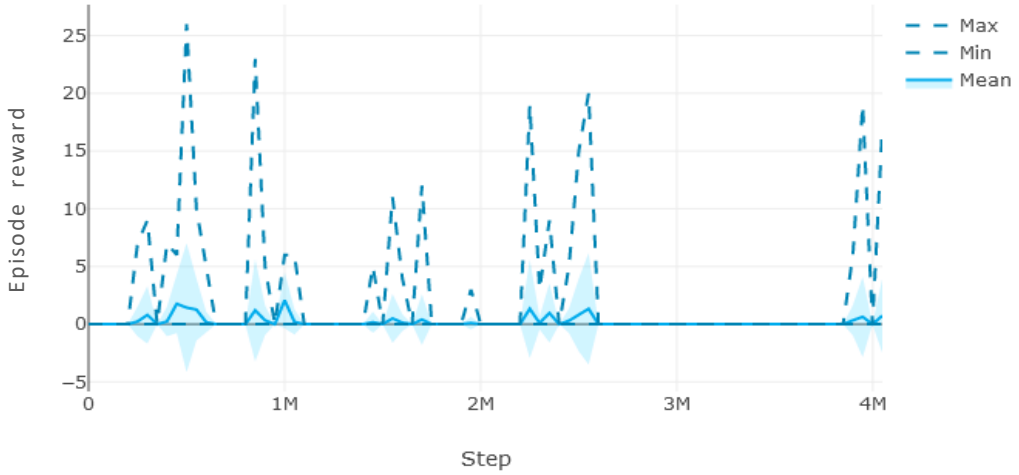


Figure 1. Rendered IRB120 MuJoCo model

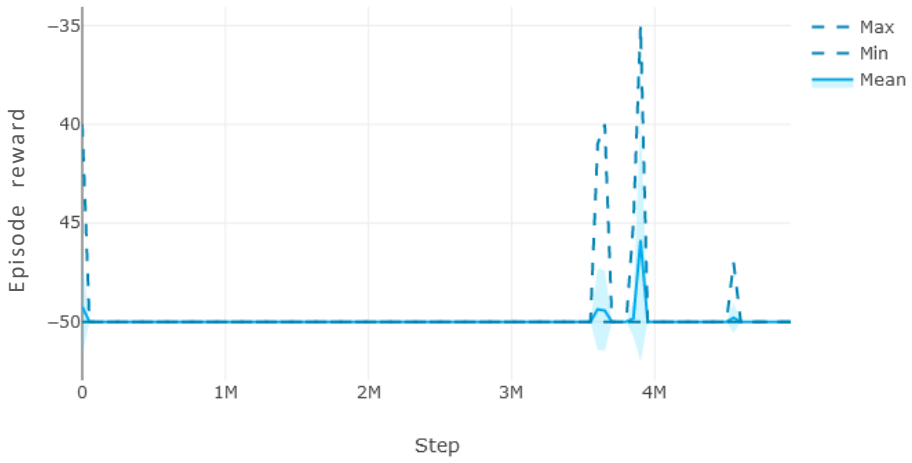
3. Results

Several agents are trained using different reward functions to analyze the its effect in the learning process. The agents are trained for 5 million steps. Every 50 thousand steps, the model is evaluated for 30 episodes of 50 steps each episode to observe the learner's development. The reward functions are the following:

1.a) The agent receives positive reward (+1) when the distance *d* between grip and target is below threshold, and 0 otherwise.

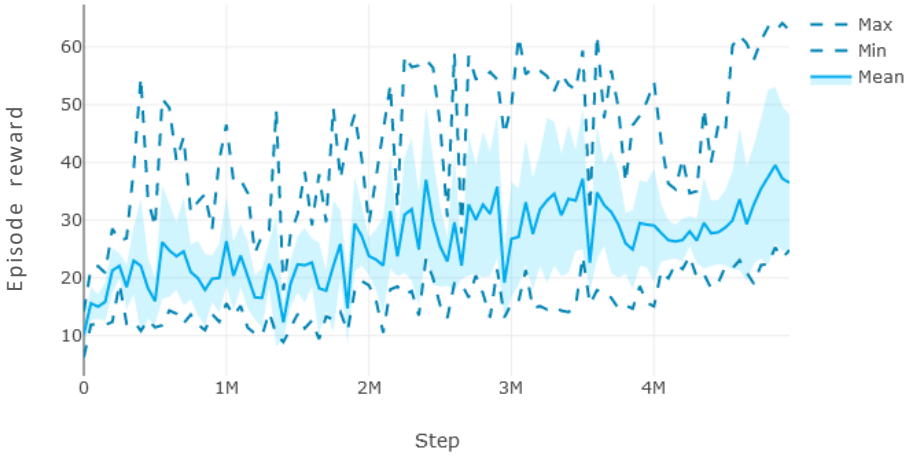


1.b) The agent is penalized (-1) is the distance is greater than the threshold, and 0 otherwise.



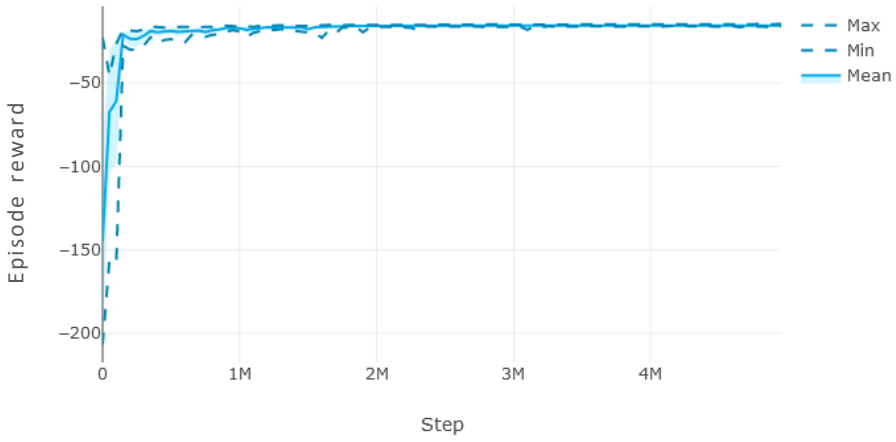
2.a) The reward is positive and inversely proportional to the distance between target and grip:

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



2.b) The reward is negative and proportional to the distance squared:

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



From Figures 2 and 3, where sparse reward is used, the learning is almost negligible after 5 million training steps. Although there are isolated cases where reasonable rewards are achieved, the average reward obtained per evaluation (30 episodes) is close to the minimum possible values.

Figures 4 and 5 show the learning when using shaped reward functions. The average reward shows an increasing trend. There is a noticeable difference between the maximum and minimum rewards obtained for the case 2.a, while the reward for 2.b converges to a stable value.

To observe the performance of the model obtained by training the agent using the reward function 2.b, the simulation is rendered for 30 episodes. By considering a successful episode when the robot places the grip at a distance below 5 cm from the target, it reaches a 96.67 % success rate when the robot resets to the same initial joint angles for every episode. However, the success rate drops to 50 % when the initial joint angles are randomized. A new model is trained using the same reward function and random initial joint angles. The obtained result is satisfactory with 90 % success rate when evaluated with random initial joint angles, and 96.67 % when starting with the same joint angles. Same success rates are obtained when

changing the shape and color of the target considering that there is only one target in the environment.

When several targets are placed, the robot tries to reach the red colored one, since it is the color used during training.

4. Conclusions

The implementation of a reinforcement learning algorithm (A3C) where a 3D model of the robotic manipulator IRB120 is capable of learning to reach a target in its workspace by using simple 64x64 RGB images and a reward function shows the great potential of reinforcement learning.

Although reinforcement learning shows great possibilities, it still requires a lot of time to train an agent using such algorithms. In this project, several agents have been trained using different reward functions. Each training period required 5 million steps, which translates to 330 minutes based on the computations offered by the machine used to develop this project. This project showed the importance of the reward function when it comes to the time required for training and convergence. It is necessary for the reward function to provide gradual feedback to the agent so that it can learn whether it is improving and getting closer to the target, accelerating hence the learning process.

5. References

[1] 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

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Actualmente, los robots están siendo ampliamente utilizados en la industria. En la mayoría de los centros de fabricación, los robots están sustituyendo a las personas para realizar tareas repetitivas y pesadas. Es sencillo introducir robots en los centros de fabricación gracias a que se tratan de entornos relativamente estáticos y fáciles de controlar y monitorizar. A pesar de la simplicidad y estatismo de estos entornos, la implementación de los algoritmos de control y planificación de los movimientos del robot son procesos largos y tediosos, por lo que es todo un reto la implementación en entornos dinámicos y complejos encontrados en la vida cotidiana. Gracias a los avances en la capacidad computacional en los últimos años y su mayor accesibilidad,

el papel del aprendizaje automático «Machine Learning» (ML) está tomando cada vez mayor relevancia ya que permite que los sistemas puedan aprender sin ser programados específicamente para realizar una determinada tarea. Dentro de los algoritmos de ML, cabe destacar los algoritmos de aprendizaje por refuerzo en el ámbito de la robótica.

Aprovechando la infraestructura disponible en el laboratorio de Automatización y Robótica Industrial, donde hay 4 brazos róboticos IRB120 de ABB, se propone desarrollar e implementar algoritmos de aprendizaje por refuerzo para que estos robots puedan realizar tareas sencillas como alcanzar un objetivo aleatoria situado dentro de su entorno de trabajo.

Si bien las técnicas de aprendizaje por refuerzo están siendo impulsadas por diversos grupos investigadores en varios ámbitos, sobre todo en los juegos de Atari [1], la complejidad del movimiento de los brazos robóticos puede parecer a priori un hándicap para aplicar este proceso de aprendizaje que requiere de numerosos episodios para que el agente explore y aprenda a partir de prueba y error. Sin embargo, mediante el entrenamiento en entornos simulados y su posterior transferencia al mundo real se evitan los riesgos asociados a movimientos del robot que puedan resultar en posiciones singulares o en daños al medio y se favorece un aprendizaje más rápido ya que se infieren los parámetros desde el modelo virtual y no se está limitado por restricciones físicas.

1.2. Objetivos del proyecto

En este proyecto se implementará un algoritmo de aprendizaje por refuerzo profundo para entrenar un modelo virtual del robot IRB120 en la realización de la tarea de alcanzar un objetivo dentro de su área de trabajo. 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. 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. Se considera episodio exitoso si la distancia euclídea entre el punto terminal del robot y el objetivo es menor de 5 cm.

Para lograr estos objetivos se procederá a dividir 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 implemen-

tación 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.

1.3. Estructura de la memoria

La estructura del resto de la tesis es la siguiente:

- En el **Capítulo 2** se definen los fundamentos requeridos para el desempeño de este proyecto.
- En el **Capítulo 3** se presenta las herramientas utilizadas para el desarrollo del proyecto.
- En el **Capítulo 4** se explica la implementación del aprendizaje por refuerzo profundo del proyecto, así como alguno de los resultados obtenidos.
- En el **Capítulo 5** se expone las conclusiones obtenidas y las direcciones a seguir para un trabajo futuro.

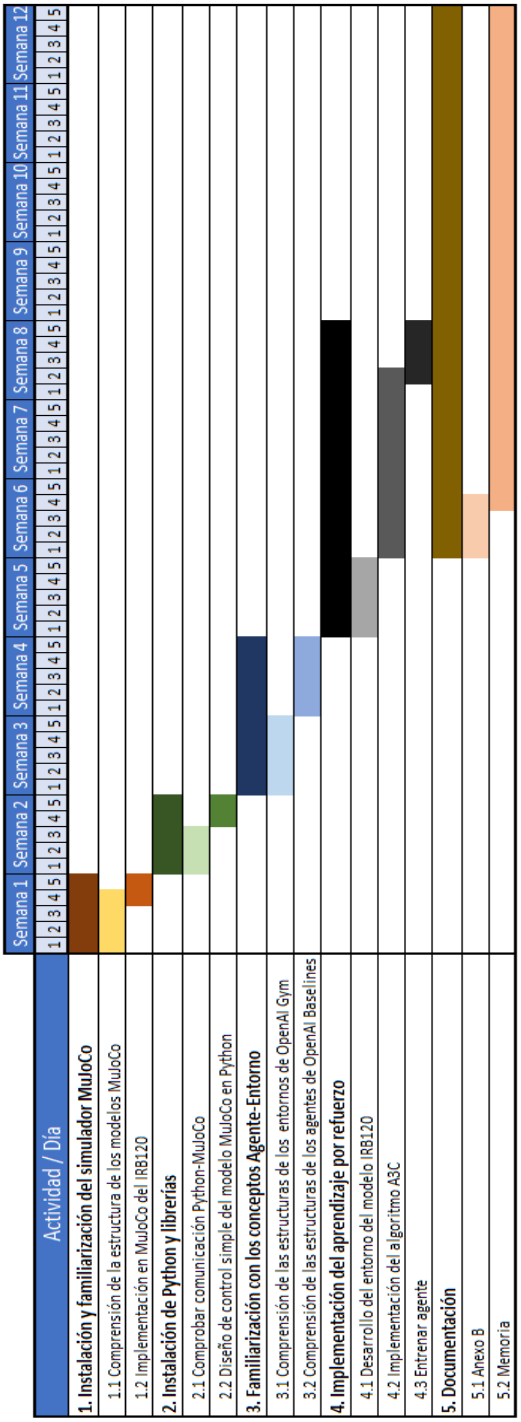


Figura 1.1: Cronograma del Trabajo de Fin de Máster

Capítulo 2

Estado del arte

2.1. Redes neuronales artificiales

Las redes neuronales artificiales (ANN) están compuestas por un gran número de elementos más simples llamadas neuronas. Cada neurona toma decisiones sencillas, pero juntas proporcionan respuestas a problemas complejos como visión artificial o procesamiento de lenguaje natural.

Las redes neuronales superficiales (*shallow*) están compuestas por una capa de neuronas de entrada, una sola capa oculta, y una capa de salida para proporcionar la salida del modelo, como se muestra en la Figura 2.1. Cuando hay varias capas ocultas, se trataría de una red neuronal más compleja y con mayor capacidad de abstracción

[2].

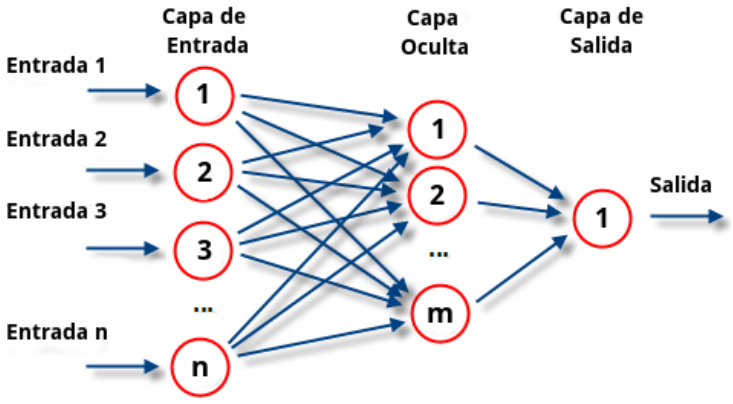


Figura 2.1: Diagrama de una red neuronal

La unidad básica de una red neuronal es la *neurona*. En la Figura 2.2 se puede observar una representación de una neurona que cuenta con m señales de entrada con sus respectivos pesos w.

En cada neurona se realiza el sumatorio ponderado de las señales de entrada más un término de bias b.

$$z = b + \sum_{i=1}^{m} x_i * w_i \tag{2.1}$$

El valor resultante z pasa por una función de activación $f(.)$ para dar una señal de salida o activación a.

$$a = f(z) \tag{2.2}$$

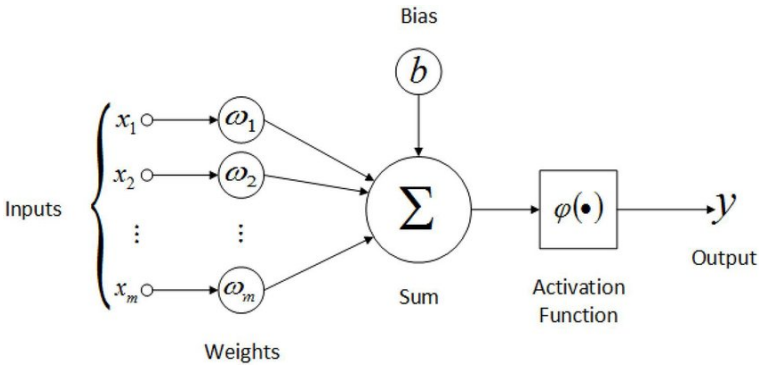


Figura 2.2: Representación de una neurona artificial

Esta señal de salida se sirve de entrada para las neuronas de la siguiente capa de la red neuronal, o la salida estimada $\hat{y}$ si se tratase de una neurona en la capa de salida.

Este proceso de alimentar las señales de entrada a las neuronas de una capa oculta para que las procesen y pase la información a las capas sucesivas hasta obtener la salida estimada se denomina propagación hacia adelante (*Forward propagation).*

Tras la propagación hacia adelante, se utilizan algoritmos de optimización para maximizar o minimizar una función objetivo con el fin de determinar los parámetros de peso y bias. En el caso de algoritmos de aprendizaje supervisado o aprendizaje por refuerzo se utiliza la propagación hacia atrás (*Back propagation)*, para calcular las derivadas con respecto a los parámetros, con algoritmos de optimización como el

descenso del gradiente para minimizar una función objetivo.

Al hablar de minimizar una función objetivo, se hace referencia a la función de pérdidas si se trata de una sola muestra de entrenamiento, y función de coste si se refiere a la media de las pérdidas de toda la muestra de entrenamiento.

2.1.1. Funciones de activación

Las funciones de activación son ecuaciones matemáticas que determinan si una neurona debe ser activada o no [3]. Estas funciones pueden ser lineales o no lineales. Se utilizan funciones de activación no lineales para introducir propiedades para que el modelo pueda generalizar y aprender a partir de datos complejos.

Se escogen las funciones de activación en función de las mejoras computacionales que ofrece a la red neural y del tipo de salida que se quiera obtener.

Sigmoide: La función Sigmoide o función logística suele aparecer en las capas de salida para predecir salidas probabilísticas y se ha empleado exitosamente en problemas de clasificación binaria. Su expresión matemática es la siguiente:

$$f(x) = \left(\frac{1}{1 + exp^{-x}}\right) \tag{2.3}$$

La función Sigmoide fue comúnmente utilizado en redes neuronales dada la sim-

plicidad de su derivada a la hora de aplicar la propagación hacia atrás. Sin embargo, debido a que su derivada tiende a 0 en los extremos, como se puede observar en la Figura 2.3, el aprendizaje es lento o nulo para valores altos de x. Este problema recibe el nombre de desvanecimiento de gradiente (*Vanishing gradient*). Debido a esto, rara vez se ve la función Sigmoide en las capas ocultas, excepto en redes superficiales o capas de salida para la clasificación binaria.

Para solventar el problema de desvanecimiento de gradiente, surgen otras funciones de activación como la unidad lineal rectificada (ReLU).

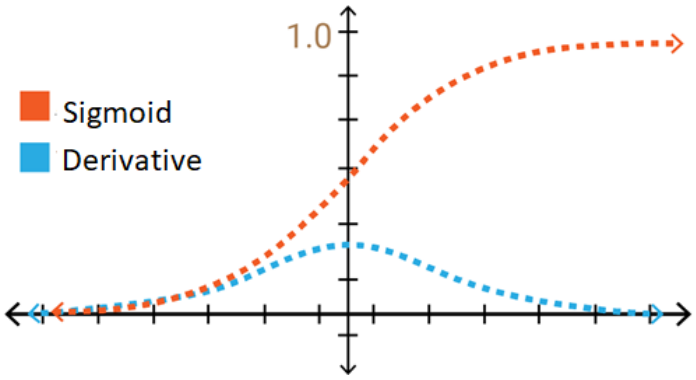


Figura 2.3: Función Sigmoide y derivada

ReLU: La unidad lineal rectificada fue propuesta por Nair y Hinton en 2010, y se ha convertido en la función de activación más utilizada [4]. La función ReLU presenta propiedades del modelo lineal haciendo que sea fácil de optimizar con métodos de descenso del gradiente y al ser una función no lineal permite la propagación hacia atrás.

ReLU es una función a trozos donde $f(x) = x$ para $x \geq 0$ y $f(x) = 0$ para $x < 0$:

$$f(x) = max(0, x) = \begin{cases} x, & x \geq 0 \\ 0, & d < 0 \end{cases}$$

Además de no sufrir los efectos de desvanecimiento del gradiente, al presentar un gradiente constante de 1 para valores mayores que 0, la ventaja principal de ReLU es que garantiza rapidez computacional al introducir *"sparsity"* en el modelo, dado que todo valor por debajo de 0 es rectificado a 0. Así pues, estas neuronas no se activan, resultando en un modelo más simple y eficiente.

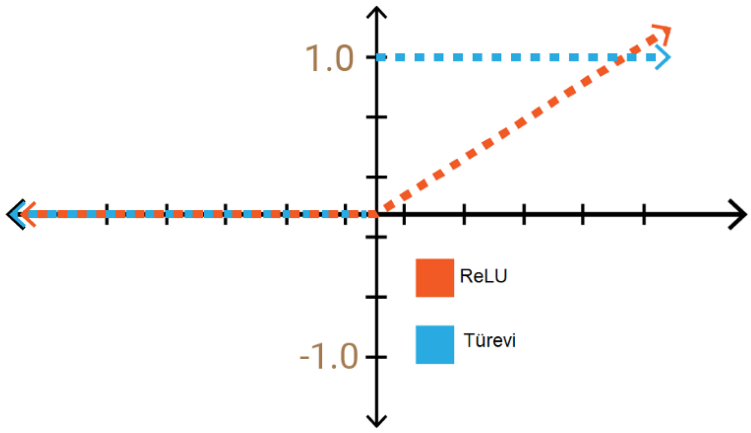


Figura 2.4: ReLU y derivada

Función Softmax: La función Softmax se usa para calcular la distribución de probabilidades de un vector de números reales. Produce una salida con valores entre 0 y 1, cuya suma de probabilidades es igual a 1. Su expresión está dada por:

$$f(x_i) = \frac{e^{x_i}}{\sum_{n=1} e^{x_j}} \tag{2.4}$$

La función Softmax aparece en las capas de salida y se utiliza en modelos de clasificación múltiple, donde devuelve probabilidades de las clases a clasificar. La clase objetivo es aquel con la mayor probabilidad. Presenta una estructura similar a la función Sigmoide, con la diferencia de que Sigmoide es aplicado en clasificación binaria y Softmax en tareas de clasificación múltiple.

2.1.2. Optimización

El objetivo de los algoritmos de aprendizaje automático es intentar minimizar una función de coste $J(w, b)$. Esta función computa la media de las pérdidas de toda la muestra de entrenamiento y mide cómo de bien lo hace los parámetros w y b en toda la muestra de entrenamiento.

Para obtener los parámetros w y b que minimice la función de coste $J(w, b)$, se utilizan algoritmos de optimización como el descenso del gradiente.

Descenso del gradiente

El descenso del gradiente es uno de los algoritmos más populares de optimización y es la forma más común de optimizar redes neuronales.

El descenso del gradiente es una forma de minimizar una función objetivo $J(\theta)$, con parámetros de un modelo $\theta \in \mathbb{R}^d$, mediante la actualización de los parámetros en la dirección opuesta al gradiente de la función objetivo $\nabla_\theta J(\theta)$ respecto de los parámetros [5]. El ratio de aprendizaje α representa el tamaño de los pasos que se toma hasta alcanzar el mínimo local.

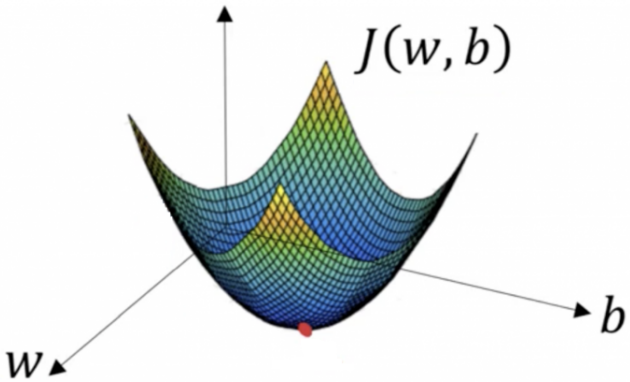


Figura 2.5: Representación simple de la función de coste $J(w, b)$ respecto de los parámetros w y b

$$w = w - \alpha \cdot \nabla_w J(w, b) \tag{2.5}$$

$$b = b - \alpha \cdot \nabla_b J(w, b) \tag{2.6}$$

Es importante escoger un ratio de aprendizaje α adecuado para agilizar el aprendizaje. Si el ratio es demasiado pequeño, el entrenamiento progresará muy despacio ya que los parámetros experimentan cambios minúsculos en cada iteración. Mientras que

para ratios demasiado grandes, puede resultar en grandes oscilaciones e incluso comportamientos de divergencia indeseados, Figura 2.6.

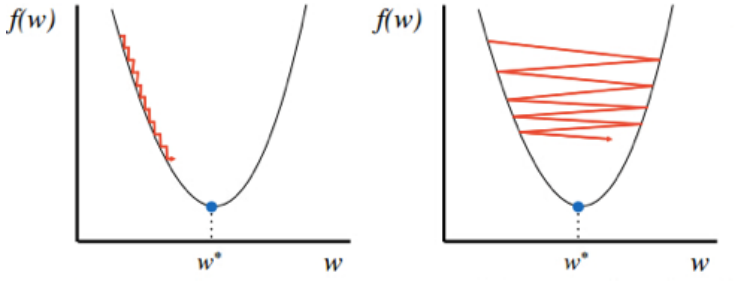


Figura 2.6: Efecto del hiperparámetro ratio de aprendizaje α en el descenso del gradiente

Encontrar un ratio de aprendizaje adecuado es un proceso tedioso y lento, ya que es necesario reentrenar la red neuronal varias veces hasta dar con un valor aceptable. Para evitar tener que modificar manualmente el ratio, surgen métodos de ratio de aprendizaje adaptativos como Adagrad, Adadelta, RMSprop o Adam [6].

2.1.3. Redes neuronales convolucionales

Una de las mayores limitaciones de las redes neuronales tradicionales es su baja capacidad para procesar imágenes debido al elevado número de parámetros, y por tanto, la alta capacidad computacional requerida [7].

Las bases de datos de referencia utilizados en el aprendizaje automático como

MNIST[1] son aptos para la mayoría de redes neuronales tradicionales debido a la baja dimensionalidad de sus imágenes (28 x 28 x 1). Con esta base de datos, cada neurona en la primera capa contendría 784 parámetros de peso, lo cual es asumible para la mayoría de redes neuronales tradicionales. No obstante, al aumentar un poco el tamaño de las imágenes, los parámetros requeridos aumentarían sustancialmente, haciendo inviable estos modelos debido a problemas de tiempo, de capacidad computacional, y de sobreajuste. Las redes neuronales convolucionales (CNN), en cambio, están enfocados específicamente al tratamiento de imágenes.

2.1.3.1. Arquitectura

Las CNNs están compuestas por tres tipos de capas: capas de convolución, capas de reducción y capas completamente conectadas, Figura 2.7.

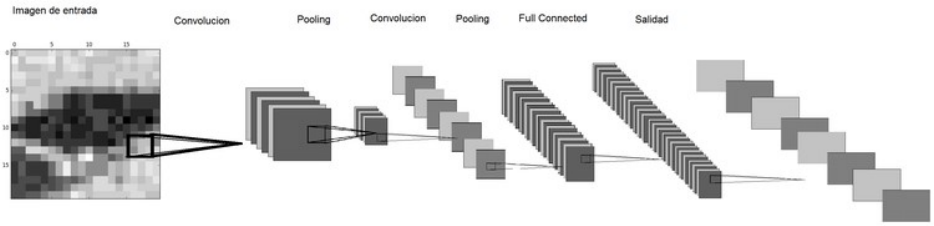


Figura 2.7: Estructura de una red neuronal convolucional

Capa convolucional

Una capa convolucional contiene una serie de filtros o kernels cuyos parámetros se

[1]Base de datos de dígitos escritos a mano

aprenden. Las primeras capas convolucionales se encargan de detectar características de bajo nivel como bordes y formas, mientras que capas más profundas son capaces de detectar características más complejas.

Estos filtros se usan para obtener diferentes mapas de características mediante el producto escalar entre la capa de entrada y los filtros, Figura 2.8. Matemáticamente, el valor de característica $z^l_{i,j,k}$ en la posición (i, j) del mapa de características número k de la capa l se obtiene según la siguiente expresión:

$$z^l_{i,j,k} = w^{l^T}_k x^l_{i,j} + b^l_k \tag{2.7}$$

donde w^l_k y b^l_k son los términos de peso y bias del filtro número k de la capa l respectivamente.

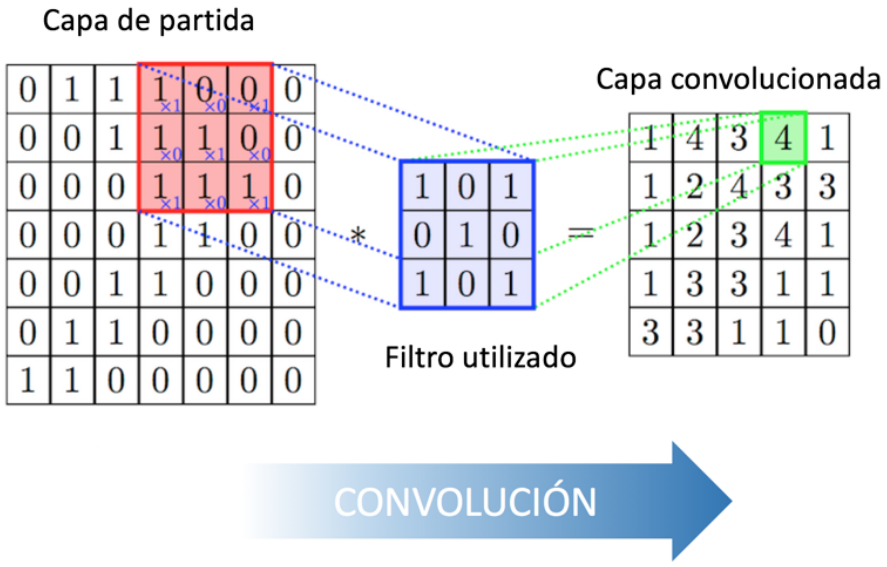


Figura 2.8: Operación de convolución. La matriz de salida se denomina mapa de características

Se puede observar que el filtro w_k^l que genera el mapa de características $z_{i,j,k}^l$ se comparte para todas las posiciones locales. Gracias a que se comparte los pesos, esto reduce la complejidad del modelo, haciendo que sea más fácil el entrenamiento.

El tamaño del mapa de características viene definido por los siguientes hiperparámetros:

Profundidad n_C : corresponde al número de filtros utilizados en cada capa convolucional. Cuanto más filtros, mayor capacidad de reconocimiento de patrones.

Zancada (stride) s : este hiperparámetro hace referencia al número de pixeles que se saltan tanto horizontal como verticalmente al deslizar el filtro sobre la imagen de entrada para calcular el producto escalar. Cuanto mayor sea la zancada, menor será las dimensiones del mapa de características. Esto tiene un efecto de submuestreo de imagen y reduce la carga computacional.

Padding p : consiste en agregar píxeles alrededor de la imagen. Tiene dos funcionalidades. 1) Se utiliza cuando información de interés que se encuentra en los laterales de la imagen no pierda importancia tras la operación de convolución. 2) Evita que los sucesivos mapas de características reduzcan demasiado de tamaño.

Siendo el mapa de características de una determinada capa de dimensión n x n x n_C, donde n hace referencia al ancho y alto de la matriz del mapa de características y n_C su profundidad, se tiene que $n = \frac{n_i+2p-f}{s} + 1$, donde n_i son el ancho y alto de

la imagen de entrada, f el tamaño de filtro, p el número de padding y s el valor de zancada.

Tras la operación de convolución, se llama a la función de activación, como Sigmoide o ReLU (sección 2.1.1), para introducir no linealidades al sistema. Siendo $a^l_{i,j,k}$ la activación de la característica $z^l_{i,j,k}$, se tiene que:

$$a^l_{i,j,k} = a(z^l_{i,j,k}) \tag{2.8}$$

Existen variantes de convolución que intentan mejorar la habilidad de representación, como *tiled convolution*, *transposed convolution*, y *dilated convolution* [8].

Capa de reducción o pooling

Las capas de pooling se colocan entre capas convolucionales. Tiene como objetivo reducir las dimensiones de las mapas de características y retener la información más relevante. Al reducir el número de parámetros, ayuda a reducir la carga computacional, así como problemas de sobreajuste. Los dos tipos de pooling más usados son *Maximum pooling* y *Average pooling*.

Esta reducción de dimensionalidad aporta a la red neuronal la capacidad de ser invariante a pequeñas traslaciones de la entrada. Es decir, si las posiciones de los objetos cambian en la imagen, la red neuronal sigue siendo capaz de detectarlos.

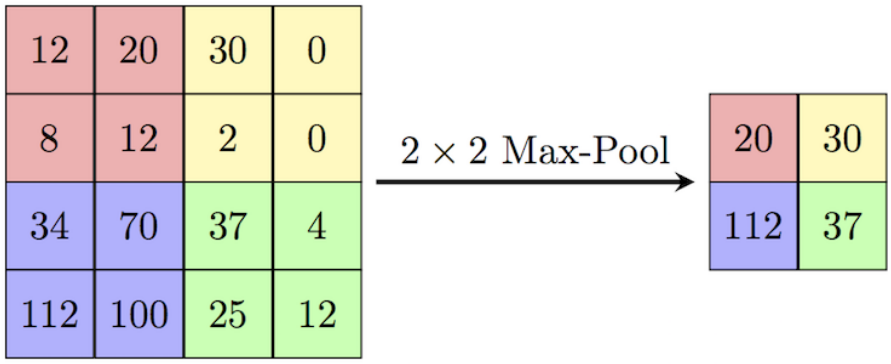


Figura 2.9: Operación de Maxpooling

Capa completamente conectada

Una capa completamente conectada es un perceptrón multicapa que utiliza un Softmax como función de activación en la capa de salida. Su función consiste en el razonamiento a alto nivel a partir de las características obtenidas de las capas de convolución y pooling.

2.2. Aprendizaje por refuerzo

El aprendizaje por refuerzo es una técnica de aprendizaje dentro del campo del aprendizaje automático que permite a un *agente* aprender a cómo comportarse en un *entorno* donde la única realimentación consta de una señal de recompensa escalar, la

cual indica cómo de bien lo está haciendo en el momento inmediato. El objetivo del agente consiste en ejecutar acciones que maximicen la recompensa en el largo plazo o retorno[9].

El aprendizaje por refuerzo se considera el tercer paradigma del aprendizaje automático junto con el aprendizaje supervisado y el aprendizaje no supervisado.

En el aprendizaje supervisado se aprende a partir de un conjunto de ejemplos de entrenamiento clasificados a priori por un experto. Cada ejemplo describe una situación con la correcta especificación o clase para determinar la acción correcta que se debería tomar en esa situación. Con esto se intenta que el sistema sea capaz de extrapolar o generalizar una serie de acciones correctas en situaciones no presentes en los ejemplos de entrenamiento. Este tipo de aprendizaje no es adecuado cuando se intenta aprender a partir de la interacción con el entorno, dado que no es práctico obtener el comportamiento correcto para todas las posibles situaciones a los que se debe enfrentar el agente. Es por ello que el agente debe aprender a partir de su propia experiencia.

El aprendizaje no supervisado trata de encontrar la estructura o patrón detrás de un conjunto de datos no clasificados. Uno puede pensar que el aprendizaje por refuerzo puede estar relacionado con el aprendizaje no supervisado ya que no depende de ejemplos cuyo comportamiento óptimo es conocido. Aunque descubrir la estructura de la experiencia del agente pueda ser útil, esto no ayuda a resolver el problema de maximizar la recompensa, por lo que estos dos aprendizajes no son tratados como la misma categoría.

La configuración típica del aprendizaje por refuerzo se muestra en la Figura 2.10.

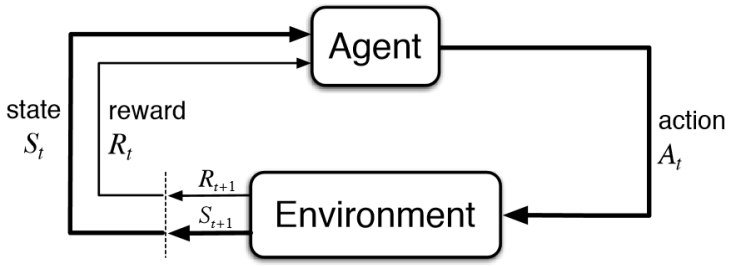


Figura 2.10: Interacción agente-entorno

Si un problema de aprendizaje por refuerzo cumple con la propiedad de Markov, entonces el problema se puede formular como un proceso de decisión de Markov (MDP).

La propiedad de Markov declara que el futuro es independiente del pasado dado el presente. Un estado S_t es Markov si:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t] \tag{2.9}$$

Si se dispone de un modelo perfecto del entorno, el MDP se puede resolver con técnicas de programación dinámica. Por otro lado, si se desconoce el modelo del entorno, se aplican técnicas de diferencia temporal.

2.2.1. Procesos de decisión de Markov

Casi todo problema de aprendizaje por refuerzo se puede formular matemáticamente como procesos de decisión de Markov. Los procesos de decisión de Markov o MDPs son la formulación clásica de toma de decisiones secuenciales, donde las acciones influyen no sólo en las recompensas inmediatas sino también las futuras, introduciendo la idea de la recompensa retrasada y la necesidad de sacrificar la recompensa inmediata por la recompensa retrasada [5].

Un MDP modela la interacción entre un agente y su entorno e incluye los siguientes elementos:

- Un conjunto de estados $s \in S$, donde S es el espacio de estado.
- Un conjunto de acciones $a \in A$, donde A es el espacio de acción.
- La probabilidad de transición, $P^a_{ss'} = P[S_{t+1} = s' | S_t = s, A_t = a]$
- La función de recompensa, $R^a_s = E[R_{t+1} | S_t = s, A_t = a]$
- Factor de descuento, $\gamma \in [0, 1]$.

El comportamiento de un agente está definido por la política π, el cual se encarga de relacionar los estados del agente con las posibles acciones en esos estados. Esta función es determinista si dado un estado s devuelve la acción a, es decir, $a = \pi(s)$ o estocástica, si devuelve las probabilidades de las acciones dado que se encuentra en un estado s: $\pi(a|s) = P[A_t = a | S_t = s]$.

Para conocer que acciones llevan al agente a maximizar las recompensas futuras, es necesario una función de valor que se encarga de evaluar la recompensa futura estimada. Se distinguen dos tipos de función de valor: a) Función de estado-valor, que estima la recompensa en función del estado en el que se encuentra y b) función de acción-valor, también conocida como Q-valor, que estima la recompensa en función de la acción que se toma dado que se encuentra en un estado s. Estas funciones de valor son dependientes del comportamiento del agente π.

$$V_\pi(s) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s] \tag{2.10}$$

$$Q_\pi(s,a) = E_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ...|S_t = s, A_t = a] \tag{2.11}$$

Para una función estado-valor, una política π se considera mejor que una política π' si $V_\pi(s) \geq V_{\pi'}(s)$ para todo $s \in S$.

Para cualquier MDP existe una política óptima π^*, que se define como la política que maximiza la función de valor. La función de valor óptimo correspondiente a la política óptima se define como:

$$V_*(s) = V_{\pi^*}(s) = \max_\pi V_\pi(s) \tag{2.12}$$

$$Q_*(s,a) = Q_{\pi^*}(s,a) = \max_\pi Q_\pi(s,a) \tag{2.13}$$

La función fundamental para resolver la política óptima es la ecuación de optimización de Bellman.

$$V_*(s) = \max_{a \in A(s)} \sum_{s'} P^a_{ss'} R^a_{ss'} + \gamma V_*(s') \tag{2.14}$$

$$Q_*(s,a) = \max_{a \in A(s)} \sum_{s'} P^a_{ss'} R^a_{ss'} + \gamma Q_*(s',a') \tag{2.15}$$

Si las probabilidades de transición y la función de recompensa son conocidas, los valores de $V_*(s)$ y $Q_*(s,a)$ se pueden calcular de manera recursiva usando métodos de programación dinámica. Los algoritmos que cuentan con las probabilidades que describen el sistema se conocen como algoritmos basados en modelo (*model-based*). Para los casos que se desconocen el modelo del entorno, se utilizan algoritmos denominados de modelo libre (*model-free*) para estimar los valores de $V_*(s)$ y $Q_*(s,a)$. Dentro de los algoritmos libres de modelo se distinguen entre métodos basados en función de valor (*value-based*) y métodos basados en política (*policy-based*).

2.2.2. Algoritmos de modelo libre

Los algoritmos libres de modelo intentan adquirir una política de comportamiento efectivo mediante prueba y error. Su objetivo consiste en optimizar la calidad de la política de comportamiento del agente en función del total de la recompensa descontada [10]. Los algoritmos de modelo libre se han aplicado en numerosos áreas como videojuegos y robótica [11]. La mayoría de los métodos libres de modelo intentan, o aprender la función de valor e inferir la política óptima a partir de allí, o directamente buscar los parámetros de la política para encontrar la política óptima. Los métodos que calculan la función de valor se conocen como métodos basados en función de valor y los que se centran en los parámetros de la política se conocen como métodos basados en política.

Los métodos de modelo libre también se diferencian entre *on-policy* y *off-policy*.

Un método es *on-policy* si la política de comportamiento es la misma que la política objetivo, mientras que un método es *off-policy* si la política comportamiento es distinto de la política objetivo. Se entiende como política de comportamiento a la política que utiliza el agente para determinar sus acciones en un estado dado, y la política objetivo como la política que emplea el agente para aprender a a partir de las recompensas obtenidas por sus acciones, es decir, es la política utilizada para determinar la función de valor actualizada.

2.2.2.1. Métodos basados en función de valor

Método de Monte Carlo

El término Monte Carlo se usa a menudo en métodos de estimación que se basan en el muestreo aleatorio con repetición. Los métodos de Monte Carlo en el aprendizaje por refuerzo permite estimar valores directamente a partir de la experiencia, secuencia de estados, acciones y recompensas. Estos valores (el retorno esperado) se obtienen promediando un gran número de muestras aleatorias.

Para diseñar un método de Monte Carlo, hay que basarse en el esquema de la iteración de política generalizada (GPI). El GPI se divide en dos pasos. El primer paso, conocido como evaluación de la política, consiste en aproximar una función de valor a partir de la política actual. Y en el segundo paso, llamado mejora de la política, la política es mejorada con respecto a la función de valor actual. Alternando los pasos 1 y 2

se termina con función de valor y política óptimas, empezado con una política aleatoria π_0. Aunque la implementación de los métodos de Monte Carlo es relativamente sencilla, requieren de un gran número de iteraciones hasta conseguir converger y sufren de una gran varianza en la estimación de la función de valor comparado con otros métodos como diferencia temporal.

Método de diferencia temporal

Al igual que Monte Carlo, los métodos de diferencia temporal (TD) aprenden a partir de experiencia sin necesidad de conocer las dinámicas del entorno. Sin embargo, en vez de usar la recompensa total acumulada (el resultado al final de un episodio completo), TD aprende a partir de episodios incompletos y la obtención de estimaciones a partir de estimaciones (*bootstrapping*). Estando en el tiempo t, los métodos TD actualizan la función de valor tras observar la recompensa R_{t+1}. El método TD más simple se denomina $TD(0)$ y está dado por la siguiente expresión:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma V(s_{t+q}) - V(s_t)] \tag{2.16}$$

Dos de los métodos TD más utilizados en el aprendizaje por refuerzo son $SARSA$ y *Q-learning*.

$SARSA$ (State Action Reward State Action) es un algoritmo TD y on-policy utilizado para determinar una función de acción-valor. Se estima una función de acción-valor Q_π para la política comportamiento actual π y para todos los estados s y acciones

a. Su función de actualización es la siguiente:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2.17}$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

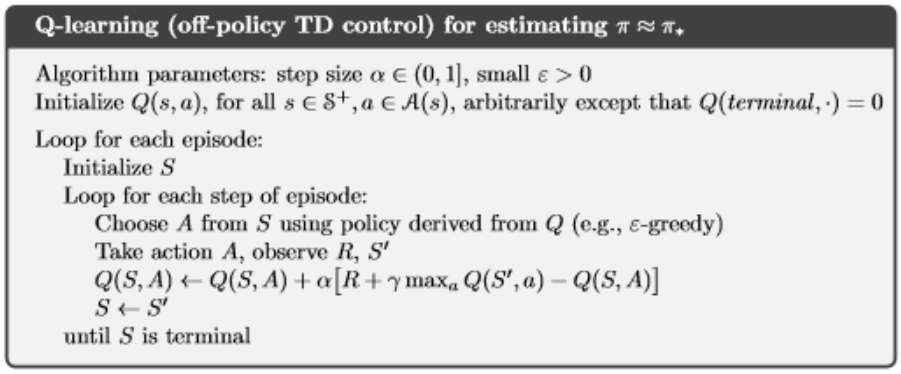
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 2.11: Seudocódigo SARSA. Introduction to Reinforcement learning by Sutton and Barto — Chapter 6

Q-learning es un algoritmo TD y off-policy con la siguiente función de actualización:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{2.18}$$

En este caso, la función acción-valor Q obtenido aproxima la función acción-valor óptimo independientemente de la política que se haya seguido.



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

Figura 2.12: Seudocódigo Q-learning. Introduction to Reinforcement learning by Sutton and Barto — Chapter 6

2.2.2.2. Método basado en política

Los métodos basados en política aprenden una política *parametrizada* para seleccionar las acciones sin la necesidad de consultar una función de valor. Siendo $\theta \in R^{d'}$ el vector de parámetros de la política, se tiene que $\pi(a|s, \theta) = P[A_t = a|St_{=}s, \theta_t = \theta]$ es la probabilidad de tomar acción a en el tiempo t dado que el entorno se encuentra en el estado s con parámetro θ.

Los algoritmos de *política gradiente* son métodos basados en la política donde el aprendizaje de los parámetros de la política se basa en el gradiente de una función objetivo $J(\theta)$ con respecto a los parámetros. La idea consiste en encontrar θ que maximice J.

$$\theta = \theta + \alpha \nabla J(\theta) \tag{2.19}$$

Existen numerosas variantes de métodos de política gradiente [12]. Los métodos basados en política cuentan con ciertas ventajas con respecto a los métodos basados en valor. 1) Presentan mejores propiedades de convergencia. Al seguir el gradiente para encontrar los mejores parámetros, se garantiza la convergencia. 2) Son más efectivos en casos con espacios de acciones grandes, o cuando se tratan de acciones continuas. 3) Se aprenden políticas estocásticas, permitiendo exploración al agente al no tomar siempre las mismas acciones en cada estado.

Las desventajas de los métodos basados en política es que pueden terminar convergiendo en máximos locales en vez de los máximos globales y se tardan más en entrenar.

Capítulo 3

Herramientas y librerías

3.1. MuJoCo

MuJoCo o Multi-Joint dynamics with Contact [13] es una herramienta de simulación de física diseñado para facilitar la investigación y desarrollo en áreas donde sistemas de compleja dinámica requieren de simulaciones rápidas y precisas, como es el caso de la robótica.

3.1.1. Instalación y puesta en marcha

MuJoCo es un producto comercial. Para usarlo, es necesario registrarse y solicitar una licencia. Es posible obtener una licencia personal gratis durante un año siendo estudiante. Si es aprobada la solicitud, se recibirá la llave de activación en formato txt por correo.

Una vez obtenida la licencia, se puede iniciar la descarga de la versión de MuJoCo Pro requerido considerando las características de la máquina utilizada (Sistema operativo). Este proyecto se ha desarrollado en una máquina Windows de 64-bits por lo que se descargó mujoco200_win64.zip. El zip resultante hay que extraerlo en la carpeta de usuario dentro de una carpeta con nombre ".mujoco", ej. C:/Users/username/.mujoco/. Es necesario pegar el archivo txt que contiene la llave de activación tanto en la carpeta ".mujoco" como "bin"para poder usar MuJoCo más adelante con Python.

La carpeta MuJoCo viene con unos modelos y códigos ejemplo que se pueden usar para tener un 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 3.1 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.

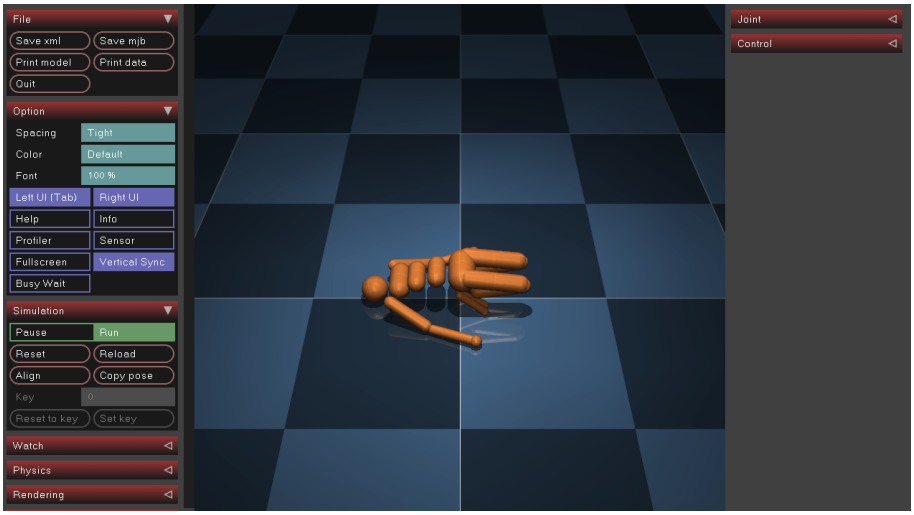


Figura 3.1: Interfaz del simulador MuJoCo 2.0 con el modelo Humanoid

3.1.2. Modelado

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 solo 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 un tipo de geometría simplificado. 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 ineria 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 *position*, cuyas señales de control son par, velocidad y posición respectivamente.

3.2. 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 va utilizar Python 3.6.

3.2.1. 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 [14].

- **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.

3.2.2. Instalación y puesta en marcha

El primer paso es crear un entorno virtual con la versión de Python 3.6. Existen varias formas de crear un entorno virtual dependiendo del sistema operativo. En este proyecto se ha utilizado el símbolo de sistema de Miniconda por su simplicidad.

- Creación del entorno: conda create –n <nombre_del_entorno>ej. conda create –n proyecto1
- Activación del entorno: conda activate <nombre_del_entorno>ej. conda activate proyecto1
- Desactivación del entorno: conda deactivate
- Instalación de la versión de Python: conda install python=3.6
- Instalación de librerías: pip install <librería>ej. pip install numpy

La mayoría de las librerías necesarias se pueden instalar siguiendo la última línea de código expuesta arriba con el correspondiente nombre de la librería. La única librería que resulta más compleja de instalar es *mujoco-py*. Para instalarlo, es necesario seguir los siguientes pasos [15]:

1. Instalar la librería git para poder clonar repositorios: conda install -c anaconda git
2. Clonar repositorio de mujoco-py: git clone https://github.com/openai/mujoco-py.git
3. Acceder a la carpeta clonada: cd mujoco-py
4. Instalar requirements: pip install -r requirements.txt
5. Instalar dev requirements: pip install -r requirements.dev.txt

6. Abrir los scripts *gen_ wrappers.py* y *wrappers.pxi* situados en /mujoco-py/scripts y /mujoco-py/mujoco-py/generated respectivamente.

7. Sustituir todas las instancias de $isinstance(addr, (int, np.int32, np.int64))$ por $hasattr(addr, '\_\_int\_\_')$

8. Volver a la carpeta de mujoco-py y forzar compilación de mujoco-py: python -c "import mujoco_p"

9. Instalar mujoco-py: python setup.py install

10. Comprobar que se ha instalado correctamente: python import mujoco_py

Tras importar mujoco_py, si toda ha salido correctamente, debería salir unos errores pidiendo que añadas la librería de mujoco al PATH junto con la línea de código de cómo hacerlo. Simplemente se debe copiar y pegar la línea de código en el símbolo de sistema. Es necesario asegurarse que el PATH mostrado en el error es donde está situado la carpeta de MuJoCo Pro.

Capítulo 4

Implementación

En esta capítulo se explicará el diseño del entorno y el agente A3C (Asynchronous Advantage Actor-Critic) que se utilizará para entrenar un agente cuya tarea consiste en alcanzar un objetivo como se muestra en la Figura 4.1.

4.1. Agente-Entorno

El objetivo del agente es tratar de mover las articulaciones del modelo MuJoCo IRB120 con el fin de acercar la pinza del robot lo más próximo posible al cubo de color rojo como se muestra en la Figura 4.1.

El agente recibe como entradas: 1) Una imagen renderizada del entorno de simu-

lación, que representa el estado actual, y 2) la recompensa asociada, y devuelve como salida un vector que contiene la información de las acciones para pasar del estado actual al estado siguiente. El entorno recibe estas acciones para actualizar los estados físicos del modelo MuJoCo y recalcula la recompensa en función de la distancia euclídea entre la pinza del robot y objetivo.

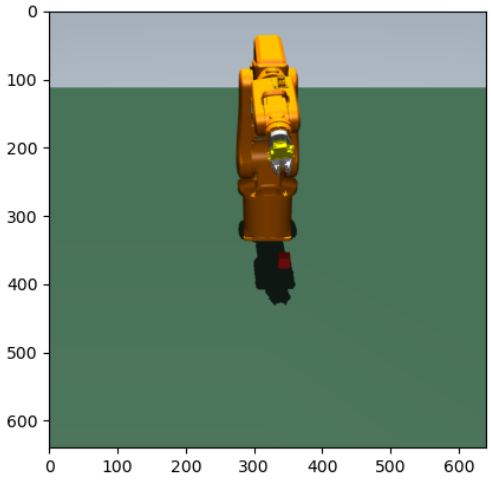


Figura 4.1: Muestra de imagen renderizada del entorno

4.1.1. Estructura del entorno

El entorno cuenta con los siguientes métodos públicos y los respectivos métodos privados a los que acceden: (Código completo en el Anexo B)

```
def step(self, a)
    def _do_simulation(self, ctrl)
    def _get_obs(self)
```

La función *step* recibe como argumento la salida del agente. Es un vector de dimensión (1, 6) donde el elemento 0 corresponde a la primera articulación y así sucesivamente. Los elementos son enteros entre 0 y 4. Cada uno de estos enteros se traduce en un sentido y par de giro al haber definido los actuadores de tipo motor en el modelo MuJoCo. La Tabla 4.1 muestra las acciones que corresponde a cada entero.

Clase	Sentido de giro	Par de giro
0		0
1	+	max/2
2	+	max
3	-	max/2
4	-	max

Tabla 4.1: Espacio de acciones discretas

Una vez discretizada las acciones, se actualizan los estados de la simulación y se obtendrán las observaciones. En este caso será una imagen RGB de 64 x 64 que utilizará el algoritmo A3C con una red convolucional para calcular las acciones recomendadas (Actor) y el valor de la recompensa futura estimada (Critic).

En la función *step*, además de actualizar los estados de la simulación y obtención

de los observaciones, se calcula la recompensa asociada. Su valor depende de la distancia euclídea entre las pinzas y el objetivo.

```
def reset(self)
    def _reset_target(self)
    def _set_qpos_qvel(self, qpos, qvel)
    def _get_obs(self)
```

La función *reset* se llama al principio de cada episodio para reiniciar los estados del modelo MuJoCo y obtener las observaciones iniciales, además de actualizar la posición del objetivo. Se medirá el efecto que tiene entrenar el agente con posiciones iniciales del entorno constantes a iniciarlos con posiciones iniciales variables.

```
def render(self, mode=None, width=DEFAULT_SIZE, height=DEFAULT_SIZE,
               camera_id=None, camera_name=None)
    def _get_viewer(self, mode)
    def _viewer_setup(self)
```

La función *render* presenta dos modos: *'human'* que permite visualizar en tiempo real la simulación y *'rgb_ array'* que permite renderizar una imagen RGB del tamaño especificado. Ambos modos presentan las configuraciones de cámara que se especifican en def _viewer_setup.

4.1.1.1. Función recompensa

Se ha definido en el entorno posibles combinaciones de funciones de recompensa para observa el efecto que tiene a la hora de entrenar el agente.

Para un primer caso, la función de recompensa será binaria (sparse reward). Se comparará la diferencia entre:

1.a) 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.

$$Recompensa = \begin{cases} 1 & d \leq umbral \\ 0 & d > umbral \end{cases}$$

1.b) Recompensa negativa (-1) si la distancia es mayor del umbral y 0 si está dentro del límite establecido.

$$Recompensa = \begin{cases} -1 & d > umbral \\ 0 & d \leq umbral \end{cases}$$

Para un segundo caso, la función recompensa será continua (shaped reward). Se observará el efecto que tiene si:

2.a) La recompensa es positiva y es inversamente proporcional a la distancia

entre objetivo y pinza.

$$Recompensa = \left(\frac{1}{d + 0{,}01}\right) * 0{,}1 \tag{4.1}$$

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

$$Recompensa = -\left(2d\right)^2 \tag{4.2}$$

4.1.2. Estructura del agente A3C

Se ha decidido escoger el agente A3C siguiendo los pasos de un grupo de investigación de DeepMind que ha obtenido unos resultados satisfactorios presentados en su paper *"Sim-to-Real Robot Learning from Pixels with Progressive Nets"*[16].

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.

4.1.2.1. Algoritmo A3C

Las siglas A3C hacen referencia a Asynchronous Advantage Actor-Critic.

Como se ha explicado anteriormente (sección 2.2.2), 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 des-

contada 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.

4.1.2.2. Estructura de la red neuronal

En el algoritmo A3C se emplea una red neuronal para hallar los Actors o acciones por una parte, y el Critic o función de valor por otra.

En este proyecto, se utilizará la estructura planteada en [16]. El agente recibe como observaciones una imagen RGB de 64x64, por lo que la red neuronal está compuesto primero por capas convolucionales para procesar y extraer las características de la imagen. La salida resultante de las capas convolucionales es alimentado, a una capa neuronal convencional (Fully-Conected Layer). Se incluye una capa recurrente (LSTM). Por último, la capa de salida está compuesto por seis Actors, dado que el robot cuenta con seis articulaciones, y un Critic. Cada uno de los Actors, tras la función de activación Softmax, devuelve cinco probabilidades. Cada probabilidad representa a una de las cinco acciones del espacio de acciones discretas explicado en la sección 4.1.1, Tabla

4.1.

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.

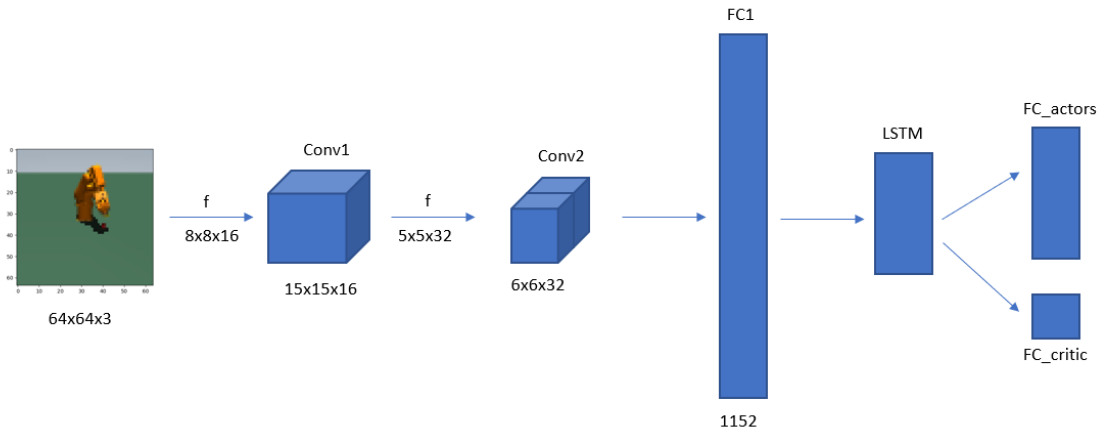


Figura 4.2: Arquitectura de la red neuronal

4.1.2.3. Función de pérdidas

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:

$$L_{critic} = (R - V(s))^2 = A(s)^2 \tag{4.3}$$

$$L_{actor} = -\log(\pi(a|s)A(s) \tag{4.4}$$

En [17] incluyen un término de entropía $H(\pi) = \sum(\log(p(x))p(x))$ en el cálculo de pérdidas del Actor para favorecer la exploración.

$$L_{actor} = -\log(\pi(a|s)A(s) - \beta H(\pi)) \tag{4.5}$$

, 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 [18].

$$A_{gae} \leftarrow A_{gae}\lambda\gamma + TD_{residual} \tag{4.6}$$

$$TD_{residual} = r + \gamma V(s') - V(s) \tag{4.7}$$

, 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:

$$L(\theta) = A(s)^2 - \log(\pi(a|s)A_{gae}(s) - \beta H(\pi)) \tag{4.8}$$

4.1.2.4. Optimización

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:

$$\theta \leftarrow \theta + \alpha A(s) \nabla_\theta \log(\pi(a|s) + \beta \nabla_\theta H(\pi) \tag{4.9}$$

$$w \leftarrow w - \alpha \nabla_w A(s)^2 \tag{4.10}$$

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

$$g = \nabla_\theta L(\theta) \tag{4.11}$$

$$sq = \alpha sq + (1 - \alpha) g^2 \tag{4.12}$$

$$\theta \leftarrow \theta - \eta \frac{g}{\sqrt{sq + \epsilon}} \tag{4.13}$$

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

4.2. Resultados

En este apartado se expone el efecto que tiene la función de recompensa en el aprendizaje del agente en la tarea de alcanzar un objetivo aleatorio situado en un área de trabajo y cuya posición inicial del robot es la misma. Con cada función de recompensa

expuesta en la sección 4.1.1.1, se han entrenado los agentes (5 hilos asíncronos) durante 5 millones de pasos. Cada 50 mil pasos, se evalúa el modelo durante 30 episodios de 50 pasos cada episodio para observar la evolución de las recompensas.

Para el caso 1.a (recompensa binaria positiva), la recompensa máxima ideal por episodio sería 50. Considerando que el objetivo está en el suelo y el robot tarda un par de pasos en alcanzarlo, se esperarían unas recompensas máximas acumuladas de entre 30 y 35 si hubiese aprendizaje. En el caso 1.b (recompensa binaria negativa), la recompensa máxima ideal sería 0 y recompensas reales de entre -20 y -15.

Se pueden observar en las Figuras 4.4 y 4.3 que el aprendizaje es despreciable tras los 5 millones de pasos. Aunque hay episodios puntuales donde se alcanzan unas recompensas notables, la media obtenida en todos los periodos de evaluación (de 30 episodios cada uno) son próximos a cero para el caso 1.a y -50 para el caso 1.b.

Este escenario era el esperado para el caso 1.a, dado que el agente no tiene información de cómo de bien o mal lo está haciendo al recibir solo ceros durante la mayor parte de la exploración. Este razonamiento se puede extender también para al caso 1.b. Aunque el agente es penalizado por no situarse dentro del umbral, no sabe cómo de mal lo está haciendo al penalizarse de la misma forma para todas las distancias superiores al umbral. Es posible que el agente sea capaz de realizar la tarea correctamente si se entrenase durante mucho más tiempo, pero por cuestiones de tiempo no se explorará el caso.

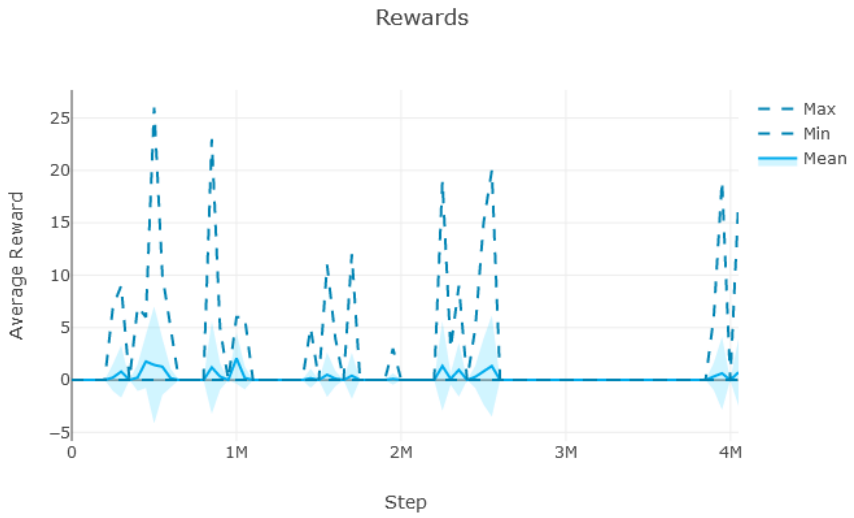


Figura 4.3: Función de recompensa caso 1.a

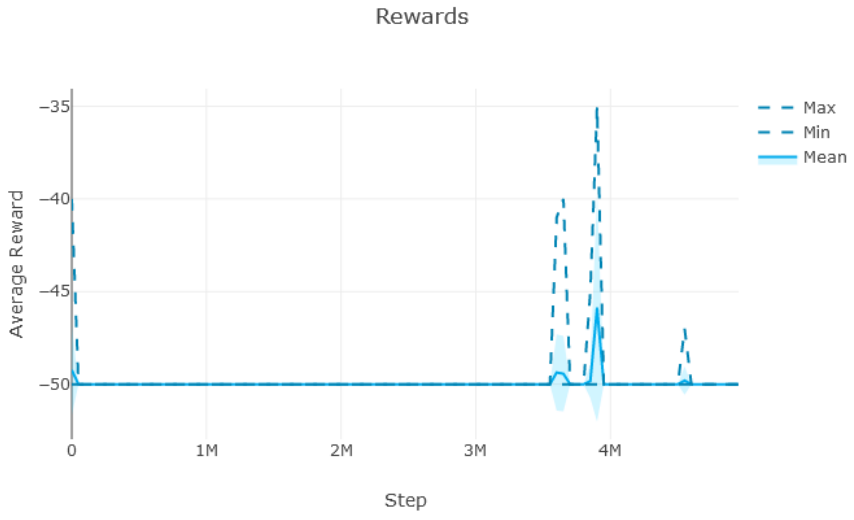


Figura 4.4: Función de recompensa caso 1.b

Teniendo en consideración el razonamiento anterior, se han entrenado los agentes con recompensas continuas, es decir, la recompensa es proporcional a la distancia entre objetivo y pinza.

En la Figura 4.5 y 4.6 se muestra el aprendizaje con las funciones de recompuestas continuas positiva (2.a) y negativa (2.b) respectivamente, Ec.4.1 y Ec.4.2. 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.

Figura 4.5: Función de recompensa caso 2.a

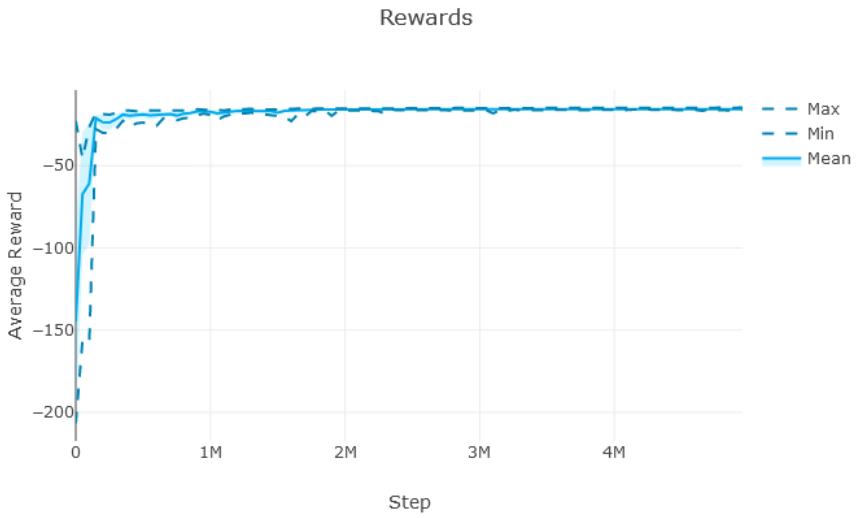


Figura 4.6: Función de recompensa caso 2.b

Como era de esperar, en estos dos casos, se puede apreciar un aprendizaje constante del agente ya que recibe una mayor información a partir de la función de recompensa continua. No obstante, el caso 2.b aporta un aprendizaje mucho más estable comparado con el caso 2.a. La razón se encuentra en la forma de la curva de las Ec.4.1 y Ec.4.2. Se puede observar en la Figura 4.7 como el gradiente es muy pronunciado para valores cercanos a 0 en el eje x, resultando en gran variabilidad de recompensa por cambios pequeños en distancia, dando lugar a las oscilaciones que se observan en la Figura 4.5. En la Figura 4.8 por el contrario, el gradiente tiende a 0 cuanto menor es la distancia, ayudando a la convergencia de la recompensa.

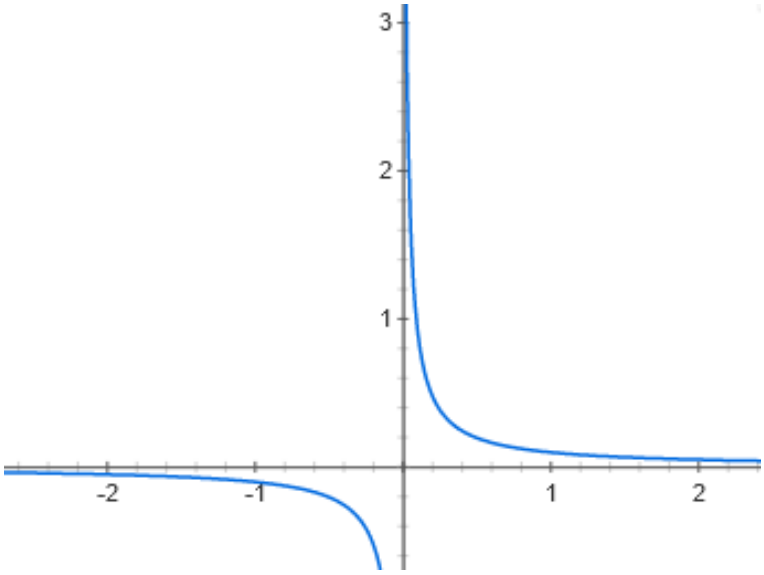


Figura 4.7: Curva $y = (1/(x + 0{,}01)) * 0{,}1$

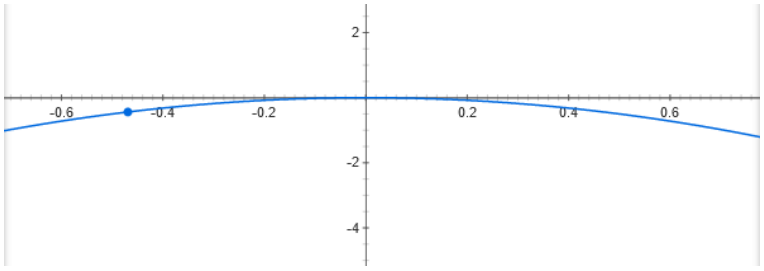


Figura 4.8: Curva $y = -(2x)^2$

Utilizando el modelo obtenido de entrenar el agente con la función recompensa del caso 2.b, se ha renderizado una 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 % (29 de cada 30) asumiendo que el robot empieza desde la misma posición inicial. No

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

Tabla 4.2: Tabla comparativa de agentes entrenados con las funciones de recompensa

obstante, la probabilidad de éxito de este modelo baja a los 50 % si se aleatoriza la posición inicial del robot en cada episodio, por lo que se ha procedido a reentrenar el 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.

Capítulo 5

Conclusión y trabajo futuro

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 en la sección 4.1.1.1. 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.

5.1. Siguientes pasos

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 entorno 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.

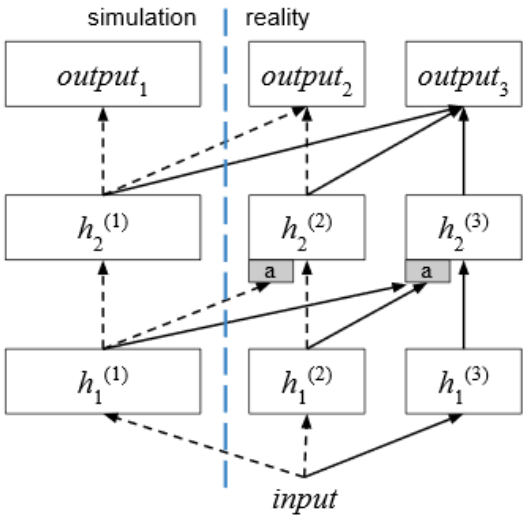


Figura 5.1: Red neuronal progresiva

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 [19], que permite la

transferencia de aprendizaje mediante conexiones laterales entre redes neuronales como se muestra en Figura 5.1, 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.

Bibliografía

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[2] Artur M Schweidtmann and Alexander Mitsos. Global deterministic optimization with artificial neural networks embedded. *CoRR*, abs/1801.07114v2, 2018.

[3] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *CoRR*, abs/1811.03378, 2018.

[4] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. Searching for activation functions. *CoRR*, abs/1710.05941, 2017.

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

[6] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[7] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *CoRR*, abs/1511.08458, 2015.

[8] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, and Gang Wang. Recent advances in convolutional neural networks. *CoRR*, abs/1512.07108, 2015.

[9] Jens Kober and Jan Peters. *Reinforcement Learning - State-of-the-Art*, volume 12 of *Adaptation, Learning, and Optimization*, chapter Reinforcement Learning and Markov Decision Processes, pages 3–39. Springer, 2012.

[10] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. *CoRR*, abs/1702.08892, 2017.

[11] J. Kober and J. Peters. *Reinforcement Learning in Robotics: A Survey*, volume 12, pages 579–610. Springer, Berlin, Germany, 2012.

[12] Konstantinos I. Chatzilygeroudis, Vassilis Vassiliades, Freek Stulp, Sylvain Calinon, and Jean-Baptiste Mouret. A survey on policy search algorithms for learning robot controllers in a handful of trials. *CoRR*, abs/1807.02303, 2018.

[13] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

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

[15] Sayan Mandal. Install openai gym with box2d and mujoco in windows 10, July 2019. Last accessed 1 July 2020.

[16] 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.

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

[18] 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.

[19] 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.

Anexo A

Objetivos de Desarrollo Sostenible

Tras el cumplimiento del plazo de los 8 Objetivos de Desarrollo del Milenio (ODM), los estados miembros de la ONU aprobaron en 2015 la Agenda 2030 sobre el Desarrollo Sostenible. La Agenda cuenta con 17 Objetivos de Desarrollo Sostenible (ODS), formulados para hacer frente al cambio climático, erradicar la pobreza, así como para promover el bienestar de todos.

Este proyecto está fuertemente relacionado con el objetivo 9 (Industria, innovación e infraestructuras) de ODS, en especial, con la industria e innovación. Actualmente, los robots están siendo ampliamente utilizados en el sector manufacturero para realizar tareas pesadas y repetitivas de manera eficiente, aumentando la productividad y reduciendo desperdicios, además de permitir que el valioso recurso humano pueda ser redistribuido a posiciones que aportan un mayor valor añadido a las empresas. El uso

de técnicas de aprendizaje automático, en especial el aprendizaje por refuerzo, favorece y ayuda a los procesos industriales, así como, permite optimizar el uso de los recursos y como consecuencia, un sector industrial más sostenible.

Anexo B

Entorno IRB120

```
import os
import mujoco_py
import numpy as np
from gym.utils import seeding
from timeit import default_timer as timer
DEFAULT_SIZE = 64
class IRB120Env():
    def __init__(self, width, height, frame_skip, rewarding_distance,
        control_magnitude,reward_continuous):
         self.frame_skip = frame_skip
         self.width = width
         self.height = height
         self.viewer = None
         self._viewers = {}
```

```
model_path = "irb120.xml"
fullpath = os.path.join(
    os.path.dirname(__file__), "assets", model_path)
if not os.path.exists(fullpath):
    raise IOError("File %s does not exist" %fullpath)
model = mujoco_py.load_model_from_path(fullpath)
self.sim = mujoco_py.MjSim(model)

self.init_state = self.sim.get_state()
self.init_qpos = self.sim.data.qpos.ravel().copy()
self.init_qvel = self.sim.data.qvel.ravel().copy()

self.actuator_bounds = self.sim.model.actuator_ctrlrange
self.actuator_low = self.actuator_bounds[:, 0]
self.actuator_high = self.actuator_bounds[:, 1]
self.actuator_ctrlrange = self.actuator_high
                          - self.actuator_low
self.num_actuators = len(self.actuator_low)

self.sim.data.ctrl[:] = np.zeros(self.num_actuators)

self.seed()
self.reward_continuous = reward_continuous
self.sum_reward = 0
self.rewarding_distance = rewarding_distance
self.max_threshold = 0.3


self.target_bounds = np.array(((0.4, 0.6), (0., 0.2), (0.02,
    0.02)))
```

```
        self.target_reset_distance = 0.2


        self.control_values = self.actuator_ctrlrange * control_magnitude

        self.num_actions = 5
        self.action_space = [list(range(self.num_actions)) ] * self.
            num_actuators
        self.observation_space = ((0, ), (height, width, 3))


        self.reset()

    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def _set_qpos_qvel(self, qpos, qvel):
        assert qpos.shape == (self.sim.model.nq, ) and qvel.shape == (
            self.sim.model.nv, )
        self.sim.data.qpos[:] = qpos
        self.sim.data.qvel[:] = qvel
        self.sim.forward()

    def reset(self):
        qpos = self.init_qpos
        # qpos[:1] = np.round(np.random.uniform(-1, 1, 1), 2)
        # qpos[1:2] = np.round(np.random.uniform(-0.3, 0.6, 1), 2)
        qvel = self.init_qvel
        self._reset_target()
```

```
        self._set_qpos_qvel(qpos, qvel)

        return self._get_obs()

    def _reset_target(self):
        # Randomize goal position within specified bounds
        self.goal = np.random.rand(3) * (self.target_bounds[:, 1] - self.
            target_bounds[:, 0]) + self.target_bounds[:, 0]
        geom_positions = self.sim.model.geom_pos.copy()

        prev_goal_location = geom_positions[1]

        while (np.linalg.norm(prev_goal_location - self.goal) <
                self.target_reset_distance):
            self.goal = np.random.rand(3) * (self.target_bounds[:, 1] -
                self.target_bounds[:, 0]) + self.target_bounds[:, 0]

        geom_positions[1] = self.goal

        self.sim.model.geom_pos[:] = geom_positions

    def render(self,
               mode=None,
               width=DEFAULT_SIZE,
               height=DEFAULT_SIZE,
               camera_id=None,
               camera_name=None):
        if mode == 'rgb_array':
            if camera_id is not None and camera_name is not None:
                raise ValueError("Both `camera_id` and `camera_name`
```

```
                cannot be specified at the same time.")

            no_camera_specified = camera_name is None and camera_id is
                None
            if no_camera_specified:
                camera_name = 'track'

            if camera_id is None and camera_name in self.model.
                _camera_name2id:
                camera_id = self.model.camera_name2id(camera_name)

            self._get_viewer(mode).render(width, height, camera_id=1)

            data = self._get_viewer(mode).read_pixels(width, height,
                depth=False)
            # original image is upside-down, so flip it
            return data[::-1, :, :]

        elif mode == 'depth_array':
            self._get_viewer(mode).render(width, height)

            data = self._get_viewer(mode).read_pixels(width, height,
                depth=True)[1]

            return data[::-1, :]
        elif mode == 'human':
            self._get_viewer(mode).render()

    def _get_viewer(self, mode):
        self.viewer = self._viewers.get(mode)
```

```
        if self.viewer is None:

            if mode == 'human':
                self.viewer = mujoco_py.MjViewer(self.sim)
            elif mode == 'rgb_array' or mode == 'depth_array':
                self.viewer = mujoco_py.MjRenderContextOffscreen(self.sim
                    , -1, opengl_backend='glfw')

            self._viewer_setup()
            self._viewers[mode] = self.viewer
        return self.viewer

    def _viewer_setup(self):
        body_id = self.sim.model.body_name2id('robot0:base_link')
        lookat = self.sim.data.body_xpos[body_id]
        for idx, value in enumerate(lookat):
            self.viewer.cam.lookat[idx] = value
        self.viewer.cam.distance = 2
        self.viewer.cam.azimuth = 180
        self.viewer.cam.elevation = -30


    def _get_obs_joint(self):
        return np.concatenate(
            [self.sim.data.qpos.flat[:], self.sim.data.qvel.flat[:]])

    def _get_obs_rgb_view1(self):
        self._get_viewer('rgb_array').render(self.width, self.height,
            camera_id=-1)
        # start = timer()
```

```
        obs_rgb_view1 = self._get_viewer('rgb_array').read_pixels(self.
            width, self.height, depth=False)
        # end = timer()
        # print(f"Get_image: {end-start}" )
        return obs_rgb_view1[::-1, :]

    # def _get_obs_rgb_view2(self):
    #     self._get_viewer('rgb_array').render(self.width, self.height,
        camera_id=1)
    #     obs_rgb_view2 = self._get_viewer('rgb_array').read_pixels(self.
        width, self.height, depth=False)
    #     return obs_rgb_view2[::-1, :]

    def _get_obs(self):
        return (self._get_obs_joint(), self._get_obs_rgb_view1())

    def _do_simulation(self, ctrl):
        '''Do one step of simulation, taking new control as target

        Arguments:
            ctrl {np.array(num_actuator)} -- new control to send to
                actuators
        '''
        # ctrl = np.min((ctrl, self.actuator_high), axis=0)
        # ctrl = np.max((ctrl, self.actuator_low), axis=0)

        self.sim.data.ctrl[:] = ctrl

        for _ in range(self.frame_skip):
            self.sim.step()
```

```
def step(self, a):

    done = False
    new_control = np.copy(self.sim.data.ctrl).flatten()

    dist = np.linalg.norm(
        self.sim.data.get_site_xpos("robot0:grip") - self.goal)

    ####continuous or sparse positive reward
    if self.reward_continuous:
        reward = (((dist)+0.01)**-1)*0.1
    else:

        if dist < self.rewarding_distance:
            reward = 1
        else:
            reward = 0
    ###### Continuous and sparse positive + negative reward
    # if self.reward_continuous:

    #     if dist > 2*self.rewarding_distance:
    #         reward = -(2*dist)**2
    #     else:
    #         reward = (((dist)+0.01)**-1)*0.1
    # else:

    #     if dist < self.rewarding_distance:
    #         reward = 1
    #     else:
```

```
#         reward = -1

##### Continuous and sparse negative reward
# if self.reward_continuous:
#     reward = -(2*dist)**2
# else:
#     if dist > self.rewarding_distance:
#         reward = -1
#     else:
#         reward = 0
# # print(dist)
# if dist < 0.05:
#     done = True
# Transform discrete actions to continuous controls
for i in range(self.num_actuators):
    '''
    0 = 0 velocity
    1 = small positive signal
    2 = large positive signal
    3 = small negative signal
    4 = large negative signal
    '''
    if a[i] == 0:
        new_control[i] = 0
    if a[i] == 1:
        new_control[i] = self.control_values[i] / 2
    if a[i] == 2:
        new_control[i] = self.control_values[i]
    if a[i] == 3:
        new_control[i] = -self.control_values[i] / 2
```

```
            elif a[i] == 4:
                new_control[i] = -self.control_values[i]

        # Do one step of simulation
        self._do_simulation(new_control)
        self.sum_reward  += reward

        return self._get_obs(), reward, done
```

Anexo C

Modelo IRB120 MuJoCo

```
<mujoco model="irb120">
  <compiler inertiafromgeom='true' angle="radian" settotalmass="25.5"
      meshdir="stl/" />

    <option timestep="0.002" gravity="0 0 0"/>

    <asset>

        <mesh name="base" file="base.stl" scale="0.001 0.001 0.001"/>
        <mesh name="link1" file="link1.stl" scale="0.001 0.001 0.001"/>
        <mesh name="link2" file="link2.stl" scale="0.001 0.001 0.001"/>
        <mesh name="link3" file="link3.stl" scale="0.001 0.001 0.001"/>
        <mesh name="link4" file="link4.stl" scale="0.001 0.001 0.001"/>
        <mesh name="link5" file="link5.stl" scale="0.001 0.001 0.001"/>
```

```
        <mesh name="link6" file="link6.stl" scale="0.001 0.001 0.001"/>
        <mesh name="connection_part" file="connection_part.stl" scale
            ="0.001 0.001 0.001"/>
        <mesh name="pinza_izq" file="pinza_izq.stl" scale="0.001 0.001
            0.001"/>
        <mesh name="pinza_der" file="pinza_der.stl" scale="0.001 0.001
            0.001"/>
        <mesh name="schunk" file="schunk.stl" scale="0.001 0.001 0.001"/>

    <texture type="skybox" builtin="gradient" rgb1=".2 .3 .4" rgb2="1 1
        1" width="100" height="100"/>
    <texture name="groundplane" type="2d" builtin="checker" rgb1=".25 .26
         .25" rgb2=".22 .22 .22" width="100" height="100" mark="none"
        markrgb=".8 .8 .8"/>
    <texture name="greenground" type="2d" builtin="gradient" rgb1="0.19
        0.28 0.23" rgb2="0 0 0" width="100" height="100" mark="none"
        markrgb=".8 .8 .8"/>
    <material name="MatViz" specular="1" shininess=".1" reflectance="0.5"
         rgba=".07 .07 .1 1"/>
    <material name="MatGnd" texture="groundplane"  texrepeat="5 5"
        specular="1" shininess="0" reflectance="0"/>

    </asset>

    <contact>
        <pair geom1 = "geo_link1" geom2 = "geo_base" condim= "1"/>
    </contact>

  <default class="robot0:fetch">
    <joint armature="0" damping="10" frictionloss="0" stiffness="0"></
```

```
        joint>

        <default class="robot0:fetchGripper">
            <joint armature="0" damping="50" limited="true" solimplimit
                ="0.9 0.999 0.01" solreflimit="0.01 1" type="slide"></
                joint>
        </default>
    </default>
<default class ="ground">
    <geom type="plane" margin="0.001" contype="1" />
</default>


<worldbody>

    <light cutoff="200" diffuse="1.2 1.2 1.2" dir="-0 0 -1.3"
        directional="true" exponent="1" pos="0 0 1.3" specular=".1 .1
        .1" castshadow="false"/>

    <camera name="view1" pos="0.2 0.1 2.0" zaxis="0 0 1" fovy="45" ipd
        ="0.068"/>
    <camera name="view2" pos="1.1 0.9 0.3" xyaxes="-1 1 0 0 0 1" fovy
        ="45" ipd="0.068"/>
    <!-- <camera name="view3" pos="1 0 0.3" axisangle="1 0 0 1" fovy
        ="45" ipd="0.068"/> -->


    <geom name="ground" class="ground" type='plane' pos="0 0 0" rgba
        ="0.19 0.30 0.23 1" size="2 2 1"/>
```

```
<body name="robot0:base_link" pos="0.1 0.1 0" axisangle="1 0 0
    1.57079632679">

        <geom name = "geo_base" type="mesh" rgba="1 0.5 0 1" mesh="base
            " />
        <body name="link1" pos= "0 0 0">
            <joint name="joint0" axis="0 1 0" range = "-2.879 2.879"
                limited="true"/>
            <!-- <inertial pos="0 0.238 0" mass="3.067" diaginertia
                ="0.0142 0.0144 0.0105 "/> -->
            <geom name = "geo_link1" pos = "0.0 0.0 0.0" type="mesh"
                rgba="1 0.5 0 1" mesh="link1" />
            <body name="link2" pos="0 0 0">
                <joint name="joint1" axis="0 0 -1" range = "-1.91986
                    1.91986" limited="true" pos="0 0.290 0" />
                <!-- <inertial pos="0 0.392 0" mass="3.9087"
                    diaginertia="0.0603 0.0416 0.0260" />    -->
                <geom name = "geo_link2" pos="0 0.560 0.058" type="mesh
                    " rgba="1 0.5 0 1" mesh="link2" />
                <body name="link3" pos="0 0 0" >
                    <joint name="joint2" axis="0 0 -1" range =
                        "-1.91986 1.22173" limited="true" pos="0 0.560 0
                         " />
                    <!-- <inertial pos="0.02281 0.6179 0 " mass
                        ="2.9437" diaginertia="0.0084 0.0167 0.0127" />
                         -->
                    <geom name = "geo_link3" pos="0.07195 0.49385 0 "
                        type="mesh" rgba="1 0.5 0 1" mesh="link3" />
                    <body name="link4" pos = "0 0 0 ">
                        <joint name="joint3" axis="1 0 0" range =
```

```
    "-2.79 2.79" limited="true" pos="0.134
    0.630  0 " />
<!-- <inertial pos="0.2246 0.6304 0" mass
    ="1.3251" diaginertia="0.0028 0.004 0.0052"
    /> -->
<geom name = "geo_link4" pos="0.302 0.630 0 "
    type="mesh" rgba="1 0.5 0 1" mesh="link4"/>
<body name="link5" pos = "0 0 0">
    <joint name="joint4" axis="0 0 -1"  range =
        "-2.0944 2.0944" limited="true" pos
        ="0.302 0.630 0"/>
    <!-- <inertial pos="0.300 0.630 0" mass
        ="1.4" diaginertia="0.0004 0.0009
        0.0008" />  -->
    <geom name = "geo_link5" pos="0 0 0" type="
        mesh" rgba="1 0.5 0 1" mesh="link5" />
    <body name="link6" pos ="0 0 0">
        <joint name="joint5" axis="1 0 0"
            range = "-6.9813 6.9863" limited="
            true" pos="0.361 0.630 0 "/>
        <!-- <inertial pos="0.3669 0.630 0"
            mass="1.4" diaginertia="0.0000001
            0.0000001 0.0000001" />  -->
        <geom name = "geo_link6" pos="0.361
            0.630 0" axisangle="0 1 0
            1.57079632679" type="mesh" rgba="0 0
             0 1" mesh="link6" />
        <body name="connection_part" pos ="0 0
            0">
```

```
                    <geom name = "geo_connection_part"
                        pos="0.38887 0.630 0" quat="0.5
                        0.5 -0.5 0.5" type="mesh" rgba
                        ="0.67 0.67 0.67 1" mesh="
                        connection_part" />
                    <body name="robot0:gripper_link" pos
                        ="0.40296 0.64122 0" >
<geom type="mesh" mesh="schunk" name="robot0:
    gripper_link" rgba="1 1 0 1" quat="0.71 0 0.71
    0"></geom>

<body childclass="robot0:fetchGripper" name="robot0
    :r_gripper_finger_link" pos="-0.0065 -0.06
    -0.031">
  <joint axis="0 0 1" name="robot0:
      r_gripper_finger_joint" range="0 0.05"></joint
      >
  <geom type="mesh" mesh="pinza_der" rgba="1 1 1 1"
       name="geo_pinza_der" quat="0 0.71 0 0.71"></
      geom>
</body>
<body childclass="robot0:fetchGripper" name="robot0
    :l_gripper_finger_link" pos="-0.0065 -0.06
    0.031">
  <joint axis="0 0 -1" name="robot0:
      l_gripper_finger_joint" range="0 0.05"></joint
      >
  <geom type="mesh" mesh="pinza_izq" rgba="1 1 1 1"
       name="geo_pinza_izq" quat="0 0.71 0 0.71"></
      geom>
```

```
                    </body>
                    <site name="robot0:grip" pos="0.06 -0.06 0" rgba
                        ="0.5 0.5 0.5 0" size="0.01 0.01 0.01"></site>
                </body>
            </body>
        </body>
    </body>
</body>
</body>
</body>
<geom name="target" pos="0.6 0.1 0.02" rgba="0.9 0.2 0.2 1" size
    ="0.02 0.02 0.02" type="box" contype='0' conaffinity="0"/>
</worldbody>
<actuator>
<motor ctrlrange="-2 2" ctrllimited="true" name="joint0_motor" joint
    ="joint0" />
<motor ctrlrange="-2 2" ctrllimited="true" name="joint1_motor" joint
    ="joint1"/>
<motor ctrlrange="-2 2" ctrllimited="true" name="joint2_motor" joint
    ="joint2"/>
<motor ctrlrange="-2 2" ctrllimited="true" name="joint3_motor" joint
    ="joint3"/>
<motor ctrlrange="-2 2" ctrllimited="true" name="joint4_motor" joint
    ="joint4"/>
<motor ctrlrange="-2 2" ctrllimited="true" name="joint5_motor" joint
    ="joint5"/>
<!-- <motor ctrlrange="-1 0" ctrllimited="true" name="r_grip" joint="
    robot0:r_gripper_finger_joint"/>
```

```
    <motor ctrlrange="-1 0" ctrllimited="true" name="l_grip" joint="
        robot0:l_gripper_finger_joint"/> -->
  </actuator>
</mujoco>
```