# 

Escuela Internacional de Posgrado

Master Universitario en Ciencia de Datos e Ingeniería de Computadores

Trabajo de Fin de Master

**Análisis y estudio en frameworks de simulación sobre la generación de datos en problemas robótica industrial y su aplicación en tareas de aprendizaje automático.**

Presentado por: Hodei Zia López

Tutores: Alberto Luis Fernández Hilario y Mikel Galar Idoate

15/09/2022

Resumen

Este proyecto pretende abordar la problemática que se genera a la hora de buscar con qué datos trabajar en un problema de predicción de trayectorias para la realización de tareas en robótica industrial. Para ello, comenzaremos analizando algunos de los frameworks de simulación más utilizados para este tipo de proyectos. Es por este motivo, que además de analizar diferentes frameworks, realizaremos también un minucioso estudio sobre otros trabajos realizados dentro de este campo con el objetivo de obtener un mayor conocimiento con el que abordar el problema. A continuación, tras haber realizado este estudio inicial, comenzaremos con el apartado más importante del proyecto, la generación de datos. En este apartado, intentaremos extraer la mayor cantidad posible de datos de la tarea a analizar para poder evaluar sus posibles configuraciones en la fase de entrenamiento del modelo mediante una red neuronal. Una vez obtengamos el modelo, volveremos al simulador para probar la calidad de los datos utilizados así como de la red en cuestión. Todo esto lo haremos sobre diferentes entornos de simulación para evitar un sesgo en los datos y con el fin de generalizar qué tipo de datos se han de utilizar para futuros problemas relacionados con la robótica dentro de este ámbito.

Palabras clave

* Robótica industrial
* Generación de datos
* Entorno de simulación
* Aprendizaje automático
* RLBench
* Metaworld

Índice

[1](#_Toc109735851)

[Resumen 3](#_Toc109735852)

[Palabras clave 3](#_Toc109735853)

[Capítulo 1 5](#_Toc109735855)

[Introducción 5](#_Toc109735856)

[1.1. Objetivo 6](#_Toc109735857)

[Capítulo 2 7](#_Toc109735858)

[Marco teórico 7](#_Toc109735859)

[2.1. Estado del arte 7](#_Toc109735860)

[2.2. Frameworks de simulación utilizados 7](#_Toc109735861)

[2.2.1. CoppeliaSim 8](#_Toc109735862)

[2.2.2. MuJoCo 11](#_Toc109735863)

[2.3. Librerías utilizadas 12](#_Toc109735864)

[2.3.1. RLBench 13](#_Toc109735865)

[2.3.2. Metaworld 18](#_Toc109735866)

[Capítulo 3 23](#_Toc109735867)

[Desarrollo e implementación 23](#_Toc109735868)

[3.1. Fase 1: Generación de los datos 23](#_Toc109735869)

[3.1.1. Generación de datos en MuJoCo-Metaworld 24](#_Toc109735870)

[3.1.2. Generación de datos en CoppeliaSim-RLBench 30](#_Toc109735871)

[3.2. Fase 2: Entrenamiento del modelo 35](#_Toc109735872)

[3.3. Fase 3: Simulación y validación del modelo 39](#_Toc109735873)

Capítulo 1

Introducción

La robótica industrial es una rama de la ingeniería que pretende realizar múltiples procesos industriales tales como la manipulación de objetos haciendo uso de robots con el objetivo de completar diversas tareas en cadena de forma automática de modo que necesitemos la mínima supervisión humana posible. Partiendo de esta definición, podemos encontrarnos tanto con tareas que no dependan del entorno y que por tanto, se realicen aplicando los mismos movimientos sobre el robot como con otras en las que tanto el entorno como los objetos u obstáculos que se encuentren en él sean cambiantes, por lo que necesitemos hacer que nuestro robot adapte sus movimientos, decisiones o incluso fuerza en base a lo que se encuentre en cada momento. Para ello, debemos hacer que el robot consiga manejarse en distintas situaciones, haciendo que aprenda de una cantidad de escenarios lo suficientemente amplia como para que sepa cómo actuar frente a un obstáculo inesperado nunca antes visto o aprendido en la fase de entrenamiento. Es aquí donde cobra una importancia real la realización de un estudio minucioso sobre la generación de datos como pueden ser distintos escenarios, la clase de imágenes sobre el entorno que van a ser capaces de aportar una mayor información, el tipo de acciones con las que el robot va a ser capaz de generalizar mejor el problema…

En este contexto, es común simular distintas tareas que pueda realizar un brazo robótico tales como pueden ser coger una pelota, apilar unos cubos, entre otros, y evaluar los datos devueltos en busca de los casos de estudio más adecuados. A partir de los mismos, se realizará un proceso de entrenamiento para generar un modelo que sea capaz de replicar estas tareas en sus simuladores correspondientes. Adicionalmente, se estudiará la mejor sinergia entre caso de estudio y simulador con vistas a formular una generalización de los tipos de datos que se han de utilizar para futuros problemas relacionados con la robótica en el ámbito de la inteligencia artificial.

# Objetivo

El objetivo principal será obtener una configuración general de datos con el que se caractericen distintos casos de uso en robótica industrial. Para ello, desglosamos entre los siguientes subobjetivos:

1. Estudiar y analizar en profundidad tanto los distintos estudios y proyectos realizados en este ámbito como los diferentes frameworks de simulación utilizados, dejando claras tanto sus diferencias como sus similitudes.
2. Generar datos para un conjunto de tareas con cada uno de los frameworks que sean los que utilicemos para generar los distintos modelos en la fase de entrenamiento.
3. Analizar las mejores configuraciones posibles de datos en busca de un patrón capaz de generar una generalización para futuros problemas de modo que sepamos de donde partir inicialmente sea cual sea el problema a evaluar. Para esto trataremos de encontrar las configuraciones que maximicen el RMSE como métrica estándar.
4. Simular en los robots los modelos generados y visualiza los resultados obtenidos mediante diferentes aproximaciones.

Capítulo 2

Marco teórico

En este capítulo vamos a exponer toda la parte teórica recopilada tras realizar una investigación documental sobre distintos trabajos centrados en el mismo tema. De igual forma, una vez finalizado dicho apartado, pasaremos a hablar sobre los diferentes frameworks de simulación que hemos utilizado.

# 2.1. Estado del arte

# 2.2. Frameworks de simulación utilizados

Como ya hemos ido mencionado a lo largo del inicio de este documento, un entorno de simulación es una aplicación software que permite al usuario trabajar tanto con una amplia variedad de robots como de tareas y entornos virtuales totalmente personalizables a los gustos y necesidades de cada uno. Al ser un entorno virtual, no requiere de la disposición de ninguna clase de herramienta física, haciendo así que ahorremos tiempo y dinero en la compra y adjudicación de la puesta en marcha de todo lo necesario para simular cualquier tipo de tarea.

El objetivo principal de cualquier entorno de simulación es el de emular y validar tareas dentro de un mundo virtual antes de hacerlo en el mundo físico para así, tal y como hemos comentado antes, evitar sobrecostes y ahorrar tiempo, ya que es mucho más rápido y menos costoso simular una tarea dentro de un software que podemos modificar, iniciar o detener a nuestro antojo antes que hacerlo sobre un robot físico el cual muy posiblemente requiera de unos cuidados y puesta en marcha muy específicos. De esta manera, podemos probar distintas configuraciones de un modo muy sencillo para posteriormente evaluar cual es la que finalmente debemos llevar a cabo sobre nuestro robot o entorno físico.

Para ello, dentro de casi cualquier entorno, tenemos la libertad para crear y modificar materiales, objetos e incluso comportamientos de una manera rápida que en el mundo real no seriamos capaces de realizar tan fácilmente.

Para la realización de este proyecto, y con el objetivo principal de evitar un sesgo en los datos por culpa de las características específicas de cada framework, hemos decidido utilizar dos de los entornos de simulación de datos más conocidos en problemas de robótica industrial: MuJoCo y CoppeliaSim. Cada uno de estos entornos de simulación es capaz de utilizar distintas librerías que son las encargadas de especificar al sistema como generar, simular y trabajar con sus propias herramientas. Nosotros hemos seleccionado las dos más utilizadas en este tipo de proyectos: RLBench y Metaworld.

## 2.2.1. CoppeliaSim

Este entorno de simulación, antes conocido como V-Rep, es uno de los más utilizados para la simulación de cualquier tipo de tarea dentro de la comunidad robótica. Fue desarrollado por Coppelia Robotics en Marzo de 2010 y a día de hoy, según el servicio de búsqueda de patentes *The Lens [x]*, cuenta con un total de 1.282.860 citas distribuidas entre 73.775 patentes, por lo que podemos hacernos a la idea del prestigio que tiene.

Diagrama

Descripción generada automáticamente Cuenta con una interfaz gráfica muy bien estructurada en donde podemos ver toda la información sobre la escena que tenemos delante así como modificar parámetros o insertar nuevos objetos a ella. A continuacion se muestra un pantallazo de lo que seria esta interfaz grafica con una escena cargada en ella:

Figura x. Interfaz gráfica de CoppeliaSim

Como podemos apreciar en la figura anterior, al lado izquierdo tenemos la parte que más ocupa en pantalla, la visualización de la escena en tiempo real. Dentro de ella podemos ver tanto los objetos que participan activamente en la escena (mesas, cubos, robots…) como los objetos pasivos (cámara, dirección de la luz…). Todos estos objetos, sean del tipo que sean, pueden ser añadidos desde la ventana llamada *Model browser* que se encuentra en el lado derecho de la pantalla, en donde tenemos una estructura jerárquica de directorios en la cual podemos encontrar todo tipo de objetos además de visualizarlos debajo tras hacer click en cualquiera de ellas. Entre medias de las dos pantallas comentadas, tenemos la denominada como *Scene hierarchy*. Esta pantalla engloba la jerarquía de todos los objetos escogidos y puestos a escena por partes, es decir, teniendo la opción de modificar cada objeto por trozos, lo cual resulta muy útil, ya que amplía la capa de personalización disponible. Finalmente, arriba del todo, cogiendo toda la pantalla en el eje horizontal, tenemos la barra de simulación en donde podemos ajustar la tasa de refresco, modificar la posición de los objetos, movernos por la escena, acelerar o decelerar la simulación y muchas más acciones relacionadas con las ahora mencionadas.

Al estar basado en una arquitectura de control distribuida, podemos programar escenas mediante seis métodos distintos:

* **Scripts embebidos**
* **API remota de cliente**
* **Add-ones**
* **Plugins**
* **Nodo ROS/ROS2**
* **Nodo ZeroMQ**

Cada cual presenta ventajas y desventajas respecto al resto de métodos tal y como vamos a mostrar a continuación:

Tabla

Descripción generada automáticamente

Figura x. Interfaz gráfica de CoppeliaSim

De entre todos estos métodos de programación, nosotros vamos a trabajar en Python con scripts embebidos provenientes de la librería RLBench [x] y modificaciones que haremos sobre ellos, de modo que tan solo usemos la interfaz gráfica para extraer información sobre la escena en todo momento tal y como explicaremos más adelante.

## 2.2.2. MuJoCo

Este otro entorno, pese a estar junto a CoppeliaSim entre los más conocidos para este tipo de problemas, no es tan utilizado. Esto lo podemos ver de igual modo que hicimos con CoppeliaSim si buscamos las citas que ha recibido en distintas patentes. Según el servicio de búsqueda de patentes *The Lens,* tan solo ha recibido 42 citas distribuidas en un total de 15 patentes. Con este dato podemos hacernos a la idea de la magnitud que tiene CoppeliaSim en la robótica industrial, dejando tanto MuJoCo como al resto de entornos de simulación más apartados para el ámbito de estudio.

Al contrario que pasaba con CoppeliaSim, MuJoCo no cuenta con una interfaz gráfica tan compleja y bien estructurada. Esto lo podemos observar en la siguiente figura:

Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Figura x. Interfaz gráfica de MuJoCo

Las diferencias entre ambas interfaces son bastante apreciables. Por un lado, vemos como la estructura es mucho más simple, careciendo de todos los menús de opciones y estructuración de los objetos de la escena. Tan solo tenemos unas cuantas opciones puestas en texto plano sobre la escena las cuales sirven para la visualización de esta, en las cuales podemos escoger la velocidad, transparencia, cámara, visualización de los ejes sobre los objetos y demás opciones que tienen que ver con la simulación de la tarea en tiempo real. Además de esto, si nos fijamos en la escena, vemos como los objetos carecen de la complejidad que

tenían los de CoppeliaSim, llegando a tener unas texturas y unos efectos de luz mucho menos realistas.

Pese a que a simple vista parezca un entorno muy distinto a CoppeliaSim, es cuando utilizamos librerías cuando le dotamos de una mayor complejidad llegando a hacer prácticamente lo mismo que el anterior entorno de simulación.

MuJoCo, al estar basado también en una arquitectura distribuida, es capaz de recibir órdenes de distintas maneras. Sin embargo, es algo más limitado que CoppeliaSim, llegando a solo soportar Scripts embebidos, Plugins de Unity y API remota de cliente. Del mismo modo que hicimos con el otro entorno, hemos decidido optar por los scripts embebidos, ya que en todo momento, nuestra intención ha sido intentar que las implementaciones para ambos entornos de simulación fueran lo más parecido posible para poder comparar los resultados de una manera más realista.

Un añadido que tiene MuJoCo frente a la mayoría de entornos de simulación y que lo hace muy interesante es el poder programar tanto en Julia como en Swift con él. Hasta la fecha, la mayoría de frameworks tan solo usaban Python, C/C++ y Java, por lo que esta característica convierte a MuJoCo en uno de los entornos más atractivos para programadores de estos dos lenguajes que hasta la fecha tenían que cambiar de lenguaje si querían trabajar con esta clase de entornos.

# 2.3. Librerías utilizadas

Como ya hemos ido mencionando a lo largo del apartado anterior, los entornos de simulación necesitaban de una librería que les aportara todas las opciones necesarias para poder trabajar en la simulación de cualquier tipo de tarea posible en el ámbito de la robótica industrial.

Han sido muchas las opciones posibles para ambos frameworks. Sin embargo, hemos decidido quedarnos con RLBench [x] para CoppeliaSim y Metaworld [x] para MuJoCo. Estas librerías están adaptadas al simulador en cuestión y vienen con un conjunto de escenas, objetos y funciones que hacen que podamos partir de una base sólida de donde partir para realizar casi cualquier tarea que se nos ocurra en el ámbito de la robótica.

## 2.3.1. RLBench

RLBench, también conocida como Robot Learning Benchmark and Learning Environment, es una amplia librería y entorno de aprendizaje diseñada para facilitar el trabajo de los investigadores que trabajen en el campo de la robótica dentro de las áreas del aprendizaje por refuerzo, aprendizaje por imitación, aprendizaje multitarea, visión por computador y en particular el aprendizaje por medio de pocas pruebas o few-shot learning.

Fue desarrollada por Stephen James, Zicong Ma, David Rovick y Andrew J.Davison en septiembre de 2019 y ha sido desde entonces una de las librerías y entornos de aprendizaje más utilizadas dentro de este área de trabajo.

Esta librería engloba un total de 100 tareas diferentes realizables con la ayuda de brazos robóticos articulados. Dentro de estas tareas se encuentran algunas como recoger una pelota, pulsar un botón o incluso abrir y cerrar una puerta, tal y como se puede observar en la siguiente figura:



Figura x. Algunas de las tareas disponibles con RLBench

Podemos apreciar como la estructura de cada escena es la misma. Poseemos el mismo robot, misma mesa, y lo único que cambia es la tarea a realizar con distintos objetos además del código interno de cada una de las tareas que es el cual guía al robot a la hora de realizar pequeñas demostraciones las cuales utilizaremos más adelante para entrenar nuestro modelo.

Por otro lado, con el objetivo de dotar de una mayor complejidad a cada tarea, RLBench es capaz de generar variaciones sobre una misma tarea haciendo que cada prueba sea completamente distinta a la anterior. Esto lo hace no solo inicializando los objetos en distintos lugares cada vez que llamamos a la ejecución de la escena, sino que también modificando los colores de los objetos que se encuentran en ella.

Diagrama, Mapa

Descripción generada automáticamente

Figura x. Ejemplo de las variaciones por prueba en cada tarea

En la figura anterior, podemos observar cómo serían las modificaciones que hemos comentado. Cada variación es una tarea distinta. En este ejemplo, la variación 0 sería la tarea de apilar un solo bloque rojo encima de la marca, mientras que en la variación 1 seria apilar dos bloques rojos, en la variación 2 apilar tres bloques rojos y en la variación V apilar un bloque de color marrón.

Además, por cada una de las variaciones de la tarea, tenemos distintos episodios, los cuales son la repetición de la tarea marcada por la variación modificando tanto los lugares de inicialización de los objetos como los colores de los objetos con baja relevancia, es decir, si nuestra tarea es apilar un bloque rojo, al menos tendremos un bloque rojo, viéndose modificado el color del resto de los bloques en cada una de las pruebas de forma aleatoria.

El objetivo de esta librería es el de dotar de las herramientas necesarias a CoppeliaSim para poder simular tareas con las que trabajar dentro del campo del aprendizaje automático y de este modo conseguir obtener y validar buenos modelos con los que trabajar en el mundo real.

No solo posee un amplio abanico de tareas, sino que estas son grabadas por un total de cinco cámaras colocadas en distintos ángulos de modo que podamos obtener toda la información posible de cada uno de los movimientos del brazo robótico. A continuación se muestra un ejemplo de cada una de las cámaras para un mismo instante:

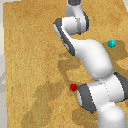
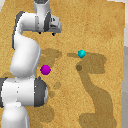


Figura x. Imágenes de Front, Left, Overhead, Right y Wrist de RLBench

Además de estas cámaras que son capaces de mostrarnos la escena en RGB, RLBench puede generar también imágenes en profundidad y mascaras desde los mismos ángulos que los mostrado en la figura de arriba.

Imagen que contiene Texto

Descripción generada automáticamenteUna caricatura de una persona

Descripción generada automáticamente con confianza bajaImagen que contiene tabla, espejo, diferente, sostener

Descripción generada automáticamente

Figura x. Imágenes en RGB, profundidad y mascara

Con el objetivo de poder realizar las pruebas lo más comparables posibles, hemos optado por usar únicamente las imágenes obtenidas mediante cámaras RGB, ya que no todos los simuladores son capaces de sacar imágenes tanto en profundidad como en forma de mascara generando una segmentación de los objetos de la escena.

Como ya hemos comentado, todas las escenas, independientemente de la tarea que pretendan realizar, poseen una configuración común, la cual vamos a explicar a continuación poniendo la siguiente figura como ejemplo:

Imagen que contiene interior, tabla, pequeño, verde

Descripción generada automáticamente

Figura x. Configuración común de una escena

La escena por defecto es la mostrada en la anterior figura. Podemos ver como tenemos un brazo robótico puesto sobre el borde de una mesa. Este brazo en concreto es del modelo Franka Emika Panda [x] de siete articulaciones. Está valorado en 25 mil euros y es capaz de soportar una carga máxima de 3kg. Siguiendo con la escena, en la figura podemos ver en azul tres ejemplos de donde estarían colocadas las cámaras de Left, Right y Wrist. Pese a ser cámaras fijas, podemos programarlas para incluir más o modificar las ya existentes.

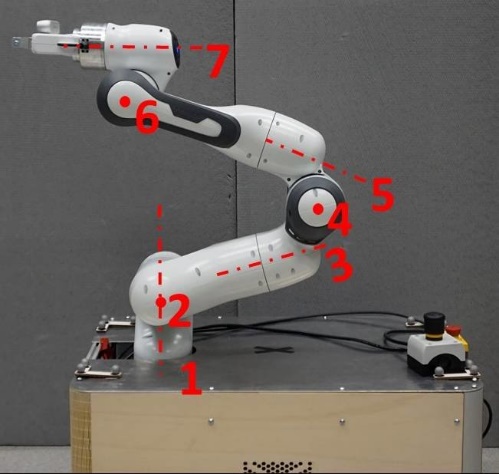


Figura x. Robot Franka Emika Panda en el mundo real

Como podemos observar en la anterior figura, el robot posee un total de siete articulaciones las cuales se mueven únicamente en un eje.

La librería no solo es capaz de ofrecernos la información en forma de imagen por cada instante para observar y evaluar las tareas, sino que proporcionan al usuario una variedad de acciones con las que complementar sus experimentos.

Estas acciones vienen dadas como un vector de siete elementos, uno por cada una de las articulaciones, entre las cuales tenemos:

* **Velocidad absoluta**: Se trata de la velocidad con la que se mueve cada articulación en un instante de tiempo. Viene dada en radianes/segundos.
* **Velocidad relativa**: Se trata de la diferencia de velocidad entre instantes dada por cada articulación. Viene dada en radianes/segundos.
* **Posición absoluta**: Se trata de la posición en la que se encuentra cada articulación en el espacio.
* **Posición relativa**: Se trata de la diferencia de posiciones en cada instante en la que se encuentra cada articulación en el espacio.
* **End-effector pose absoluto**: Se trata únicamente de la posición de la pinza en el espacio de coordenadas xyz. Con esta acción, es como si tiráramos de la pinza haciendo que el resto de articulaciones se movieran únicamente por dicha fuerza.
* **End-effector pose relativo**: Se trata únicamente de la diferencia de la posición de la pinza en el espacio de coordenadas xyz por cada instante de tiempo.
* **Fuerza absoluta**: Se trata de la fuerza realizada por cada una de las articulaciones.
* **Fuerza relativa**: Se trata de la diferencia de fuerza realizada por cada una de las articulaciones.

En próximas actualizaciones de RLBench se comenta sobre la posibilidad de incluir nuevas métricas de acciones tales como los ángulos absolutos y relativos, los cuales serán medidos en radianes.

## 2.3.2. Metaworld

Metaworld es una librería de simulación de tareas centrada en el aprendizaje por refuerzo y en el aprendizaje multitarea.

Fue desarrollada por Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn y Sergey Levine en el año 2019 y ha estado en constante desarrollo desde entonces implementando nuevas tareas, funciones...

A diferencia de RLBench, la cual se centraba en muchos aspectos del aprendizaje automático, esta librería se centra única y exclusivamente en los dos campos ya mencionados y que se pueden ver en la siguiente figura:

Imagen que contiene Icono

Descripción generada automáticamente

Figura x. Explicación de lo que pretende realizar esta librería

Gracias a esta figura, podemos llegar a comprender de una manera muy básica que es lo que pretende esta librería. Por un lado, dado un conjunto amplio de tareas, se pretende aprender y automatizar la forma en la que se realizan tareas ya vistas mediante el método del aprendizaje por refuerzo en el cual se dota de recompensas a la maquina en función de su grado de acierto o proximidad a la solución final. Por otro lado, dado el mismo conjunto de tareas, se pretende aprender a realizar nuevas tareas nunca antes vistas o vistas muy pocas veces en comparación con el resto de una manera muy rápida.

Para ello, se han diseñado un total de 50 tareas, muy parecidas a las que tiene RLBench, con las que se han generado seis métodos distintos de evaluar los distintos algoritmos de aprendizaje que el usuario genere. Cada uno de estos métodos de evaluación posee una dificultad y objetivo distinto al resto tal y como vamos a ver a continuación.

* Interfaz de usuario gráfica, Aplicación

  Descripción generada automáticamente**Meta-Learning 1 (ML1)**: El ML1 es el método más sencillo de evaluación de meta-aprendizaje. Se escoge una sola tarea y se entrena realizando muchas pruebas con ella en donde se modifican las inicializaciones de los objetos y de sus metas. Finalmente, se realiza una validación con otro conjunto de pruebas en donde tengamos objetos inicializados en posiciones nunca antes vistas en el conjunto de entrenamiento.

Figura x. Ejemplo del funcionamiento del ML1

* **Multi-Task 1 (MT1)**: El MT1 es bastante parecido al ML1, ya que tan solo usa una única tarea para realizar el aprendizaje. Sin embargo, realiza la evaluación sin necesidad de realizar ninguna validación previa.

Diagrama

Descripción generada automáticamente

Figura x. Ejemplo del funcionamiento del MT1

* **Meta-Learning 10 (ML10)**: El ML10 es un método de meta-aprendizaje más complejo que el ML1 en donde entrenamos con un conjunto de 10 tareas distintas. Finalmente, realizamos la validación sobre 5 tareas no vistas en la fase de entrenamiento.

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

Figura x. Ejemplo del funcionamiento del ML10

* **Diagrama

  Descripción generada automáticamenteMulti-Task 10 (MT10)**: El MT10 se asemeja al MT1 con la diferencia de que en lugar de aprender de una tarea y replicarla sin necesidad de validación, aprende de 10 tareas distintas para tratar de reproducirlas posteriormente.

Figura x. Ejemplo del funcionamiento del MT10

* **Imagen que contiene Escala de tiempo

  Descripción generada automáticamenteMeta-Learning 45 (ML45)**: El ML45 es el modelo más completo de meta-aprendizaje en donde se entrena un conjunto de 45 tareas dejando de lado las 5 restantes para validación.

Figura x. Ejemplo del funcionamiento del ML45

* **Imagen que contiene Escala de tiempo

  Descripción generada automáticamenteMulti-Task 50 (MT50)**: El MT50 es el modelo más completo de multitareas, ya que usa el 100% de estas sin dejar ninguna para la validación.

Imagen que contiene Escala de tiempo

Descripción generada automáticamente

Figura x. Ejemplo del funcionamiento **del** MT50

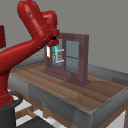
Las cincuenta tareas que se pueden observar son controladas por el robot de igual forma que en RLBench, con acciones que son pasadas instante a instante a nuestro robot.

Al contrario que con RLBench en donde teníamos un amplio abanico de acciones con el cual probar, en Metaworld tan solo tenemos las posiciones relativas en el espacio de coordenadas XYZ junto con el estado de la pinza, el cual puede estar representado por un cero en caso de estar abierta o de un 0.1 en caso de estar en proceso de cerrarse.

En cuanto al robot, al igual que en RLBench, se utiliza un robot del modelo Franka Emika Panda con la única diferencia de que este se encuentra pintado en tonos rojizos en lugar de tener sus colores originales.

Las imágenes utilizadas por Metaworld, en cambio, sí que son bastante parecidas a las que podemos obtener mediante RLBench, ya que se encuentran situadas en distintas zonas del entorno haciendo así que podamos obtener el máximo de información posible en todo momento de manera visual.

Imagen que contiene exterior, parado, calle, tabla

Descripción generada automáticamenteUna sala de estar

Descripción generada automáticamente con confianza baja

Figura x. Imágenes de BehindGripper, Corner, Corner2, Gripper, Top y Corner3 de Metaworld

Podemos ver como existen tipos de imágenes que son muy similares con algunos de los tipos de RLBench como son las que muestran tanto la primera persona de la pinza como el entorno desde arriba y desde ambos lados.

En definitiva, vemos como ambas librerías, pese a tener grandes diferencias por el modo en el que están hechas y por el entorno de simulación en el que son utilizadas, sirven para lo mismo, pues podemos obtener los mismos datos e información de tareas completamente iguales para entrenar nuestro modelo.

Además, es muy interesante ver la capa de personalización que poseen ambas, haciendo que pese a tener funciones y tareas ya programadas y funcionales, seamos capaces de modificarlas o crearlas desde cero a nuestro antojo. Esto último sí que es verdad que se nota mucho más en Metaworld, en donde no solo tenemos los scripts mejor explicados sino que al utilizar un entorno de simulación menos complejo, dependemos de menos variables para realizar nuestras modificaciones.

Capítulo 3

Desarrollo e implementación

A continuación, una vez hemos hablado tanto sobre en qué consiste nuestro proyecto como sobre los distintos entornos y librerías que vamos a utilizar para ello, vamos a pasar a explicar como lo hemos hecho.

# 3.1. Fase 1: Generación de los datos

Esta primera fase es una de las más importantes del proyecto, ya que consiste en entender tanto los entornos de simulación como las librerías y sus tareas para poder realizar pruebas con ellas y extraer sus datos, los cuales serán utilizados posteriormente a la hora de entrenar el modelo.

Como ya hemos comentado previamente, los entornos de simulación utilizados no son del todo iguales. Además, al estar utilizando librerías adaptadas a cada uno de ellos, los métodos para la generación y obtención de datos no es del todo igual pese a que hayamos intentado adecuar nuestros scripts para que sean lo más parecidos posibles, de modo que podamos estar alternando de un entorno al otro sin necesidad de volver a entender todo de nuevo.

Además, algo también a tener en cuenta es como dependiendo de la tarea que estemos ejecutando, las tres fases se van a ver afectadas, ya que no cuesta lo mismo coger una pelota que abrir una ventana, sobre todo si tenemos en cuenta la información de las observaciones, cosa que hace que en MuJoCo tengamos que adaptarnos a cada una de las tareas realizadas tal y como vamos a explicar a continuación.

## 3.1.1. Generación de datos en MuJoCo-Metaworld

Ya hemos comentado previamente como Metaworld dispone de diversos métodos de evaluación de tareas programados y listos para ejecutar de modo que podamos observar como el robot es capaz de simular tareas nuevas a partir de un conjunto de tareas diferentes o aprender a realizar una tras haber entrenado con ella.

Sin embargo, y pese a que nuestro objetivo pasa por hacer que el modelo aprenda de una tarea para poder simularla posteriormente con los datos que hemos obtenido en la generación de esta, necesitamos desarrollar un script que sea capaz de utilizar las tareas ya creadas por la librería para poder generar los conjuntos de datos que vayamos a utilizar para la fase de entrenamiento del modelo.

Partiendo de esta idea, hemos comenzado analizando el código interno de la librería para ver qué datos podemos obtener a la hora de simular cualquier tipo de tarea.

Dentro del script situado en *metaworld/envs/mujoco/mujoco\_env.py,* podemos apreciar como existe una función capaz de devolvernos seis tipos distintos de imágenes en base a donde este situada la cámara con la resolución que deseemos. Estos seis tipos, tal y como se pueden apreciar de manera ordenada en la figura x, son los siguientes:

* **BehindGripper**: Cámara fija situada boca abajo que sigue los movimientos de la pinza situándola en el centro de la imagen.
* **Corner**: Cámara fija situada de modo que veamos la tarea desde la zona noreste del robot.
* **Corner2**: Cámara fija situada de modo que veamos la tarea desde la zona noroeste del robot de manera más alejada que en Corner1.
* **Gripper**: Cámara fija situada dentro de la pinza en donde podemos observar el entorno que se presenta bajo la pinza.
* **Top**: Cámara fija situada sobre nuestro entorno.
* **Corner3**: Cámara fija situada de manera similar que en Corner2 estando un poco más cercana al robot con una mayor altura.

Además de las imágenes, necesitamos extraer un conjunto de valores numéricos que representen la acción a realizar por el robot en cada instante de tiempo. Observando el modo en el que se realizan las ejecuciones de cada tarea, hemos visto como dentro del script situado en el directorio *metaworld/policies/policy.py* existe una función que tras recibir un objeto de la clase *Observation,* devuelve un vector de cuatro valores los cuales utiliza para definir el movimiento que ha de realizar el robot en cada instante de tiempo.

Un ejemplo del vector que obtenemos es el siguiente:

**[0.02345, -3.01789, 4e10-4, 0]**

Mientras que los tres primeros valores equivalen a la posición de la pinza en el eje de coordenadas XYZ, el ultimo es un valor binario que representa el estado de la pinza, el cual se encuentra como 0 si la pinza o bien está abierta o bien se encuentra en proceso de ser abierta, o como 0.1 en caso de que la pinza se encuentre en proceso de ser cerrada. Este último valor se pasa por una función dentro de este mismo script y calcula el valor continuo del grado de apertura de la pinza mediante una formula pasando de 1 si se encuentra completamente abierta a 0 si se encuentra completamente cerrada.

Esta posición es relativa, lo cual quiere decir que tan solo posee la diferencia de movimiento respecto al instante anterior, por lo que sitúa la pinza del robot dentro del eje de coordenadas base (0,0,0) antes de aplicar la acción.

Este vector nos será de mucha utilidad, ya que dispone de los valores que tratamos de predecir y que por tanto, debemos aplicar en el entrenamiento junto con los diferentes tipos de imágenes.

Otros valores que también hemos podido obtener son los que devuelven la información restante de los objetos de la clase *Observation*. Estos objetos se reinician al comienzo de cada instante de tiempo y disponen de la información relevante tanto de la pinza como del objetivo al que llegar. En concreto, hemos decidido extraer la información sobre el valor continuo del estado real de la pinza que se calcula mediante una formula y que hemos comentado antes y la posición absoluta de la pinza del robot en el eje XYZ que también podemos extraer.

Al final, el objetivo de esta fase es tener un conjunto de datos representativos de cada instante de tiempo con el que realizar las pruebas en la fase de entrenamiento.

Para la obtención de todos los datos aquí mencionados, hemos generado un script denominado como *dataset\_generator\_X.py* en donde la X es el nombre de la tarea que estamos generando, el cual ejecutamos con los siguientes parámetros de entrada:

* **#PRUEBAS:** Cantidad de pruebas sobre una misma tarea con infinitas variaciones a realizar.
* **#ITERACIONES:** Cantidad de iteraciones máximas marcadas como umbral que una tarea es capaz de ejecutar.
* **%TRAIN:** Porcentaje del reparto entre los conjuntos de train y test.
* **NOMBRE:** Nombre de la carpeta que guardara el dataset generado.

El objetivo de este script tal y como el nombre indica es generar un dataset con todos los datos mencionados previamente dentro su subconjunto predeterminado el cual puede ser tanto train como test.

Para ello, comenzamos asignando de forma aleatoria cada una de las pruebas al subconjunto de datos de train o de test de modo que cumplamos con el porcentaje pasado por parámetro. Al tener dos subconjuntos de datos, debemos guardar la información de cada subconjunto separada de la del otro. Es por esto por lo que en base a si una prueba va destinada a train o a test, debemos tratarla de forma diferente. Almacenaremos sus datos dentro de una carpeta o csv único para su subconjunto, de modo que finalicemos la ejecución teniendo una carpeta por cada subconjunto con la misma estructura en donde almacenemos las imágenes en formato .png y las acciones dentro de un fichero con extensión .csv.

Una vez sabemos a qué subconjunto ha de ir destinada cada prueba, comenzamos su ejecución paso a paso. Con cada iteración no solo iremos guardando los datos ya descritos, sino que iremos teniendo en cuenta otros valores procedentes del objeto de la clase *Environment* el cual, al igual que pasa con los objetos de la clase *Observation*, se van actualizando con cada iteración. El valor del que hablamos nos devuelve un conjunto de valores numéricos como la distancia de la pinza al objetivo final, la recompensa por cercanía u otros tipos de recompensas. Es con estos valores con los que detendremos la prueba para pasar

a la siguiente antes de llegar al tope máximo de iteraciones marcado mediante parámetro de entrada. Cabe destacar que dependiendo de la tarea que estemos ejecutando, tendremos que usar unos parámetros u otros, y para saber cuáles hemos de usar, debemos fijarnos en cómo se modifican a medida que llegan a su objetivo o se alejan de este.

Además, durante la ejecución no solo almacenaremos las imágenes, acciones u observaciones, sino que con la finalización de cada prueba, guardaremos en otro fichero de extensión .csv la cantidad de iteraciones con las que esa prueba ha finalizado, de modo que en el futuro podamos saber dónde comienza o finaliza cada una de las pruebas, ya que al finalizar, se van a encontrar todas juntas dentro de su carpeta o fichero .csv correspondiente de manera ordenada.

Diagrama

Descripción generada automáticamenteAl haber realizado tantas pruebas y tener un conjunto tanto de datos como de variables tan extenso, podemos tener la problemática de que sobrepasemos la capacidad de nuestra memoria RAM. Para solventar dicho problema, a la hora de ejecutar el script no realizamos todas las pruebas de golpe, dado que esto provocaría lo ahora descrito, sino que ejecutaríamos el mismo script una media de cuatro veces repartiendo las pruebas por ejecución para posteriormente, mediante otro script llamado *dataset\_generator\_apilator.py*, podamos juntar las pruebas y concatenar los datos en un solo conjunto de train y test. Todo este proceso lo realizaremos mediante un *Makefile* para poder de esta manera automatizar el lanzamiento de todas las ejecuciones necesarias para realizar el dataset final que albergue todos los datos necesarios.

Figura x. Proceso para la generación del dataset con MuJoCo

Dentro del directorio en donde tenemos el dataset finalizado, guardamos los datos con la siguiente estructura:

Escala de tiempo

Descripción generada automáticamente

Figura x. Estructura de directorios del dataset con MuJoCo

Dentro de cada directorio de imágenes, tenemos todas las imágenes de ese subconjunto ordenadas por pruebas e iteraciones y renombradas desde 0.png hasta N.png, siendo N el valor de la última iteración para dicho subconjunto. De igual forma, dentro de los ficheros que guardan el vector de acciones *Train\_Actions.csv* y *Test\_Actions.csv* tendremos N y M filas o vectores de acciones, siendo N nuevamente el valor de las iteraciones realizadas en Train y M el valor de las iteraciones realizadas en Test. Lo mismo ocurrirá con los ficheros *Train\_Observaciones.csv* y *Test\_Observaciones.csv*. Finalmente, tenemos los ficheros *Tam\_pruebas\_train.csv* y *Tam\_pruebas\_test.csv*, las cuales tendrán un total de K y L filas, siendo K el numero de pruebas realizadas en Train y L el numero de pruebas realizadas en Test. Hay que recordar que el porcentaje de división entre train y test se mantiene siempre a 0.8, por lo que esa será la relación de valores tanto entre N y M como entre K y L, teniendo en cuenta que no será exacta al ser cada prueba de un tamaño distinto de iteraciones o instantes.

Una problemática que nos surgió a la hora de analizar las imágenes generadas para este primer dataset fue el rango en donde se inicializaban los objetos de cada tarea.

Nos fijamos como los rangos eran muy pequeños y tomaban como base el centro de la mesa con un diámetro de aleatoriedad muy bajo. Esto nos hizo pensar en intentar aumentar dicho rango con el objetivo de añadir una mayor complejidad a la hora de generar y simular tareas, ya que con el rango actual eran muchas las pruebas que coincidían en la posición de sus objetos o se parecían mucho a otras.

Analizando los scripts internos de la librería, encontramos como dentro de *metaworld/envs/mujoco/Sawyer\_xyz/v2* se encontraban los scripts de personalización del entorno de cada una de las 50 tareas disponible en Metaworld. Es aquí en donde probamos a cambiar los valores de inicialización de los objetos pertinentes en cada tarea (teniendo en cuenta las restricciones de movilidad del robot). Un ejemplo de la modificación realizada para una de las tareas probadas, en donde en rojo marcamos el rango inicial y en verde el final, es el siguiente:

Imagen que contiene interior, edificio, montar a caballo, tablero

Descripción generada automáticamente

Figura x. Comparativa entre rango inicial y final

Una vez analizada cada tarea para ampliar los rangos lo necesario, procedimos a comenzar a generar los conjuntos de datos.

## 3.1.2. Generación de datos en CoppeliaSim-RLBench

Con RLBench, al contrario que con Metaworld, disponemos de una menor capa de personalización de pruebas, tanto por la complejidad del entorno de simulación como por los ficheros de extensión .xml que disponen de toda la información necesaria de cada tarea y es dependiente de un conjunto amplio de ficheros los cuales son muy difíciles de modificar sin realizar cambios inoportunos o romper alguna de las dependencias de la tarea.

Pese a ello, algo que hace que no necesitemos realizar ninguna modificación como las realizadas en Metaworld, es como las inicializaciones de los objetos de la escena ya cuentan con un rango lo suficientemente amplio como para necesitar aumentarlo más.

Dicho esto, comenzamos analizando los scripts internos de cada directorio de la librería para ver que datos son los que podemos llegar a obtener tras simular cualquier tipo de tarea.

Dentro del script ubicado en *RLBench/gym/rlbench\_env.py,* disponemos de un conjunto de funciones las cuales nos informan sobre las distintas cámaras que se utilizan dentro de esta librería. Observamos como son un total de 5 cámaras capaces de obtener imágenes de tres clases diferentes. Los cinco tipos de imágenes que se pueden observar de manera ordenada en la figura x son las siguientes:

* **Front:** Cámara fija situada en la parte frontal del robot capaz de mostrar toda la escena.
* **Left:** Cámara fija situada a la izquierda del robot.
* **Overhead:** Cámara fija situada sobre la escena.
* **Right:** Cámara fija situada a la derecha del robot.
* **Wrist:** Cámara fija situada dentro de la pinza del robot.

En cuanto a los tres tipos de imágenes que cada una de estas cámaras es capaz de sacar, en la figura x tenemos un ejemplo de como serian en comparativa. Nosotros hemos decidido usar únicamente las del tipo RGB, ya que aparte de ser el tipo de imagen por defecto y la más económica dentro del mundo real, no todos los entornos de simulación eran capaces de sacar imágenes de otro tipo que no fuera ese.

Una vez sabemos que imágenes podemos extraer de la escena, necesitamos hacernos con un conjunto de acciones con las que guiar al robot para llegar a los objetivos marcados por sus tareas. Esto lo podemos encontrar dentro del script situado en *RLBench/action\_modes.py,* en donde podemos ver como pese a tener un amplio conjunto de acciones posibles, muchas no han sido aun implementadas a expensas de una nueva actualización del entorno de simulación.

De entre las disponibles que hemos comentado anteriormente, hemos decidido probar con la velocidad, fuerza y posición, ya que el end-effector-pose era muy complicado de obtener y generaba demasiadas complicaciones mas adelante a la hora de simular los datos.

A continuación mostraremos un ejemplo de cada una de las tres acciones obtenidas:

**Velocity:**

**[-0.0017, -0.0191, 9.05e-05, -0.0602, 2.86e-05, 0.0218, 0.0002]**

**Force:**

**[-0.0185, 22.897, -0.0492, -15.466, -0.0152, 1.2232, 0.0397]**

**Positions:**

**[-5.93e-05, 0.1769, 1.12e-05, -0.8745, -1.2787, 0.1699, 0.8996]**

Podemos ver como los tres tipos de acciones disponen de un total de siete valores, los cuales representan a cada una de las siete articulaciones del robot, ya que en comparativa con Metaworld, en RLBench para realizar un movimiento debemos de pasar el valor de la acción que debe de ejecutar cada una de las articulaciones, no solo la pinza y que el resto de articulaciones se adapten a esta tal y como ocurría en Metaworld. Esto será interesante, ya que es una diferencia de acciones lo suficientemente grande entre ambos entornos de simulación como para sacar conclusiones de cual es mejor aplicar en el mundo real.

Para la obtención de las acciones, a la hora de ejecutar cada prueba se genera un fichero de tipo .pkl el cual dispone de toda la jerarquía de objetos de Python como una secuencia de bytes la cual debemos de extraer.

Una vez sabemos que datos vamos a extraer de cada prueba y como los vamos a extraer, comenzamos con la generación de los scripts que tengan como objetivo devolver un dataset con todos los datos necesarios de manera similar a como lo hacemos con Metaworld.

Como ya hemos dicho, RLBench es mucho mas complejo que Metaworld por diversos motivos. Es por esto que tras ver la longitud del script inicial que generamos, decidimos dividirlo en tres partes, de modo que tengamos un primer script que realice las pruebas y almacene tanto las imágenes como los ficheros .pkl de cada prueba; un segundo script que partiendo de los .pkl genere tres ficheros de extensión .csv para cada clase de acción a simular y un script final que realice el reparto entre train y test y deje la carpeta lista para ser manipulada en la fase de entrenamiento del modelo.

En nuestro primer script denominado como *dataset\_generator.py*, pasamos los siguientes parámetros de entrada:

* **#PRUEBAS:** Cantidad de pruebas sobre una misma tarea con infinitas variaciones a realizar.
* **#MODIFICACIONES:** Cantidad de modificaciones aleatorias sobre una misma tarea que queremos realizar. Un ejemplo de esto seria hacer que en lugar de apilar un bloque rojo siempre, apile dos o apile uno de diferente color. No se trata de las variaciones sobre una prueba como son la inicialización de las posiciones de los objetos.
* **TAREA:** Se trata de la tarea a realizar.

Una vez hemos terminado con la ejecución del primer script, lanzamos el segundo llamado *Data\_getter.py* al que le pasamos los siguientes parámetros:

* **TYPE:** Se trata de un booleano que usamos para decirle al código si lo que queremos es realizar una ejecución normal (0) o una ejecución en donde tan solo guardemos el 60% inicial de cada prueba de modo que tengamos un dataset exclusivo para el aprendizaje del comienzo de cada tarea.
* **TAREA:** Se trata de la tarea a realizar.

Finalmente ejecutamos *Train\_test\_split\_multimage.py* con los siguientes parámetros:

* **%TRAIN:** Porcentaje del reparto entre los conjuntos de train y test.
* **TAREA:** Se trata de la tarea a realizar.

Al ser tres script dependientes entre sí, con el objetivo de no tener que estar pendientes de cuando finaliza uno para lanzar el siguiente, hemos generado un Makefile que los lance en cascada.

El modo en el que funcionan es idéntico al script utilizado en Metaworld, ya que hemos tratado de realizar el mismo procedimiento adaptándonos a las dependencias de cada entono y librería, por lo que no vamos a entrar nuevamente en detalle de que realiza cada uno.

Diagrama

Descripción generada automáticamenteAlgunas de las diferencias entre ambos entornos de ejecución, aparte de la división en tres partes del código, es la no necesidad de tener que fijar un umbral para el máximo de iteraciones de cada prueba, ya que del modo que las realiza internamente RLBench, cada prueba es única y no depende de un iterador que la vaya realizando iteración a iteración, lo cual resulta mucho más cómodo. Por otro lado, por el modo en el que se ejecuta CoppeliaSim, no es necesario tener que dividir el dataset para después apilarlo. Esto se debe a que internamente, CoppeliaSim va borrando la cache y liberando espacio, algo que viene muy bien si queremos realizar muchas pruebas de golpe. Por otro lado, la librería de RLBench se encuentra mejor optimizada que la de Metaworld, motivo por el cual ahorramos también en memoria RAM.

Figura x. Proceso para la generación del dataset con CoppeliaSim

Imagen que contiene Escala de tiempo

Descripción generada automáticamente De este modo, una vez hayamos finalizado, tendremos la siguiente estructura de directorios:

Figura x. Estructura de directorios del dataset con CoppeliaSim

Podemos ver como una diferencia de esta estructura respecto a la generada con MuJoCo es la falta de un fichero de extensión .csv que nos marque el inicio y final de cada prueba para poder manejarla en el entrenamiento. Esto se debe a que en la siguiente fase, gracias a las diferencias que existen entre valores iniciales y finales de cada prueba, somos capaces de distinguir cuando comienza y cuando finaliza cada prueba sin necesidad de almacenar dichos valores previamente.

Al igual que hacíamos para el anterior entorno de simulación, dentro de cada carpeta de imágenes almacenamos cada iteración ordenada desde 0 hasta N, siendo N el valor del total de iteraciones. Por otro lado, los tres ficheros de acciones disponen de K filas para Train y de L filas para Test, siendo K y L el numero de iteraciones realizadas en train y test.

# 3.2. Fase 2: Entrenamiento del modelo

Una vez hemos finalizado con la generación de datasets para ambos entornos de simulación, vamos a comenzar con la siguiente fase, la cual tiene como objetivo utilizar los datos recopilados para entrenar el modelo que sirva para simular los datos en la última fase.

Para ello, vamos a tratar de aplicar una red neuronal no muy compleja sobre distintas combinatorias de nuestros datos para intentar obtener la mejor configuración posible la cual tomemos como referencia en futuros proyectos de este tipo.

Lo primero que hemos hecho para esta fase ha sido generar un cuaderno de Jupyter [x] con el que ir ejecutando celda a celda para visualizar los resultados e ir modificando por trozos el código de modo que fuera más sencillo e intuitivo.

Como una vez llegados a este punto lo único que diferencia los datasets de los distintos entornos de simulación es un tipo de imágenes de más para Metaworld y la longitud del vector de acciones a predecir que pasa de 3 en el caso de Metaworld a 7 para RLBench, hemos decidido realizar un solo cuaderno el cual iremos adaptando para las pequeñas diferencias de ambos datasets que acabamos de comentar. La implementación de dicho cuaderno es la que viene a continuación.

Nuestro cuaderno de Jupyter comienza con la creación de tres clases para la extracción de los datos, sus transformaciones y generación de la cabeza de la red neuronal.

La primera de las clases se llama *ObsActionGetter()* y es la encargada de recibir la ruta de los ficheros tanto .csv como de la carpeta en donde se encuentran los subdirectorios con las imágenes para devolvernos posteriormente el conjunto de imágenes junto con las acciones ya listas.

Los datos que queremos utilizar para el entrenamiento son tanto las imágenes como las acciones. Sin embargo, nos surge la problemática una vez llegados a este punto de que al tener que predecir las acciones, no podemos pasarle la acción del estado actual, ya que no tendría ningún sentido que intentara predecir uno de los parámetros que le pasamos. Por ello, decidimos utilizar por

cada instante o estado, las imágenes actuales y previas junto con el vector de acciones del estado anterior. De este modo, tendremos una información visual gracias a las imágenes del movimiento realizado junto con la acción previa que representa dicha transición. Cabe destacar que esta es una de las configuraciones posibles, pero en caso de así quererlo, podríamos entrenar con un conjunto de imágenes de mas de un estado de igual forma que podríamos hacerlo con el vector de acciones, el cual sí que quedaría restringido al estado anterior como pronto.

Una duda que nos surge al idear esta idea es la de que hacer con las acciones previas del estado inicial, ya que no existen. Esto hemos decidido solventarlo haciendo que en el estado inicial, las acciones previas sean un vector de ceros, ya que esto implica la negación del movimiento y por tanto, el inicio de cada prueba.

Algo parecido hemos hecho para las imágenes, en donde en caso de estar en el primer estado, como hemos dicho que la acción va a ser un vector de ceros el cual implica no movimiento, tendremos las imágenes del estado previo iguales a las del estado actual.

Para ayudarnos a la hora de saber cuando comienza una prueba y cuando finaliza, usamos el fichero de extensión .csv que incluye estos valores, de modo que si el nombre de una imagen, el cual hace referencia a su instante, se encuentra dentro de dicho fichero, querrá decir que acabamos de comenzar una prueba y por tanto debemos tanto inicializar a ceros el vector de acciones como hacer que las imágenes previas sean iguales a las actuales.

De este modo, gracias a esta clase almacenamos dentro de variables para el entrenamiento las 12 imágenes y el vector de acciones previo. Además, guardamos también el vector de acciones actual para predecir.

Una vez tenemos almacenados todos estos valores, se los pasamos junto con las transformaciones que debemos realizar a la siguiente clase llamada como *ObsActionDataset().* Esta clase es la encargada de aplicar dichas transformaciones sobre las imágenes de manera individual para concatenarlas entre si por canales una vez transformadas. El procedimiento es sencillo, cada una de las 12 imágenes de 128x128x3 se convierte en Tensor [x] y posteriormente se normaliza teniendo en cuenta tanto la media como la desviación estándar que es calculada para el total de las imágenes de Train. Inicialmente utilizábamos las medias y

desviaciones proporcionadas por el dataset de ImageNet, ya que poseen unos valores generalistas para problemas de visión por computador. Sin embargo, más tarde decidimos calcular ambos valores en base al conjunto de train que tuviéramos. Una vez transformadas, nos encontramos con que cada imagen es un tensor de 3x128x128. Estas se concatenan por canales en base a su estado, es decir, las 6 imágenes de cada estado se concatenan entre si para obtener dos tensores de 3x128x768 las cuales se vuelven a concatenar entre si por canales llegando a tener finalmente un tensor de 3x128x1536. Este tensor final será el objeto que devolvamos de esta clase y que pasemos a nuestra red junto con el vector de acciones del estado previo.

La ultima clase que tenemos se llama *MultiImage()* y es la clase encargada de generar la red. Para ello, le pasaremos el extractor de características y la cabeza de la red personalizada, inicializándose con un *AdaptiveAvgPool2d()*, la cual como el nombre indica, hace un pooling de media adaptativo en dos dimensiones con un tamaño de salida de 1x1 para el tensor de imagen. Tras esto, usa un *Flatten()* para aplanar su rango contiguo de dimensiones de modo que obtengamos un tensor, ya que tras aplicar el pooling de media adaptativo, dejamos de tener un objeto de la clase tensor. Finalmente, concatenamos el tensor de imágenes junto con el vector de acciones previo y devolvemos dicho objeto como la cabeza de nuestra red y encargada de comenzar con ella.

Tras esto, cargamos nuestros conjuntos de train y test gracias a la función *DataLoader(),* a la cual le pasamos los objetos devueltos por la clase *ObsActionDataset()* yel batch size fijado a 32 entre otros parámetros.

En cuanto a la red que vamos a utilizar, vamos a hacer uso de una *ResNet18* [x] sin preentrenar de la cual cambiaremos la primera convolución por una personalizada y adaptada a nuestro problema. Una vez hayamos hecho esto, le eliminaremos el avgpool y las capas fully conected quedándonos únicamente con el extractor de características. En cuanto al resto de la red, crearemos una cabeza personalizada adaptada también a nuestro problema. Tanto el extractor de características como esta cabeza personalizada serán los objetos que pasemos a la clase *MultiImage()* durante toda la ejecución del entrenamiento.

Además de esto, durante las primeras cinco épocas de aprendizaje, congelaremos nuestras capas de aprendizaje de modo que no actualicen parámetros haciendo que la optimización de los restantes sea mas eficiente al

realizar menos cómputos de gradientes. A partir de la quinta época, las capas serán descongeladas y comenzarán a actualizar sus pesos.

Por otro lado, y con el objetivo de optimizar el ratio de aprendizaje al máximo, antes de comenzar cada entrenamiento ejecutaremos un test de búsqueda del mejor ratio posible. Lo que haremos será incrementarlo exponencialmente entre dos umbrales prefijados. El umbral más bajo será el que diga a la red donde empezar a converger. A medida que incrementemos este ratio, empezara a ser demasiado grande hasta el punto de que la red diverja. Un ejemplo de lo que sería la evolución del ratio y el que debemos de escoger de esta evolución se puede apreciar en la siguiente figura:

Gráfico, Gráfico de líneas

Descripción generada automáticamente

Figura x. Evolución del LR en el test inicial

Como podemos ver en esta figura a medida que vamos aumentando el valor del LR vamos teniendo una menor perdida. Sin embargo, llega un momento en el que es tan grande que llegamos a mínimos locales y comenzamos a ganar perdida. Es antes de este momento en el que debemos de escoger el LR, con un pequeño margen de modo que no nos llegue a divergir la red.

Para todas las pruebas, tras estudiarlo muy detenidamente, hemos decidido realizar un total de 20 épocas, ya que es sobre la numero 15 sobre la que empezamos a obtener buenos resultados y es a partir de la numero 25 sobre la que tiende a sobreaprender devolviéndonos muy buenos RMSE a la hora de realizar la validación pero errando tras repetir patrones a la hora de realizar las simulaciones.

# 3.3. Fase 3: Simulación y validación del modelo

En esta ultima fase, una vez hemos obtenido el modelo entrenado, volveremos a nuestro entorno de simulación con el objetivo de intentar aplicarlo de modo que el robot se guie por las salidas que este le devuelva.

Con este objetivo, hemos generado un nuevo script para cada uno de los entornos de simulación. Este script, recibe el modelo y en base a las acciones que devuelva, va realizando la simulación de un total de 100 pruebas de las cuales va almacenando las imágenes y los resultados en formato csv para que podamos ver en caso de fallar que es lo que está ocurriendo.

Para ello, comenzamos asignando una semilla fija la cual iremos aumentando en uno por cada prueba que avancemos. De este modo, siempre que queramos probar un modelo lo haremos sobre las mismas pruebas, por lo que podremos ver si un modelo es capaz de solventar errores que otro realizaba o si es capaz de llegar al objetivo de una manera más eficiente.

Una vez tengamos fijada nuestra semilla, comenzaremos con la primera de las cien pruebas que vamos a ejecutar. Asignamos un umbral de fin de prueba para pasar a la siguiente en caso de que el robot no sea capaz de llegar al objetivo en un tiempo límite. Esta acción de ir monitorizando las iteraciones era algo que ya realizábamos en Metaworld a la hora de generar los datos al no tener una función que hiciera que se ejecutara una prueba al completo. Sin embargo, con RLBench no teníamos que preocuparnos de este problema en la primera fase. Aquí sin embargo, al poder llegar a un estado en el que el robot no sea capaz de alcanzar el objetivo, hemos tenido que desglosar las funciones proporcionadas por RLBench de modo que el paso de un estado a otro sea realizado manualmente sin tener que hacer uso de las funciones predeterminadas para ello.

Tras esto, inicializamos la prueba. Para poder predecir qué acción ha de realizar el robot, debemos pasarle al modelo el conjunto de imágenes del estado actual junto con las del estado previo además del vector de acciones previo. Cuando nos encontremos en el estado inicial, no existirá un conjunto previo de datos, por lo que haremos lo mismo que realizábamos en la segunda fase de entrenamiento del modelo, asignar a ceros el vector de acciones e igualar el

conjunto de imágenes del estado previo al del estado actual para de esta manera hacer ver al modelo que no se ha realizado ningún movimiento. De este modo, pasaremos al modelo estos datos para recibir la primera acción a realizar y reasignaremos las variables de manera que los datos del estado actual pasen a ser los del estado previo y extraigamos las imágenes del estado al que hemos avanzado para asignárselas al conjunto de imágenes del estado actual. Repetiremos este proceso hasta que o bien superemos el umbral de iteraciones máximas o bien alcancemos el objetivo y demos paso a la siguiente prueba.

Estas dos maneras de pasar de una prueba a otra tienen un significado muy distinto, ya que si alcanzamos el umbral máximo, querrá decir que no hemos finalizado la prueba y por tanto hayamos fallado. Sin embargo, si alcanzamos el objetivo, querrá decir que la prueba ha sido finalizada satisfactoriamente y que por tanto, hemos obtenido un acierto.

Para saber si hemos alcanzado el objetivo, realizamos diferentes comprobaciones en base al entorno de simulación que estemos utilizando. En el caso de estar trabajando con Metaworld, tal y como ya mencionamos en el apartado de la primera fase de generación de datos, existe un objeto que almacena toda la información del entorno. Es este objeto el que nos devolverá los valores que debemos de evaluar para saber si una prueba a finalizado o no, del mismo modo que hacíamos a la hora de generar los datos cuando queríamos saber si habíamos llegado al final de una prueba. El valor que debemos de analizar difiere dependiendo del tipo de tarea que estemos simulando, ya que puede darse el caso de que exista una tarea que finalice siempre que uno de los valores se encuentre por encima de cierto umbral del mismo modo que puede haber otra que finalice siempre y cuando la diferencia de un valor entre dos estados contiguos sea lo suficientemente grande o pequeña. Esto es algo que tendremos que analizar en la primera fase y que nos servirá llegados a esta ultima fase de igual forma. En cuanto a RLBench, al poder calcular las posiciones tanto del centro de la pinza como del objetivo al igual que el estado de la pinza, tan solo debemos de analizar si ambas posiciones se encuentran lo suficientemente cerca como para poder confirmar que se ha llegado al objetivo. Además de esto, tendremos que tener en cuenta el estado de la pinza, ya que no es lo mismo llegar a una pelota que llegar y cogerla.

Con el objetivo de poder evaluar a posteriori cada simulación, almacenaremos las imágenes dentro de una carpeta separada en el tipo de cámara que se usa de la misma manera que hacíamos a la hora de generar el dataset. Además de esto, almacenaremos dos ficheros con extensión .csv. El primero de ellos, almacenara las acciones predichas durante la simulación para que podamos sacar estadísticos de estas y compararlas junto a las imágenes correspondientes a dicho estado para saber que esta ocurriendo en todo momento. En cuanto al segundo de los ficheros, almacenaremos un estadístico por cada prueba realizada que será el que evalué el correcto comportamiento de cada modelo en el simulador. Este fichero csv almacenara la siguiente información:

* **Inicio**: Sera el valor de la iteración en la que comienza dicha prueba.
* **Final**: Sera el valor de la iteración en la que finaliza dicha prueba.
* **Iteraciones**: Sera el valor de las iteraciones realizadas de inicio a fin.
* **Finaliza**: Sera un valor booleano fijado en 0 si no llega al objetivo y a 1 si es capaz de llegar al objetivo.
* **Acierto**: Sera el porcentaje acumulativo de acierto por cada prueba. El valor que realmente nos interesara será el que se muestre junto a la ultima de las pruebas, ya que será el porcentaje total de las 100 pruebas.

A continuación se muestra un extracto de este fichero de resultados:

Tabla

Descripción generada automáticamente

Figura x. Extracto del fichero de resultados devuelto tras la simulación

En esta figura podemos observar como existe una prueba la cual no llega al objetivo y se consume tras las 200 iteraciones máximas. Esto hace que baje el porcentaje de acierto acumulativo el cual es recalculado en base al numero total de pruebas que hemos realizado hasta el momento. Esto se debe a que a medida que realicemos más y más pruebas, un acierto o un error tendrá un menor impacto en el porcentaje final, ya que no es lo mismo fallar una prueba de dos y tener un 50% de acierto que fallar una de cien y tener un 99%.

Capítulo 4

Resultados obtenidos