Resumen

El proyecto explora las oportunidades que ofrece el aprendizaje automático por refuerzo al campo de la robótica mediante la implementación del algoritmo por refuerzo profundo *DDPG*, inspirado en los gradientes de políticas deterministas, para evaluarlo en una serie de entornos diferentes, con diferentes arquitecturas y parámetros. También se compara su rendimiento con el del planificador de última generación *KPIECE* en el campo de la planificación de movimientos. El proyecto se diseña con un enfoque práctico, con posibilidad de llegar a implementar los métodos estudiados en robots reales.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La planificación de movimientos sigue siendo, a día de hoy, un problema importante en la implementación de sistemas robóticos en el mundo real, debido principalmente al alto coste computacional y temporal que supone ejecutar los planificadores, por lo que los principales esfuerzos en el campo se centran en el diseño de algoritmos más eficientes[16]. Los problemas de planificación de movimientos sufren de la *maldición de la dimensión*[1]; la complejidad de los problemas crece de forma exponencial con relación al número de dimensiones del espacio de configuraciones[23]. Es común encontrar la necesidad de realizar varios millones de cálculos de colisiones para encontrar un camino libre en escenas con robots de 6 grados de libertad en entornos medianamente complejos, una situación típica en la industria[16].

Este problema se ve aún más acentuado en el caso de entornos en los que existen objetos móviles, con los cuales el robot puede interactuar. Dichos problemas se ven sujetos a una explosión combinatoria en complejidad debido a la necesidad de generar caminos múltiples y objetivos intermedios[23], además de sufrir de una penalización de precisión en comparación con las trayectorias exentas de colisiones debido a las limitaciones de los modelos de simulación física que utilizan[6].

Es precisamente debido a estas dificultades que los recientes avances en aprendizaje automático se muestran tan prometedores. El aprendizaje automático profundo, todo aquel que utilice redes neuronales profundas, consigue escapar, al menos en parte, de la *maldición de la dimensión*, al ser capaz de abstraer información al transmitirla de capa en capa, buscando patrones y dibujando una representación jerárquica de la información de entrada[45], pudiendo aprender a generalizar[6]. El uso de redes neuronales profundas también ha demostrado ser una alternativa más rápida y robusta que los

[1] Conocida en la literatura de aprendizaje automático como *curse of dimensionality*, en inglés.

planificadores tradicionales en entornos con objetos móviles[6].

Son estas características las que dotan al aprendizaje automático profundo por refuerzo del potencial del que goza en la actualidad, sobre todo en el campo de la robótica, donde resulta muy útil la capacidad de aprender de la experiencia y generalizar para enfrentarse a situaciones noveles. En octubre de 2017, el algoritmo *AlphaGo Zero*, desarrollado por *DeepMind Technologies Limited* aprendió a jugar al juego de Go de forma completamente independiente, sin más datos que las reglas del juego y aprendizaje automático profundo por refuerzo[43], superando a todas sus iteraciones previas, las última de las cuales había vencido en 2015 a Fan Hui[2], siendo esta la primera vez que un ordenador había vencido a un profesional en igualdad de condiciones[41], y a Lee Sedol[3] en 2016[2]. Tal es la potencia y aparente creatividad del algoritmo, que se está estudiando su estilo de juego y extrayendo nuevas ideas y perspectivas que están influenciando al mundo del Go[26], lo cual es impresionante si se tiene en cuenta que se trata del juego más antiguo aún jugado en su forma original, con su nacimiento datando hace 2.500 a 4.000 años atrás[1].

Este proyecto nace con el objetivo de aplicar un algoritmo de aprendizaje automático por refuerzo a un problema robótico real, utilizando hardware, para comparar su rendimiento con el de otros planificadores convencionales. Para ello, se ha escogido el algoritmo *DDPG*[27] (*Deep deterministic policy gradients*[4]), un algoritmo de refuerzo basado en los resultados obtenidos por el algoritmo *DQN* (*Deep Q-Networks*[5]), con el que se consiguieron muy buenos resultados en 2015, alcanzando niveles humanos de destreza al jugar a juegos de *ATARI* utilizando solamente los píxeles que estos mostraban por pantalla como entradas[33], pero expandiéndolo a un dominio continuo y haciendo uso de gradientes de políticas deterministas[42].

1.1. Objetivos del proyecto

A continuación se expone una declaración concisa de los objetivos del proyecto:

- Implementar un sistema de planificación de movimiento utilizando el algoritmo *DDPG*.

[2]Profesional de 2 dan, campeón del campeonato Europeo de Go de 2013, 2014 y 2015[41].

[3]Campeón de 18 títulos mundiales y reconocido como el mejor jugador de la década[2].

[4]*Gradientes de políticas deterministas profundas*, en castellano.

[5]*Redes de valores Q profundas*, en castellano.

- Elaborar una comparativa del rendimiento del algoritmo implementado y otros planificadores de última generación en simulaciones.

- Elaborar una comparativa del rendimiento del algoritmo implementado y otros planificadores de última generación en un robot real *YuMi* de *ABB*.

1.2. Alcance del proyecto

La realización del proyecto consta de dos componentes principales. Por un lado, se realiza un análisis tanto del algoritmo escogido, *DDPG*, como de cada una de las partes que lo conforman, entendiendo así mejor la segunda parte experimental.

Dicha parte experimental cuenta a su vez con 3 problemas diferenciados. En primer lugar, se escoge, diseña e implementa el sistema de generación de trayectorias a utilizar, el cual será controlado por las salidas del algoritmo pero del que no forma parte estrictamente.

Las dos últimas partes consisten en la elaboración y puesta en marcha del software necesario para implementar y ejecutar tanto el algoritmo como el planificador en el simulador y en el robot real, con tal de poder elaborar una comparativa que servirá como los resultados del proyecto.

En resumen, el proyecto detalla como se lleva la idea de un algoritmo a una aplicación real, diseñando el software necesario para su implementación tanto simulada como en un entorno físico y módulos de comparación con otros planificadores de última generación.

1.3. Organización de la memoria

La memoria se divide en 6 capítulos; en el primer capítulo, *Introducción*, se exponen los objetivos, la temática y la motivación del proyecto.

A continuación se pasa a un breve resumen del estado del arte de los campos del aprendizaje por refuerzo, del aprendizaje automático en general y de la planificación de movimientos en el capítulo 2, *Estado del arte*, para contextualizar el proyecto.

En el capítulo 3, *Algoritmo*, se examina más a fondo el algoritmo de aprendizaje por refuerzo escogido, *DDPG*, así como sus partes integrantes.

En el capítulo 4, *Detalles experimentales*, se discute la implementación del proyecto y los experimentos, tanto el apartado de *software* como la disposición experimental física.

El capítulo 5, *Resultados*, muestra los resultados de los experimentos, las curvas de aprendizaje del algoritmo, las diferencias entre distintas implementaciones y las comparaciones entre los planificadores.

Por último, el capítulo 6, *Conclusiones y trabajo futuro*, pretende valorar los resultados del capítulo anterior y extraer información útil que pueda ayudar a implementaciones futuras o a introducir posibles mejoras.

Capítulo 2

Estado del arte

Uno de los atractivos del proyecto es el gran número de campos que cubre, pues sirve de puente entre el aprendizaje automático y el aprendizaje por refuerzo en particular, con la planificación de movimientos robóticos, en especial en entornos poblados. A continuación se expone un breve estado del arte de cada uno de estos subcampos para situar el resto del documento en contexto.

2.1. Aprendizaje automático

El aprendizaje automático podría definirse como la ciencia o el campo dedicado a conseguir que los ordenadores aprendan a actuar sin haber sido programados explícitamente para comportarse de una manera u otra ante un estímulo. Tom Mitchell define el aprendizaje dentro de un programa de la siguiente manera[31]:

> Se dice que un programa informático *aprende* de una experiencia E con respecto a una tarea T y una medida de rendimiento P si su rendimiento en T, medido a partir de P, mejora con la experiencia E.

El aprendizaje automático goza de una popularidad creciente en la actualidad, impulsada principalmente por dos factores, el surgimiento de cantidades de información masivas con las cuales se pueden entrenar algoritmos, el denominado *big data*, y el aumento en potencia del hardware necesario para hacer posible un procesamiento de datos a este nivel[13]. Sobre todo en internet, su uso ya es común para tareas como recomendaciones de vídeos a ver[14], o qué comprar[5] y han dado lugar a implementaciones impresionantes de reconocimiento visual de objetos en tiempo real[39] o procesado de lenguaje[13], por nombrar unas pocas aplicaciones.

El aprendizaje automático se suele subdividir en 3 subcategorías:

El aprendizaje automático supervisado: Su objetivo es el de aprender una transformación de una serie de entradas a una serie de salidas, para ello dispone de información *etiquetada* sobre estas tuplas. Dispone de la respuesta correcta para cada entrada y comparando sus salidas con las respuestas correctas, estos métodos son capaces de corregirse hasta aprender la función deseada. Si las salidas son categóricas, se suele hablar de problemas de clasificación, mientras que cuando estas son continuas, se trata de problemas de regresión[35].

El aprendizaje automático no supervisado: Este tipo de métodos no cuentan con respuestas correctas para aprender relaciones específicas, sino que su objetivo es el de encontrar *patrones interesantes* en conjuntos de datos[35].

El aprendizaje automático por refuerzo: Esta clase de métodos se diferencia de los dos anteriores por basarse en una interacción explícita con algún tipo de entorno del cual extrae su propia información. Dado un estado, los métodos de aprendizaje por refuerzo tratan de tomar la mejor acción con tal de maximizar una señal de recompensa dada, dependiente de los estados siguientes. Es por esto que presenta un problema adicional al cual escapan los otros dos métodos, el de encontrar un equilibrio entre la *exploración* del espacio de estados y la *explotación* de las acciones que presentan mayor recompensa futura[45].

2.2. Aprendizaje por refuerzo

En esta sección se comentarán los principales tipos de métodos que emplean los algoritmos de aprendizaje por refuerzo.

2.2.1. Métodos tabulares

Llamados así porque tratan de entornos donde los espacios de estados y de acciones son suficientemente pequeños como para poder ser representados mediante tablas, es posible almacenar todos sus valores. Esto permite a los algoritmos encontrar la solución óptima, una vez se han explorado por completo todos los estados[45].

Programación dinámica: Esta colección de algoritmos se basan en la existencia de un modelo perfecto del entorno, con el cual son capaces de

computar políticas óptimas. No suelen ser prácticos en casos reales debido a la carga computacional que conllevan y el hecho de requerir modelos precisos del entorno. Se basan en la actualización del valor de un estado a partir del valor de todos los estados sucesivos posibles, junto a la probabilidad de que ocurran. Después de visitar los estados y recibir recompensas por su comportamiento en ellos suficientes veces, los valores de estos acaban convergiendo y la política se vuelve óptima. Al concepto de actualizar estimaciones de valores a partir de otras estimaciones se le conoce como *bootstrapping*[45].

Métodos Monte Carlo: A diferencia de la programación dinámica, estos métodos son capaces de aprender de la experiencia, necesitando únicamente un modelo que sea capaz de generar muestras de posibles transiciones, sin tener en cuenta las probabilidades exactas de que estas ocurran. Se basan en promediar retornos, sumas de recompensas futuras descontadas, de pares acción-estado que parten de cada estado y por lo tanto no hacen *bootstrapping*[45].

Aprendizaje por diferencias temporales: Presentan ideas de la programación dinámica y de los métodos de Monte Carlo; son capaces de aprender de la experiencia sin un modelo de las dinámicas del entorno y son capaces de actualizar sus estimaciones basándose en otras estimaciones, sin tener que esperar al final del episodio para actualizarse, realizan *bootstrapping*. Son los métodos más utilizados actualmente debido a su simplicidad y capacidad de ser ejecutados en línea[45].

2.2.2. Métodos de solución aproximada

Este tipo de métodos son capaces de hacer frente a espacios de estados y acciones arbitrariamente grandes. Para ello, estos métodos deben ser capaces de generalizar, siendo capaces de actuar de forma similar al encontrar un estado nuevo pero similar a uno ya explorado. La forma más común de hacer frente a este problema es mediante la aproximación de funciones[45].

Predicción *on-policy* con aproximación: Mediante aproximación parametrizada de funciones, estos métodos son capaces de predecir valores para los estados, corrigiéndolos después con el retorno real de ese estado utilizando esa política[45].

Control *on-policy* con aproximación: Estos métodos recogen las ideas de la predicción *on-policy* con aproximación y trasladando las predicciones al espacio de acciones[45].

Métodos *off-policy* con aproximación: Esta clase de métodos se fundamenta en las mismas ideas que los métodos *on-policy*, pero presenta unos problemas añadidos en cuanto a estabilidad del aprendizaje y la distribución de las actualizaciones. En contrapartida, aportan mayor flexibilidad en el conflicto entre exploración y explotación que las versiones *on-policy*[45].

Métodos de gradientes de políticas: En este tipo de métodos la política parametrizada es actualizada con tal de maximizar alguna medida de rendimiento utilizando su gradiente respecto los parámetros de dicha política. No es necesaria, aunque puede utilizarse para aprender los parámetros de la política, una función que aproxime el valor de los estados para escoger acciones. Los algoritmos que aprenden aproximaciones a las funciones de la política y de valores se conocen como algoritmos *actor-critic* (actor-crítico). Una propiedad importante de esta categoría de algoritmos es que pueden tratar de forma natural con espacios de estados continuos[45].

2.2.3. Otros conceptos

A continuación se definen un par de conceptos recurrentes al hablar de algoritmos de aprendizaje por refuerzo.

***On-policy* y *off-policy*:** Un algoritmo es *on-policy*, siguiendo la política, si se actualiza mediante las acciones que tomaría su política actual, a la vez que mueve su política hacia la óptima. Un algoritmo es *off-policy*, o no siguiendo la política, si sus actualizaciones se realizan respecto al óptimo global, de manera que es capaz de optimizar una política óptima empleando cualquier otra. Un ejemplo de un algoritmo *off-policy* sería el *Q-learning*[45].

***model-based* y *model-free*:** En el aprendizaje por refuerzo, los algoritmos con modelo, *model-based algorithms*, son aquellos que poseen algún tipo de capacidad para realizar inferencias sobre como se comportará el entorno a partir de un estado y unas acciones, permitiendo más o menos planificación en el futuro. Los algoritmos sin modelo, *model-free algorithms*, carecen de esta capacidad y funcionan estrictamente a partir de prueba y error[45].

2.3. El potencial del aprendizaje automático profundo

El aprendizaje automático profundo es aquel que emplea redes neuronales de más de dos capas, el denominado *deep learning*. Una sola capa, con un número suficiente de unidades, es capaz de aproximar cualquier tipo de función[45] pero el uso de más capas ofrece beneficios importantes.

El uso de redes neuronales profundas supone una oportunidad importante para el aprendizaje automático en general y para el aprendizaje por refuerzo en particular, ya que tal y como se comenta en la introducción, poseen la capacidad de abstraer información y extraer patrones complejos a partir de información básica, como podrían ser los píxeles de una cámara. Para poner en contexto la importancia que el uso de redes neuronales puede tener en el aprendizaje por refuerzo, el juego de *backgammon* dispone de unos 1.020 estados diferenciados, lo que supone un gran inconveniente a la hora de conseguir experiencia para todos ellos. Sin embargo, Gerry Tesauro fue capaz de entrenar a un programa de aprendizaje por refuerzo utilizando redes neuronales en la década de los 90 para que jugara a un nivel equivalente al de los mejores jugadores humanos del planeta, gracias a su capacidad de abstraer estados y relacionar estados similares entre si, todo de forma implícita[45]. Tal y como se expuso con anterioridad, la tecnología ha avanzado desde entonces para permitir un poderío de las máquinas indiscutible en juegos a los que se pensaba que no sería capaz de destronar a los mejores humanos, como es el caso del GO[41]. Mucha literatura de aprendizaje por refuerzo se basa en el dominio de juegos, ya que estos se presentan a los algoritmos como un 'mundo' simplificado, fácil de entender y evaluar[33] [15].

Uno de los problemas más importantes a los que se enfrenta la tecnología de las redes neuronales a la hora de verse implementada en sistemas críticos o que de alguna manera puedan causar situaciones de peligro, categoría relevante al campo de la planificación de movimientos robóticos, es su *no auditabilidad*. Este problema se conoce también por el nombre de *the black box problem*[1] y consiste, en esencia, en la alta dificultad que supone extraer el *porqué* del resultado de la red neuronal, debida a la forma en la que dichas redes almacenan la información aprendida, dispersa en los pesos sinápticos entre sus unidades[11].

El problema es más grave de lo que puede parecer en un principio, pues el no poder depurar responsabilidades desemboca en la imposibilidad de realizar juicios de valor sobre la lógica detrás de las decisiones que pueda tomar la red, por lo que esta se vuelve vulnerable a llegar a conclusiones erróneas debidas

[1] *El problema de la caja negra*, en castellano.

a un proceso de aprendizaje poco cuidadoso, como fue el caso de un estudio acerca del riesgo que presentaban diferentes pacientes de neumonía dados diferentes datos. El estudio comparaba diferentes métodos de predicción del riesgo, y descubrió que, aunque las redes neuronales ofrecían los mejores resultados en cuanto a precisión, no eran una buena elección, puesto que, incluido en el resto de métodos evaluados, se encontraba también un método de aprendizaje basado en reglas, y una de las reglas que había aprendido, regla que sin duda también había aprendido la red neuronal, aseguraba que los pacientes con asma presentaban un riesgo menor que el resto. El problema radicaba en los datos que habían sido suministrados para el aprendizaje, los pacientes con asma eran colocados directamente en cuidados intensivos, por lo que reducía su riesgo, con lo que los algoritmos, al actuar sin lógica alguna, llegaron a una conclusión peligrosa[10]. En este caso se pudo detectar el error gracias a la presencia de un algoritmo transparente, pero la naturaleza de las redes neuronales hace que sean inescrutables por si solas, no es posible saber, al día de hoy, que otros errores pueden estar ocultando. Otros casos más recientes incluyen los accidentes de coche de *Tesla*[22] y *Google*[48] al interpretar erróneamente la situación en la que se encontraban.

Otro ejemplo de este tipo de fenómeno se puede encontrar en el trabajo del equipo del científico computacional de la Universidad de Wyoming, Jeff Clune, que consiguió crear imágenes consistentes en ruido aparentemente aleatorio o patrones geométricos, que las redes identificaban con alta precisión como objetos o seres concretos, *engañándolas* con aparente facilidad[36].

Hay quienes[2] aseguran que este tipo de comportamientos son inevitables en un mundo complejo y que no supone un problema tan grave. Sin embargo, se están tomando medidas para combatir este problema y poder analizar mejor el comportamiento de las redes neuronales. Un método propuesto es el de intentar reconstruir una imagen a partir de la representación de esta en una red neuronal ya entrenada, encontrando una serie de reconstrucciones y observando los elementos invariantes, que definen la imagen original, según la red[29]. Otro método parecido consiste en ir actualizando la imagen de entrada de manera que se parezca cada vez más a lo que una red neuronal, ya entrenada y con pesos sinápticos fijos, entiende como un objeto en concreto[34]. Este procedimiento no solamente ofrece información sobre qué entiende la red cuando identifica algo, sino que además produce resultados sorprendentemente artísticos[3].

En adición a los progresos realizados en la visualización y análisis de las

[2]Stéphane Mallat, matemático de la Escuela Politécnica de París o Pierre Baldi, investigador en aprendizaje automático de la Universidad de California en Irvine[11]

[3]El algoritmo utilizado se llama *Deep Dream* y se puede descargar en formato *python notebook* del siguiente repositorio: https://github.com/google/deepdream

redes, se están desarrollando algoritmos de depuración sistemática de redes, como *DeepXplore*[38], de tal manera que se encuentran, de forma automática, casos en los que estas erran y así se consigue mejorar su precisión.

A pesar de estas dificultades, y vistos los pasos que se están dando para mitigar las deficiencias, resulta muy interesante el desarrollo de métodos de aprendizaje automático para la generación de trayectorias robóticas.

2.4. La generación de trayectorias robóticas

La planificación de movimientos robóticos se realiza en el denominado *espacio de configuraciones*, espacio con tantas dimensiones como grados de libertad posea el robot en cuestión, permitiendo, de esta manera, reducir toda la cadena cinemática del robot a un punto y transformar el problema de llevar el robot, sujeto a restricciones, de una configuración inicial a una final, al de encontrar un camino válido de un punto a otro en el espacio de configuraciones.

Este es un enfoque muy práctico, por lo que la mayoría de planificadores lo utilizan. Sin embargo, sufre de una serie de inconvenientes[16]:

- Las superficies en el espacio de configuraciones[4] se gobiernan por ecuaciones altamente no lineales.
- Las superficies en el espacio de configuraciones no se pueden representar de forma compacta con facilidad.
- El espacio de configuraciones suele presentar una alta dimensionalidad en robots industriales (6 o más grados de libertad).

La mayoría de los planificadores contemporáneos se basan en la discretización del espacio de configuraciones, por lo que su eficacia muchas veces radica en el planteamiento de sus algoritmos de exploración y de detección de colisiones[16]. Dentro de la exploración, existen dos enfoques principales, especialmente dentro de los planificadores en forma de árbol: por dónde debería seguir la exploración, el 'guiado' de la exploración, de que nodos se debería partir, y *cómo* debería realizarse esta exploración, afrontando problemas tales como la detección de pasadizos estrechos para decidir que direcciones deberían tomarse para explorar desde un nodo[49].

Para espacios de configuración con 4 o menos dimensiones, existen algoritmos capaces de discretizar el espacio y encontrar cualquier camino libre

[4]Aquellas regiones formadas por configuraciones no asumibles por el robot, ya sea por la presencia de obstáculos, porque se encuentren fuera del volumen de trabajo del robot o por otras restricciones.

de colisiones, si este existe, en segundos o minutos[16]. Para espacios de configuraciones con una dimensionalidad más elevada, se han desarrollado toda una gama de técnicas que se expondrán a continuación.

2.4.1. Enfoque clásico

Métodos basados en la descomposición por celdas: Son los métodos más estudiados[23]; consisten en descomponer el espacio en celdas, ya sea mediante métodos *exactos*, generando celdas a partir de puntos críticos en el espacio de configuraciones o mediante descomposiciones aproximadas, subdividiendo el espacio recursivamente en formas predeterminadas, como podrían ser cuadrados, hasta que cada división permanece a un espacio completamente libre o se alcanza una resolución máxima predeterminada[23]. Una vez se ha discretizado el espacio de esta manera, se genera un grafo con uniones entre las distintas celdas y se busca un camino entre las celdas inicial y final en este grafo[23].

Métodos basados en campos de potenciales: Estos métodos se basan en la generación de un campo de potenciales en el espacio de configuraciones. Los obstáculos generan un potencial negativo mientras que la configuración objetivo genera uno positivo. Así, el robot se mueve de la posición inicial a la objetivo bajo el influjo de fuerzas repulsivas y atractivas generadas por el campo de potenciales. Son los métodos más eficaces, pero también son susceptibles a quedarse atrapados en mínimos locales[23].

Métodos basados en mapas de ruta: Estos métodos se basan en la construcción de caminos válidos (que no interseccionan con objetos en el espacio de configuraciones) que atraviesen el espacio, escogiendo después el camino más corto que lleve del punto inicial al final. Los algoritmos y tipos de mapas de ruta generados basados en esta idea básica son varios[23].

2.4.2. Métodos basados en el muestreo

Los métodos mencionados anteriormente se fundamentan en la representación explícita del espacio de configuraciones; este hecho hace que se vuelvan menos prácticos a medida que aumenta el número de dimensiones del espacio de configuraciones. Los planificadores basados en el muestreo solventan este problema valiéndose del bajo coste que supone evaluar si cierta configuración

es válida o no. Con un acceso mucho más limitado al espacio de configuraciones, estos métodos generan trayectorias tomando muestras del espacio y evaluando su validez[12].

***Probabilistic Road Maps (PRM)*:** Consta de dos partes. En una, se sondean puntos al azar del espacio de configuraciones y se conectan utilizando un planificador local simple y rápido, formando un mapa de ruta. En la segunda parte, se encuentra un camino entre los nodos inicial y final utilizando este mapa de rutas. Aplicable a casi cualquier tipo de robot holonómico. Es especialmente útil para problemas con 5 o más grados de libertad en comparación con los métodos clásicos[18].

***Rapidly-exploring Random Trees (RRT)*:** Un método diseñado para poder hacer frente a restricciones no holonómicas, problemas kinodinámicos y elevados grados de libertad. A partir de una muestra inicial, se sondea el espacio y se van generando nuevos vértices al ir conectando las muestras iniciales con muestras nuevas sesgadas hacia la posición de la muestra aleatoria, tal que el nuevo punto se encuentra en una región válida, a una distancia máxima y es posible alcanzarlo desde el anterior punto mediante un control también válido. Para dirigir la exploración hacia el punto objetivo, la muestra aleatoria hacia la cual se expande el árbol es dicho punto objetivo con cierta probabilidad. De entre sus propiedades cabe resaltar que es probabilísticamente completo bajo condiciones muy generales y muestra sesgo exploratorio hacia regiones del espacio no exploradas aún[24],[25]. Existen también variantes bidireccionales con árboles que surgen del punto inicial y objetivo[21].

***Kinodynamic Motion Planning by Interior-Exterior Cell Exploration (KPIECE)*:** Planificador con estructura de árbol basado en la discretización basada en celdas a varios niveles para estimar la cobertura del espacio de estados. A partir de esta estimación es capaz de detectar las áreas menos exploradas dentro del espacio de estados. Está específicamente diseñado para sistemas con dinámicas complejas donde es necesaria una simulación física[49].

Capítulo 3

Algoritmo

En este capítulo se presentarán la ideas principales tras el algoritmo DDPG, así como la implementación que se realizó en el artículo original[27] y se explicarán después, en más detalle, las técnicas que este incorpora procedentes de otros trabajos anteriores.

El algoritmo DDPG surge del deseo de aplicar las ideas del algoritmo DQN[33] al espacio de acciones continuo. Para ello se parte de la base de DPG[42]. El resultado es un algoritmo *actor-critic model-free off-policy* de aprendizaje por refuerzo profundo robusto con muy buenos resultados en espacios de estados y acciones tanto bajos como altos[27].

3.1. Antecedentes

Como se ha comentado con anterioridad, el algoritmo DDPG está basado en el algoritmo DQN y en DPG. A continuación se explican brevemente y se señalan las aportaciones que cada uno hizo a DDPG para así poner el algoritmo en contexto.

3.1.1. Deep Q-Networks

El algoritmo DQN, *Deep Q-Networks*, es un algoritmo de aprendizaje por refuerzo *model-free* y *off-policy* basado en el algoritmo *Q-learning*[47]. Su objetivo era el de combinar los avances recientes en *deep learning* y aprendizaje por refuerzo para crear un algoritmo que fuera capaz de aprender buenas políticas a partir únicamente de datos de vídeo, entradas de muy alta dimensionalidad. Para ello se le entrenó con juegos de *ATARI*, con la información que se mostraba en pantalla como única entrada al sistema[33].

Para procesar las entradas, hizo uso de redes neuronales convolucionales.

Se basaba en el uso de una política voraz donde se seleccionaba la acción que mayor valor tenía para un estado determinado en cada paso a la vez que se seguía una distribución de comportamiento que aseguraba una exploración adecuada[33]. Un par de avances importantes fueron el uso de *experience replay*[28] y de *target networks*[32], avances que DDPG después adoptaría.

Los resultados fueron muy prometedores, consiguiendo rendimientos superiores a todos los algoritmos anteriores y hasta superhumanos en juegos que no requerían demasiada planificación en el futuro, todo sin cambiar de hiperparámetros o arquitectura de juego en juego[33] [32].

3.1.2. Deterministic Policy Gradient

El gradiente de la política determinista, *Deterministic Policy Gradient* (DPG), μ parametrizada por una serie de parámetros θ para mapear estados s a acciones $a = \mu_\theta(s)$, presenta ventajas frente al de políticas estocásticas $\pi_\theta(a \mid s) = \mathbb{P}[a \mid s; \theta]$ en las que se selecciona una acción a de forma estocástica a partir de un estado s. En el caso estocástico, el gradiente de la política debe integrar en el espacio de estados y en el de acciones, mientras que en el caso de la política determinista, solo debe hacerlo sobre el de estados, lo que lleva a que una política estocástica necesite más transiciones para alcanzar un rendimiento similar al de una política determinista, sobre todo si el espacio de acciones es de dimensionalidad elevada[42].

En *Deterministic Policy Gradient Algorithms*[42], se introduce el teorema del gradiente de la política determinista:

Teorema del gradiente de la política determinista: Si un proceso de decisiones de Markov cumple con que $p(s' \mid s, a)$, $\nabla_a p(s' \mid s, a)$, $\mu_\theta(s)$, $\nabla_\theta \mu_\theta(s)$, $r(s, a)$, $\nabla_a r(s, a)$, $p_1(s)$ son continuos en todos los parámetros y variables s, a, s' y x, entonces $\nabla_\theta \mu_\theta(s)$ y $\nabla_a Q^\mu(s, a)$ existen y el gradiente de la política determinista también existe y es[1]:

$$\begin{aligned} \nabla_\theta J(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a) \mid_{a=\mu_\theta(s)} ds \\ &= \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a) \mid_{a=\mu_\theta(s)}] \end{aligned} \quad (3.1)$$

Los resultados con gradientes de políticas deterministas han mostrado rendimientos varios ordenes de magnitud más elevados que algoritmos que

[1]Donde μ_θ es una política determinista que mapea el espacio de estados $\mathcal{S}$ al de acciones $\mathcal{A}$: $\mu_\theta : \mathcal{S} \to \mathcal{A}$ mediante un vector de parámetros $\theta \in \mathbb{R}^n$, $J(\mu_\theta) = \mathbb{E}[r_1^\gamma \mid \mu]$ es el objetivo de rendimiento a optimizar y se define una distribución de probabilidades $p(s \to s', t, \mu)$ y una distribución de estados descontada $\rho^\mu(s)$.

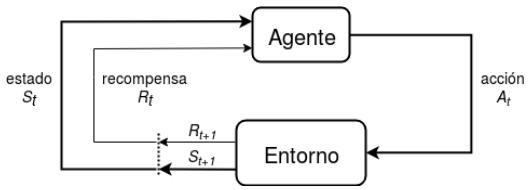

Figura 3.1: Diagrama de un agente interactuando con el entorno[45].

utilizan políticas estocásticas, sobre todo en tareas de dimensionalidad elevada, mostrando un coste computacional lineal de actualización con respecto a dimensionalidad del espacio de acciones[42].

3.2. Deep Deterministic Policy Gradients

El problema se modela como un proceso de decisión de Markov (MDP); el algoritmo que aprende y toma las decisiones se conoce como *agente* y todo aquello con lo que interacciona es el *entorno*[45]. Estos interaccionan continuamente, el agente selecciona acciones a_i de un espacio de acciones $\mathcal{A} = \mathbb{R}^N$ y el entorno devuelve un nuevo estado s_{i+1} del espacio de estados $\mathcal{S}$ y una recompensa r, según una función de recompensa $r(s_t, a_t)$, tal y como puede verse en la figura 3.1[27]. El comportamiento del agente es gobernado por una política π que actúa de mapa de estados a una distribución de probabilidad sobre las acciones $\pi : \mathcal{S} \rightarrow P(A)$ en un entorno E, que puede ser estocástico.

El retorno de un estado se define como la suma de todas las recompensas futuras descontadas:

$$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i) \tag{3.2}$$

Donde $\gamma \in [0, 1]$ es un factor de descuento[27]. Definiendo la función acción-valor como el retorno esperado al tomar una acción a_t en el estado s_t para después seguir la política π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi}[R_t \mid s_t, a_t] \tag{3.3}$$

Se utiliza la ecuación de Bellman con una política determinista μ [27]:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_t + 1))] \tag{3.4}$$

Utilizando la ecuación 3.4 en la ecuación 3.6 para actualizar la función Q definida en la ecuación 3.3 aproximada mediante una función con parámetros

θ^Q minimizando la pérdida $L(\theta^Q)$:

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t \mid \theta^Q) - y_t)^2] \tag{3.5}$$

Donde β es una política estocástica cualquiera, al ser un algoritmo *off-policy*, ρ^π es la distribución descontada de visitación para una política π, y y_t se define como:

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1} \mid \theta^Q) \tag{3.6}$$

Mediante estas actualizaciones se encuentra la función $Q(s, a)$ del crítico. Las actualizaciones del actor se basan en seguir el gradiente del retorno esperado de la distribución inicial J con respecto a los parámetros del aproximador de funciones del actor (la red neuronal del actor)[27]. Este es el gradiente del rendimiento de la política[42].

$$\begin{aligned} \nabla_{\theta^\mu} J &\approx \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a \mid \theta^Q) \mid_{s=s_t, a=\mu(s_t \mid \theta^\mu)}] \\ &= \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu) \mid_{s=s_t}] \end{aligned} \tag{3.7}$$

Uno de los problemas al usar redes neuronales en el aprendizaje por refuerzo es que es habitualmente asumido para muchos optimizadores que las muestras son independientes y siguen una distribución idéntica, hecho que no se cumple en un proceso de interacción con el entorno, donde los estados siguientes son consecuencia directa del estado actual y de la acción que se tome. El algoritmo DQN resolvió este problema con el uso de *experience replay*, explicado en más detalle en la subsección 3.2.2, y en el algoritmo DDPG se volvió a implementar con tal de solventar el mismo problema.

Otro de las idea aportadas por DQN es el uso de *target networks*, o redes objetivo, explicadas en detalle en la subsección 3.2.1, que aportan estabilidad a la actualización del crítico. En DDPG, se modificaron para adaptarlas a un algoritmo *actor-critic*. La versión modificada utiliza también una actualización suave como se puede ver en la ecuación 3.8 con un parámetro τ de actualización $\tau \ll 1$.

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{3.8}$$

Por último, se implementó *batch normalization*, explicado en detalle en la subsección 3.2.3, para escalar las entradas de la red, que pueden tener unidades y rangos diferentes al cambiar de problema. Esta técnica dota al algoritmo de un comportamiento robusto, siendo capaz de funcionar bien en distintos problemas sin tener que alterar los hiperparámetros.

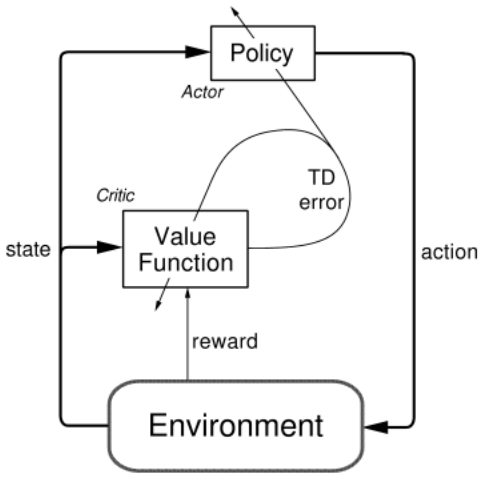

Figura 3.2: Diagrama de la estructura de un algoritmo *actor-critic*[45].

En cuanto a la exploración, se utilizó un proceso Ornstein-Uhlenbeck[46], como proceso generador de ruido temporalmente correlacionado. Este ruido se añadió a las salida del actor tal que la política pasó a ser la siguiente:

$$\mu'(s_t) = \mu(s_t \mid \theta_t^{\mu}) + \mathcal{N} \tag{3.9}$$

El algoritmo original se muestra en el algoritmo 1. De forma más conceptual, su funcionamiento es el siguiente:

1. Primero, se inicializan las dos redes neuronales que actuarán de actor μ y de crítico Q, así como las dos redes objetivo μ' y θ' y el *buffer* de interacciones R. Líneas 1 a 3.

2. A continuación comienza un proceso iterativo en se ejecutan M episodios. Líneas 4 a 19.

3. Para cada episodio, se inicializa el generador de ruido de exploración $\mathcal{N}$ y se extrae el estado inicial s_1. Líneas 5 y 6.

4. A continuación se itera para cada paso del episodio. Líneas 7 a 18.

5. Se calcula una acción a_t a partir del actor con el estado actual s_t como entrada y se le suma el ruido exploratorio antes de ejecutar dicha acción y recibir el estado del siguiente paso s_{t+1} y la recompensa asociada a la combinación (s_t, a_t). El tupla $(s_t, a_t, r_t, s_t{+}1)$ constituye una interacción y se almacena en el *buffer* R. Líneas 8 a 10.

6. A continuación comienza el proceso de aprendizaje, se muestrea un *minibatch* aleatorio de N interacciones de R. Línea 11.

7. Con el *minibatch* muestreado, se calculan los valores de Q, y_i, del siguiente estado al muestreado utilizando las redes objetivo. El crítico objetivo devuelve su predicción del valor de Q para el estado s_{t+1} utilizando la salida del actor objetivo frente a s_{t+1} como entrada para sus acciones; en otras palabras, se calcula que acción habría tomado el actor objetivo en el estado s_{t+1} y se evalúa la tupla estado, acción mediante el crítico objetivo. Puesto que se quiere utilizar esta predicción del valor de la tupla acción-estado para compararlo con el valor de la tupla correspondiente al estado actual s_t del *minibatch* y se encuentran separados por un instante en el tiempo, se descuenta el valor de la Q futura y se le suma el valor de la recompensa instantánea r_t que recibe el agente al ejecutar la acción a_t en el estado s_t. Línea 12.

8. Una vez calculado el objetivo y_i, se actualiza el crítico minimizando el error cuadrático medio entre el valor que predice el crítico para (s_t, a_t) y y_i. Línea 13.

9. Se pasa ahora a la actualización del actor. Se mueven los pesos de la red neuronal del actor según el gradiente ∇_{θ^μ} que se calcula con la media del producto del gradiente del crítico con respecto a las acciones y el gradiente del actor con respecto sus pesos. Línea 14.

10. Por último, se actualizan las dos redes objetivo acercándolas a los pesos del crítico y del actor actualizados mediante una suma ponderada de estos últimos y las redes objetivo de la iteración anterior. Líneas 15 a 17.

11. Se itera hasta acabar un episodio. Línea 18.

12. Se itera hasta completar todos los episodios. Línea 19.

En la figura 3.2 se representa un diagrama con la estructura de un algoritmo *actor-critic* cualquiera para ayudar en la explicación.

Los resultados del algoritmo son buenos[2]. Ha sido capaz de aprender buenas políticas para toda una serie de problemas diferentes sin tener que cambiar sus hiperparámetros, en algunas ocasiones superando el rendimiento de planificadores que tenían acceso a modelos dinámicos de los problemas[27].

[2]Es posible ver videos de los comportamientos aprendidos en la siguiente dirección: https://goo.gl/J4PIAz[27]

Los problemas se ejecutaron de dos modos: uno con entradas de dimensionalidad reducida, donde el algoritmo veía los pares de las articulaciones u otros parámetros importantes y uno en el que las entradas del algoritmo eran directamente imágenes del entorno, como si tuviera una cámara y pudiera ver a través de ella el estado actual. Sorprendentemente, para algunas tareas, aprender de píxeles era tan rápido como aprender de espacios de estados más reducidos[27].

Algoritmo 1: Algoritmo DDPG[27]

Inicializar aleatoriamente la red del crítico $Q(s, a \mid \theta^{\mu})$ y el actor $\mu(s \mid \theta^{\mu})$ con pesos θ^{Q} y θ^{μ}.
Inicializar las redes objetivo Q' y μ' con pesos $\theta^{Q'} \leftarrow \theta^{Q}$,$\theta^{\mu'} \leftarrow \theta^{\mu}$
Inicializar el *buffer* de interacciones R
for *episodio=1, M* **do**
    Inicializar proceso aleatorio N para la exploración del espacio de acciones
    Recibir estado observacional inicial s_1
    for *t=1, T* **do**
        Seleccionar acción $a_t = \mu(s_t \mid \theta^{\mu}) + N_t$ de acorde con la política actual y el ruido exploratorio
        Ejecutar acción a_t y observar la recompensa r_t y el nuevo estado s_{t+1}
        Almacenar transición (s_t, a_t, r_t, s_{i+t}) en R
        Muestrear un *minibatch* aleatorio de N transiciones (s_i, a_i, r_i, s_{i+1}) de R
        Sea $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'})$
        Actualizar el crítico minimizando la pérdida:
        $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i \mid \theta^{Q}))^2$
        Actualizar la política del actor utilizando el gradiente de la política muestreado:
        $\nabla_{\theta^{\mu}} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a \mid \theta^{Q}) \mid_{s=s_i, a=\mu(s_i)} \nabla_{\theta^{\mu}} \mu(s \mid \theta^{\mu}) \mid_{s_i}$
        Actualizar las redes objetivo:
        $\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'}$
    end
end

3.2.1. Target networks

La idea detrás de introducir un retraso temporal a la actualización de los generadores de los objetivos a optimizar es la de aumentar la estabilidad del aprendizaje. Sin redes objetivo, una actualización que incrementaba el valor de $Q(s_t, a_t)$ también solía incrementar el valor de $Q(s_{t+1}, a)$, lo que movía toda la actualización, creando un sesgo que podía acabar en oscilaciones o incluso divergencia para la política. Las redes objetivo alivian este problema en cierta medida[32].

3.2.2. Experience replay

El método de *experience replay* consiste en mantener un *buffer* de transiciones pasadas disponible para actualizar el algoritmo con ellas. Esta técnica no solamente acelera el proceso de aprendizaje y aumenta la eficiencia de la exploración[28] [33] [9], sino que se ha demostrado son vitales para la estabilidad del aprendizaje[9].

El actualizar el agente a partir de interacciones pasadas guardadas permite que una sola interacción pueda evaluarse múltiples veces, cada vez en un momento diferente, y por lo tanto con una política diferente, lo que aumenta la eficiencia de la exploración inicial[33] [9], además de permitir el uso de *minibatches*, lo que aumenta la eficiencia computacional al entrenar el algoritmo en una unidad gráfica[9].

Al sondear interacciones de forma aleatoria para formar cada *minibatch*, se rompen las correlaciones temporales de las interacciones consecutivas, reduciendo la varianza de las actualizaciones ya que los algoritmos de optimización esperan muestras independientes[33] [9].

También ayuda a evitar mínimos locales evitando que el agente se actualice únicamente con la última política. Al disponer de transiciones variadas, es menos probable que escoja un comportamiento que lleve al agente a concentrarse únicamente en una pequeña sección del espacio, puesto que también reevaluará experiencias pasadas fuera de ese espacio con nuevas perspectivas[33] [9].

Otra manera en la que aumenta la estabilidad del aprendizaje es la que proviene del hecho de que la distribución del comportamiento del agente se ve promediada sobre muchos estados pasados[33].

Sin embargo, existen problemas con esta técnica. Notablemente, el *buffer* es limitado, por lo que las experiencias iniciales se acabarán olvidando eventualmente, siendo reemplazadas por experiencias nuevas, reduciendo la variedad en una base de datos en la que ya dominan las interacciones de la mejor política que ha encontrado el agente, ya que este problema suele

aparecer más tarde en el aprendizaje, cuando el agente ya ha dado con un comportamiento prometedor. Este comportamiento acaba resultando en rendimientos reducidos y sobreajuste. Se crea el problema adicional de decidir que interacciones salvaguardar con tal de disponer siempre de *replay buffer* variado y representativo[9].

3.2.3. Batch normalization

El problema del desplazamiento covariable, *covariate shift*, consiste en el cambio de la distribución de las entradas a las capas de una red neuronal. Supone un problema puesto que al cambiar la distribución, la red debe aprender un nuevo patrón que se ajuste a esta nueva distribución. El problema se acentúa en redes profundas, ya que cambios en las primeras capas se propagan hasta las últimas, moviendo la distribución de las salidas de cada capa[17].

Este problema se ha solucionado convencionalmente con una inicialización cuidadosa de los parámetros iniciales y ratios de aprendizaje más bajos, ralentizando el aprendizaje[17].

Batch normalization soluciona este problema al normalizar las entradas a cada capa de la red neuronal, fijando una media de 0 y una varianza de 1 para cada *minibatch*. Así, aunque las entradas a cada capa cambien, la distribución no variará tan drásticamente y se reduce el riesgo de divergencia. Se consigue desacoplar en cierta medida las capas entre si, reduciendo la dependencia de los gradientes con la escala de los parámetros o de sus valores iniciales[17].

Aplicando *batch normalization* se han conseguido precisiones idénticas a métodos convencionales con hasta 14 veces menos iteraciones de aprendizaje, además de mostrar una menor dependencia de las redes con respecto a sus parámetros iniciales[17].

3.2.4. Implementación original

Se exponen los detalles de la implementación original del algoritmo DDPG[27] en el cuadro 3.1.

Elemento	**Descripción**
Arquitectura del crítico	2 capas internas de 400 y 300 unidades respectivamente. Todas las capas tienen activación ReLU. Las acciones no se incluyen hasta la segunda capa. Se utilizó *batch normalization* en las entradas de todas las capas previas a la entrada del vector de acciones.
Arquitectura del actor	2 capas internas de 400 y 300 unidades respectivamente. Todas las capas tienen activación ReLU excepto la salida, que usa una tanh. Se utilizó *batch normalization* en las entradas de todas las capas.
Regularización L_2 *weight decay* para el crítico	10^{-2}
Optimizador para el crítico	ADAM
Optimizador para el actor	ADAM
Ratio de aprendizaje para el crítico	10^{-3}
Ratio de aprendizaje para el actor	10^{-4}
Factor de descuento γ	0.99
Factor τ para las actualizaciones de las *target networks*	0,001
Tamaño del *replay buffer*	10^6
Proceso Ornstein-Uhlenbeck	$\theta = 0{,}15$, $\sigma = 0{,}2$

Cuadro 3.1: Resumen de los detalles de implementación del algoritmo DDPG original para el caso de baja dimensionalidad[27].

Capítulo 4

Detalles experimentales

En este capítulo se discutirán los detalles de la implementación de la parte práctica del proyecto: el tipo de planificación de trayectorias utilizado, las modificaciones realizadas sobre el algoritmo original y el planteamiento de los experimentos así como las herramientas utilizadas para lograr los resultados propuestos.

4.1. Generación de trayectorias

La salida del agente consistirá, en última instancia, en una trayectoria para el punto central de la herramienta del robot. La forma que está adoptará es importante tanto para la estructura interna del agente (una codificación adecuada puede suponer una mayor facilidad para encontrar buenas soluciones) como para los requerimientos cinemáticos y dinámicos del robot.

Una trayectoria se define como la unión de un camino geométrico $p = p(u)$, una curva parametrizada p dependiente de un parámetro u y una ley de movimiento $u = u(t)$ que relaciona el parámetro u con el tiempo t[7].

La curva geométrica viene dada por la tarea a realizar, mientras que la ley de movimiento viene condicionada por las limitaciones físicas del robot[7].

4.1.1. Elección de un sistema de generación de trayectorias

Aun siendo común en la literatura de aprendizaje por refuerzo el control de los pares de las articulaciones o direcciones de movimiento como salidas de los agentes[27],[6],[9], se propuso para este proyecto el utilizar métodos de generación de trayectorias comunes en el campo de la planificación de movimientos convencional, como son los métodos de interpolación entre puntos,

aplicados normalmente en la fase de post-proceso, después de haber establecido una trayectoria básica.

Estos métodos proporcionan una serie de ventajas importantes a la hora de aplicar el algoritmo propuesto, tales como:

- La tarea se simplifica a encontrar una trayectoria fija, fuera de línea, por episodio, que después se ejecuta y evalúa, reduciendo el problema a una iteración por episodio, el agente se actualiza mediante la recompensa que ha surgido de la aplicación de la trayectoria directamente, sin depender de la predicción del crítico para auto actualizarse. Se intuye que esta propiedad debería acortar el tiempo de iteración entre episodios al no tener que calcular comandos intermedios y el tiempo de convergencia, como mínimo, del crítico.

- Si la trayectoria se define entre el punto inicial y el punto objetivo como puntos fijos de salida y terminación, alterando únicamente el camino que lleva de uno al otro, se asegura que el agente siempre llegue a la posición objetivo, lo que reduce en gran medida la exploración necesaria: el robot empezará cumpliendo su objetivo y aprenderá a mejorar la forma en el que lo consigue, ahorrándose el paso previo de aprender como cumplirlo.

Sin embargo, también conlleva algunos inconvenientes en comparación con la aplicación de vectores de velocidad sobre la configuración de la herramienta terminal del robot, como:

- Serán necesarias más salidas del agente para parametrizar adecuadamente la trayectoria, lo que puede afectar negativamente al rendimiento.

- El algoritmo dependerá de la parametrización de la curva y es posible que las soluciones que pueda ofrecer se vean fuertemente limitadas por dicha parametrización.

Para dibujar la trayectoria, se buscó una forma sencilla y dependiente de pocos parámetros (ya que estos dependen de las salidas del agente) pero lo más reconfigurable posible para dotar al agente de la flexibilidad necesaria para resolver los problemas. Se optó por el uso de puntos de paso, por los que debe pasar el robot, como parámetros de salida, asegurando de esta manera un camino básico. Sin embargo, es necesario también dotar a la trayectoria de una interpolación adecuada entre estos.

La elección de un sistema de interpolación es dependiente de cada problema[44], sin embargo, para la interpolación de varios puntos concatenados, es más habitual la interpolación mediante polinomios[30] que mediante la unión de

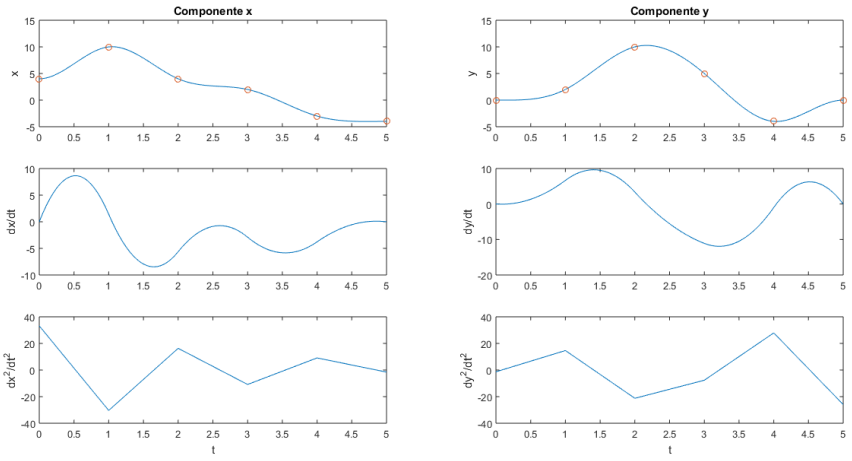

Figura 4.1: Trayectorias creadas mediante *clamped splines* cúbicas junto a sus primeras y segundas derivadas. Los puntos resaltados en rojo en las trayectorias son los puntos de control.

segmentos primitivos (parábolas, rectas, exponenciales, etc.), que suelen presentar discontinuidad paramétrica aun poseyendo continuidad geométrica[7].

En lo que respecta la interpolación mediante polinomios, son de especial interés las *splines*, o funciones polinómicas definidas a trozos, donde un polinomio se encarga de conectar un punto de control con el siguiente; la continuidad de la trayectoria depende entonces del grado de los polinomios utilizados.

Las *splines* cúbicas pasan por los puntos de control, se encuentran inequívocamente definidas por dichos puntos de control y ofrecen continuidad C^2, es decir, continuidad hasta su segunda derivada, lo que permite utilizar una parametrización simple linear de la ley de movimiento para generar la trayectoria continua en aceleración para el robot[1], simplificando el problema. En la figura 4.1 se encuentran representadas dos *splines* y sus derivadas de primer y segundo orden. En el caso de que el robot no pudiera seguir alguna trayectoria debido a valores de velocidad o aceleración no asumibles por alguno de sus actuadores, es posible reparametrizar la curva tal que estos valores vuelvan a entrar dentro de los parámetros aceptables[7].

Es por estas características que se ha optado por el uso de *splines* cúbicas para la generación de las trayectorias del robot. La trayectoria de cada eje del plano horizontal de trabajo se representa mediante el uso de *splines*, que

[1]Una trayectoria continua en aceleración evita problemas de vibraciones inducidas en el manipulador y control ineficaz[44], por lo que son preferidas ahí donde sean posibles.

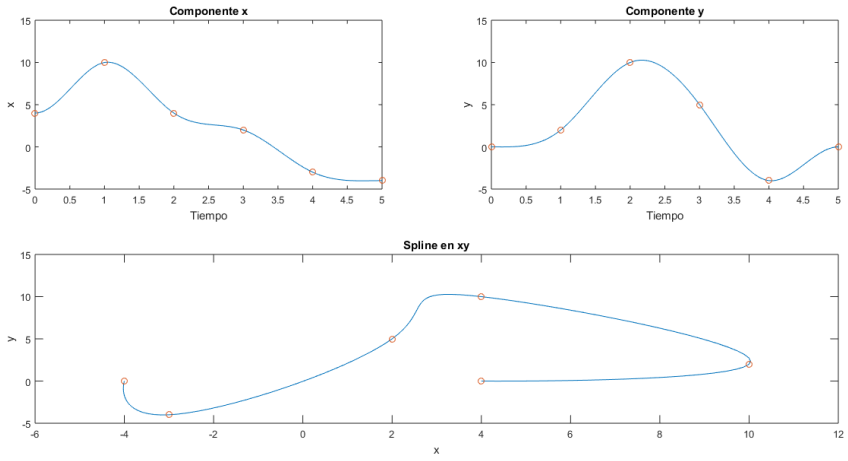

Figura 4.2: Trayectorias creadas mediante *clamped splines* cúbicas para dos dimensiones x e y y la trayectoria combinada en el plano xy. Los puntos resaltados en rojo en las trayectorias son los puntos de control

se combinan para ofrecer una trayectoria en el plano, tal y como se puede observar en la figura 4.2.

A continuación se definen matemáticamente:

Una spline cúbica es una función formada a trozos por n polinomios de la forma:

$$y_i(u) = a_i + b_i u + c_i u^2 + d_i u^3, u \in [0, 1] \tag{4.1}$$

Que unen cada uno de sus $n+1$ puntos de control. Ya que se deben fijar 4 coeficientes a_i,b_i,c_i y d_i por polinomio, la trayectoria presenta $4n$ incógnitas. Para resolver el sistema de ecuaciones, se aplican las siguientes restricciones:

- Los polinomios deben pasar por los puntos de control D_i, ver las ecuaciones 4.2 y 4.3, lo que supone n restricciones.
- Continuidad geométrica (ecuación 4.4), de la primera derivada (ecuación 4.5) y de la segunda derivada (ecuación 4.6). Todas ellas suman un total de $3(n-1)$ restricciones.

En total, aplicando las condiciones de continuidad, se obtienen $4n-2$ ecuaciones, por lo que son necesarias 2 condiciones de contorno adicionales para acabar de definir el problema.

Existen diferentes condiciones de contorno aplicables, y las *splines* cúbicas adoptan diferentes nombres propios según cuales utilizan. Así pues, por ejemplo, una *spline* cúbica que fija un valor nulo de su segunda derivada en

los extremos recoge el nombre de *spline natural*. En el caso de este proyecto, se ha optado por forzar las velocidades finales e iniciales a 0, lo que se conoce como una *clamped spline*, o *complete spline*, ya que de esta manera se describe como el robot comienza en reposo y alcanza una pose de *pre-grasp* decelerando hasta quedarse otra vez en reposo (ecuación 4.7).

$$\begin{cases} y_i(0) = D_i, i = 1, ..., n-1 & (4.2) \\ y_n(1) = D_n & (4.3) \end{cases}$$

$$y_i(1) = y_{i+1}(0), i = 1, \ldots, n-1 \quad (4.4)$$

$$y_i^{(1)}(1) = y_{i+1}^{(1)}(0), i = 1, \ldots, n-1 \quad (4.5)$$

$$y_i^{(2)}(1) = y_{i+1}^{(2)}(0), i = 1, \ldots, n-1 \quad (4.6)$$

$$y_1^{(1)} = y_n^{(1)} = 0 \quad (4.7)$$

El parámetro u está relacionado con el tiempo según una ley de movimiento. Si $u \in [0, 1]$ para cada segmento de la trayectoria, una ley de movimiento proporcional tomaría la siguiente forma:

$$u(t) = \lambda t \quad (4.8)$$

donde λ es un factor de proporcionalidad. En el caso de que una velocidad o aceleración no sean válidas, se puede reparametrizar la curva $p(u(t))$ mediante *constant scaling*[7]:

$$p(u(t))^{(1)} = \frac{dp}{du}\lambda \quad (4.9)$$

$$p(u(t))^{(2)} = \frac{d^2p}{du^2}\lambda^2 \quad (4.10)$$

$$p(u(t))^{(3)} = \frac{d^3p}{du^3}\lambda^3 \quad (4.11)$$

$$\vdots \quad (4.12)$$

Así pues, se puede calcular λ tal que no se supera velocidad, aceleración, sobreaceleración... máximas según:

$$\lambda = \text{mín} \left\{ \frac{v_{\max}}{|p(u(t))^{(1)}|_{\max}}, \sqrt{\frac{a_{\max}}{|p(u(t))^{(2)}|_{\max}}}, \sqrt[3]{\frac{j_{\max}}{|p(u(t))^{(3)}|_{\max}}}, \ldots \right\} \quad (4.13)$$

Elemento	Descripción
Dimensiones del área de trabajo	$19 \times 38\text{cm}^3$.
Dimensiones de los obstáculos	Cilindros de $11,5$cm de altura, $6,6$cm de diámetro.
Dimensiones del TCP	Esfera de 7cm de diámetro[4].

Cuadro 4.1: Resumen de los detalles del espacio experimental

4.2. Detalles de implementación del algoritmo

El algoritmo se implementó de forma gradual, probando antes su eficacia en problemas sencillos de *AI Gym*[37], `pendulum-v0` y `MountainCarContinuous-v0` antes de afrontar el problema más complicado de aplicarlo a una simulación del experimento con el robot. La simplicidad de estos problemas iniciales fue muy útil, ya que gracias a la rapidez con la que se solucionaban, permitieron que se experimentara con distintos parámetros y técnicas, para después implementar las más interesantes en el problema de simulación, de duración mucho más elevada[2].

Todos los programas de aprendizaje automático fueron implementados en `python` mediante `Tensor-Flow`[3].

4.2.1. Problema objetivo

El problema objetivo que se intenta simular es el de realizar un movimiento del TCP de un robot *YuMi* de *ABB*[4], desde una posición inicial a una final, en un entorno poblado donde las colisiones con los obstáculos no solamente son válidas, sino que muchas veces necesarias. Se penaliza tanto la longitud de la ruta escogida (cuanto más larga es, peor es considerada) como el desplazamiento de los obstáculos total (cuanto más desplace los obstáculos, peor es valorada).

Las características del espacio experimental se detallan en el cuadro 4.1.

El movimiento del robot actuando bajo DDPG se verá gobernado por una *spline* de 7 puntos de paso en total, estando el primero y el último fijados en

[2]El problema de simulación tardaba entre 20 y 25 veces más tiempo en converger en una buena política.

[3]El área de trabajo se calculó encontrando los límites del área a la cual *YuMi* podía llegar de forma sencilla con el TCP mirando hacia el suelo.

[4]Debido a limitaciones del planificador alternativo con el que se realizaron las comparaciones, se tuvo que optar por una esfera. Esto se consiguió mediante el uso de una media esfera con un soporte interior que el robot podía coger.

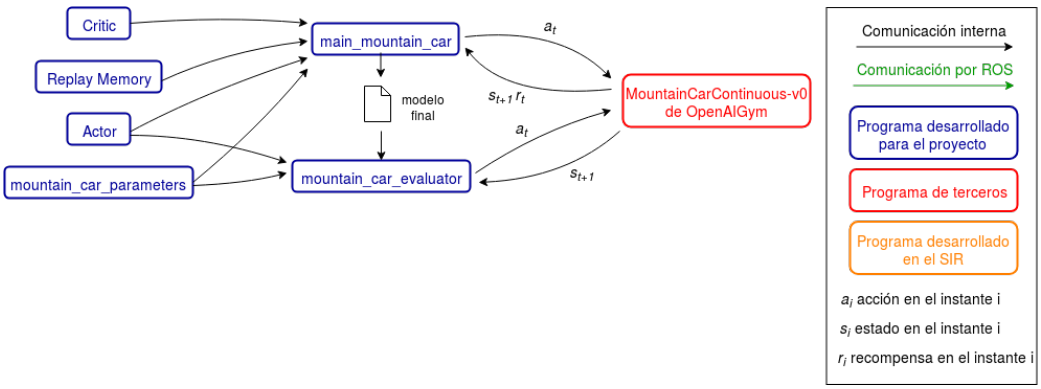

Figura 4.3: Diagrama del funcionamiento del algoritmo DDPG ejecutando el entorno `MountainCarCountinuous-v0` y su evaluador.

el punto inicial y el final, respectivamente. De esta manera, el robot moverá el TCP hacia el objetivo a una velocidad constante v_x mientras que cambia su velocidad y posición en el eje perpendicular y con tal de pasar por los puntos de paso y obedecer la trayectoria calculada con anterioridad mediante *splines*.

En el caso del planificador alternativo, se le suministró el mismo tipo de problema, pero sus soluciones no se ven restringidas a un camino creado por splines, sino que es libre de moverse por toda el área de trabajo.

Debido a problemas técnicos, al final no ha sido posible implementar los resultados finales en el robot *YuMi*, por lo que todas las comparaciones entre el planificador con DDPG y el planificador alternativo se realizan en simulación. Sin embargo, se incluye todo el *software* desarrollado para el control del robot en el trabajo.

4.2.2. Problemas sencillos

A continuación se explican detalles sobre los entornos proporcionados por *AI Gym*.

4.2.2.1. Pendulum

El objetivo de este entorno consiste en balancear un péndulo sólido aplicando un par a su articulación tal que este acabe quedando en una posición vertical, con el peso en el punto más alto. En la figura 4.5 se muestra una representación gráfica del problema.

Se trata de un problema sin fin de episodio marcado, por lo que no existe una recompensa terminal y y_i con el cual se compara la salida del crítico objetivo siempre es:

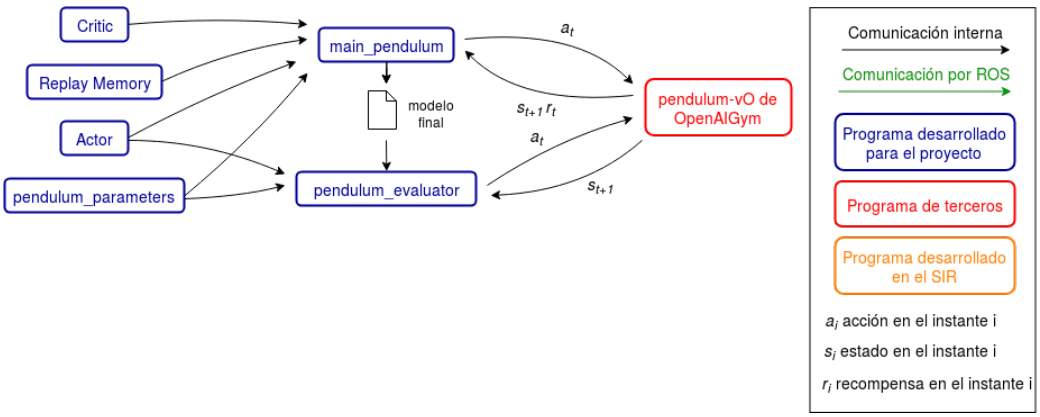

Figura 4.4: Diagrama del funcionamiento del algoritmo DDPG ejecutando el entorno `pendulum-v0` y su evaluador.

Figura 4.5: Imagen del entorno `pendulum-v0` de *Open AI Gym*[37]. El tamaño de la flecha representa la intensidad del par aplicado.

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1} \mid \theta^{\mu'}) \mid \theta^{Q'}) \tag{4.14}$$

La figura 4.4 muestra un diagrama de su funcionamiento y de su evaluador.

4.2.2.2. MountainCarContinuous

El objetivo de este entorno consiste en balancear un coche para que este consiga escalar una montaña. Solo recibe recompensa positiva si alcanza el objetivo, a la vez que se le penaliza el esfuerzo, por lo que es necesaria una exploración más agresiva para que pueda ver la recompensa al menos una vez y comenzar a intentar optimizar su política para llegar con más facilidad al objetivo.

Este entorno si posee de estado terminal, por lo que el objetivo y_i con el

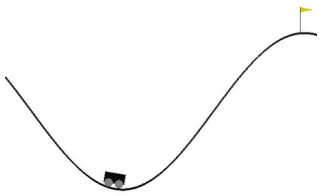

Figura 4.6: Imagen del entorno `MountaincarContinuous-v0` de *Open AI Gym*[37].

cual se compara la salida del crítico objetivo depende de si la interacción se produce en en estado terminal o no, adoptando la forma vista en la ecuación 4.14 si no es un estado terminal y la forma de la ecuación 4.15 en caso contrario.

$$y_i = r_i \tag{4.15}$$

La figura 4.3 muestra un diagrama de su funcionamiento y de su evaluador.

4.2.3. Simulación en GAZEBO

El entorno simulado del problema objetivo se realizó en GAZEBO[20]. El agente recibe como entradas un vector de posiciones en el plano x,y de cada uno de los obstáculos y responde con un vector de acciones de longitud $n-2$, donde n es el número de puntos de control de las *splines* utilizadas, ya que el primer y el último punto de control están fijados. Esta salida pasa por un gestor del entorno que la convierte en una trayectoria junto con otros parámetros adicionales que se han fijado antes de la ejecución del programa, tales como la duración del movimiento.

`Gazebo` recibe la trayectoria y mueve el TCP simulado según las instrucciones dadas. Al final, devuelve un estado final, con las posiciones de todos los obstáculos, que el gestor convierte en una señal de recompensa según una función arbitraria. Se genera un nuevo estado y el ciclo vuelve a comenzar. La función de recompensa elegida se encuentra representada en la ecuación 4.16, donde o es la suma de todos los desplazamientos de los obstáculos y d es la distancia total del recorrido. Se llego a esta forma modificándola hasta que el agente presentaba un comportamiento satisfactorio.

En la figura 4.9 se ha representado un diagrama que visualiza el funcionamiento explicado. En el cuadro 4.2 se muestran los hyperparámetros del agente.

$$r_t = -o - 0,5d \tag{4.16}$$

Debido a que se trata de un generador de trayectorias en lazo abierto, es decir, no es capaz de reaccionar a estados intermedios, únicamente actúa ante la situación inicial y entonces ejecuta la trayectoria, sin cambiarla pase lo que pase por el camino, no existen estados intermedios. Dicho de otra manera, todos los estados siguientes al estado inicial de cada episodio son estados terminales, por lo que la actualización del crítico siempre se realiza mediante la ecuación 4.15, con lo que las redes objetivo nunca llegan a jugar ningún papel y se puede prescindir de ellas. El algoritmo modificado es el algoritmo 2.

Algoritmo 2: Algoritmo DDPG simplificado

Inicializar aleatoriamente la red del crítico $Q(s, a \mid \theta^\mu)$ y el actor $\mu(s \mid \theta^\mu)$ con pesos θ^Q y θ^μ.
Inicializar el *buffer* de interacciones R
for *episodio=1, M* **do**
    Inicializar proceso aleatorio N para la exploración del espacio de acciones
    Recibir estado observacional inicial s_1
    for *t=1, T* **do**
        Seleccionar acción $a_t = \mu(s_t \mid \theta^\mu) + N_t$ de acorde con la política actual y el ruido exploratorio
        Ejecutar acción a_t y observar la recompensa r_t y el nuevo estado s_{t+1}
        Almacenar transición (s_t, a_t, r_t, s_{i+t}) en R
        Muestrear un *minibatch* aleatorio de N transiciones (s_i, a_i, r_i, s_{i+1}) de R
        Sea $y_i = r_i$
        Actualizar el crítico minimizando la pérdida:
        $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i \mid \theta^Q))^2$
        Actualizar la política del actor utilizando el gradiente de la política muestreado:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a \mid \theta^Q) \mid_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s \mid \theta^\mu) \mid_{s_i}$
    end
end

Elemento	**Descripción**
Arquitectura del crítico	2 capas internas de 400 unidades respectivamente. Todas las capas tienen activación ReLU.
Arquitectura del actor	2 capas internas de 400 unidades respectivamente. Todas las capas tienen activación ReLU excepto la salida, que usa una tanh.
Regularización L_2 *weight decay* para el crítico	10^{-2}
Optimizador para el crítico	ADAM
Optimizador para el actor	ADAM
Ratio de aprendizaje para el crítico	10^{-3}
Ratio de aprendizaje para el actor	10^{-4}
Factor de descuento γ	0.99
Factor τ para las actualizaciones de las *target networks*	0,01
Tamaño del *replay buffer*	10^5
Proceso Ornstein-Uhlenbeck	$\theta = 0{,}15$, $\sigma = 0{,}2$

Cuadro 4.2: Resumen de los detalles de implementación del algoritmo DDPG original para el caso de baja dimensionalidad[27].

4.2.4. Experimentos

La lista de experimentos por entorno se muestra en el cuadro 4.3. A continuación se explica el racional tras estos experimentos:

Normal: Estos experimentos se llevaron a cabo como *grupo de control*, para comparar el impacto del resto de modificaciones.

Acciones en la segunda capa: En la implementación original se describe una arquitectura en la cual el vector de entrada de acciones no entra en el crítico hasta la segunda capa sin justificación técnica. Estas pruebas tienen como objetivo averiguar si aportan alguna ventaja sobre concatenar las acciones a los estados en la entrada del crítico.

***Batch normalization*:** Esta prueba se realiza con la intención de medir el impacto del uso de *batch normalization*.

Sin *warmup*: El *warmup*, o precalentamiento, consiste en hacer correr el entorno sin entrenar al agente, dejando que este explore libremente para que cuando empiece a aprender pueda disponer de un *buffer* de interacciones más amplio y variado. Se intenta medir el impacto que tiene desactivar esta función.

Como el artículo original: Consiste en ejecutar el algoritmo DDPG tal y como se describe en el artículo original, es una combinación de los casos 'Normal', 'Acciones en la segunda capa' y '*Batch normalization*'.

Entradas reordenadas: Tal y como se diseño el entorno simulado de la tarea robótica, las entradas al sistema eran las posiciones de los obstáculos, en orden. Se intenta ver si al reordenar las entradas tal que los obstáculos ahora entran al agente ordenados según su proximidad al punto de partida en la dimensión de avance se mejora su rendimiento. La lógica tras este razonamiento está en que al reordenar las entradas de esta manera, para el agente, la distribución de los obstáculos variará menos, concentrando sus distribuciones por el espacio de estados, al estar los obstáculos cercanos siempre cercanos y los lejanos siempre lejanos.

Como el artículo original y con entradas reordenadas: Se trata de una combinación de los casos 'Como el artículo original' y 'Entradas reordenadas'.

Experimento	**Entornos**	**Descripción**
Normal	Pendulum Mountain Car Simulación	DDPG básico, sin batch normalization.
Acciones en la segunda capa	Pendulum Mountain Car	Como en el algoritmo original, las acciones se añaden en la segunda capa del crítico.
Batch normalization	Pendulum Mountain Car	DDPG con batch normalization.
Sin *warmup*	Pendulum Mountain Car	Sin episodios de calentamiento.
Como el artículo original	Pendulum Mountain Car	DDPG tal y como se describe en el artículo original.
Entradas reordenadas	Simulación	DDPG básico, sin batch normalization pero con las entradas reordenadas.
Como el artículo original y con entradas reordenadas	Simulación	DDPG tal y como se describe en el artículo original y con las entradas reordenadas.

Cuadro 4.3: Cuadro con los experimentos realizados por entorno.

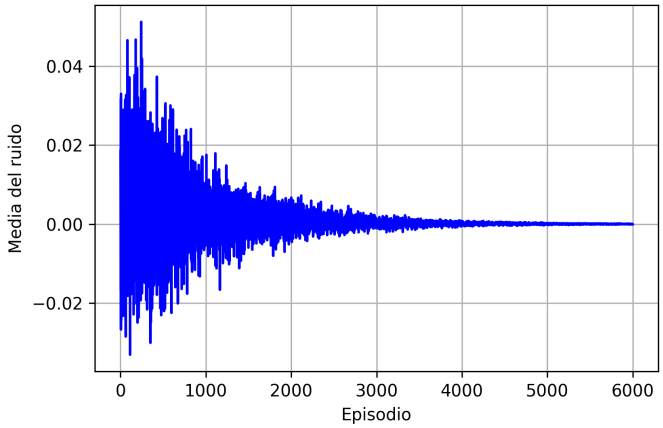

Figura 4.7: Media del ruido aportado en cada episodio por el proceso *Ornstein–Uhlenbeck*.

4.2.5. Proceso Ornstein–Uhlenbeck

La exploración se llevó a cabo como en el artículo original[27], mediante un proceso generador de ruido Ornstein–Uhlenbeck, tanto en los experimentos con problemas sencillos de *AI-Gym*, en los cuales se generaba toda la distribución para cada episodio y después se muestreaba cada elemento, por orden, a cada paso del episodio, como en el caso del planificador de movimientos, en el cual se ejecutaba el proceso para que diera tantas muestras como dimensiones del espacio de acciones.

En las figuras 4.7 y 4.8 se muestra como el ruido se fue atenuando a medida que se avanzaba en el aprendizaje, con tal de permitir que la política convergiera en un buen comportamiento y acabara de ajustarse cuando se consideraba que había realizado suficiente exploración. Algunos problemas, como el `MountainCarContinuous-v0` requerían mucha más exploración que otros y el atenuamiento se codificó de forma diferente.

4.2.6. Planificador comparativo KPIECE

Los resultados del algoritmo se han comparado con los del planificador de última generación Kinodynamic Motion Planning by Interior-Exterior Cell Exploration (KPIECE), un planificador de árbol basado en el muestreo. Presume de de tiempos de cálculo y memoria reducidos en comparación con otros planificadores actuales y es capaz de trabajar con simulación física, razones por las que se escogió para esta comparativa[49].

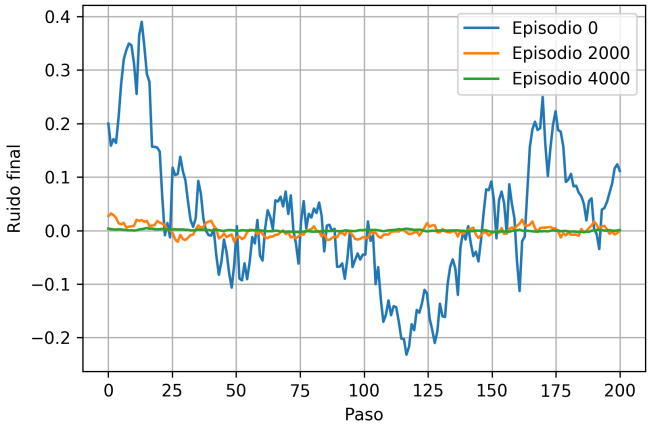

Figura 4.8: Perfiles del ruido aportado por el proceso *Ornstein–Uhlenbeck* a cada paso para diferentes episodios.

Su principio de funcionamiento se basa en decidir desde que partes del árbol exploratorio se debería explorar más. Consigue esto mediante proyecciones del espacio de estados a otros de menor dimensionalidad y realizando un seguimiento de los límites del espacio explorado mediante una discretización a varios niveles con tal de llevar una evaluación continua del progreso de la exploración[49].

El algoritmo KPIECE se ejecuta dentro del programa `The Kautham Project`, un entorno de desarrollo de algoritmos de planificación de movimientos desarrollado por el instituto de organización y control de sistemas industriales de la ETSEIB[40].

Para poder realizar una comparación justa, tanto el algoritmo DDPG como KPIECE deben ejecutarse en el mismo entorno simulado. Puesto que `The Kautham Project` resuelve los problemas en su propio entorno, se optó por extraer las trayectorias que KPIECE generaba dentro de `The Kautham Project` para después ejecutarlas en `GAZEBO` con el mismo problema. De esta manera es posible extraer los rendimientos de los dos planificadores operando en el mismo simulador físico. Este proceso se muestra representado en la figura 4.9.

4.2.7. Aplicación en el robot YuMi

Aunque no ha sido posible poner en marcha la ejecución de los planificadores utilizando el robot YuMi, si que se diseñó y se implementó gran parte del código necesario para hacerlo.

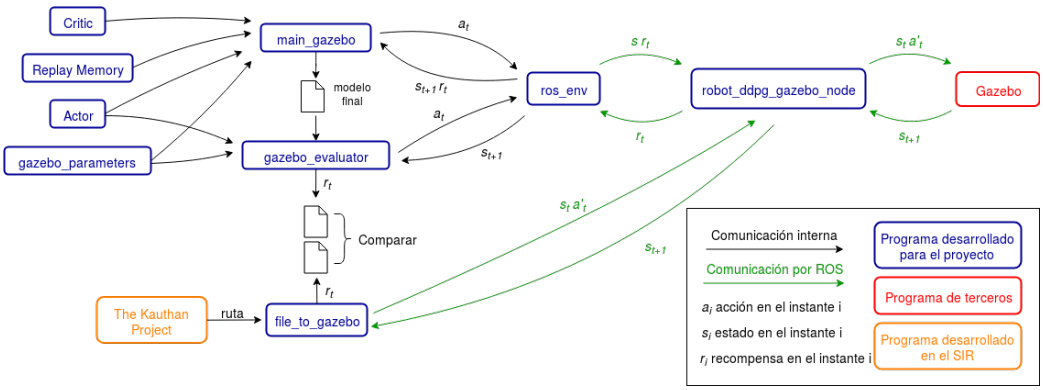

Figura 4.9: Diagrama del funcionamiento del algoritmo DDPG ejecutando la simulación en `GAZEBO` y la comparación de resultados con la trayectoria computada por `The Kautham Project`. Las acciones a representan los puntos de paso de las *splines* mientras que a' son trayectorias.

Puesto que ya no se trabaja con una simulación, el proceso es más elaborado, tal y como se puede comprobar en la figura 4.10.

4.3. Detalles experimentales

4.3.1. Software

Las *splines* fueron implementadas con la librería `ALGLIB 3.13.0` para `C++`[8]. Para la comunicación con el robot YuMi, el simulador `GAZEBO` y otros paquetes se utilizó `ROS`citeQuigley09.

Todo el *software* desarrollado se encuentra disponible en los anexos del trabajo. A continuación se resume el objetivo de cada paquete:

`robot_ddpg_gazebo:` El paquete se encarga de generar trayectorias a partir de puntos de paso de splines y genera recompensas a partir de los resultados de dichas trayectorias, actuando como mediador entre el agente y la simulación `GAZEBO`[5].

`robot_ddpg_agent:` El paquete contiene todos los agentes creados, así como un envoltorio para el agente que trabaja con la simulación `GAZEBO` para que pueda comunicarse por `ROS` con `robot_ddpg_gazebo`[6].

[5]Este paquete también se encuentra disponible en el siguiente repositorio: `https://github.com/GuillermoUrcera/robot_ddpg_gazebo`

[6]Este paquete también se encuentra disponible en el siguiente repositorio: `https://github.com/GuillermoUrcera/robot_ddpg_agent`

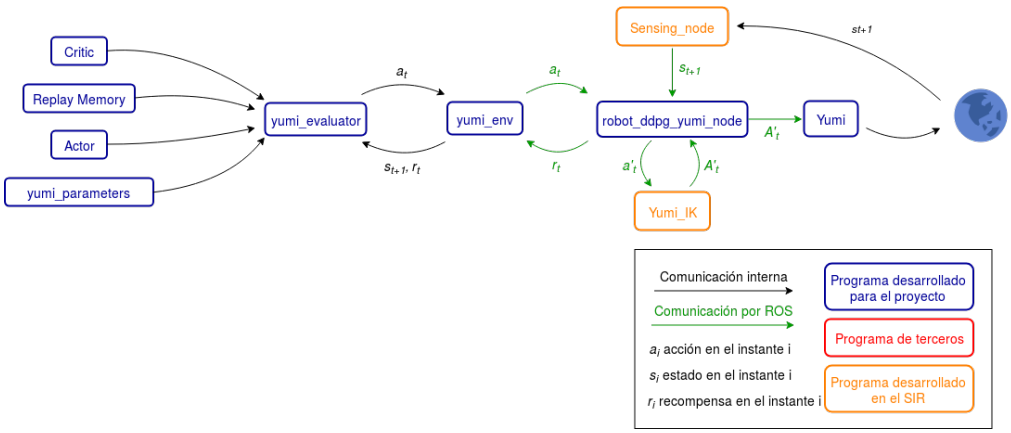

Figura 4.10: Diagrama del funcionamiento de la cadena de *software* para ejecutar trayectorias en YuMi. las acciones a representan los puntos de paso, a' es la trayectoria de la *spline* y A' es la trayectoria de todas las articulaciones del YuMi para que el TCP siga la *spline*.

`robot_ddpg_agent_ros:` El paquete contiene a `robot_ddpg_agent` como submódulo, actuando como envoltorio `ROS`. También dispone de un ejecutable capaz de leer archivos de trayectorias generadas en `The Kautham Project` y enviarlas a `robot_ddpg_gazebo`[7].

`robot_ddpg_yumi:` El paquete actúa de forma análoga a `robot_ddpg_gazebo` pero sobre el entorno físico del robot YuMi, cargando la política del agente guardada y llamando a otros nodos auxiliares como el de cálculo de la cinemática inversa del YuMi o el del sensado mediante cámara para evaluar el desplazamiento de los obstáculos. Este paquete no consiguió ponerse en marcha.

4.3.1.1. Precisión contra velocidad

Las simulaciones físicas, como las efectuadas en el simulador `Gazebo`, suponen un gasto computacional, y por tanto de tiempo, importante, llegando a formar una fracción considerable del tiempo total de ejecución del algoritmo. Se vuelve entonces de interés intentar minimizar la carga que este supone para de esta manera agilizar el proceso de aprendizaje.

Una manera de conseguir esto es incrementando el paso de la simulación física, reduciendo el número de operaciones que esta debe realizar por episodio. Sin embargo, con un paso más elevado se incurre en una mayor

[7]Este paquete también se encuentra disponible en el siguiente repositorio: `https://github.com/GuillermoUrcera/robot_ddpg_agent_ros`

Figura 4.11: Gráfica del coeficiente de variación y del tiempo por episodio para diferentes tamaños de paso de la simulación en `Gazebo`.

imprecisión en los cálculos, tal y como se puede observar en la figura 4.11. Observando esta figura, se puede ver como a un menor paso se incrementa el tiempo de simulación de forma exponencial, mientras que la precisión, expresada por medio del coeficiente de variación, muestra un comportamiento más complicado pero del cual se puede extraer una relación más linear con respecto al tamaño del paso.

Para generar los datos mostrados en la figura 4.11 se ejecutaron 1.000 episodios por cada valor de paso. Cada episodio consistía en preparar la escena, ejecutar los movimientos y devolver la recompensa.

Teniendo en cuenta estos datos y el tiempo del cual se disponía para realizar el estudio e iterar sobre el, se escogió un valor de $0,001$ para el paso de la simulación.

Capítulo 5

Resultados

En este capítulo se exponen los resultados de las diferentes implementaciones y de la comparativa del algoritmo con KPIECE. La métrica más importante es la recompensa total en cada episodio, pues es el objetivo a maximizar por el algoritmo. Sin embargo, las medidas de recompensas pueden ser muy ruidosas. Para solucionar este problemas, se optó por las siguientes medidas:

- Se pasó a calcular la media del valor a medir cada cierto número de episodios, ejecutando el algoritmo una cantidad de episodios adicionales en los que el agente no se actualizaba. Estos episodios de evaluación se encuentran también exentos de ruido.

- Se optó por mostrar también el valor de las predicciones del crítico, métrica que da una idea del retorno que experimenta, o espera experimentar, el agente. Una ventaja sobre la recompensa es que es mucho menos ruidosa[33].

5.1. Experimentos en problemas sencillos

A continuación se muestran los resultados de los experimentos propuestos en el capítulo anterior con los problemas `pendulum-v0` y `MountainCarContinuous-v0` de *Open AI Gym*. Con tal de

5.1.1. Pendulum

Es difícil de sacar conclusiones sólidas de este problema, pues se demostró que el algoritmo encontraba una muy buena política a los pocos episodios,

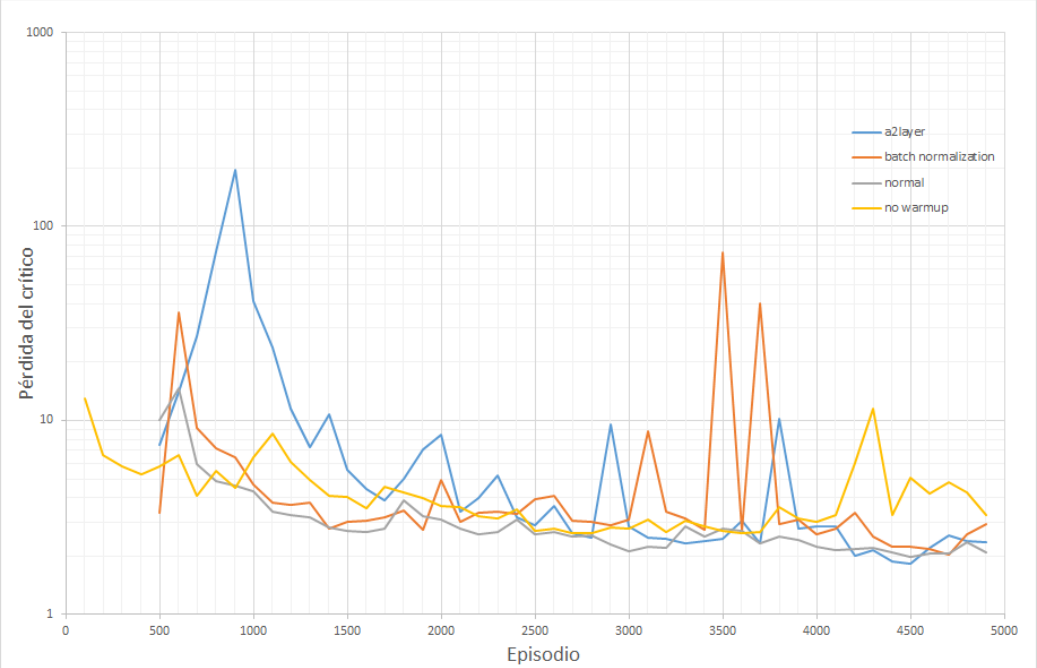

Figura 5.1: Gráfica de la pérdida del crítico según el episodio para diferentes implementaciones del problema `pendulum-v0`

por lo que la evolución del aprendizaje no es fácilmente estudiable. Lo único que parece relamente haber tenido efecto es el añadir las acciones a la segunda capa del crítico, tal y como se puede observar en la figura 5.3, en la cual se puede observar un efecto negativo en la velocidad de aprendizaje. Contra las expectativas, eliminar el precalentamiento no parece resultar en efectos negativos, puede que sea por la sencillez de la topología del espacio de recompensas del problema.

5.1.2. MountainCar

En la figura 5.6 se pueden observar las recompensas reales que obtenía el agente durante su entrenamiento. Esto es debido a que la técnica de la evaluación *on-line* discutida al principio del capítulo no ofrecía buenos resultados. En este problema si que se puede apreciar un mejor comportamiento con la arquitectura del artículo original en la figura 5.6. Otra vez se demuestra la poca importancia del precalentamiento, sobretodo en la figura 5.5, donde se esperaba ver como el crítico sufría de *overfitting* al principio del entrenamiento al actualizarse siempre con las mismas muestras.

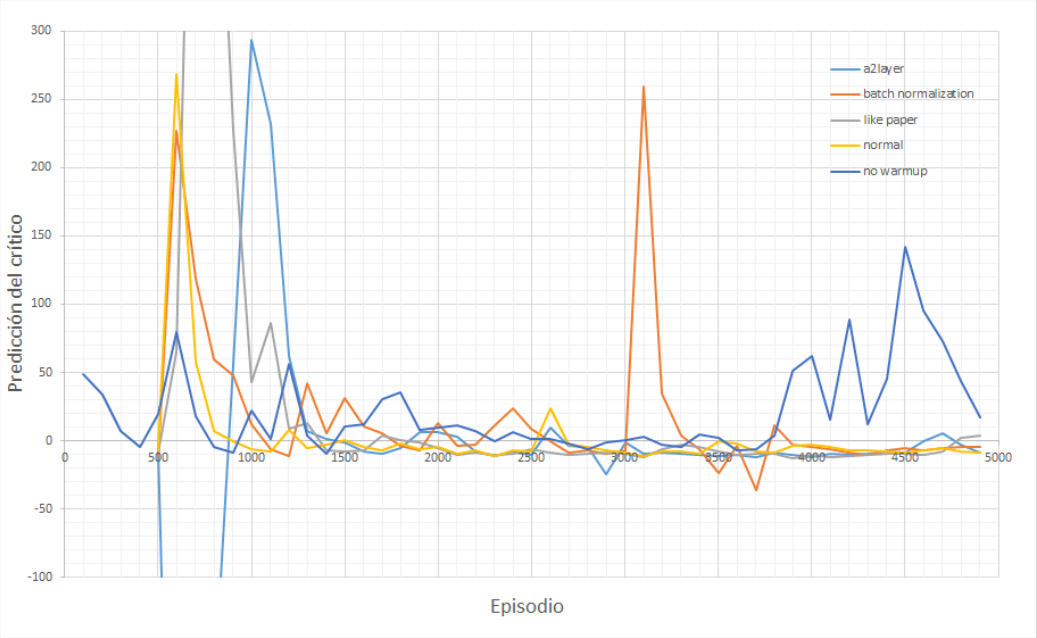

Figura 5.2: Gráfica de la predicción del crítico del retorno esperado según el episodio para diferentes implementaciones del problema `pendulum-v0`

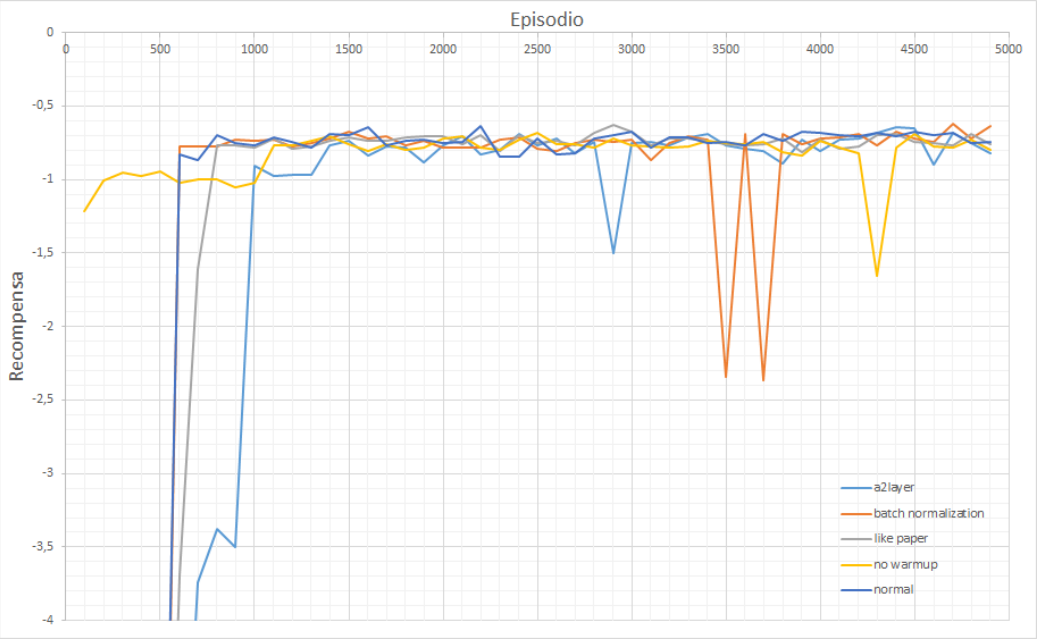

Figura 5.3: Gráfica de la recompensa según el episodio para diferentes implementaciones del problema `pendulum-v0`

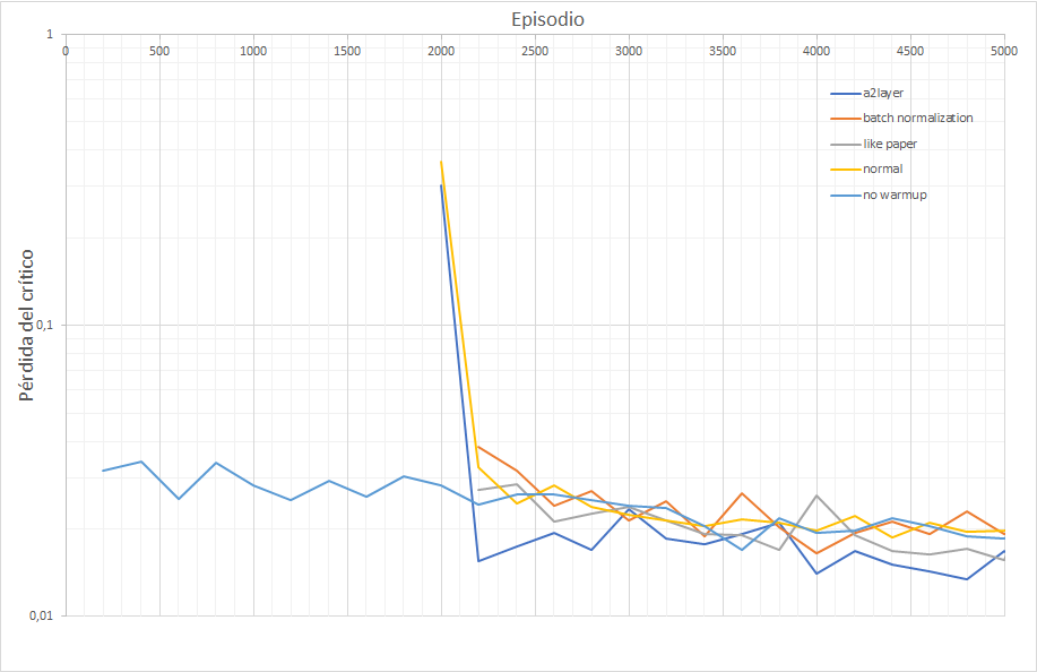

Figura 5.4: Gráfica de la pérdida del crítico según el episodio para diferentes implementaciones del problema `MountainCarContinuous-v0`

5.2. Experimentos en la simulación GAZEBO

En las figuras 5.8 y 5.9 se puede observar claramente como el reordenamiento de las entradas sí ha tenido un efecto positivo, aunque pequeño, en el aprendizaje. En la figura 5.7 se puede ver como la arquitectura del artículo original ayuda a estabilizar el aprendizaje.

5.3. Comparativa con KPIECE

En la figura 5.10 se pueden observar algunos de los caminos escogidos por KPIECE para cada uno de los tests utilizados para comparar el algoritmo con DDPG. Al ser un planificador aleatorio, a veces encuentra una solución sencilla nada más comenzar a explorar, mientras que otras veces, puede quedarse atascado hasta que encuentre una solución válida, lo cual explica la dispersión tan elevada vista en las figuras 5.11, 5.12 y 5.14 respecto a la observada para DDPG.

El algoritmo DDPG suele encontrar soluciones más cortas y con mu-

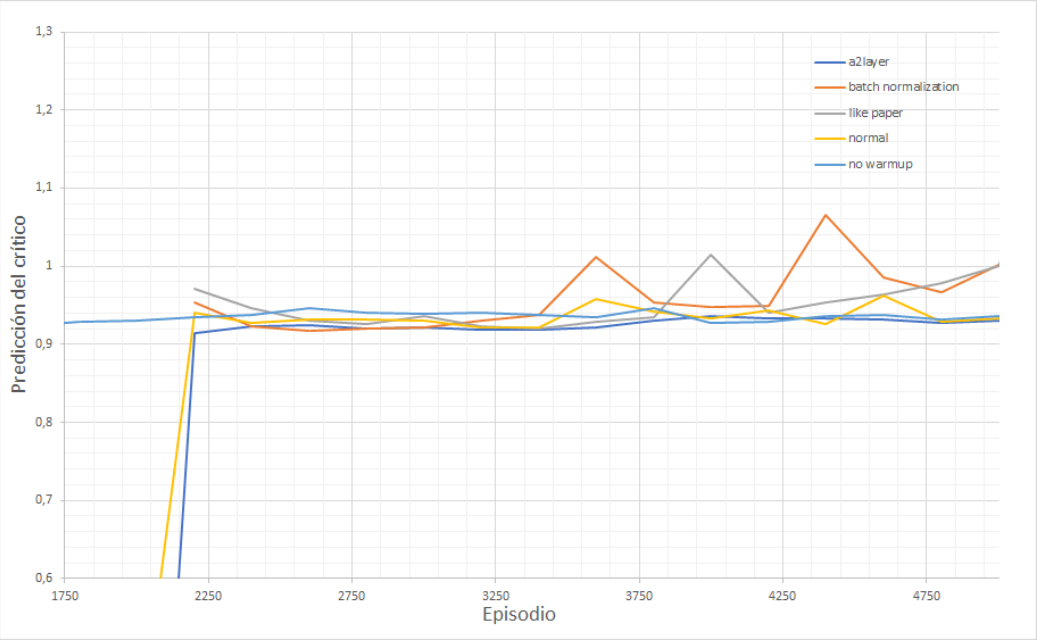

Figura 5.5: Gráfica de la predicción del crítico del retorno esperado según el episodio para diferentes implementaciones del problema `MountainCarContinuous-v0`

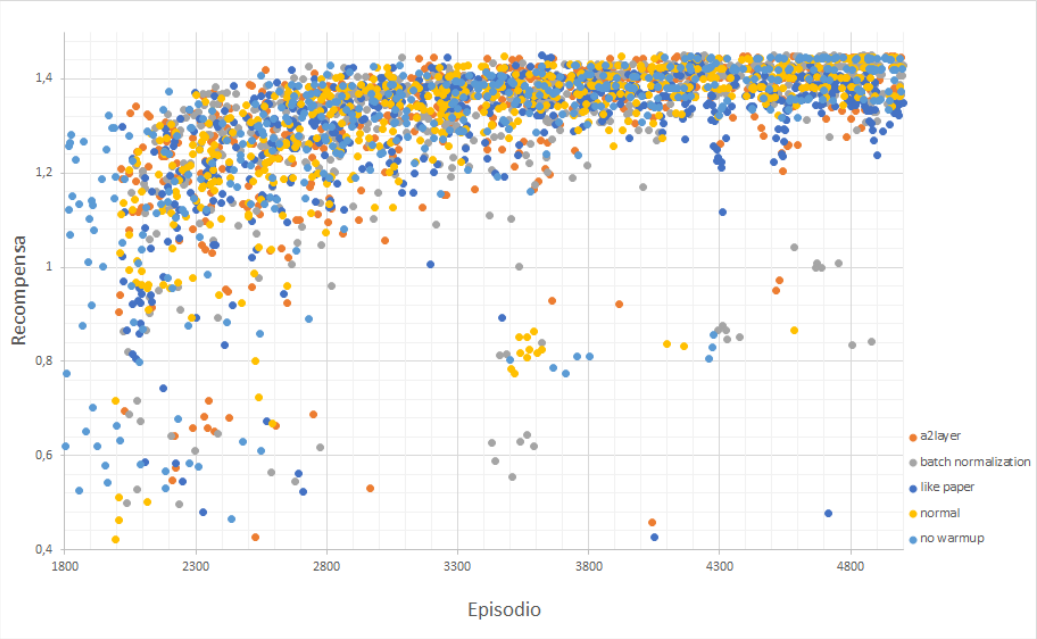

Figura 5.6: Gráfica de la recompensa según el episodio para diferentes implementaciones del problema `MountainCarContinuous-v0`

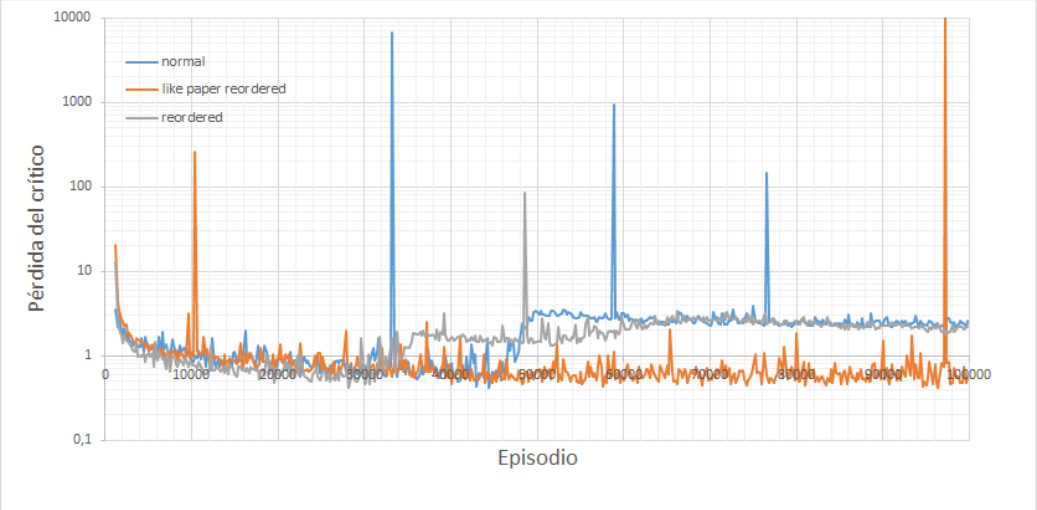

Figura 5.7: Gráfica de la pérdida del crítico según el episodio para diferentes implementaciones del problema en simulación

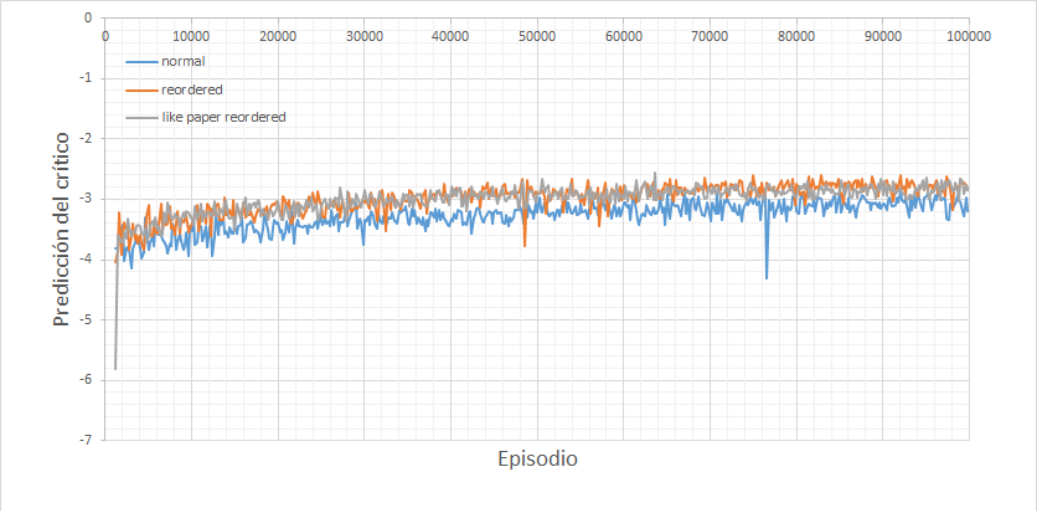

Figura 5.8: Gráfica de la predicción del crítico del retorno esperado según el episodio para diferentes implementaciones del problema en simulación

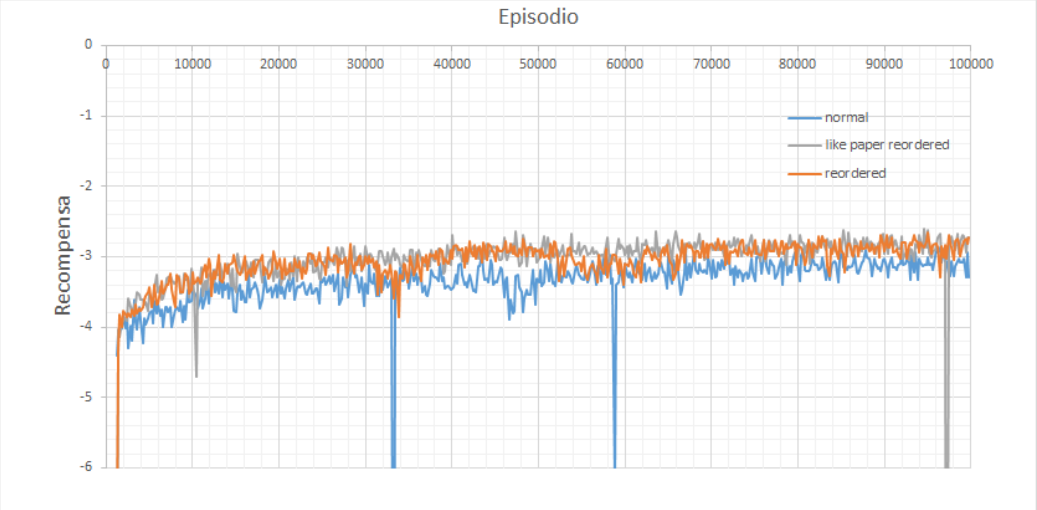

Figura 5.9: Gráfica de la recompensa según el episodio para diferentes implementaciones del problema en simulación

cha más consistencia[1]. En cambio, sufre más penalizaciones por colisiones con los obstáculos. El resultado final es que ambos algoritmos se encuentran bastante igualados en cuanto a la recompensa recibida por la trayectoria (cabe destacar que el algoritmo KPIECE desconoce la función de recompensa completamente), lo cual aporta confianza en la calidad de las trayectorias generadas mediante DDPG.

Cuando se llega al tiempo de cálculo, sin embargo, DDPG es ordenes de magnitud más veloz, como era de esperar. Se podría criticar la comparación argumentando que, siendo un algoritmo en lazo abierto que además está forzado a avanzar hacia delante, escoja lo que escoja (y estando el punto objetivo siempre en su trayectoria), no se trata de una comparación justa. Es un punto válido. Se esta comparando un algoritmo muy restringido en su entorno de funcionamiento ideal con un algoritmo mucho más generalista.

[1]Cabe recordar que la distancia máxima que puede llegar a recorrer una trayectoria generada con el planificador se ve acotada dentro del rango que fija la *spline* utilizada en su generación.

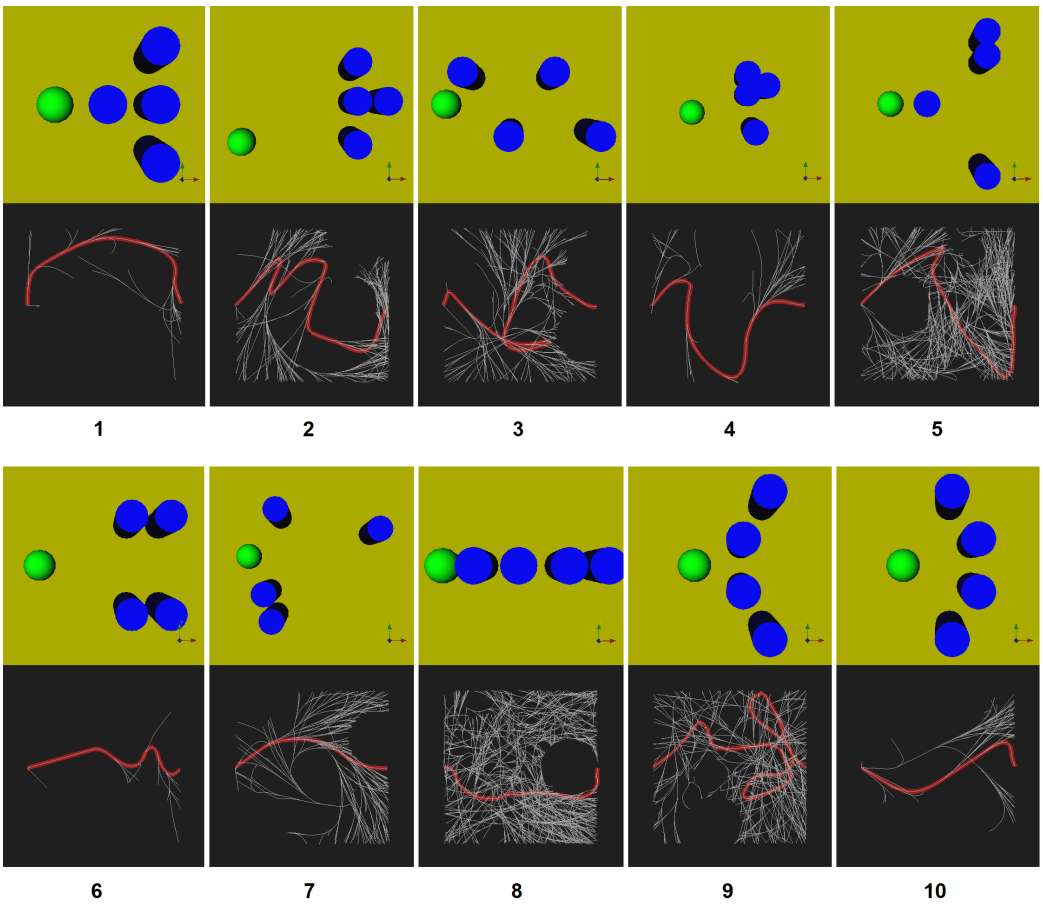

Figura 5.10: Cada uno de los 10 escenarios de prueba, vistos en `The Kautham Project`. Arriba, la vista gráfica del TCP, la esfera, y los obstáculos. Abajo, el espacio de configuraciones con la exploración realizada por KPIECE.

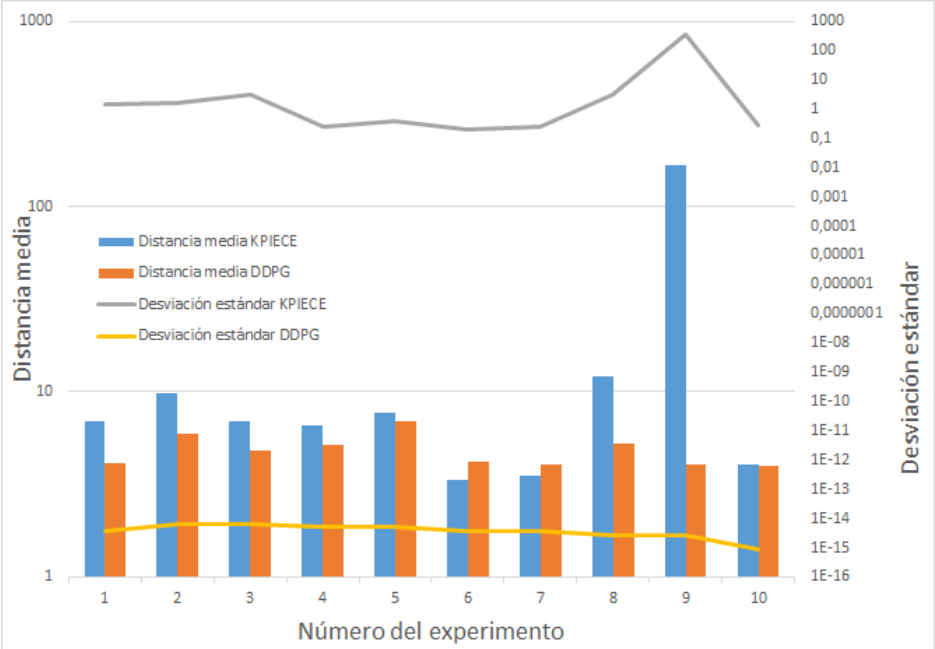

Figura 5.11: Gráfica de la longitud media de la trayectoria del algoritmo DDPG y KPIECE, junto a sus desviaciones estándar.

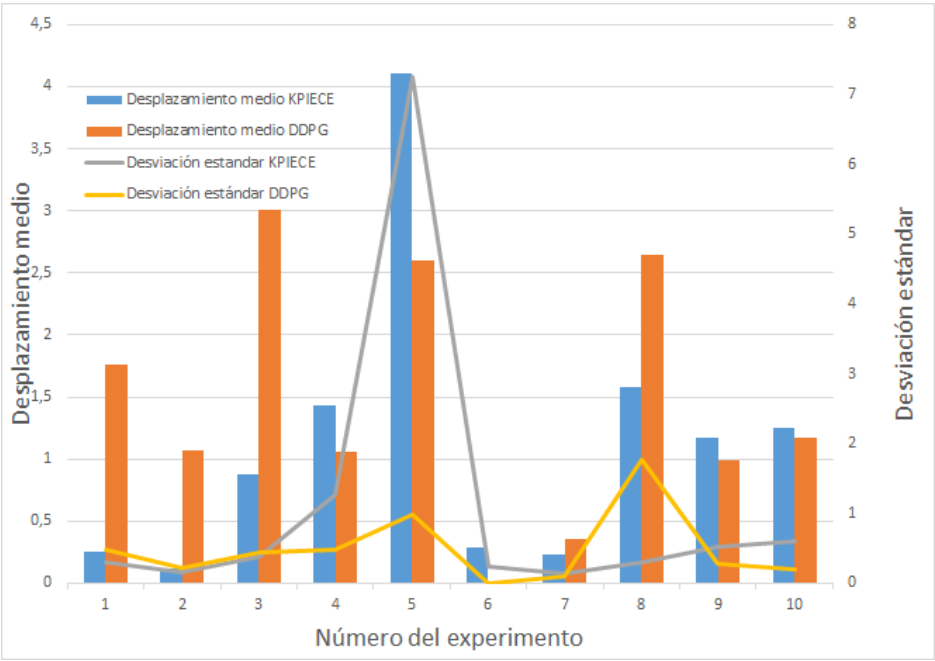

Figura 5.12: Gráfica del desplazamiento medio de obstáculos del algoritmo DDPG y KPIECE, junto a sus desviaciones estándar.

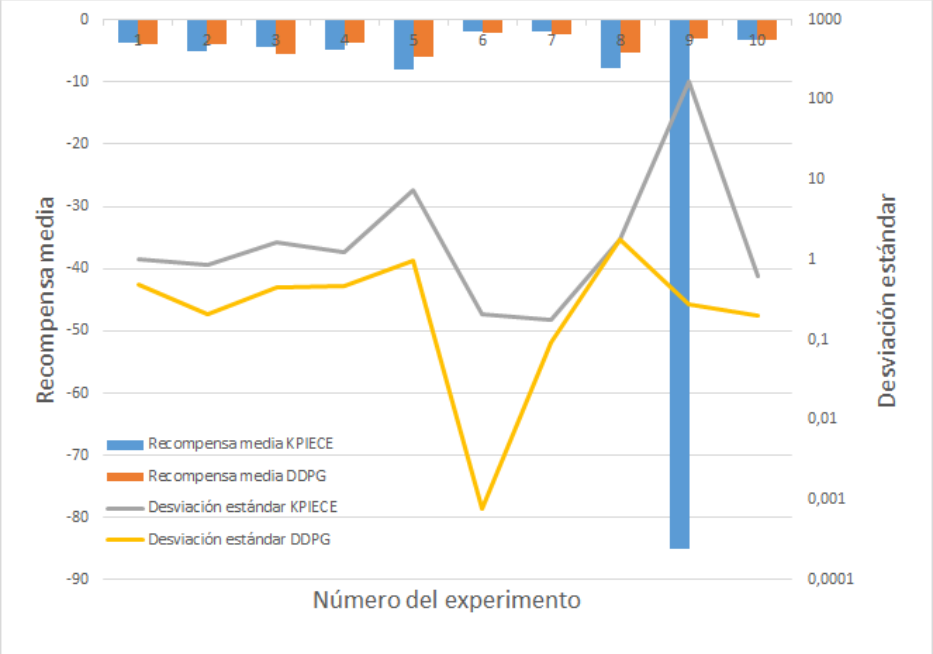

Figura 5.13: Gráfica de la recompensa media conseguida por algoritmo DDPG y KPIECE, junto a sus desviaciones estándar.

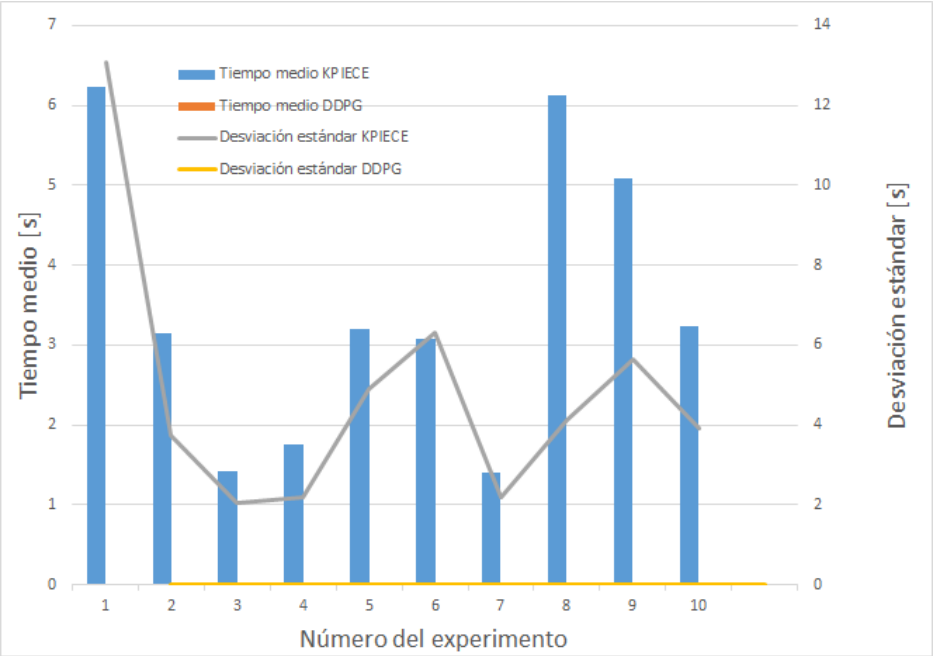

Figura 5.14: Gráfica del tiempo de cálculo medio de la trayectoria para el algoritmo DDPG y KPIECE, junto a sus desviaciones estándar.

Capítulo 6

Conclusiones y trabajo futuro

El aprendizaje profundo por refuerzo muestra gran potencial en el mundo de la robótica, antes intratable debido a experiencia costosa de conseguir, además de ruidosa, dispersa, sobre un espacio continuo y difícil de reproducir[19] [33], pero que se está volviendo más accesible con avances recientes en aprendizaje automático.

La capacidad de las redes neuronales de operar en espacios de dimensionalidad elevada sin sufrir la *maldición de la dimensionalidad* de forma tan severa y sobre todo su capacidad para generalizar experiencia y extraer patrones de información importantes las hace idóneas para su aplicación a algoritmos de aprendizaje por refuerzo en el campo de la robótica. Otra de las grandes ventajas de las que gozan estos métodos es su rapidez de respuesta al tratarse de un proceso determinista, hecho que ha demostrado ser decisivo en la comparación con planificadores del estado del arte actual. Sin embargo, aun son necesarios grandes avances para hacer que el aprendizaje por refuerzo pase de las simulaciones a robots reales, donde es difícil justificar la gran inversión en aprendizaje que exigen y su falta de trazabilidad.

Como trabajo futuro, construyendo sobre lo expuesto en este trabajo, sería muy interesante la ampliación del generador de trayectorias desarrollado para que operara en bucle cerrado, ya fuere controlando el *TCP* mediante un vector de velocidades o incluso llegando a controlar todas las articulaciones de un robot simulado para después llevarlo a la práctica.

Para ello, se intuye que será necesario acabar de refinar los métodos, muy noveles, que se han expuesto en este trabajo y que utiliza *DDPG* para estabilizar su aprendizaje. Sería interesante trabajar en mejoras al *experience replay*, tal que se consiguieran aliviar algunos de los problemas expuestos.

Agradecimientos

Me gustaría mostrar mi agradecimiento a mi tutor, Jan Rosell, por toda la ayuda y orientación prestada. También me gustaría dar las gracias a Leopold Palomo por sus consejos y su ayuda técnica, a Aliakbar Akbari por su ayuda con el robot YuMi, a Mohammed Diab por su ayuda con el sensado y a Muhayyuddin por su ayuda con `The Kautham Project`.

Bibliografía

[1] *A Brief History of Go | American Go Association*. http://www.usgo.org/brief-history-go, visitado el 2018-04-11.

[2] *AlphaGo | Deepmind*. https://deepmind.com/research/alphago/, visitado el 2018-04-11.

[3] Abadi, M., A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, J. Yangqing, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. ÉMan, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. GaséVi, V. Oriol, P. Warden, M. Wattenberg, M. Wicke, Y. Yu y X. Zheng: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, 2015. https://www.tensorflow.org/.

[4] ABB: *YuMi*. http://new.abb.com/future/yumi, visitado el 2018-04-26.

[5] Amazon: *Amazon DSSTNE*. https://github.com/amzn/amazon-dsstne, visitado el 2018-04-24.

[6] Bejjani, W., R. Papallas, M. Leonetti y M. R. Dogar: *Learning Deep Policies for Physics-Based Manipulation in Clutter*. arXiv, 2018. http://arxiv.org/abs/1803.08100.

[7] Biagiotti, L. y C. Melchiorri: *Trajectory Planning for Automatic Machines and Robots*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN 978-3-540-85628-3. http://link.springer.com/10.1007/978-3-540-85629-0.

[8] Bochkanov, S.: *ALGLIB*. www.alglib.net.

[9] Bruin, T. de, J. Kober, K. Tuyls y R. Babuska: *The importance of experience replay database composition in deep reinforcement learning.* Deep Reinforcement Learning Workshop, Advances in Neural Information Processing Systems (NIPS), págs. 1–9, 2015. https://pdfs.semanticscholar.org/091d/1712b97c0353b9a79e8563e7f174a8fd5450.pdf{%}5Cnrll.berkeley.edu/deeprlworkshop/papers/database{_}composition.pdf.

[10] Caruana, R., Y. Lou, J. Gehrke, P. Koch, M. Sturm y N. Elhadad: *Intelligible Models for HealthCare: Predicting Pneumonia Risk and Hospital 30-day Readmission.* En *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, págs. 1721–1730, New York, New York, USA, 2015. ACM Press, ISBN 9781450336642. http://dl.acm.org/citation.cfm?doid=2783258.2788613.

[11] Castelvecchi, D.: *El problema de la caja negra.* Investigación y Ciencia, págs. 14–17, apr 2017.

[12] Choset, H., K. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. Kavraki y S. Thrun: *Principles of robot motion.* The MIT press, London, 1ª ed., 2005, ISBN 978-0-262-03327-5.

[13] Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu y P. Kuksa: *Natural Language Processing (Almost) from Scratch.* Journal of Machine Learning Research, 12:2493–2537, 2011, ISSN 0891-2017.

[14] Covington, P., J. Adams y E. Sargin: *Deep Neural Networks for YouTube Recommendations.* En *Proceedings of the 10th ACM Conference on Recommender Systems - RecSys '16*, págs. 191–198, New York, New York, USA, 2016. ACM Press, ISBN 9781450340359. http://dl.acm.org/citation.cfm?doid=2959100.2959190.

[15] Dosovitskiy, A. y V. Koltun: *Learning to Act by Predicting the Future.* http://arxiv.org/abs/1611.01779, nov 2016.

[16] Gupta, K. y A. P. del Pobil: *Practical motion planning in robotics: current approaches and future directions.* John Wiley & Sons, Bath, 1ª ed., 1998, ISBN 0-471-98163-X.

[17] Ioffe, S. y C. Szegedy: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.* CoRR, abs/1502.0, 2015, ISSN 0717-6163. http://arxiv.org/abs/1502.03167.

[18] Kavraki, L., P. Svestka, J. C. Latombe y M. Overmars: *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*. IEEE Transactions on Robotics and Automation, 12(4):566–580, 1996, ISSN 1042296X. http://ieeexplore.ieee.org/document/508439/.

[19] Kober, J., J. Andrew Bagnell y J. Peters: *Reinforcement Learning in Robotics: A Survey*. En *Springer Tracts in Advanced Robotics*, vol. 97, págs. 9–67. 2014, ISBN 9783642276446.

[20] Koenig, N. y A. Howard: *Design and use paradigms for Gazebo, an open-source multi-robot simulator*. En *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, págs. 2149–2154 vol.3, sep 2004.

[21] Kuffner, J. y S. LaValle: *RRT-connect: An efficient approach to single-query path planning*. En *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, págs. 995–1001, 2000, ISBN 0-7803-5886-4. http://ieeexplore.ieee.org/document/844730/.

[22] Lambert, F.: *Understanding the fatal Tesla accident on Autopilot and the NHTSA probe*, 2016. https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/, visitado el 2018-04-06.

[23] Latombe, J. C.: *Robot motion planning*. Kluwer Academic Publishers, London, 8ª ed., 2004, ISBN 0-7923-9129-2.

[24] LaValle, S. M.: *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. In, 129:98–11, 1998, ISSN 1098-6596. http://scholar.google.com/scholar?hl=en{&}btnG=Search{&}q=intitle:Rapidly-exploring+random+trees:+A+new+tool+for+path+planning{#}0.

[25] LaValle, S. M. y J. J. Kuffner Jr.: *Rapidly-Exploring Random Trees: Progress and Prospects*. En *Algorithmic and Computational Robotics: New Directions*, págs. 293–308. 2000.

[26] Lee, M.: *Go Players Excited About 'More Humanlike' AlphaGo Zero*, 2016. http://koreabizwire.com/go-players-excited-about-more-humanlike-alphago-zero/98282, visitado el 2018-04-11.

[27] Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver y D. Wierstra: *Continuous control with deep reinforcement learning*. En *ICLR 2016*, 2016, ISBN 2200000006. http://arxiv.org/abs/1509.02971.

[28] Lin, L. J.: *Reinforcement Learning for Robots Using Neural Networks*. Tesis de Doctorado, Carnegie Mellon University, 1993.

[29] Mahendran, A. y A. Vedaldi: *Understanding deep image representations by inverting them*. En *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 07-12-June, págs. 5188–5196. IEEE, jun 2015, ISBN 978-1-4673-6964-0. http://ieeexplore.ieee.org/document/7299155/.

[30] Melchiorri, C.: *Trajectory Planning for Robot Manipulators*.

[31] Mitchell, T. M.: *Machine Learning*. 1997, ISBN 0071154671.

[32] Mnih, V., K. Kavukcuoglu, D. Silver y E. Al.: *Human-level control through deep reinforcement learning*. Nature, 518(February):529–533, 2015, ISSN 10450823. http://dx.doi.org/10.1038/nature14236.

[33] Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra y M. Riedmiller: *Playing Atari with Deep Reinforcement Learning*. arXiv, dec 2013, ISSN 0028-0836. http://arxiv.org/abs/1312.5602http://www.nature.com/articles/nature14236.

[34] Mordvintsev, A., C. Olah y M. Tyka: *Inceptionism: Going Deeper into Neural Networks*. https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html, visitado el 2018-04-06.

[35] Murphy, K.: *Machine Learning: A Probabilistic Perspective*. 2012, ISBN 9780262018029. http://link.springer.com/chapter/10.1007/978-94-011-3532-0{_}2{%}0Ahttp://www.springerreference.com/index/doi/10.1007/SpringerReference{_}35834.

[36] Nguyen, A., J. Yosinski y J. Clune: *Deep neural networks are easily fooled: High confidence predictions for unrecognizable images*. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 07-12-June:427–436, 2015, ISSN 10636919.

[37] OpenAI: *OpenAI Gym*. https://gym.openai.com/, visitado el 2018-04-26.

[38] Pei, K., Y. Cao, J. Yang y S. Jana: *DeepXplore: Automated Whitebox Testing of Deep Learning Systems*. En *Proceedings of the 26th Symposium on Operating Systems Principles*, págs. 1–18, Shanghai, China, 2017. ISBN 9781450350853. http://arxiv.org/abs/1705.06640{%}0Ahttp://dx.doi.org/10.1145/3132747.3132785.

[39] Redmon, J. y A. Farhadi: *YOLOv3: An Incremental Improvement*. arXiv, 2018.

[40] Rosell, J., A. Pérez, A. Aliakbar, L. Palomo y N. García: *The Kautham Project: A teaching and research tool for robot motion planning*. En *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation*, Barcelona, 2014. Institute of Industrial and Control Engineering (IOC) - Universitat Politecnica de Catalunya (UPC) – Barcelona Tech.

[41] Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel y D. Hassabis: *Mastering the game of Go with deep neural networks and tree search*. Nature, 529(7587):484–489, 2016, ISSN 14764687. http://dx.doi.org/10.1038/nature16961.

[42] Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra y M. Riedmiller: *Deterministic Policy Gradient Algorithms*. En *Proc. of the 31st International Conference on Machine Learning*, págs. 387–395, 2014, ISBN 9781634393973. http://jmlr.org/proceedings/papers/v32/silver14.html.

[43] Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel y D. Hassabis: *Mastering the game of Go without human knowledge*. Nature, 550(7676):354–359, 2017, ISSN 14764687. http://dx.doi.org/10.1038/nature24270.

[44] Strandberg, M.: *Robot Path Planning : An Object-Oriented Approach*. Department of Signals, Sensors and Systems Royal Institute of Technology (KTH), Stockholm, Sweden, 2004, ISBN 91-7283-868-X.

[45] Sutton, R. y A. Barto: *Reinforcement Learning: An Introduction - Complete Draft-*. The MIT Press, Cambridge, Massachusetts; London, England, second edi ed., sep 2018.

[46] Uhlenbeck, G. y L. Ornstein: *On the theory of the brownian motion*. Physical Review, 36(September):823–841, 1930.

[47] Watkins, C. J. C. H. y P. Dayan: *Q-learning*. Machine Learning, 8(3-4):279–292, 1992, ISSN 0885-6125. `http://link.springer.com/10.1007/BF00992698`.

[48] Ziegier, C.: *A Google self-driving car caused a crash for the first time*, 2016. `https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report`, visitado el 2018-04-06.

[49] Şucan, I. A. y L. E. Kavraki: *A sampling-based tree planner for systems with complex dynamics*. IEEE Transactions on Robotics, 28(1):116–131, 2012, ISSN 15523098.