Implementación de Algoritmos de Aprendizaje por Refuerzo Avanzados para el Control en Espacio Continuo

Application of Advanced Reinforcement Learning Algorithms for Continuous Control

Trabajo de Fin de Grado
Curso 2023–2024

Autores

Pablo Pardos Medem
Carlo Sebastiano Dubini Marqués

Director

Raúl Fernández Fernández

Grado en Ingeniería Informática e Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

Implementación de Algoritmos de Aprendizaje por Refuerzo Avanzados para el Control en Espacio Continuo

Application of Advanced Reinforcement Learning Algorithms for Continuous Control

Trabajo de Fin de Grado en Ingeniería Informática e Ingeniería del Software

Autores

Pablo Pardos Medem

Carlo Sebastiano Dubini Marqués

Director

Raúl Fernández Fernández

Convocatoria: Junio 2024

Grado en Ingeniería Informática e Ingeniería del Software

Facultad de Informática

Universidad Complutense de Madrid

27 de Mayo de 2024

Agradecimientos

A nuestras familias y amigos, por hacer posible que nos dedicaramos a esto, apoyándonos durante todo este proceso.

Resumen

Implementación de Algoritmos de Aprendizaje por Refuerzo Avanzados para el Control en Espacio Continuo

En la actualidad, el control automático se está introduciendo más en las tareas de la industria y de nuestras vidas, volviéndose algo muy común para ser el objeto de estudio de la implementación de técnicas de aprendizaje automático. Entre estas tareas destaca la aplicación a nuevas tareas complejas en espacio continuo, en concreto su aplicación en robótica y control automático. En este documento se abordan las diferentes características del aprendizaje por refuerzo y las redes neuronales, estudiando sus diferentes variantes y extensiones, subrayando su importancia en el aprendizaje automático y sus aplicaciones prácticas. Posteriormente se explican los métodos de Policy Gradient, responsables de tratar problemas en espacio continuo sin necesidad de discretizar el espacio, con énfasis en DDPG y HER dentro de estos. Estas técnicas han sido implementadas a lo largo de este trabajo de fin de grado para ejecutar el control de un péndulo y un brazo robot industrial. Con los datos posteriores al entrenamiento de este brazo robot en MuJoCo se han encontrado los parámetros que se consideran más importantes para el modelo. Se ha hecho una iteración sobre los posibles cambios a las estructuras, optimizadores y parámetros de control para realizar un análisis completo que permita más adelante añadir consideraciones de objetivos y obstáculos adicionales para un posible trabajo futuro. Por último, se ha diseñado la memoria de tal forma que se explican los pasos a seguir para su correcta implementación.

Palabras clave

Aprendizaje por refuerzo, Deep Learning, Robótica, Aprendizaje automático, Redes neuronales, Deep Deterministic Policy Gradient, Hindsight Experience Replay

Abstract

Application of Advanced Reinforcement Learning Algorithms for Continuous Control

In current times, automatic control is being introduced increasingly in the industrial processes and our lives, becoming something quite common to be the object of study through the implementation of machine learning techniques. Among these tasks and processes, complex tasks in continuous spaces stand out, highlighting its application in robotics and automatic control. This paper discusses the different characteristics of reinforcement learning and neural networks, studying their variants and extensions as we emphasize their importance in machine learning and their practical applications. Subsequently, Policy Gradient Methods are explained in detail, responsible for dealing with problems in continuous space without a real need for discretization, emphasizing DDPG and HER within these. These techniques have been implemented throughout this Bachelor's Degree Final Project to execute the control of a pendulum and an industrial robot arm. With the data following the training of this robot arm in MuJoCo, we have found the parameters that are considered the most important for the robot arm by iterating on possible changes to the structures, optimizers, and control parameters to perform a complete analysis to add additional goals and obstacle considerations for future work. Finally, this paper has been designed to explain the steps that must be taken for its correct implementation.

Keywords

Reinforcement Learning, Deep Learning, Robotics, Machine Learning, Neural Networks, Deep Deterministic Policy Gradient, Hindsight Experience Replay

Índice

1. Introducción **1**
1.1. Motivación 1
1.2. Objetivos 2
1.3. Plan de trabajo 3

2. Estado de la Cuestión **5**

3. Descripción del Trabajo **7**

4. Aprendizaje por refuerzo **9**
4.1. Introduccion del Aprendizaje por Refuerzo 9
4.2. Procesos de decisión finitos de Markov: F-MDP 10
4.2.1. Objetivos y recompensas 12
4.2.2. Políticas, optimalidad y Funciones valor 13
4.3. Programación dinámica 13
4.3.1. Iteración de políticas y valores 14
4.4. Aprendizaje por diferencia temporal: TDL 15
4.4.1. Sarsa: TDL On-policy 16
4.5. Q-Learning: TDL Off-policy 17
4.5.1. Double Q-learning y sesgo de maximización 17
4.6. Aplicaciones del aprendizaje por refuerzo. 18
4.7. Puntos clave sobre aprendizaje por refuerzo. 19

5. Deep Learning **21**
5.1. Arquitectura de Redes Neuronales Artificiales 22
5.1.1. Capas en redes neuronales 23
5.1.2. Arquitecturas generales 24
5.2. Entrenamiento y algoritmos de optimización 26
5.2.1. Proceso de entrenamiento de una red neuronal 26

5.2.2. Regularización y optimización de modelos 27
5.2.3. Algoritmos de optimización 29
5.3. Aplicaciones del Deep Learning 31
5.4. Puntos clave sobre Deep Learning 31

6. Desafíos del espacio continuo **33**
6.1. Problemas de Aprendizaje Continuo 34
6.2. Aproximación de Políticas 35
6.3. Deep Q-learning 37
6.4. Deep Deterministic Policy Gradient (DDPG) 38
6.4.1. Aprendizaje de política en DDPG 39
6.4.2. Exploración y explotación 40
6.4.3. Extensiones y Aplicaciones 41
6.4.4. Hindsight Experience Replay: HER 42

7. Tecnologías utilizadas **45**
7.1. Python 45
7.2. Plataformas de desarrollo 46
7.2.1. Visual Studio Code 46
7.2.2. Github 46
7.3. Herramientas de desarrollo 46
7.3.1. Gym 46
7.3.2. Gymnasium-Robotics 47
7.3.3. Keras 47
7.3.4. TensorFlow 47
7.3.5. MuJoCo 48

8. Implementación **49**
8.1. Algoritmo de DDPG en Python 49
8.1.1. HER:Hindsight Experience Replay 52
8.2. Agente 53
8.2.1. Actor 55
8.2.2. Critic 56
8.2.3. Replay Buffer 56
8.2.4. Funcionamiento del Agente 57
8.3. Péndulo 60
8.3.1. Configuración y Diseño 61
8.3.2. Resultados 61

9. Controlando un robot manipulador en un espacio continuo 63
9.1. Robot Manipulador 63
9.1.1. Mujoco y Fetch 63
9.1.2. Fetch-Reach 65
9.2. Variables estudiadas 66
9.3. Variación de función de recompensa 68
9.4. Variación de uso de HER 70
9.5. Variación de las capas y número de neuronas 73
9.6. Variación de las funciones de activación 75
9.7. Variación de los optimizadores 77
9.8. Variación de otros hiperparámetros 79
9.8.1. Variación de los parámetros de aprendizaje 79
9.8.2. Variación de Gamma 80
9.8.3. Variación de Tau 81
9.8.4. Variación del tamaño de lote 82
9.8.5. Variación de ruido 83
9.9. Mejor modelo encontrado 85

10.Conclusiones y Trabajo Futuro 87

Conclusions and Future Work 91

Contribuciones Personales 95

Bibliografía 99

Índice de figuras

4.1. Estructura básica de señales en MDP 12
4.2. Diagrama estructural de doble Q-Learning 18

5.1. Red neuronal con capas conectadas de diferente número de neuronas. 23
5.2. Estructura básica del perceptron. 24
5.3. Ejemplo de red convolucional de Cecbur(2019) 25

6.1. *Q-Learning vs Deep Q-Learning* por (Sebastianelli et al., 2021) 38
6.2. Estructura de un actor y crítico con estados de tres variables y acciones de dos y tres variables 39

8.1. Diagrama de clase del agente 53
8.2. Función de tangente hiperbólica, Geek3(2014), CC BY 3.0 56
8.3. *Illustration of the replay buffer in the RL.* como se ve en Lee y Lee (2020) CC BY-NC-ND 4.0 57
8.4. Imagen del entorno del péndulo 60
8.5. Resultado de las medias de entrenamiento en entorno péndulo no normalizadas 62

9.1. Imagen del entorno FetchReach 64
9.2. Gráfica de resultados: Reward 1 68
9.3. Gráfica de resultados: Reward 2 69
9.4. Gráfica de resultados: HER A 70
9.5. Gráfica de resultados: HER B 71
9.6. Gráfica de resultados: HER C 71
9.7. Gráfica de resultados: HER D 72
9.8. Gráfica de resultados: Capas 1 74
9.9. Gráfica de resultados: Capas 2 74
9.10. Gráfica de resultados: Funciones de activación 76
9.11. Gráfica de resultados: Optimizadores 77

9.12. Gráfica de resultados: Parámetros de aprendizaje 79
9.13. Gráfica de resultados: Gamma 80
9.14. Gráfica de resultados: Tau 81
9.15. Gráfica de resultados: BatchSize 82
9.16. Gráfica de resultados: Ruido 83

Índice de tablas

8.1. Espacio de observación del péndulo 60
8.2. Espacio de acciones del péndulo 60
8.3. Parámetros para el péndulo 61

9.1. Espacio de Acciones 64
9.2. Parámetros para el robot manipulador 65
9.3. Resultados de explotación de recompensas 69
9.4. Resultados de explotación de cambios en HER 73
9.5. Resultados de explotación de neuronas 75
9.6. Resultados de explotación de diferentes funciones de activación 76
9.7. Resultados de explotación de diferentes optimizadores 78
9.8. Resultados de explotación de parámetros de aprendizaje 80
9.9. Resultados de explotación de Gamma 81
9.10. Resultados de explotación de Tau 82
9.11. Resultados de explotación de tamaño de lote 83
9.12. Resultados de explotación de ruido 84
9.13. Parámetros para el modelo final óptimo 85

10.1. Parámetros para el modelo final óptimo. Bis 88
10.2. Final parameters for the optimal model. 92

Capítulo 1

Introducción

1.1. Motivación

El aprendizaje por refuerzo (AR) ha sido un enfoque con mucho potencial en el campo de la inteligencia artificial durante los últimos años. Este método de aprendizaje automático permite a los agentes aprender de las experiencias que tienen y tomar decisiones óptimas mediante la interacción con su entorno. Además, se ha vuelto particularmente relevante en aplicaciones de control automático y robótica. Esto se debe a que es en estos casos es dificil aplicar un control clásico cuando los sistemas deben adaptarse y funcionar en espacios continuos y dinámicos.

Con estos desafíos en cuenta, en este campo aún hay problemas sin resolver como la generalización de políticas en entornos continuos. Por ejemplo, se necesitan algoritmos capaces de manejar la complejidad y la incertidumbre de estos espacios para avanzar en áreas como la robótica autónoma o la optimización de procesos. Es también necesario en muchos de estos casos que no se pierda precisión ni robustez, lo que nos lleva el plantearnos el uso de algoritmos más avanzados de AR.

En este trabajo nos centraremos principalmente en estudiar tanto el AR como el Deep learning para sentar unas bases que nos permita implementar dos algoritmos avanzados de AR. Estos son conocidos como DDPG (Deep Deterministic Policy Gradient) y HER (Hindsight Experience Replay) y que actúan en espacio continuo y ayudan a la exploración en entornos con recompensas esparcidas.

Al implementarlos, buscamos estudiar las mejoras de la precisión, velocidad y robustez de este modelo, una versión generalizada de lo que aplicaciones más concretas de brazos robóticos pueden hacer. Consideramos que esto podría tener un impacto significativo en diversas industrias y contribuir al impacto que la aplicación de esta disciplina tiene en la industria y vida diaria.

1.2. Objetivos

En este trabajo nos centraremos principalmente en el estudio de algoritmos avanzados de AR para el control de un brazo robot en espacio continuo, utilizando dos algoritmos conocidos como DDPG (Deep Deterministic Policy Gradient) y HER (Hindsight Experience Replay). Estos algoritmos representan enfoques punteros para la resolución de tareas complejas en el campo del control de sistemas dinámicos y gran espacio de estados, donde las decisiones deben tomarse en un espacio de acción continuo.

Para lograr esta implementación, es fundamental entender en profundidad como funcionan los algoritmos de aprendizaje por refuerzo y las redes neuronales profundas. En particular, los algoritmos DDPG y HER son de gran interés para la creación de este controlador debido a su capacidad para manejar tareas de control continuo con alta dimensionalidad. DDPG combina técnicas de aprendizaje supervisado con la capacidad de tomar decisiones en un espacio de acciones continuo, mientras que HER mejora la eficiencia del aprendizaje al reconstruir las experiencias fallidas en estados exitosos, lo cual es especialmente útil en entornos con recompensas esparcidas.

El proceso de implementación de estos algoritmos implicará una exploración exhaustiva e iterativa de las posibles configuraciones, con el objetivo de identificar combinaciones óptimas de hiperparámetros, estructuras de la red neuronal y función de recompensa. Esta etapa será importante para entender qué relevancia tienen y el impacto que generan diferentes configuraciones en el desempeño del controlador.

Para ilustrar la aplicabilidad y eficacia de estos algoritmos, se realizará una implementación en dos casos de estudio encontrados en la librería gym [7.3.1]: el control de un péndulo y el control de un brazo robot. El entorno del péndulo invertido es un problema que ya ha sido estudiado de forma amplia en el ámbito del control y el aprendizaje por refuerzo, debido a su simplicidad y al mismo tiempo su riqueza en dinámicas no lineales. Este problema servirá como un primer paso para validar los algoritmos implementados en un entorno rápido de entrenar y probar. Una vez establecido un buen rendimiento en el péndulo, el objetivo cambiará al control de un brazo robot, que presenta una mayor complejidad debido a sus múltiples grados de libertad y la necesidad de precisión en el posicionamiento y movimiento.

La implementación exitosa de estos algoritmos no solo permitirá obtener controladores efectivos y eficientes, sino que también se busca que estos sean fácilmente entendibles y escalables a entornos más complejos y/o reales. Es el desarrollo de este tipo de controladores adaptativos el que tiene el potencial de revolucionar la manera en se interactúa con sistemas automatizados.

Es con estas ideas en mente que se extiende este documento, de avanzar en la materia de controladores en espacio continuo. En este trabajo no solo se propone explorar y mejorar algoritmos avanzados de control continuo, sino también contribuir al conocimiento sobre como estos algoritmos pueden aplicarse y optimizarse en

problemas de la vida real. También es nuestro objetivo el entender de mejor manera como diversas configuraciones afectan al rendimiento en entornos de control para lograr un mejor uso de esfuerzo y recursos en estas implementaciones.

1.3. Plan de trabajo

Para conseguir estos objetivos y metas que se han propuesto se va a seguir un plan de trabajo por etapas, que también se refleja en como se ha estructurado esta memoria.

Inicialmente se investigará y estudiará la documentación existente sobre aprendizaje automático, aprendizaje por refuerzo y Deep Learning en profundidad. Esta etapa se considera crucial, ya que se requiere un conocimiento sólido de conceptos fundamentales como los Procesos de Decisión de Markov (MDP) y las características diversas que pueden presentar las redes neuronales profundas.

Durante esta fase, se revisarán artículos académicos y libros, con un enfoque particular en la aplicabilidad de estos conceptos al control de sistemas continuos. Se prestará especial atención a como los algoritmos DDPG y HER, ya que han sido utilizados en estudios previos, y se analizarán sus ventajas y limitaciones en contextos similares. De maneara adyacente, se explorarán las arquitecturas de redes neuronales comunes en estos algoritmos, tales como las técnicas de regularización y estructuras que puedan mejorar la estabilidad y eficiencia del aprendizaje de nuestro controlador.

Una vez se haya logrado, se investigarán las herramientas disponibles, en forma de entornos en Gymnasium y posibilidades de implementación de las redes neuronales que se tengan que usar para DDPG. Se analizarán las características de estos entornos, tales como la complejidad del espacio de estado y acción, la disponibilidad de funciones de recompensa adecuadas y la capacidad de personalización dentro de estas.

Una vez identificadas las herramientas y entornos adecuados, se procederá con la implementación inicial de los algoritmos. Esta etapa implicará la configuración de las redes neuronales y la definición de los hiperparámetros iniciales basados en prácticas comunes y la literatura revisada. Se desarrollarán scripts en Python [7.1] para entrenar los modelos en los entornos de simulación seleccionados.

Las pruebas preliminares se centrarán en validar el funcionamiento básico de los algoritmos. Su principal enfoque estará en que las redes neuronales aprendan y mejoren su rendimiento a lo largo del tiempo. Una vez con el entorno final implementado se realizarán múltiples ejecuciones para ajustar las capas, funciones de recompensa, elección del optimizador y validez de las técnicas.

Las pruebas finales tendrán lugar cuando se haya obtenido una estructura sólida y eficiente para realizar ajustes a los hiperparámetros y otras características intere-

santes del modelo. Es con estos cambios que se pretende conseguir el modelo final del controlador, siendo elegido con la estructura y parámetros más eficientes para esta implementación.

Capítulo 2

Estado de la Cuestión

Se ha observado un progreso significativo en los últimos años en el campo del aprendizaje por refuerzo y Deep Learning, principalmente motivado por la creciente capacidad de cómputo que han hecho estas técnicas más atractivas que la simple algoritmia. Esto ha llevado a la creación de nuevas técnicas para aumentar la eficiencia del entrenamiento (Luo et al., 2005) y que se proponga el desarrollo de algoritmos más sofisticados (Sutton y Barto, 2018; Belousov et al., 2021).

Cabe mencionar que esta sección proporciona una visión general, pero no profundiza en los detalles de como funcionan estos métodos. La explicación técnica de los algoritmos de mayor interés como DDPG y HER, así como de otros conceptos fundamentales del deep learning y aprendizaje por refuerzo, se encuentra en los capítulos posteriores de este trabajo.

El interés por estas técnicas ha crecido también por las aplicaciones que tienen en la manufactura e ingeniería inteligente (Li et al., 2023) así como en el sector energético (Perera y Kamalaruban, 2021), en el económico (Mosavi et al., 2020) y el de la conducción, utilizando arquitecturas que ahora mismo se consideran estado del arte como Ape-X (Hussonnois y Jun, 2022).

Los algoritmos avanzados de aprendizaje por refuerzo han demostrado ser herramientas poderosas para abordar problemas complejos en robótica (Dankwa y Zheng, 2020) y otras áreas como la gestión de energía en redes inalambricas de sensores y el control de drones (Qiu et al., 2019; Tagesson, 2021). Esto es debido a que estas áreas requieren un control preciso y que se adapte a diferentes aplicaciones con una flexibilidad notable.

El DDPG es una de las posibles variantes de los métodos de actor-crítico en el aprendizaje por refuerzo que combina las ventajas de los Policy Gradient Methods con las de los métodos de valor Q. Esto permite a los agentes aprender políticas complejas en espacios de acción continua con un desarrollo mayor que las técnicas tradicionales de aprendizaje por refuerzo profundo. Esto es especialmente patente en ciertas aplicaciones que necesitan una gran precisión como es el control de dro-

nes (Tagesson, 2021) y la planificación de caminos en vuelos de vehículos aereos no tripulados (UAVs) (Bouhamed et al., 2020).

Por otro lado, el HER se ha popularizado ya que es una técnica que ayuda a mejorar la eficiencia del aprendizaje por refuerzo en entornos con recompensas dispersas (Andrychowicz et al., 2017). Esta combina el aprendizaje online que realizan estos algoritmos con una parte offline que computa experiencias óptimas con el objetivo de que el sistema aprenda como llegar hasta este punto.

La integración de estas técnicas ha llevado a la creación de sistemas autónomos más robustos y versátiles (Li et al., 2024). Sin embargo, aún existen desafíos que no se pueden pasar por alto, como lo son la necesidad de grandes cantidades de datos para el entrenamiento (Ramezan et al., 2021), la interpretación de los modelos de deep learning(Li et al., 2022), y la transferencia de conocimiento entre tareas (Wang et al., 2020), con grandes frentes de investigación abiertos hoy día.

Es importante también mencionar la creciente importancia que están cobrando la seguridad y la ética en IA y concretamente en el aprendizaje automático. Las decisiones tomadas por sistemas autónomos pueden tener consecuencias significativas en el mundo real como pueden ser en el sector de la medicina (Chen et al., 2021) o conducción (Geisslinger et al., 2021).

El estado actual en el aprendizaje automático y sus técnicas asociadas refleja un campo en rápida evolución, con un gran potencial para transformar diversas industrias y aspectos de nuestras vidas, desde el plano cotidiano al laboral. A medida que avanzamos, será cada vez más importante que los ingenieros e investigadores en aprendizaje automático continúen tratando los problemas que se puedan encontrar con un enfoque multidisciplinar que incluya consideraciones sobre la técnica, la moral y auditabilidad de estas.

Capítulo 3

Descripción del Trabajo

Como se ha explicado en el apartado de plan de trabajo este documento seguirá unas pautas y extensión que imitará el proceso de desarrollo que se ha seguido para implementar el brazo robot y hacer el análisis de parámetro consecuente.

Este estará dividido en capítulos, siendo los tres primeros, Aprendizaje por refuerzo [Capítulo 4], Deep Learning [Capítulo 5] y Desafíos del espacio contínuo [Capítulo 6] bases teóricas sobre las que expandir y terminar implementando.

Se ha considerado de gran relevancia desarrollar en el capítulo sobre el aprendizaje por refuerzo la importancia de conceptos como los MDP, la función recompensa y como esta se relaciona con los objetivos dados, las políticas y funciones valor. De la misma manera se estudiarán antecedentes relevantes al algoritmo a implementar para entender como funcionan los métodos actuales de aprendizaje por refuerzo como lo son la programación dinámica o la técnica de iteración de valores. También se verán métodos más usados como lo son los métodos de aprendizaje por diferencia temporal. Sus principales agentes se pueden entender como Sarsa para los métodos *on-policy* y el pariente más cercano dentro de la resolución de los F-MDP al DDPG, el Q-Learning. Del Q-Learning se heredarán ideas de exploración y explotación que, con algunas modificaciones, se aplicarán más adelante en el DDPG.

En el capítulo de Deep Learning [5] se establecerán conceptos sobre como funcionan las redes neuronales artificiales y como maximizar su utilidad dentro de los campos a tratar. Entre estos temas se verá la elección de una arquitectura adecuada al problema a resolver, como cambia el entrenamiento dependiendo de diferentes algoritmos de optimización, tipos de regularización y elección de la función de pérdida adecuada para la tarea a realizar. Además, también vemos aplicaciones relacionadas con esta disciplina que recalcan su importancia actual en todos estos contextos.

El capítulo donde se introducen los algoritmos de DDPG y HER es el de Desafíos del espacio continuo [6]. Estos métodos se basan en la mejora directa de su política, maximizando la recompensa de esta manera utilizando estimados estocásticos. Estos estimados eliminan la necesidad de discretizar el entorno o las acciones

del modelo. En este capítulo también se verán conceptos como el funcionamiento de la aproximación de políticas y las ventajas que esta trae, así como la parametrización de políticas. Se explorarán a su vez las diferentes partes, variaciones y extensiones consideradas estado del arte de DDPG como lo son Twin Delayed DDPG y Soft Actor-Critic.

Durante los capítulos consecutivos de Tecnologías utilizadas [7], la Implementación del algoritmo DDPG y HER [8] y el diseño del robot manipulador en un estado continuo [9]. Se tratarán en detalle tanto los programas y librerías utilizados para la implementación como esta en sí y los resultados posteriores al análisis de diferentes factores dentro de esto.

La implementación se ha realizado en Python (3.10) con las librerías de Tensorflow y Keras, con implementaciones de numpy para ciertas estructuras. Esto puede verse representado en el capítulo sobre estas tecnologías, donde también se defiende el propósito de estas elecciones en su contexto.

La implementación que hemos hecho y los pasos seguidos para esta se encuentran detallados en el capítulo de Implementación. En este no solo detallamos el proceso iterativo que se ha seguido, con el antecedente del péndulo como entorno de prueba, sino que también exploraremos las estructuras utilizadas y exploraciones del algoritmo de DDPG en pseudocódigo.

Finalmente, se terminará discutiendo la implementación del modelo del brazo manipulador y los resultados hallados. Estos se considerarán tras el análisis de los datos de entrenamiento y explotación obtenidos con diferentes versiones de los modelos. En este se tratarán tanto cambios en la estructura de las redes como en las funciones de recompensa, optimizadores y cambios en los hiperparámetros utilizados.

Como capítulo final se llegarán a unas conclusiones concretas sobre los puntos que se han expuesto. Aquí se verá cuál es el modelo con mejores resultados y se comentarán los resultados de los análisis realizados. También se tratarán y detallarán puntos sobre trabajo futuro que se pueda realizar utilizando como base esta implementación y documento.

Capítulo 4

Aprendizaje por refuerzo

"A algunas personas les preocupa que la inteligencia artificial nos haga sentir inferiores, pero cualquier persona en su sano juicio debería tener un complejo de inferioridad cada vez que mira una flor"
— Alan Kay

4.1. Introduccion del Aprendizaje por Refuerzo

El aprendizaje por refuerzo (AR) es un método de aprendizaje automático [1] cuyo objetivo es el de encontrar un agente que sea capaz de resolver un problema concreto a través de su interacción con un entorno, real o simulado (Sutton y Barto, 2018).

Este se diferencia de otros métodos de aprendizaje automático en su manera de aprender. En otros sistemas es el usuario el que entrega un conjunto de ejemplos para que el sistema pueda aprender (Goodfellow et al., 2016) de una manera previa al entrenamiento u *offline*. Con el AR, el sistema es el que recibe los ejemplos en base a la recompensa que ha obtenido durante su interacción con el entorno. El usuario define como quiere que sea la tarea con el diseño de como es esta recompensa

El modelo no tiene un conjunto de ejemplos de como debe actuar y es mediante la interacción con el entorno que se va modificando su comportamiento. Esto hace que el aprendizaje por interacción con el entorno sea la única manera para que pueda solucionar situaciones inciertas. El AR se basa en 3 principios:

- **El agente:** Este toma decisiones secuenciales para maximizar una señal de recompensa, aprendiendo a través de la experiencia y la retroalimentación del entorno. El proceso de aprendizaje se basa en la exploración de acciones y la recompensa que estas conllevan en el entorno, mejorando así su desempeño mediante la optimización de su comportamiento.

[1] También referido como AA o *machine learning*

- **El entorno:** Es el escenario donde el agente toma decisiones. Este puede ser real o simulado y establece las reglas de la simulación, las acciones disponibles para el agente, y como estas acciones afectan el estado del entorno. Con esto proporciona el marco en el que se desarrolla el proceso de adaptación del agente para lograr sus objetivos.

- **La recompensa:** La recompensa es una señal que actúa como la retroalimentación que recibe el agente del entorno después de tomar una acción en un estado dado. Esta es diseñada por la persona encargada de definir los objetivos del agente y es una medida de cuán favorable fue la acción tomada para completar la tarea dada. La recompensa actúa como un incentivo para que el agente aprenda a través de la experiencia, buscando maximizar la acumulación de recompensas a lo largo del tiempo mediante la selección de acciones que conduzcan a resultados favorables

El funcionamiento del AR trata de una secuencia de decisiones donde este observa su entorno para decidir qué acción realizar teniendo en cuenta una política dada. Al ejecutar una acción, el agente modifica su estado[2] y recibe una recompensa en función de como de correcta ha sido la acción tomada por el agente con el objetivo dado.

La tarea del agente durante estos pasos temporales es maximizar su recompensa. En función de las recompensas que el agente que reciba, este irá aprendiendo a tomar mejores decisiones en base a los resultados de sus acciones anteriores, es decir, el camino. Este camino se refiere al recorrido secuencial a través de los estados del entorno que el agente experimenta.

Un buen diseño de la recompensa es capital para el correcto aprendizaje del agente en la toma de decisiones, debido a que un mal diseño puede llevar a que estas representen tareas adyacentes no deseadas o que lo cumpla de una manera que no sea válida. Un buen diseño de recompensa debe ser capaz de representar un objetivo de manera inequívoca y no poder ser maximizada de realizar acciones en contra de los objetivos.

4.2. Procesos de decisión finitos de Markov: F-MDP

Una parte fundamental y origen del Aprendizaje por refuerzo son los procesos matemáticos conocidos como los Markov decision processes (MDP) estos modelos permiten al modelo tomar decisiones secuencialmente en entornos estocásticos (Sigaud y Buffet, 2013).

En los procesos MDP también se utilizan los conceptos de agente, entorno y recompensa, que son esenciales para entender como funcionan estos procesos y como, esencialmente, son la versión generalizada del tipo de algoritmo que presenta el AR. Aquí se introducen los conceptos del estado y las probabilidades asociadas a las

[2]Conjunto de variables que definen su posición única dentro del entorno

transiciones. Estos elementos juntos han permitido modelar diferentes entornos de manera formal y predecir como las acciones del agente afectan su trayectoria futura.

"*Los MDPs son una versión matemáticamente idealizada de los problemas de aprendizaje por refuerzo a los que se pueden aplicar enunciados teóricos precisos*" Sutton y Barto (2018).

Las interacciones entre el agente y el entorno se hacen en pasos temporales discretos t = 0,1,2... En cada uno de los pasos es el agente el que recibe un conjunto de características propias del entorno, llamadas estado. Con este estado, es el agente el encargado de elegir una acción de las que tenga disponibles para transicionar a uno siguiente. Como consecuencia de la acción, el agente recibe una recompensa r_{t+1} y se mueve al nuevo estado s_{t+1} según las funciones R y ϕ. El proceso se repita hasta que t sea igual a un valor concreto o se cumpla otra condición.

Concretamente nos interesan los MDP finitos o F-MDP, que tienen la particularidad que en estos, todos los conjuntos y la función de recompensa (Que se puede entender también como un conjunto) tienen un número discreto de elementos.

Un MDP se representa típicamente de la siguiente manera:

$$M =< S, A, \phi, R > \tag{4.1}$$

Donde:

- S es el conjunto de estados s en el que se puede encontrar el agente y/o el entorno. Este puede tener estados avanzados en los que lo que cambia no es el estado de las variables sino la política a seguir, es decir, que solo se diferencian en el estado computacional.
- A: es el conjunto de acciones que puede tomar el agente, siendo este el conjunto de unión entre las acciones individuales que puede tomar para cada estado en S.
- ϕ: es la función de transición. $\phi(s'|s, a)$. La función de transición es la probabilidad de alcanzar $s' \in S$ al realizar la acción a en el estado s.
- R: es la función de recompensa que define la meta que debe de conseguir el agente y genera la recompensa de cada decisión $r(s, a)$.

Todas las interacciones del sistema se dividen en episodios. Los episodios son todos los pasos temporales que transcurren en una ejecución completa de la tarea o debido a que se alcanza una condición final de parada. Cada episodio termina en un estado especial llamado estado terminal. Es importante notar también que el comienzo de un episodio es independiente del episodio anterior.

En estos procesos todo lo referente al funcionamiento del aprendizaje se puede resumir en tres señales principales, una que va del agente al mundo (acciones), otra que va del mundo al agente (estado) y otra que define la recompensa para el agente según la situación como se ve en la figura [4.1].

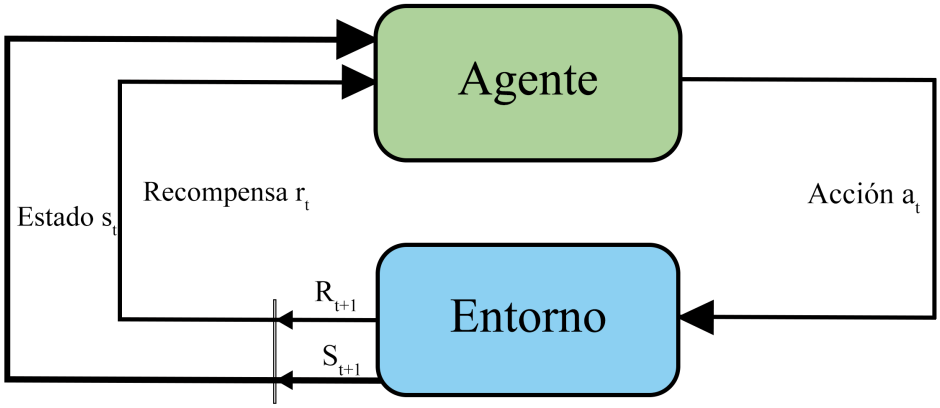

Figura 4.1: Estructura básica de señales en MDP

4.2.1. Objetivos y recompensas

Para realizar un buen entrenamiento se deben diseñar las recompensas para que el agente pueda cumplir con el objetivo de la tarea. Hay muchas maneras de hacer esto para un problema dado, dependiendo del tipo de tarea. Los dos tipos principales de recompensas son las recompensas densas y las esparcidas.

Las recompensas densas son aquellas que se otorgan con frecuencia durante la interacción entre el agente y el entorno, ya sea en un paso temporal o en un alto porcentaje de ellos. Una recompensa esparcida se otorga solo ocasionalmente o al final de largos periodos de interacción

Un ejemplo muy sencillo de recompensa densa seria en el caso de que el actor sea un atleta olímpico y tenga que llegar a un destino, premiando al actor cada vez que se acerque mas al objetivo. En contraposición, una recompensa esparcida le daría una recompensa positiva únicamente cuando este llegue a la meta, si es que lo hace durante el episodio.

Como se ve en Belousov et al. (2021) en su capítulo sobre las funciones recompensa, durante mucho tiempo se han intentado mejorar los algoritmos de AR tomando la función de recompensa como algo dado. Esta consideración era ajena a la decisión de una implementación por parte del usuario. En su conclusión se demuestra que es tan importante la elección de la función de recompensa utilizada como el propio algoritmo elegido.

El último elemento que falta por definir es el factor de descuento. Este elemento se introduce con dos objetivos: asegurar la convergencia en casos en los que podamos tener episodios con infinitos estados; y reducir el impacto de estados muy alejados en el futuro sobre el agente actual. De este modo, este determina cuánto valor se asigna a las recompensas futuras en relación con las recompensas actua-

les. Según este se acerque a cero o a uno respectivamente servirá para modelar su preferencia por las recompensas inmediatas o las futuras.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.2}$$

El factor de descuento, representado como γ determina la importancia de las recompensas futuras en la valoración total tomando valores entre 0 y 1. Un factor de descuento ajustado de acuerdo al objetivo y al entorno puede ayudar al agente a tomar decisiones más efectivas y a desarrollar estrategias que optimicen su desempeño en la tarea.

4.2.2. Políticas, optimalidad y Funciones valor

La política es la manera que tenemos de maximizar la recompensa. Es por esto que es la que define el comportamiento del agente con su entorno, una política se dedica a mapear los estados a las acciones. La política de un modelo se representa con el símbolo (π).

Para poder resolver una tarea, el agente debe encontrar una política que le guíe hacia el objetivo y obtenga una recompensa total máxima. La política que obtiene la máxima recompensa posible se le llama política óptima.

Las funciones valor se utilizan para estimar cuánto valor tiene un estado. Esta función valor intenta estimar cuánta recompensa obtendrá en base a las observaciones y a la experiencia previa (Sutton y Barto, 2018). Es decir, el agente aprende la función valor y la usa para estimar la recompensa futura de cada acción.

Existen dos tipos principales de funciones valor:

- Función valor de un Estado ($V(s)$):
  Estima el valor de un estado s e indica la recompensa acumulada que espera el agente desde el estado s siguiendo la política que tiene.
- Función valor de una Acción ($Q(s, a)$):
  Estima el valor de una acción a en un estado s. Indica la recompensa acumulada que espera el agente después de tomar la acción a en el estado s.

Para aprender las funciones valor, se utilizan principalmente los algoritmos Q-Learning y SARSA. Estos métodos se explicarán posteriormente, ya que antes se ha de comprender la programación dinámica, de la que originan.

4.3. Programación dinámica

La Programación Dinámica (PD) es un enfoque computacional utilizado para resolver problemas de optimización y toma de decisiones en situaciones donde las

decisiones se toman de manera secuencial y las soluciones óptimas exhiben una estructura de subproblemas superpuestos. La solución de cada problema radica en las soluciones de diferentes subproblemas.

Su objetivo principal es encontrar la política óptima, es decir, la secuencia de decisiones que maximiza (o minimiza) la medida de rendimiento deseada.

Normalmente este acercamiento se aplica a un F-MDP, pero algunas ideas de PD se pueden llegar a aplicar a problemas en espacios continuos realizando ciertas modificaciones.

La manera más sencilla de organizar soluciones a estas tareas es cuantificar los estados y las acciones y entonces convertirlo en un F-MDP al limitar estos conjuntos a conjuntos finitos. Esto se basa en discretizar los valores de las acciones y el entorno, volviendo un MPD en un F-MPD con unos cuantos (mínima división de una variable) definidos.

La idea principal de PD es el usar funciones valor para organizar la estructura y búsqueda de buenas políticas. Debido a la cuantificación de todos los estados y acciones, la PD es un método muy costoso si queremos tener una buena precisión en entornos continuos, por lo que su uso es recomendable en entornos pequeños y discretos

Dentro de la PD, la evaluación de políticas calcula el valor de cada estado bajo la política a evaluar iterando hasta la convergencia. Esto se realiza mediante la ecuación de Bellman actualizando las estimaciones de los valores hasta que converjan a los valores verdaderos.

$$V_\pi = \sum_a \pi(a|s) \sum_{s+1,r} p(s+1, r|s, a)[r + \gamma \upsilon_\pi(s+1)] \tag{4.3}$$

4.3.1. Iteración de políticas y valores

La iteración de políticas consiste en calcular la función valor de seguir una política particular en todos los estados del entorno y mejorarla al actualizarla seleccionando, en cada estado, la acción que maximiza el valor esperado. Con este método se busca la convergencia hacia la función verdadera, lo que puede ser un proceso muy largo porque requiere determinar la función de valor para la política en cada iteración. Esto puede tener un gran coste computacional según el caso.

A veces, no es necesario esperar a una convergencia exacta, y podemos acortar, o truncar, este proceso. Un enfoque eficiente es la iteración de valores, que conlleva detener la evaluación de la política después de un solo barrido por los estados. La evaluación de la política es el proceso de determinar la función de valor para la política dada. Este método se enfoca en actualizar iterativamente la función de valor de cada estado hasta que converge a la función de valor óptima en un solo paso. Esto

aún garantiza que vaya a converger y es relativamente más rápido que la iteración de políticas completa. Este método se representa de la siguiente manera:

$$v_{k+1}(s) = max_a E[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a] \tag{4.4}$$

$$= max_a \sum_{s_t+1,r} p(s_{t+1}, r|s, a)[r + \gamma v_k(s_{t+1})] \tag{4.5}$$

La primera ecuación representa la actualización de la función valor para un estado s en la iteración $k+1$. La función valor $v_{k+1}(s)$ se actualiza tomando la máxima recompensa sobre todas las posibles acciones a que pueden tomarse en el estado s, donde $E[\cdot]$ denota el valor esperado. Esto significa que estamos considerando la recompensa inmediata R_{t+1} más el valor descontado del siguiente estado $v_k(S_{t+1})$ ponderado por la probabilidad de transición $p(s+1, r|s, a)$.

La segunda ecuación termina por darnos la noción de que la recompensa final de un estado es la suma de las recompensas de todos los posibles estados a los que se puede llegar ponderadas por las probabilidades de llegar a cada estado con la acción elegida.

Por ejemplo, si hay dos estados posibles $E1(30\,\%, r = 10)$ y $E2(70\,\%, r = 100)$, la recompensa total será $R_{total} = 10 \times 0{,}3 + 100 \times 0{,}7 = 73$.

4.4. Aprendizaje por diferencia temporal: TDL

Una de las mayores novedades dentro del AR es el aprendizaje por diferencia temporal (TDL). El TDL consiste en la combinación entre la programación dinámica y los métodos de Monte Carlo donde el objetivo principal es encontrar la política óptima. Los métodos de Monte Carlo son una técnica de simulación que utiliza muestreo aleatorio repetido para estimar propiedades de sistemas complejos o estocásticos. Un método de TDL aprende directamente en base de su experiencia en el entorno (Monte Carlo) y se actualiza mediante estimaciones ya aprendidas (Programación Dinámica) (Sutton y Barto, 2018). Este método se registra cada vez que se visita un estado sin importar cuántas veces ocurra, se calcula el retorno promedio de todas las visitas a ese estado y se utiliza ese valor promedio para estimar su valor.

A esta estimación se le conoce también como *Bootstraping* y se suele utilizar para espacios muy grandes. Debido a que calcular todas los pasos de entrenamiento para todos los estados es computacionalmente caro, o incluso prohibitivo, se actualizan los valores en los estados que hayan sido explorados. Esto es una gran diferencia, ya que con otros métodos para poder actualizar las estimaciones se tenia que explorar todo el entorno y terminar el episodio, mientras que ahora los valores se van actualizando gradualmente.

Normalmente se utiliza la fórmula siguiente para actualizar la función valor en su versión más sentilla, $TD(0)$ (Sutton, 1988):

$$V_{St} = V_{St} + \alpha[R_{t+1} + \gamma V_{s+1} - V_s] \tag{4.6}$$

Donde el valor de un estado V_{St} se actualiza en función en función del valor del siguiente V_{St+1} y la recompensa asignada a ese estado (Suponiendo que el siguiente siempre es el que conduce la acción elegida para el estado actual).

Existen dos formas principales de implementar el aprendizaje por diferencia temporal. La primera es con el método SARSA (State-Action-Reward-State-Action) que es un método *on-policy*. La segunda es con Q-learning que es un método *off-policy*. Ambos tienen como objetivo encontrar la función valor, normalmente referida como Q, de cada pareja de estado y acción. Aquí el valor se almacena en la función valor, que puede tomar varias formas, desde una matriz a redes neuronales [6.3]. Inicialmente, en toda la representación los valores están vacíos debido al desconocimiento del entorno, siendo la fase del entrenamiento la que provocan que esta converja hacia sus valores reales.

4.4.1. Sarsa: TDL On-policy

Sarsa es un método TDL que aprende y mejora la política mientras que explora el entorno utilizando esta misma política. Es esto lo que define a Sarsa como un método *on-policy*.

En Sarsa necesitamos tratar con el concepto de la función Q, que también se ha reconocido como la función de valor de acción, la cual representa el valor esperado acumulado de tomar una acción específica en un estado particular y luego seguir una política específica para elegir acciones futuras. Formalmente, la función $Q(s, a)$ denota el valor esperado esperado de tomar la acción a desde el estado s.

Sarsa sigue la siguiente ecuación para actualizar su política a través del entrenamiento:

$$Q(S_t, A_t) \leftarrow (1 - \alpha) * Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4.7}$$

Sarsa utiliza una política ε-greedy, que va oscilando entre exploración y explotación al funcionar de la manera siguiente:

- Usando la política vigente, tomamos el mayor valor de Q(s,a), con una probabilidad de 1-ε
- Con la probabilidad de ε tomamos una acción aleatoria con la finalidad de explorar el sistema.

La política ε-greedy permite al modelo explorar el sistema y al mismo tiempo poder explotar las acciones mas valiosas que ha recopilado. Esta política de exploración forma parte del proceso de aprendizaje. Esto hace que el modelo converja al valor óptimo de la política final, que utiliza una explotación greedy completa.

4.5. Q-Learning: TDL Off-policy

Q-Learning es el método de aprendizaje TDL *off-policy* más utilizado. El funcionamiento del Q-Learning, a diferencia de la actualización tras elegir una acción en Sarsa, se basa en actualizar los valores de Q según la elección óptima en cada estado, siguiendo luego la política de exploración que se haya decidido de manera independiente (Sutton y Barto, 2018). Además, en este se utiliza el factor de aprendizaje (α) para no sobrescribir directamente los nuevos valores Q con los viejos.

El algoritmo sigue la siguiente ecuación:

$$Q(S_t, A_t) \leftarrow (1 - \alpha) * Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (4.8)$$

A Diferencia del anterior algoritmo de aprendizaje SARSA, Q-Learning es un algoritmo *off-policy* donde tiene dos políticas diferentes. Una para la exploración y otra para la explotación.

En la fase de exploración utiliza una política ε-greedy para conocer nuevos estados y así actualizar los valores Q de una manera balanceada, buscando el camino óptimo pero sin caer en máximos locales.

4.5.1. Double Q-learning y sesgo de maximización

Todos los algoritmos que se han visto hasta este punto utilizan la maximización para construir sus políticas objetivo (en Q-learning la política objetivo es la política greedy dado los valores actuales, lo que se traduce en el máximo, y en Sarsa se utiliza la política ϵ-greedy). En estos algoritmos se utilizan los máximos entre los estimados lo que puede llevar a un sesgo positivo.

Debido a las incertidumbres introducidas por las estimaciones, es posible que algunos pares de estado-acción tengan valores sobrestimados en comparación con otros que revisamos repetidamente. Esto puede llevar a que ciertos valores que son los verdaderos máximos no se exploren adecuadamente. A este fenómeno se conoce como sesgo de maximización (Hasselt, 2010).

El aprendizaje doble surge para evitar el sesgo de maximización en las estimaciones de valor utilizando dos funciones Q. La idea es que, en lugar de usar una sola función Q para determinar y estimar el valor de las acciones, se usan dos funciones Q separadas. En cada paso, una de las funciones Q se utiliza para determinar la acción que maximiza su valor, mientras que la otra función Q se usa para obtener la estimación del valor correspondiente (Sutton y Barto, 2018). Alternando entre estas dos funciones, se reduce el riesgo de sobrestimar los valores, siendo que en cada paso solo se actualiza el valor de una. Esta doble actualización evita el sesgo de maximización al irse alternando los roles de estimaciones.

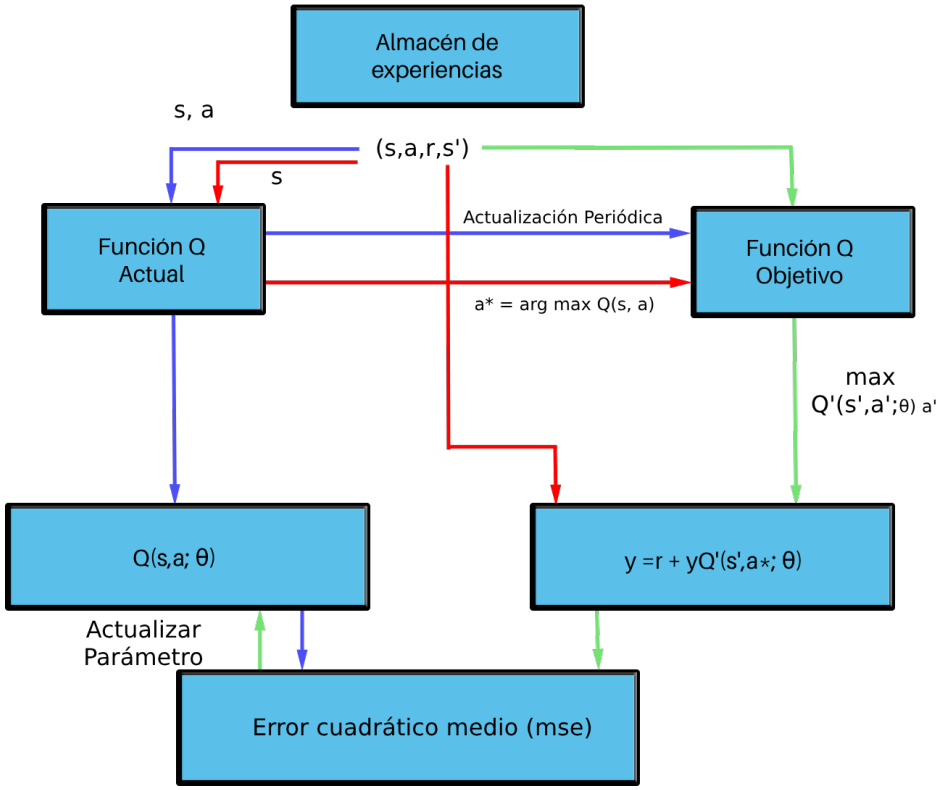

Figura 4.2: Diagrama estructural de doble Q-Learning

Esta doble actualización evita el sesgo de maximización al irse alternando los roles de los estimadores. Este proceso de alternancia continua reduce la probabilidad de que las estimaciones de valor se inflen indebidamente, mejorando la estabilidad y precisión del aprendizaje del agente. Esto termina requiriendo el doble de memoria para ejecutarse, aunque el coste computacional no cambie de orden de complejidad.

4.6. Aplicaciones del aprendizaje por refuerzo.

El AR se ha utilizado para muchas aplicaciones debido a que es una herramienta versátil y útil para resolver cierto tipo de problemas, desde la robótica (Hussonnois y Jun, 2022; Li et al., 2023) hasta videojuegos, la gestión de recursos (Perera y Kamalaruban, 2021) y la resolución de problemas algorítmicamente complejos.

En su variante discretizada, es decir, utilizando métodos TDL *on* u *off-policy* se ha utilizado con éxito para resolver problemas informáticos NP-Hard (Ardon, 2022) y siendo combinado con técnicas de Deep Learning [5] para eliminar el sesgo de exposición con modelos de resumen de lenguajes abstractos, dando mejores resultados que solo con Deep Learning(Paulus et al., 2017).

En su variante más popular en los últimos años, el Deep Reinforcement Learning , que combina conceptos del Deep Learning con el AR como en el Deep Q-Learning [6.3] ha habido una mayor investigación, logrando aplicaciones en la detección de brechas de seguridad en redes (Alavizadeh et al., 2022) y en la reducción del consumo de energía de dispositivos integrados respecto a otras planificaciones de uso (Zhang et al., 2019).

4.7. Puntos clave sobre aprendizaje por refuerzo.

El aprendizaje por refuerzo (AR) conforma una parte del aprendizaje automático en el que un agente aprende a tomar decisiones secuenciales para maximizar una recompensa acumulada a lo largo del tiempo. En este marco, el agente va interactuando con un entorno, percibiendo estados y tomando acciones. Estas acciones a cambio influyen en estado del entorno, generando observaciones nuevas y recompensas que van guiando el aprendizaje del agente.

Un estado (s) representa la situación actual que se tiene, proporcionando al agente la información del entorno necesaria para tomar decisiones. Una acción (a) es una decisión, movimiento o proceso de cómputo que el agente puede realizar en un estado dado. La recompensa (r) es un valor que recibe el agente como retroalimentación inmediata después de tomar una acción, y que indica el valor que tiene en el contexto del objetivo establecido. La interacción entre estos elementos se formaliza mediante los Procesos de Decisión de Markov (MDP), que describen la dinámica del entorno a través de estados, acciones, probabilidades de transición y recompensas.

Para resolver ciertos tipos de problemas en los que no disponemos de ejemplos directos y que se pueden configurar como MDP es muy interesante utilizar técnicas que exploten una política construida. Esta política óptima deberá ser capaz de, con su secuencia de acciones, completar el objetivo dado en el entorno con la mayor recompensa acumulada posible.

Dentro de los algoritmos de AR, la programación dinámica (PD) es una técnica utilizada para encontrar la política óptima para entornos no muy grandes. La PD se basa en la idea de descomponer un problema más grande o complejo en subproblemas más pequeños y resolverlos de manera iterativa.

Dos enfoques que se pueden considerar principales de la PD son la evaluación de políticas y la mejora de políticas, que colaboran para decidir qué política maximiza la recompensa esperada. La iteración de valores es una técnica eficiente dentro de la PD que actualiza iterativamente la función de valor de cada estado hasta converger a la función de valor óptima.

Además de la PD, el aprendizaje por diferencia temporal (TDL) es otra técnica importante en AR. TDL combina elementos de la PD con métodos de Monte Carlo

para aprender directamente de la experiencia del agente. Algoritmos populares de TDL incluyen SARSA y Q-Learning, que difieren en su enfoque hacia la política (*on-policy* vs *off-policy*) y como actualizan los valores de acción-valor.

Estos métodos y sus expansiones y variaciones han llevado a volverse populares por su posibilidad de implementación en muchos entornos, como juegos, robots autónomos y sistemas de recomendación. Todas estas características nos llevan a utilizar sus versiones más avanzadas para crear un controlador para los dos entornos que vamos a explorar.

Capítulo 5

Deep Learning

"Inteligencia artificial, aprendizaje profundo, aprendizaje automático... te dediques a lo que te dediques, si no lo comprendes tienes que ponerte con ello y aprender qué es. Porque de lo contrario serás un dinosaurio dentro de 3 años."
— Mark Cuban

Lo que se conoce ya comúnmente como Deep Learning no es más que un subgénero dentro del aprendizaje automático (Goodfellow et al., 2016), que a su vez es una de las ramas más conocidas y utilizadas de la Inteligencia Artificial junto al AR.

Un algoritmo de aprendizaje automático es un algoritmo que es capaz de aprender de datos (Goodfellow et al., 2016) y que tiene como objetivo adaptarse a nuevas circunstancias mediante la detección y extrapolación de patrones (Russell y Norvig, 2016). Lo que tienen en común tanto el Deep Learning como el AR es la existencia de hiperparámetros que configurar para mejorar su funcionamiento al prepararlos para la tarea a solucionar.

Esta definición sobre qué es aprender de datos se puede explicar de manera sencilla con la siguiente frase *"Un programa informático se dice que aprende de experiencia E con respecto a unas tareas T y medida de su rendimiento R, si su rendimiento en las tareas T, medido en R, mejora con la experiencia E"* como explica Mitchell (1997). Esto está mejor definido y es más evidente en la sección sobre el AR [4] en la que se trata qué forma toman estos factores de experiencia, tareas y rendimiento.

Lo que diferencia al Deep Learning de otras formas de aprendizaje automático es la manera que tiene este de modelar abstracciones de alto nivel en datos. Esto lo logra mediante transformaciones no lineales múltiples e iterativas en arquitecturas computacionales que lo admitan (Bengio et al., 2012).

Esto se hace mediante Redes Neuronales Artificiales que pueden tomar diversas formas y tamaños, desde simples perceptrones hasta redes convolucionales o recurrentes. En esencia, estas redes están compuestas por capas de nodos interconecta-

dos, donde cada nodo realiza operaciones matemáticas simples, como sumar entradas ponderadas por unos pesos y aplicar una función de activación. Una función de activación es una función, normalmente no lineal, que existe a nivel de cada capa y se utiliza para modificar la forma de la salida de esta.

El proceso de aprendizaje en Deep Learning implica ajustar los pesos de estas conexiones entre nodos para que la red pueda aprender automáticamente características relevantes de los datos de entrada. Esto se logra mediante el uso de algoritmos de optimización, como el descenso de gradiente o su versión estocástica. Estos algoritmos que ajustan gradualmente los pesos de la red en función de la discrepancia entre las predicciones del modelo y los valores reales.

Estas capacidades de aprendizaje del Deep Learning lo han convertido en una herramienta muy relevante en el mundo actual en una amplia gama de aplicaciones, desde la visión artificial (Shanmugamani et al., 2018) hasta la medicina y robótica. Su capacidad para modelar sistemas de datos complejos y para extraer patrones se han considerado ideales para generar avances significativos en estos campos, siendo que su popularidad y relevancia continúan creciendo rápidamente.

5.1. Arquitectura de Redes Neuronales Artificiales

Las redes neuronales son modelos computacionales inspirados en el funcionamiento del cerebro humano, capaces de aprender y realizar tareas complejas a partir de una experiencia proporcionada. Sin embargo, la diversidad de estructuras y tipos de redes neuronales pueden hacer que su estudio y aplicación sean más complicados de lo necesario.

Una neurona es una unidad que imita el funcionamiento de una neurona biológica, recibiendo unas entradas que transforma y proporcionando una salida que puede ir a otras neuronas de la capa siguiente. La arquitectura de una red se refiere a la forma que esta tiene para tratar los datos de entrada, pasando por transformaciones no lineales que ejecutan las funciones de activación en cada capa. Una capa es un conjunto de neuronas que se encuentran conectadas a la capa anterior y la siguiente pero normalmente no entre ellas, generando pasillos por donde pasan los datos con ciertas transformaciones, como se explicará más adelante.

La elección de la arquitectura adecuada depende del tipo de datos, la naturaleza del problema y los recursos computacionales disponibles. Por lo tanto, es necesario entender como se aplican en diferentes contextos para desarrollar soluciones eficaces en cualquier área. Con esto y nuestros objetivos de implementación en mente, exploraremos en detalle las principales arquitecturas de redes neuronales, sus características distintivas y sus aplicaciones específicas, con el objetivo de proporcionar una visión amplia y clara que preceda a nuestra implementación.

5.1.1. Capas en redes neuronales

Las redes neuronales están compuestas por capas de neuronas interconectadas, cada una de las cuales realiza ciertas operaciones a sus entradas, normalmente con una ponderación por pesos. Las capas se organizan en una estructura jerárquica, con una capa de entrada, una o más capas ocultas y una capa de salida. La capa de entrada recibe los datos de entrada, las capas ocultas realizan transformaciones, lineales o no lineales, en los datos, y la capa de salida produce las predicciones o resultados finales.

Esto no quiere decir que la capa de entrada y salida no realicen transformaciones, pero pueden quedar exentas de su uso, sobre todo la final según lo que se busque de la red.

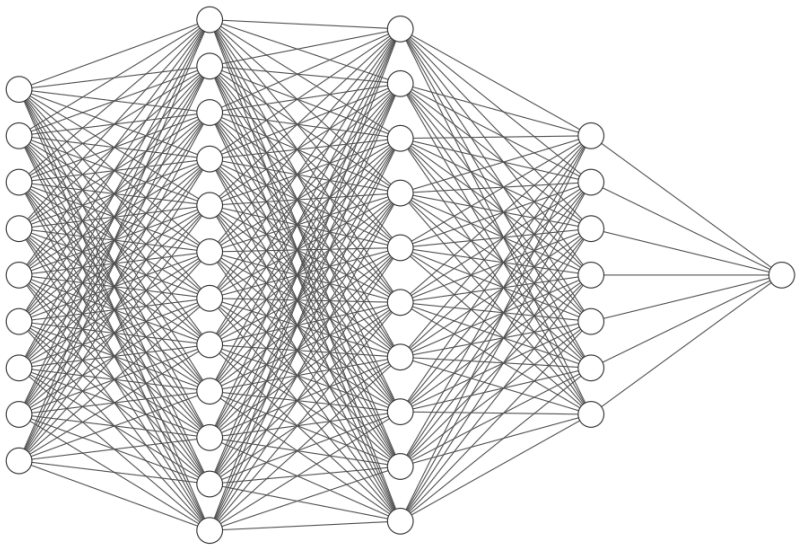

Figura 5.1: Red neuronal con capas conectadas de diferente número de neuronas.

En cuanto a elegir el número de capas ocultas en una red neuronal, hay que tener en cuenta la dificultad del problema y la cantidad de información necesaria para resolverlo. Para problemas simples con datos estructurados, como clasificación binaria de imágenes, una sola capa oculta puede captar características importantes. En cambio, muchas capas ocultas pueden ser necesarias para sobrepasar representaciones de alto nivel en casos tales como detección de objetos y traducción automática.

El número de capas y arquitecturas se pueden buscar realizando estimaciones de la complejidad del problema y otros factores como el modelo a codificar, pero existen métodos desarrollados más informados y por lo tanto útiles para esta tarea (Panchal et al., 2011).

El Criterio de Información de Akaike (AIC) es uno de los métodos más utilizados en este contexto. Fue desarrollado por Hirotugu Akaike en Akiake (1974). El AIC

es una medida que castiga la complejidad del modelo para evitar, principalmente, el sobreajuste.

En el contexto de las redes neuronales, el AIC puede emplearse para comparar varias arquitecturas y determinar cuántas capas ocultas son necesarias para explicar adecuadamente los datos sin incluir complejidad innecesaria.

$$AIC = 2k - 2\ln(\hat{L}) \tag{5.1}$$

Siendo k el número de parámetros libres en el modelo y $\hat{L}$ la función de verosimilitud del modelo ajustado, es decir, la medida de qué tan probable es observar los datos que tienes dado un modelo estadístico específico y sus parámetros.

Cuanto menor sea el valor de AIC, mejor se ajustará un modelo a los datos respecto a su complejidad relativa. De este modo, se puede utilizar el AIC como guía a la hora de seleccionar el número de neuronas por capa de red neuronal que mejor se ajuste a los datos disponibles evitando el sobre y el infraajuste.

Este no es el único método, pero tal y como se define en Panchal et al. (2011) en lo que esto se resume es que este proceso de elección "*... si la precisión del resultado es un factor crítico para una implementación, se deben utilizar más capas ocultas, pero si el tiempo es un factor importante, se debe utilizar una sola capa oculta.*"

5.1.2. Arquitecturas generales

A lo largo de los años numerosas arquitecturas han sido desarrolladas con características distintivas y aplicaciones específicas (Wilamowski, 2009), diseñadas para afrontar desafíos particulares en diferentes dominios.

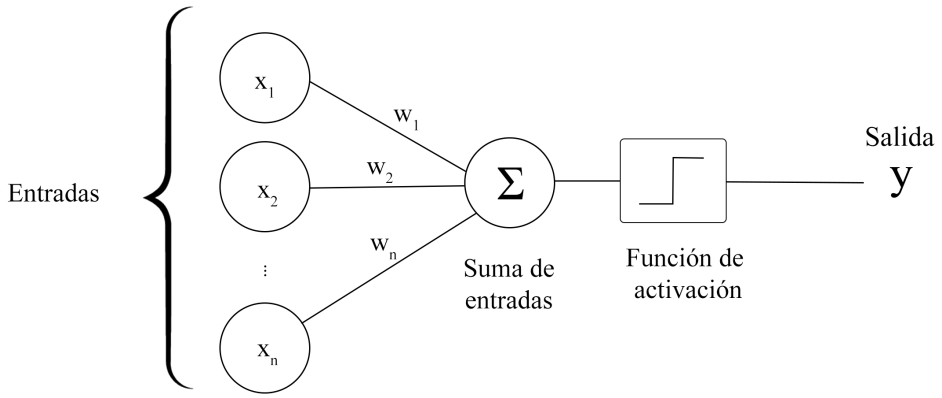

Figura 5.2: Estructura básica del perceptron.

En este contexto, el perceptrón [5.2] es, posiblemente, el ejemplo más elemental, ya que representa la forma más simple de red neuronal. Originalmente propuesto

por Rosenblatt (1957), el perceptrón es una unidad de procesamiento sencilla que toma un conjunto de entradas, las pondera por medio de pesos asociados y produce una salida binaria que normalmente clasifica los datos según el objetivo. Esta salida suele estar basada en una función de activación, normalmente de paso o *step*. La función de *step* funciona de tal manera que si la suma ponderada por sus pesos correspondientes de las entradas supera un cierto umbral, su salida se activa. En caso contrario, no se activa.

Otra arquitectura prominente es la de las redes convolucionales [5.3][1] (CNN), diseñada específicamente para trabajar con datos estructurados en forma de matrices multidimensionales, como imágenes o datos espaciales complejos. Las CNN suelen extraer mejor las características espaciales usando filtros convolucionales (Goodfellow et al., 2016) que se van pasando secuencialmente. Esto las convierte en especialmente interesantes para tareas visuales como la visión artificial, segmentación semántica y reconocimiento facial.

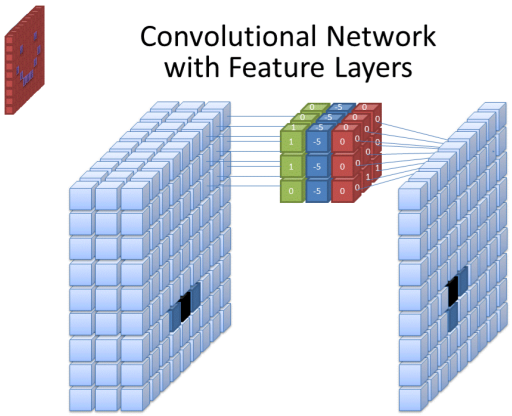

Figura 5.3: Ejemplo de red convolucional de Cecbur(2019)

Por otro lado, la arquitectura recurrente (RNR) está particularmente construida para modelar datos secuenciales donde la información depende del contexto temporal. Estas redes están especialmente preparadas para modelar datos secuenciales (Medsker et al., 2001), pudiendo recordar estados anteriores y procesar entradas de longitud arbitraria. Cada neurona en la red tiene conexiones con los nodos en la capa siguiente, pero también recibe conexiones de los nodos en la misma capa en el instante de tiempo anterior. Esto significa que cada nodo tiene en cuenta cierta información sobre el pasado mientras procesa datos en el presente. En la actualidad, estas arquitecturas encuentran aplicaciones principalmente en la IA generativa y en realidad virtual y aumentada.

Las redes neuronales generativas (GNN) destacan por la capacidad que tienen para generar nuevos datos realistas, en lugar de realizar actividades de regresión o

[1]CC BY-SA 4.0 `https://creativecommons.org/licenses/by-sa/4.0`, via Wikimedia Commons

clasificación. Los ejemplos más utilizados de este tipo de redes incluyen las Redes Generativas Adversariales (GAN), donde dos redes compiten entre sí. Una de estas redes será la generadora que produce datos falsos, mientras que otra red discriminadora intenta distinguir entre datos reales y generados por la otra red para devolverle el rendimiento obtenido a la red generadora (Creswell et al., 2018). Estos entrenamientos se realizan de manera paralela, mejorando las falsificaciones mientras que la red discriminadora aprende, cada vez de mejor manera, a distinguirlos. Este tipo de arquitecturas donde se enfrentan redes con objetivos distintos han mostrado una versatilidad notable en diversas aplicaciones, entre ellas para la clasificación, síntesis, edición semántica y superresolución de imágenes.

En resumen, la elección de la arquitectura de red adecuada depende en gran medida de la naturaleza de los datos y la tarea específica que se desea abordar. Hay tareas para que las convolucionales pueden tener un desarrollo sorprendente por como se presentan los datos, mientras que en los problemas en los que la temporalidad es relevante estas podrían llegar a ser bastante subóptimas. Es por esto que se vuelve relevante conocer cuales son los puntos débiles y fuertes de cada tipo de arquitectura, con el objetivo de así no malgastar los recursos que tenemos en estructuras subóptimas que no aprovechan las capacidades concretas de sus arquitecturas.

5.2. Entrenamiento y algoritmos de optimización

En el Deep Learning, los algoritmos de optimización y entrenamiento son áreas fundamentales a explorar si queremos sacar el mayor provecho posible de las redes neuronales profundas. En esta sección, exploraremos los conceptos clave detrás del proceso de entrenamiento de una red neuronal y los algoritmos utilizados para optimizar su rendimiento.

El proceso de entrenamiento de una red neuronal implica la ajuste de los pesos de la red para minimizar una función de pérdida, que esencialmente mide el error entre las predicciones del modelo y los valores con los que se etiquetaron los datos de entrenamiento. Este proceso se lleva a cabo utilizando algoritmos de optimización que ajustan los pesos de la red en función del gradiente de la función de pérdida.

El algoritmo de optimización más básico y fundamental es el descenso de gradiente, que se comentará más adelante y es en el que se basan, en mayor o menor medida, otros que se mencionarán y serán utilizados. A lo largo de esta sección, exploraremos como funciona el proceso de entrenamiento de una red neuronal y de qué manera se utilizan los algoritmos de optimización para mejorar su rendimiento.

5.2.1. Proceso de entrenamiento de una red neuronal

El entrenamiento de una red neuronal comienza por asignar pesos a la red, que se hacen en general siguiendo una determinada distribución aleatoria, como la distribución normal o uniforme (Thimm y Fiesler, 1995). Una vez que se inicializan los

pesos, los datos de entrada se propagan a través de la red en un proceso conocido como propagación hacia adelante. Durante esta fase, cada neurona en cada capa hace una combinación lineal de las entradas y les aplica su función de activación. Esto se repite en todas las capas hasta que se obtenga la salida final desde la red.

Después del cálculo del resultado obtenido, sigue el cálculo de una función denominada pérdida o de coste; esta indica como de cerca llegó la salida generada por la red a etiquetas o soluciones verdaderas dentro del conjunto original de datos. Un ejemplo clásico para problemas de clasificación binaria o multiclase es el uso de la función normal de entropía cruzada:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \tag{5.2}$$

La elección del tipo de función depende del problema específico bajo consideración pero usualmente para problemas de regresión se utiliza una función de pérdida cuadrática y de entropía cruzada para problemas de clasificación. Esencialmente, el objetivo fundamental de estas es medir la cantidad en que esta salida difiere del valor real. Una vez que se ha calculado la función de pérdida, se utiliza el algoritmo de backpropagation para calcular el gradiente de la función de pérdida con respecto a los pesos de la red.

Backpropagation utiliza la regla de la cadena, que implica descomponer el proceso en una serie de derivadas parciales que se multiplican entre sí a lo largo de las capas de la red. El objetivo de este algoritmo es propagar el error desde la capa de salida hasta las capas ocultas de la red (Rojas, 1996). Gracias a esta técnica se calcula el gradiente de la función de pérdida con respecto a los pesos de la red. Este gradiente se utiliza entonces para ajustar los pesos en la dirección que minimiza la función de pérdida, utilizando un algoritmo de optimización como el descenso de gradiente o los que se verán más adelante.

Con el gradiente calculado y una vez se han actualizado los pesos de la red, se repite el proceso con un nuevo lote de datos de entrenamiento. Este proceso se repite iterativamente durante un número determinado de episodios o hasta que se alcance algún otro criterio de convergencia, como la estabilización de la función de pérdida en un valor mínimo, o que se complete un objetivo dado.

Al final del proceso de entrenamiento, la red neuronal deberá haber ajustado sus pesos de manera que la función de pérdida sea mínima en el conjunto de datos de entrenamiento, lo que debería permitirle generalizar bien a nuevos datos si no ha caído en un sobreajuste.

5.2.2. Regularización y optimización de modelos

La regularización en los modelos de redes neuronales es una técnica utilizada para prevenir el sobreajuste, ya que esto puede llevar a un desarrollo deficiente en

datos que se salgan de este entrenamiento. Esta regularización implica añadir términos adicionales a la función de pérdida durante el entrenamiento del modelo para penalizar pesos muy grandes o complejos (Girosi et al., 1995).

Hay varias maneras de aplicar esta regularización, entre las que se verán las más usuales para luego explicar la aplicada dentro de la implementación elegida:

- **Regularización L1:** La regularización L1 añade el término L1 de regularización a la función de pérdida, el cual es proporcional a la suma de los valores absolutos de los pesos del modelo. A efectos prácticos esto termina provocando esparcidad en los pesos del modelo, lo que conllevará que algunos, si no muchos pesos, serán cero, lo que ayudará a simplificar el modelo y evitar el sobreajuste de la red.
- **Regularización L2:** De manera parecida a L1 este método añade el término L2 de regularización a la función de pérdida, el cual es la suma del cuadrado los valores de los pesos del modelo. Esto pone mayores penalizaciones a valores grandes del modelo respecto a los pequeños, resultando en distribuciones más suaves y modelos generalmente más simples que también puede ayudar a evitar el overfitting.
- **Regularización de Dropout:** Esta regularización es posiblemente la más sencilla de implementar. Se basa en que durante el entrenamiento de la red neuronal, aleatoriamente, se apagan un cierto porcentaje de neuronas en cada iteración. Esto ayuda a prevenir el sobreajuste al forzar a la red a aprender representaciones más robustas y redundantes de los datos.
- **Regularización por Early Stopping:** Esta regularización es la más simple, ya que consiste en detener el proceso de entrenamiento cuando el rendimiento del modelo comienza a empeorar para el conjunto de datos de validación. Este comportamiento indica que el modelo está sobreajustando hacia los datos de entrenamiento. Es importante mencionar que el conjunto de validación es un conjunto de datos independiente de los datos de entrenamiento y que no suele compartir ejemplos con este.

Como veremos más adelante con los optimizadores, la regularización en modelos de redes neuronales es muy importante. La regularización principalmente se utiliza para mejorar la capacidad de generalización del modelo y su capacidad para hacer predicciones precisas en datos que se escapen de la tipicidad de los datos de entrenamiento. Esta regularización se vuelve especialmente relevante si queremos asegurar que vaya a tener un comportamiento fiable en implementaciones reales, donde existirán ejemplos a los que tener que extrapolar con ruido y otros factores que nos lo obstaculicen.

La relevancia de estos tipos de regularización radica en su capacidad para mitigar el sobreajuste hacia los datos de entrenamiento. Este siempre ha sido uno de los desafíos más importantes en el desarrollo de modelos de aprendizaje automático y

redes neuronales en particular. Al controlar la complejidad del modelo y promover la simplicidad, la regularización ayuda a garantizar que el modelo pueda generalizar bien a datos nuevos y no vistos. Esto suele mejorar de manera notoria su utilidad y aplicabilidad en situaciones del mundo real con ruido y datos considerados como *outliers* o partes aisladas.

5.2.3. Algoritmos de optimización

El proceso de aprendizaje de una red neuronal funciona a través de variaciones en los pesos de la red según como evoluciona la función de pérdida. En el entrenamiento de una red neuronal, el algoritmo de optimización de la función de pérdida es tan importante como la elección de esta función. Estos algoritmos determinan como la red ajusta el peso de sus neuronas en respuesta al gradiente o pendiente de la función objetivo que presentan los ejemplos.

Uno de los algoritmos que más se utilizan y básicos dentro de los algoritmos de optimización es el descenso de gradiente. Este método ajusta los pesos del modelo en contra del gradiente multiplicado por un ratio de aprendizaje para que no cambie de manera drástica en cada paso temporal. Si el gradiente indica que la función de pérdida está disminuyendo más rápidamente en una dirección particular, el algoritmo de descenso de gradiente moverá los pesos ligeramente en la dirección opuesta a ese gradiente para minimizar la pérdida con cada paso. El pseudocódigo correspondiente a este algoritmo sería el siguiente:

Algorithm 1 Descenso de Gradiente

```
Inicializar los pesos de la red neuronal θ
Inicializar la tasa de aprendizaje α
Inicializar el número de episodios N
for i = 1 hasta N do
    Cálculo de la función de pérdida J(θ) utilizando los datos de entrenamiento
    Cálculo del gradiente de la función de pérdida: ∇_θ J(θ)
    Actualizar los pesos de la red: θ = θ − α · ∇_θ J(θ)
end for
```

No obstante, el descenso del gradiente, sin aplicar ninguna mejora adicional, puede ser muy lento y quedarse atascado fácilmente en mínimos locales para aplicaciones complejas. Esto ha llevado al desarrollo de diversos métodos más avanzados de optimización que buscan mejorar la eficiencia y velocidad para llegar a la solución óptima durante el proceso de aprendizaje como se ve en *Enhancing Performance of a Deep Neural Network by Comparing Optimizers Experimentally* en Noor (2020).

Algunos ejemplos incluyen:

- **Descenso de gradiente estocástico (SGD):** Este funciona igual que el descenso de gradiente clásico para cada mini-lote o fracción del espacio de

transiciones explorado. Este, en lugar de calcular el gradiente sobre todo el conjunto a la vez, utiliza estos mini-lotes acelera el proceso de entrenamiento y ayudar a evitar mínimos locales al no tratar todo el espacio a la vez.

- **Root Mean Square Propagation (RMSprop):** Este es otro algoritmo de optimización que en este caso se aprovecha de técnicas adaptativas. Esto lo hace mediante el ajuste de la tasa de aprendizaje de forma individual para cada parámetro de la red en vez de para esta como conjunto. Para realizar esto, este normaliza la tasa de aprendizaje para cada parámetro al utilizar una estimación del segundo momento del gradiente, así este obtiene mejores resultados que el descenso de gradiente simple, siendo más dinámico y flexible.

- **Adaptive Gradient Algorithm (Adagrad):** Propuesto en Duchi et al. (2011), este método es bastante similar al descenso de gradiente estocástico. Este presenta el cambio de utilizar gradientes adaptativos para mejorar la robustez a diferentes ratios de aprendizaje, tal y como explica Noor (2020) "*[... su objetivo] es obliterar el ajuste manual de su tasa de aprendizaje, mientras que su defecto más significativo es la agregación de gradientes al cuadrado en el denominador.*"

- **Adaptive Moment Estimation (Adam):** Adam es un algoritmo de optimización, también adaptativo, muy utilizado. De manera similar a otros algoritmos del mismo tipo, este algoritmo utiliza estimaciones mediante las reglas L2 del primer y segundo momento del gradiente para adaptar la tasa de aprendizaje del modelo. Este también cambia la tasa de aprendizaje de manera individual para cada parámetro como RMSprop, aunque presentando rendimientos mejores que este, siendo conocido por su gran versatilidad, rápida convergencia y eficiencia. Una variación menos conocida de este es Adan, que en vez de el momento lineal utiliza el momento de Nesterov para realizar las adaptaciones (Nesterov, 1983).

Normalmente se diría que la elección del algoritmo de optimización adecuado depende del problema específico que se esté abordando, así como de consideraciones prácticas como el tamaño del conjunto de datos y la arquitectura de la red neuronal. Sin embargo se ha encontrado que en la práctica el más utilizado es Adam e incluso variaciones y optimizaciones del mismo (Tato y Nkambou, 2018), ya que se considera el mejor para prácticamente todas las aplicaciones. Esto fue demostrado en Noor (2020) en su conclusión tras realizar un estudio con diferentes pruebas de problemas a resolver termina por concluir que "*... el Algoritmo de Optimización Adam funciona mejor con los cuatro Modelos de Redes Neuronales Profundas, en todas las circunstancias y por lo tanto es prácticamente capaz de trabajar con cualquier modelo de clasificación resultando en la mejor precisión.*"

Es por esta razón que este es el que se va a utilizar para la implementación en un primer momento, aunque se vayan a realizar pruebas y análisis de rendimiento con otros optimizadores más adelante.

5.3. Aplicaciones del Deep Learning

Con su habilidad de aprender características complejas y completar tareas que son difíciles de modelar mediante otras técnicas, el Deep Learning ha revolucionado muchos campos. Por ejemplo, en medicina ha sido utilizado para diagnosticar enfermedades a partir de imágenes médicas con una precisión comparable a la de los expertos humanos.

Una de las aplicaciones más significativas del Deep Learning fue el desarrollo de AlexNet en 2012, una red neuronal convolucional que realizó un avance importante en el campo de la clasificación de imágenes utilizando el conjunto de ImageNet (Krizhevsky et al., 2012). Desde entonces, las Redes Neuronales Convolucionales son las más utilizadas en aplicaciones de visión artificial como pueden ser el reconocimiento facial, la detección de objetos y conducción autónoma (Geisslinger et al., 2021).

En el procesamiento del lenguaje natural, por ejemplo, el Deep Learning ha impulsado progresos significativos tales como la traducción automática, análisis sentimentales, generación textual y reconocimiento vocal (Medsker et al., 2001). Entre los diferentes modelos e implementaciones destaca el transformer (Hung et al., 2021). Este es un modelo basado en un mecanismo de atención que ha permitido alcanzar mejorías masivas en la calidad de las traducciones automáticas al darle diferentes pesos a cada parte de los datos de entrada.

Adicionalmente, siendo más parecido al tipo de implementación que se realizará en nuestro caso, el deep learning ha logrado hacerse un hueco en la industria automotriz. Su principal aplicación en este se ha ajustado a mejorar los sistemas de seguridad y control, consiguiendo una conducción más segura y hasta autónoma. Los sistemas de percepción basados en redes neuronales permiten a los vehículos que lo implementan identificar peatones, señales de tráfico, obstáculos y otros vehículos en tiempo real (Hussonnois y Jun, 2022).

5.4. Puntos clave sobre Deep Learning

A lo largo de este capítulo, se han explorado diversos aspectos de Deep Learning. Principalmente se han estudiado sus fundamentos teóricos hasta llegar a examinar sus aplicaciones prácticas en diferentes áreas. Entre sus características más destacadas se encuentra su capacidad para aprender representaciones de datos jerárquicas o no lineales. Con esta capacidad es que normalmente se tratan problemas donde la extracción de características complejas y abstractas son críticas. Esta capacidad que demuestra el Deep Learning lo hace especialmente adecuado para aplicaciones en las que los datos presentan relaciones no lineales y una complejidad mayor, como el procesamiento de imágenes, de textos en lenguaje natural y especialmente, la robótica (Goodfellow et al., 2016).

En cuanto a las posibles arquitecturas de redes neuronales, hemos explorado una variedad de modelos bastante generales. Se han visto desde redes neuronales convolucionales (CNN) para el procesamiento de imágenes hasta redes neuronales recurrentes (RNN) para el procesamiento de secuencias temporales. La introducción de estructuras más avanzadas y complejas, como las redes neuronales generativas adversarias (GAN) o redes transformer, ha ampliado aún más el alcance y posibilidades de aplicación del Deep Learning en las áreas exploradas.

Se ha examinado también como el entrenamiento de modelos de Deep Learning es un proceso intensivo, que requiere una combinación de algoritmos de optimización eficientes. Entre estos se encuentran el descenso de gradiente estocástico y sus variantes como RMSprop, Adagrad y Adam. También se ha argumentado y demostrado por qué en nuestra implementación se ha decidido utilizar Adam, ya que se considera el mejor para prácticamente todos los casos de uso posible (Tato y Nkambou, 2018), entre lo que esperamos que se encuentre nuestra implementación.

Además, hemos examinado el papel de la regularización y los distintos optimizadores en el entrenamiento de modelos de Deep Learning. Todo para prevenir el sobreajuste y mejorar la generalización a datos no vistos. Estos problemas no solo se presentan en estos modelos y los consideramos críticos de cara a aplicarse a nuevos entornos con nuevos datos sin perder su capacidad de generalización.

El como se abordan problemas complejos con datos masivos ha evolucionado radicalmente al popularizarse el Deep Learning. Esto es causado por la calidad de las soluciones que estos modelos proveen en una variedad de dominios. Se considera que serán estas técnicas las que nos permitan servir de base para abordar desafíos aún más grandes en el futuro.

Capítulo 6

Desafíos del espacio continuo

"El software se ha comido el mundo, pero es la IA la que se va a comer al software."
— Jensen Huang

En una implementación del mundo real, no siempre es factible o conveniente discretizar el entorno de manera directa, ya que esto podría llevar a la pérdida de información o a una complejidad computacional de espacio y del algoritmo innecesaria. En el caso de querer discretizar espacios muy grandes lo más común es que el coste de una precisión aceptable sea prohibitivo, en estos parámetros. En tales casos, es necesario recurrir a métodos más avanzados que puedan manejar entornos continuos de manera eficiente y efectiva.

Los métodos de Policy Gradient se presentan como una alternativa novedosa e interesante para abordar problemas en entornos continuos, permitiendo que los agentes aprendan directamente una política óptima sin la necesidad de discretizar el espacio de acciones o estados.

Los métodos que veremos a continuación utilizan una política parametrizada que puede seleccionar acciones sin tener una función valor. De esta manera vamos a considerar métodos para aprender esta política parametrizada basada en el gradiente del rendimiento de un escalar $J(\theta)$ respecto al parámetro parametrizado (Sutton y Barto, 2018).

En lugar de predecir valores de acción o estado, estos métodos optimizan directamente la política del agente, maximizando la recompensa esperada a lo largo del tiempo.

$$\Theta_{t+1} = \Theta_t + \alpha \hat{G}_t \tag{6.1}$$

$$G_t = \nabla J(\Theta_t) \tag{6.2}$$

Donde la segunda parte es un estimado estocástico cuya expectativa aproxima el gradiente del rendimiento con respecto a su argumento Θ_t. Todos los métodos que siguen este esquema se pueden llamar métodos de Policy Gradient.

La ventaja clave de los métodos de Policy Gradient es su capacidad para actuar con acciones en un espacio continuo, es decir, tipos de problemas donde el entorno o las acciones no son discretas, lo que los hace especialmente adecuados para aplicaciones en las que las acciones son suaves y tienen un rango infinito de posibilidades, como el control de sistemas físicos o robóticos. Además, al aprender directamente esta política, normalmente estos métodos pueden adaptarse de manera más flexible a cambios en el entorno y a la aparición de nuevas situaciones.

Sin embargo, el entrenamiento de políticas mediante métodos de Policy Gradient presenta desafíos que no nos encontrábamos en los métodos de AR ordinarios.

La optimización de políticas suele implicar el enfrentarse a problemas que presentan alta varianza en la estimación del gradiente según el episodio. Esto habitualmente puede dificultar la convergencia del algoritmo y hacer que el rendimiento en el entrenamiento sea inestable o incluso imposible (Sutton y Barto, 2018). Para abordar estos problemas se han propuesto diversas técnicas, como la normalización de características, la reducción de la varianza del gradiente y el uso de arquitecturas de redes neuronales específicas para la representación de políticas más complejas.

En este capítulo, exploraremos en detalle uno de los métodos más destacados y prometedores de Policy Gradient. Este algoritmo es el Deep Deterministic Policy Gradient (DDPG), que funciona al combinar elementos de AR heredados de métodos como Q-learning o Sarsa, con Deep Learning para aprender políticas continuas en entornos de una alta dimensionalidad.

6.1. Problemas de Aprendizaje Continuo

Antes de introducir DDPG y las técnicas que se derivan de este hay que entender la problemática que emerge con el uso de un espacio continuo. Uno de los principales desafíos en el aprendizaje continuo es el fenómeno conocido como olvido o fallo catastrófico. En el olvido catastrófico los modelos de aprendizaje automático, ya no solo de deep learning, tienden a perder la capacidad de generalizar sobre datos previamente vistos cuando se les presenta nueva información. Esto se puede ver explorado y medido en Kemker et al. (2018) "*(MLP) y los algoritmos de entrenamiento típicos no pueden gestionar el aprendizaje incremental de nuevas tareas o categorías sin olvidar catastróficamente los datos de entrenamiento aprendidos previamente.*".

Esto se debe a que los modelos suelen estar muy optimizados para aprender sobre un conjunto específico de datos durante el entrenamiento. Al verse así de optimizados para un dominio concreto estos pueden tener dificultades para adaptarse a nuevos datos sin olvidar las características y atajos del dominio que han aprendido anteriormente.

Para abordar este problema se han propuesto varias técnicas a lo largo del tiempo.

Entre ellas se encuentra el aprendizaje incremental, donde el modelo se actualiza continuamente con nuevos datos sin perder el conocimiento previamente adquirido. Esto se logra mediante técnicas como el aprendizaje online, el cual utilizaremos en nuestra implementación para el péndulo y brazo robot, donde el modelo se va actualizando con cada nueva muestra de datos.

Sin embargo, solo actualizar el modelo de manera incremental no es suficiente en la mayoría de casos en los que se presenta el olvido catastrófico. Normalmente se utilizan paralelamente técnicas de regularización que penalizan los cambios drásticos en los pesos del modelo. Estas técnicas funcionan restringiendo las modificaciones a los parámetros del modelo durante el entrenamiento con nuevos datos, asegurando que el conocimiento previamente adquirido no se pierda de repente.

Otra técnica importante es el uso de una memoria externa o amortiguadores de experiencia, que podremos explorar más adelante en forma del búfer de repetición. Este tipo de almacén de información permiten al modelo almacenar y recuperar información de manera eficiente. Esto suele ayudar a mitigar el olvido catastrófico al darle al modelo acceso a datos anteriores durante el entrenamiento.

Además, aunque no se vaya a aplicar en el proyecto que se está desarrollando, se han propuesto enfoques basados en meta-aprendizaje. Esto se realiza mediante el entrenamiento en una variedad de tareas relacionadas aunque con objetivos de naturaleza diferente durante la fase de pre-entrenamiento. Esto puede ayudar al modelo a generalizar mejor a nuevos datos y retener información previamente aprendida.

Como resumen, los problemas de aprendizaje continuo presentan desafíos diferentes de abordar en el desarrollo de modelos que apliquen aprendizaje automático. Es importante tener en cuenta los desafíos que se presentan de manera más común para acelerar el proceso de desarrollo de sistemas inteligentes, especialmente para que puedan operar de manera efectiva en entornos dinámicos.

6.2. Aproximación de Políticas

La aproximación de políticas es una estrategia muy importante para poder modelar la toma de decisiones de un agente en entornos continuos. En este tipo de contextos, la aproximación de políticas implica representar esta estrategia de manera parametrizada, lo que va a permitir su optimización a través de un proceso de aprendizaje.

Esta política puede se puede representar de muchas maneras, siempre y cuando la función de acción según el estado y el rendimiento para cada caso se pueda diferenciar respecto a sus parámetros. Es decir, que la política debe ser parametrizada de forma que sus parámetros se puedan ajustar mediante el descenso de gradiente, idealmente mejorando el rendimiento obtenido.

Es decir, esto tiene que cumplir la condición siguiente: Sea $f : R^n \to R$ una función cuyas derivadas parciales existen y son finitas para todo x en su dominio, entonces se expresarán para sus variables de la siguiente manera:

$$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \tag{6.3}$$

Donde $x_1, x_2, \ldots, x_n$ son las variables independientes que componen el dominio de la función f (Sutton y Barto, 2018).

Esta condición termina por representar la existencia de una gran flexibilidad para parametrizar un sistema, lo cual introduce la posibilidad de elegir la manera de parametrizarse estos entornos según la aplicación concreta. Así pueden ser por una red neuronal profunda o pueden ser directamente lineales utilizando vectores de características.

Un método común de parametrización es a través de distribuciones de probabilidades, sobre todo en el caso de trabajar en espacios continuos. A diferencia del caso de los espacios de acción discretos, en lugar de asignar preferencias numéricas a cada acción, podemos modelar las distribuciones de probabilidad sobre el espacio de acción de manera directa.

La principal ventaja que trae la aproximación de políticas es la capacidad que posee para modelar políticas estocásticas. Incluso si el objetivo es obtener una política determinista, la aproximación estocástica puede ser beneficiosa para garantizar una exploración adecuada del espacio de acciones. Por ejemplo, en un enfoque E-greedy, existe una probabilidad E de seleccionar una acción aleatoria en lugar de seguir la política determinista.

Otra opción bastante utilizada para la parametrización de la función de política es utilizar una distribución de máximos débiles. Los máximos débiles estarán basados en este caso en los valores de las acciones para cada uno de los estados. Esto implica que debemos asignar probabilidades a cada acción en función de su valor estimado. De este modo se terminarán eligiendo más frecuentemente las acciones con valores más altos.

Una manera común de definir esta parametrización es mediante una función *softmax*, ya que nos ayuda con esta generación de probabilidades. Este tipo de función asigna probabilidades a cada acción de la que disponemos de acuerdo a su valor estimado según la siguiente ecuación:

$$\pi(a|s) = \frac{e^{\frac{Q(s,a)}{\alpha}}}{\sum_{a'} e^{\frac{Q(s,a')}{\alpha}}} \tag{6.4}$$

Donde $\pi(a|s)$ es la probabilidad de seleccionar la acción a dado un estado s, $Q(s, a)$ es el valor estimado de la acción a en el estado s y α es un parámetro de suavizado que controla la exploración.

Sin embargo, este enfoque tiene una limitación importante: no permite que la política alcance una forma determinista. Incluso si una acción tiene un valor mucho más alto que las demás, aún existe una probabilidad finita de que se seleccione una acción diferente. Esto se debe a que la distribución de probabilidad asignada a las acciones está influenciada por los valores estimados, pero no se limita estrictamente a seleccionar la acción con el valor más alto.

En lugar de converger hacia una política determinista, los valores estimados de las acciones convergerían gradualmente hacia los valores verdaderos a medida que el agente continúa explorando y aprendiendo sobre el entorno. Esto quiere decir que, usando una distribución estocástica, siempre habrá una variabilidad inherente en la selección de acciones, lo que impide que la política se vuelva completamente determinista.

En el caso de que necesitemos que se vuelva determinista, si se le incluye un parámetro de temperatura esta bajaría con el tiempo para volverse determinista, aunque, en la práctica, es difícil elegir un horario de reducción o incluso la temperatura inicial para que de resultados adecuados.

Sin embargo, estas aproximaciones traen tambien la ventaja de que permiten la selección de acciones con probabilidades arbitrarias. En problemas con mucha aproximación esta puede ser estocástica, por ejemplo en juegos con información imperfecta o incompleta, como en juegos de dados o cartas.

Finalmente, denotamos que la elección de política es una buena manera de añadir conocimiento experto o adicional de la forma de la política en el sistema de aprendizaje. Esta elección de política implica que se se pueda inicializar estos parámetros de tal manera que se reflejen unas reglas o heurísticas. Estas heurísticas ayudarán al sistema a apuntar en la dirección más óptima.

6.3. Deep Q-learning

El Deep Q-Learning se basa en la estimación de la función de valor de acción $Q(s, a)$ definida en la sección sobre el AR [4]. Esta estimación se realiza mediante una aproximación con una arquitectura de red neuronal profunda que se conoce como Deep Q-Network [6.1]. Esta es una red neuronal que toma como entrada el estado del entorno y produce como salida los valores Q para todas las posibles acciones (Mnih et al., 2015).

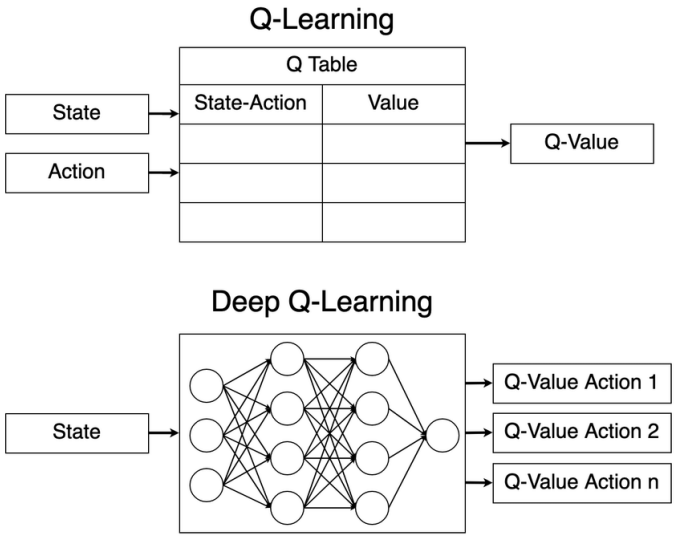

Figura 6.1: *Q-Learning vs Deep Q-Learning* por (Sebastianelli et al., 2021)

Normalmente para esta red se utiliza una función de pérdida [5.2] de diferencia cuadrática entre el valor predicho y el obtenido como lo es la siguiente.

$$L(\theta) = E\left[\left(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \theta') - Q(s_t, a_t; \theta)\right)^2\right] \tag{6.5}$$

Siendo θ los parámetros de la red y $Q(s_t, a_t; \theta)$ el valor de Q que predice la red, con θ' como los parámetros de una red objetivo que se utiliza para estabilizar el proceso de entrenamiento y evitar problemas ya explicados en la sección 5.2.1.

Al tener esta manera de almacenar la función Q, el Deep Q-Learning ha mostrado ser efectivo en sistemas donde Q-Learning puede tener el coste muy elevado o en sistemas que presentan un mayor dinamismo con el que Q-Learning no es capaz de lidiar sin la Deep Q-Network.

6.4. Deep Deterministic Policy Gradient (DDPG)

El algoritmo Deep Deterministic Policy Gradient (DDPG) es una técnica avanzada que se utiliza en el Aprendizaje por Refuerzo (AR). Esta está diseñada concretamente para resolver problemas en los que el espacio de acciones es continuo.

DDPG combina elementos de dos enfoques populares en AR: Q-learning, que se utiliza para aprender las funciones de valor de acción, y métodos Policy Gradient, que se utilizan para aprender políticas deterministas óptimas. Al integrar estos dos enfoques en el mismo algoritmo se obtienen las ventajas que presentan ambos. Con estas ventajas DDPG logra abordar la necesidad de aprender políticas óptimas en entornos complejos donde las acciones son continuas.

El funcionamiento de DDPG se fundamente en la interacción entre dos componentes principales y que exploraremos de manera más detallada: el actor y el crítico. Este enfoque tiene una serie de similitudes con el Double Q-Learning y las redes generativas adversariales. En este sistema el actor es una red neuronal que mapea estados del entorno a acciones, y su objetivo es aprender una política determinista óptima que maximice la recompensa acumulada a lo largo del tiempo. El crítico, por otro lado, es otra red neuronal que estima el valor de la acción en un estado dado, lo que proporciona retroalimentación sobre qué tan buena es la acción seleccionada por el actor en ese estado específico para así poder adaptarse mejor al dinamismo del espacio de estados.

6.4.1. Aprendizaje de política en DDPG

Este aprendizaje de política se realiza mediante el actor, cuyo objetivo es aprender una política determinista óptima que maximice la recompensa acumulada a lo largo del tiempo. A diferencia de los métodos convencionales de aprendizaje de política que pueden generar acciones de forma estocástica para métodos de Policy Gradient, el enfoque en DDPG es aprender una política determinista que asigna directamente cada estado a una acción óptima.

Esto se consigue mediante gradientes para el actor y el crítico, con los cuales mejoran sus funciones aproximadoras para decidir la acción a realizar.

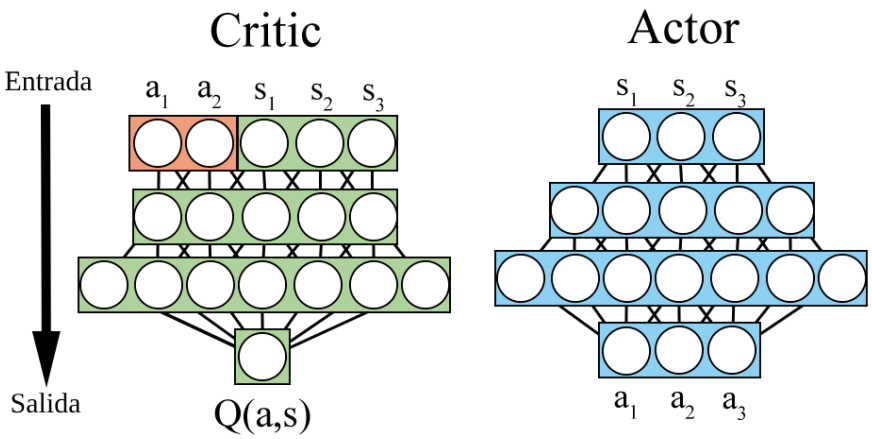

Figura 6.2: Estructura de un actor y crítico con estados de tres variables y acciones de dos y tres variables

6.4.1.1. Actor en DDPG

El aprendizaje de la política se realiza utilizando un enfoque de gradiente ascendente. En este enfoque los parámetros del actor se actualizan en la dirección que maximiza el rendimiento esperado por el agente.

Una vez que se ha calculado el gradiente de la función de rendimiento se utilizan

técnicas de optimización para actualizar los parámetros del actor en la dirección del gradiente ascendente. Esto se realiza iterativamente durante el entrenamiento para mejorar gradualmente la política del actor y maximizar el rendimiento esperado en el entorno para el que estamos entrenando el modelo.

6.4.1.2. Crítico en DDPG

Dentro de este algoritmo, el componente o red crítica utiliza una técnica derivada del Q-learning [4.5] para aprender a estimar una función de valor de acción continua. La función $Q(s, a)$ va estimando el valor esperado del retorno acumulado al tomar la acción a en el estado s. La actualización de los valores de Q se realiza utilizando la ecuación de Bellman. Esta ayuda a la estimación porque establece una relación recursiva entre los valores de Q en diferentes estados y acciones de la siguiente manera:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r(s, a) + \gamma \max_{a'} Q(s', a') \right) \tag{6.6}$$

Donde r es la recompensa obtenida por tomar la acción a en el estado s, γ es el factor de descuento que controla la importancia de las recompensas futuras, y s' es el siguiente estado observado después de tomar la acción a.

Como en este caso la función Q está siendo aproximada por una red neuronal, esta se actualizara en la dirección del gradiente descendente de la función de error cuadrático medio entre el valor predicho y el valor real de la acción.

La arquitectura del crítico también puede variar tanto en forma como en profundidad, pero típicamente implica una red neuronal profunda que aproxima la función de valor Q en una función que conoceremos como $Q*$. En nuestra sección de implementación será cuando se explore la estructura concreta de nuestras redes, maximizando la utilidad para nuestros casos de uso.

6.4.2. Exploración y explotación

En DDPG, la exploración se logra mediante la introducción de ruido en las acciones seleccionadas por el actor para forzar que no tome siempre el mismo camino en el espacio, fomentando así la exploración. Este ruido puede tomar varias formas, como ruido gaussiano aditivo o procesos de Ornstein-Uhlenbeck, y se aplica para agregar variabilidad a las acciones seleccionadas, como se expresa en:

$$a_{noisy} = a_{original} + \mathcal{N}(0, \sigma) \tag{6.7}$$

Donde N será una función que añada ruido a la salida en función de σ. Esta controlará la forma e intensidad del ruido que se le añada a la salida de la red actor.

El hiperparámetro σ puede quedarse estable o disminuir a medida que el agente

se vuelve más experto en la tarea y requiere menos exploración ya que ha pasado un buen número de los posibles mínimos locales en los que se puede estancar.

De manera contraria, la explotación se realiza seleccionando la acción determinista óptima según la política aprendida por el actor. Esta acción se elige en función de la estimación de la función de valor $Q*$, es decir, de la salida del crítico, y se selecciona como la acción que maximiza la recompensa esperada en ese estado.

Comúnmente para estos problemas encontrar el equilibrio adecuado entre exploración y explotación para un aprendizaje eficaz puede ser complejo. Una exploración excesiva puede llevar a un uso ineficiente de los recursos del agente (normalmente en cálculos por tiempo de entrenamiento) y una convergencia lenta hacia una política óptima, mientras que una explotación excesiva puede resultar en una convergencia prematura hacia óptimos locales y una falta de capacidad para descubrir nuevas estrategias efectivas.

Es por todo esto que se vuelve crucial ajustar la exploración y explotación en función de las necesidades específicas del problema y del progreso del agente en la tarea.

6.4.3. Extensiones y Aplicaciones

DDPG ha sido estudiado de forma extendida y aplicado en diferentes contextos y con diferentes objetivos, dando lugar a variantes y extensiones que buscan adaptarse mejor a ciertos dominios al mejorar su rendimiento y aplicabilidad para estos. Algunas de estas variantes y extensiones incluyen:

- **Twin Delayed DDPG (TD3):** Esta es una extensión de DDPG que introduce varios cambios para mejorar su estabilidad y rendimiento en entornos donde la complejidad sea excepcionalmente alta. Una de las mejoras clave en TD3 es el uso de dos redes de críticos en lugar de una sola, lo que ayuda a mitigar el problema de la sobreestimación de los valores de Q para $Q*$ al utilizar siempre el que de valores mínimos para esto. Además, TD3 utiliza una política de actualización retardada para los parámetros del actor y los objetivos de los críticos, lo que estabiliza el entrenamiento y mejora la convergencia para este contexto, como es demostrado en el control de un robot hormiga de cuatro patas en Dankwa y Zheng (2020).

- **Soft Actor-Critic (SAC):** Este método es otra variante de DDPG que se basa en el enfoque de máxima entropía para mejorar la exploración del espacio de acciones, es decir, que pasa a maximizar tanto el rendimiento como la entropía. Esto lo consigue al introducir una regularización de entropía en la función de rendimiento que penaliza las políticas deterministas demasiado rígidas (Haarnoja et al., 2018). Además, SAC puede aplicarse utilizando dos redes de críticos y una política de actualización retardada similar a TD3 para mejorar la estabilidad del entrenamiento.

DDPG y sus variantes han demostrado ser efectivos en una variedad de aplicaciones en robótica y control, incluyendo la manipulación de objetos, la navegación autónoma y el control de drones autónomos respecto a otras posibilidades como son el deep Q-learning (Tagesson, 2021).

6.4.4. Hindsight Experience Replay: HER

Hindsight Experience Replay (HER) es una técnica diseñada para mejorar la eficiencia del AR en entornos donde la obtención de recompensas es escasa o dispersa. Esta técnica se ha aplicado con éxito en algoritmos como DDPG para abordar problemas de muestreo ineficiente y mejorar el rendimiento del agente en tareas complejas (Andrychowicz et al., 2017).

El principio fundamental de HER es aprovechar la información obtenida de experiencias fallidas[1] para mejorar el aprendizaje del agente. En lugar de considerar una experiencia como un fracaso si no alcanza el objetivo deseado, HER reutiliza esa experiencia para entrenar al agente en el contexto de un objetivo diferente. Esto se logra modificando el objetivo de la experiencia, el objetivo concreto de este paso temporal, a uno ya cumplido o que se esté cumpliendo en ese instante, con todos los cambios que esto conlleva. Con este cambio, el algoritmo aprende de esta como si realmente hubiera ocurrido naturalmente. Al hacerlo, se enriquece el conjunto de datos de entrenamiento del agente con éxitos sintéticos y se mejora su capacidad para aprender de experiencias pasadas.

En DDPG, HER se integra mediante la modificación retrospectiva de las experiencias almacenadas en el búfer de repetición, la estructura encargada de guardar los resultados de los pasos de simulación y que le suministrará a los algoritmos los ejemplos de los que aprender. Cuando una experiencia se almacena en el búfer de repetición, se selecciona un objetivo de manera aleatoria (denominado objetivo de Hindsight) y se modifica el estado final y la recompensa asociada para que coincida con el objetivo de Hindsight. Esto crea una nueva experiencia "positiva"que el agente puede aprender, incluso si la experiencia original fue un fracaso en términos del objetivo original.

La aplicación de HER en DDPG es relevante al permitir al agente aprender de la exploración de manera más eficiente al reutilizar experiencias que inicialmente se consideraron fracasos. Esto a cambio amplía el conjunto de datos de entrenamiento del agente y mejora su capacidad para generalizar.

Es especialmente relevante en entornos donde las recompensas son escasas o difíciles de alcanzar. Por ejemplo en los modelos de recompensa esparcida, HER proporciona una manera de generar más experiencias positivas para el agente, incluso cuando no logra el objetivo deseado inicialmente.

[1] Que no hayan llegado al objetivo, no estrictamente fallidas o con errores.

En el ejemplo propuesto anteriormente de una carrera de atletismo donde a los corredores solo se les otorga recompensa si estos son capaces de llegar a la meta se aplicaría de manera directa. Si nunca se llega a meta nunca se obtendrá ninguna recompensa, por lo que se mueve la meta para que el corredor la haya cruzado y se guarda esa experiencia fabricada.

A modo de conclusión, HER facilita la transferencia de conocimiento entre objetivos similares al modificar las experiencias almacenadas enseñándole al agente como se vería un estado con un objetivo diferente o resuelto, dependiendo de la implementación.

Capítulo 7

Tecnologías utilizadas

"La realidad es que no estar preparado es una elección. Los beneficios llegan cuando vemos la IA como una herramienta, no como algo que nos aterrorice y la utilizamos en nuestras ventas."
— Anita Nielsen

En esta sección se presentarán las tecnologías empleadas para el desarrollo e implementación del proyecto, además de justificar la razón o razones por las que se ha decidido implementar con dichas herramientas.

7.1. Python

Python es un lenguaje de programación de alto nivel que es ampliamente utilizado en el desarrollo de software, siendo muy popular también para el aprendizaje automático. Una de sus principales características es que su sintaxis es de muy alto nivel y tiene una gran capacidad de adaptarse a múltiples estilos de programación (Sarkar et al., 2017).

Gracias a su flexibilidad y que es un lenguaje de código abierto; Python mantiene una de las comunidades mas extensas dentro de los lenguajes de programación, haciendo que destaque por tener una cantidad de librerías como numpy, implementada parcialmente en C y muy eficiente para el uso de matrices, y APIs[1] muy abundantes dentro de todos los lenguajes de programación.

Python es el lenguaje mas extendido dentro del desarrollo de algoritmos de inteligencia artificial y machine learning. Por esta razón, Se ha establecido Python como el principal lenguaje de programación del proyecto.

[1] *Application Programming Interface* o Interfaz de Programación de Aplicaciones

7.2. Plataformas de desarrollo

7.2.1. Visual Studio Code

Este es un editor de desarrollo creado por Microsoft que es capaz de soportar múltiples lenguajes de programación. Además incluye diferentes extensiones creadas por la comunidad que le permite ampliar la funcionalidad y facilitar el desarrollo de aplicaciones de muchos tipos, siendo utilizado tanto a nivel educativo como empresarial.

Entre el uso de extensiones, la amplia integración que tiene con Git para el control de versiones, la familiaridad de los integrantes del grupo con este entorno y su versatilidad; se ha establecido utilizar Visual Studio Code como entorno de desarrollo de la aplicación del proyecto.

7.2.2. Github

Github es una plataforma web que permite la gestión y el almacenamiento del código fuente y documentación de los proyectos utilizando el control de versiones Git, un sistema de código abierto con este propósito [2].

Github permite a los desarrolladores compartir proyectos en forma de repositorios y colaborar en los proyectos de otros equipos de manera controlada. Los desarrolladores utilizan los *commits* a nivel local para recibir los cambios y los *pushes* a nivel de rama para enviar los cambios que ellos mismos han hecho. Esto permite mantener una gran coherencia mientras que se hacen cambios y pruebas a diferentes niveles (Blischak et al., 2016).

En este proyecto se ha utilizado un repositorio de Github para el desarrollo conjunto de la aplicación y el control de versiones al ser varios miembros. Principalmente ha sido por su comodidad, familiaridad y el control con capacidades de *rollback* que este brinda.

7.3. Herramientas de desarrollo

7.3.1. Gym

Gym (recientemente ha cambiado de nombre y actualmente se llama gymnasium)[3] es una biblioteca Python de código abierto desarrollada por Open AI que tiene como finalidad proporcionar una plataforma para desarrollar aplicaciones de aprendizaje por refuerzo que normalmente proporcionan una API para utilizar la misma sintaxis con diferentes entornos.

[2]Documentación de Git: `https://git-scm.com`
[3]Documentación de Gymnasium: `https://gymnasium.farama.org/index.html`

Gym proporciona varios entornos o simulaciones diferentes para que los desarrolladores tengan un gran número de sistemas para crear los algoritmos, tanto orientados a ser resueltos con métodos TDL como entornos más complejos que se busca que se resuelvan en entorno contínuo.

Gracias a ser una plataforma flexible para desarrollar algoritmos y ser de código abierto, se ha formado un ecosistema de bibliotecas alrededor de Gym que proporcionan nuevos entornos de desarrollo o funcionalidades

7.3.2. Gymnasium-Robotics

Gymnasium-Robotics es una biblioteca que funciona dentro del ecosistema de Open AI Gym. Esta biblioteca proporciona una serie de entornos de prueba especializadas en el control de robots en un espacio continuo. Entre los principales entornos se pueden encontrar el control de un brazo robot manipulador y el control de una mano robótica. En este proyecto se ha utilizado el entorno del robot manipulador (Plappert et al., 2018).

Debido a que el objetivo de este proyecto es el desarrollo de un algoritmo de aprendizaje por refuerzo en un espacio continuo, se ha decidido utilizar la biblioteca Gymnasium-Robotics para que nos proporcione el entorno del brazo robot, concretamente utilzando el entorno Fetch.

7.3.3. Keras

Keras[4] es la biblioteca de alto nivel desarrollada por ingenieros de Google cuya función es la creación de redes neuronales con una API integrada.

La principal ventaja de Keras es que se pueden ensamblar y configurar las capas de las redes neuronales con gran simplicidad y sin necesidad de fabricar cada neurona de forma manual, pudiendo modificar funciones de activación y pérdida de manera sencilla.

Además, su arquitectura modular permite organizar las capas y los modelos sin necesidad de rediseñar la red neuronal por completo lo que hace que el proceso de construcción sea muy flexible, por lo que ha sido elegida por encima de alternativas como Pytorch.

7.3.4. TensorFlow

TensorFlow [5] es la biblioteca más utilizada para utilizar tensores, normalmente aplicado al machine learning, ya que le permite al usuario crear y desarrollar redes

[4]Documentación de la API de Keras: `https://keras.io`
[5]Documentación de Tensorflow: `https://www.tensorflow.org/`

neuronales de manera cómoda. En nuestra implementación es el principal responsable de la creación y configuración de las neuronas, así como de ser el corazón de ciertas implementaciones de optimizadores de Keras.

A diferencia de Keras, TensorFlow ofrece operaciones de control de tensores que se pueden considerar de bajo nivel, lo que permite un mayor control al usuario sobre la red neuronal y que permite utilizarlas para la implementación realizada de DDPG, actualizando los pesos directamente fuera de Keras. Es por estas razones que en este proyecto se utiliza para administrar el entrenamiento de las redes actor y crítico.

7.3.5. MuJoCo

MuJoCo[6] es un motor de físicas donde se renderizan y simulan robots u otros sistemas de forma rápida y precisa, pudiendo establecer la frecuencia de control y otras características de estos para poder así acercar de gran manera un modelo a su versión real.

Este motor es de código abierto y gracias a ello, MuJoCo está implementado dentro de la biblioteca Gymnasium-Robotics para que la biblioteca pueda crear sus entornos, siendo MuJoCo el que se encargue de la simulación de estos a un nivel más bajo del que se va a utilizar.

[6]Documentación de MuJoCo: `https://mujoco.readthedocs.io/en/stable/overview.html`

Capítulo 8

Implementación

"La inteligencia artificial e IA generativa puede ser la tecnología más importante de toda una vida."
— Marc Benioff

8.1. Algoritmo de DDPG en Python

En este apartado se mostrará la implementación del proyecto aplicando el marco teórico explicado anteriormente cuya finalidad es cumplir los objetivos del mismo. Todo este código se puede encontrar en el repositorio del proyecto, junto a datos y gráficas sacados del entorno de pruebas [1].

Debido a la poca experiencia previa en el desarrollo de redes neuronales en un entorno Python, se ha decidido utilizar una metodología de trabajo iterativa. Esta metodología de trabajo consiste en revisar y mejorar iterativamente el código haciendo que en cada iteración sea capaz de superar un entorno mas complejo hasta poder realizar un estudio de su funcionamiento en el entorno Fetch.

A pesar que se utilicen varios entornos, los pilares de la aplicación siguen siendo los mismos ya que en todos los entornos se ha utilizado el algoritmo DDPG (Tabor, 2020) con posterior aplicación del algoritmo HER en DDPG+HER.

La aplicación se basa en dos pilares que son el agente y el entorno. Gracias a las bibliotecas como Gym y Gymnasium-Robotics, nos han proporcionado varios entornos para poder desarrollar y configurar el agente.

Inicialmente hemos comenzado con un entorno básico llamado péndulo; por lo que se usará este entorno para explicar las bases de la implementación.

[1] `https://github.com/CarloDubini/ReinforcementLearningTFG`

Algorithm 2 Pseudocódigo de Main

```
Inicializar el entorno y obtener sus parámetros
Inicializar el agente con los parámetros del entorno e hiperparámetros dados
Cargar punto de control si es necesario
for i = 1 hasta el número de episodios elegido do
    Reiniciar entorno y puntuación de episodio
    while No ha finalizado el entorno y dentro del límite de pasos do
        Elegir acción para el estado actual
        Realizar acción y obtener siguiente estado φ(s'|s, a)
        Calcular recompensa con el método decidido
        if Modo de exploración then
            if Se entrena con HER then
                Aplicar HER
            end if
            Aprender de un grupo de transiciones
        end if
        Actualizar estado
    end while
    Imprimir información sobre el episodio y guardar pesos del modelo si es necesario
end for
if Modo de exploración then
    Realizar gráficos sobre el aprendizaje y guardar recompensas de episodios.
end if
```

Al ejecutar la aplicación, como se puede ver conceptualizado en el pseudocódigo 2 *Pseudocódigo de main*, se inicializa el entorno mediante una llamada a la clase env. La clase env se encarga de controlar el entorno del sistema y cuando el agente ejecuta la acción en el entorno, este se encarga de actualizar el entorno mediante su función $\phi(s'|s, a)$. Además, el entorno se encarga de enviar la información que el agente necesita para poder tomar una decisión. Por ejemplo, el entorno le muestra cual el es objetivo del agente y cual es su estado actual. Toda esta información se guarda en un diccionario llamado "*observation*".

Después de conocer el tamaño del entorno y el numero de acciones posibles, se crea la clase del agente con hiperparámetros establecidos. Se explicará el funcionamiento del agente y entorno posteriormente en sus secciones en este capítulo.

Una vez se ha configurado el entorno y el agente, se puede empezar el entrenamiento. El objetivo del entrenamiento es que el agente aprenda a cumplir su objetivo generalizando su aprendizaje. Esto permite al Agente a actuar correctamente en situaciones inciertas. Para que el Agente aprenda a generalizar, hay que evitar a toda costa el sobreajuste. Con este objetivo, el entrenamiento se divide en una serie de episodios y en cada episodio, el agente posee un número limitado iteraciones o acciones para cumplirlo.

En cada episodio el agente aparece en un lugar aleatorio dentro del entorno para que así evitar el sobreajuste al iniciar en una zona concreta. Además se aplican técnicas detalladas en la sección del agente para evitar el sobreajuste y en medida de lo posible el fallo catastrófico, del que se habla en el capítulo sobre Deep Learning [5].

El entrenamiento comienza con un bucle que ejecuta todos los episodios. En cada episodio se crean las variables de inicialización del episodio y luego el agente toma una decisión, luego de insertar la decisión en el entorno, este devuelve un nuevo estado, e inicializa la fase de aprendizaje. almacena los resultados de sus decisiones iterativamente hasta que termina el episodio. Luego de terminar el episodio, se evalúa si la red neuronal ha mejorado su modelo. En caso afirmativo se almacenan los pesos. Para saber si el nuevo modelo es mejor que el anterior, agrupamos su puntuación media en los últimos cien episodios y lo comparamos con el anterior en vez de solo el actual, como se defiende teóricamente en [6] y se puede observar en la figura [8.5]. Si el resultado es mejor, entonces se almacena el nuevo modelo como los pesos de las cuatro redes neuronales que componen el almacenamiento de la política.

Después de finalizar el entrenamiento y almacenar los pesos del agente, podemos decidir si comenzar un nuevo entrenamiento o cargar los modelos previamente almacenados para observar los resultados del entrenamiento anterior. En caso afirmativo, el agente que adquiere los pesos almacenados, deja de entrenar y procede a interactuar con el entorno sin ser evaluado en una fase de explotación. En caso contrario se procede con el entrenamiento.

8.1.1. HER:Hindsight Experience Replay

Algorithm 3 Pseudocódigo de HER

if Se activa her statistic para esta iteración **then**
  Crear nuevos estados (s y $\phi(s'|s,a)$).
  Modificar el punto objetivo a la misma posición del efector final.
  Calcular el resultado del nuevo estado modificado $\phi(s'|s,a)$
  Almacenar la transición en el replay búffer con los nuevos estados y la nueva recompensa, normalmente cercana a cero.
end if

Para la implementación de HER, cada vez que se efectúa una iteración en el entrenamiento, añadimos una nueva iteración modificada en el búfer de repetición. La modificación consiste en cambiar el objetivo a la posición del efector final y su recompensa a la recompensa óptima dentro de la iteración, que suele ser cero para todos nuestros sistemas.

Como se explicó teóricamente en la sección [6.4.4], el propósito de este diseño es que en cada iteración, el agente aprenda a asociar que el efector final tiene que estar en la misma posición que el punto objetivo. Es importante no modificar el estado de finalización (d) debido a que es necesario para el correcto aprendizaje del agente.

Con estas nuevas modificaciones se espera que los sistemas con recompensa esparcida obtengan un rendimiento similar al de recompensas densas al incorporar este nuevo tipo de experiencias.

8.2. Agente

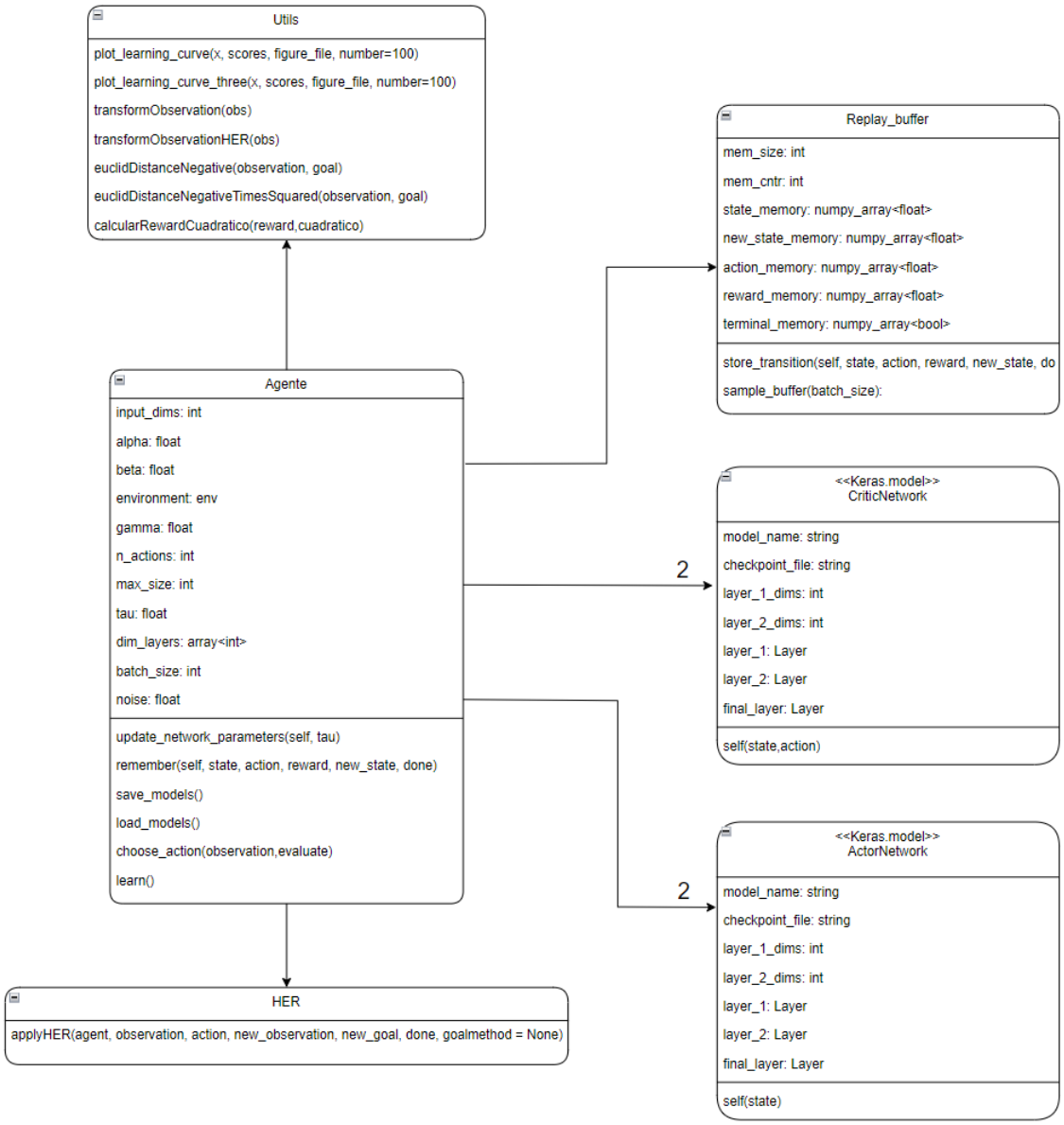

Figura 8.1: Diagrama de clase del agente

La clase agente es la que se encarga de actuar de controlador y de gestionar el proceso de entrenamiento a través de las cuatro redes neuronales secuenciales multicapa que posee. Estas redes neuronales son: el actor, el target actor, el critic y el target critic. El agente posee un replay buffer que se encarga de almacenar las ultimas iteraciones que realizó el agente.

Se ha decidido hacer que las redes sean secuenciales multicapa debido a su simplicidad y facilidad de ajuste en comparación con convolucionales y ya que se ha considerado que los estados son temporalmente independientes al contener información sobre la posición y velocidad del agente en el entorno, por lo que es innecesario

el uso de recurrentes o transformers.

Algorithm 4 Pseudocódigo de inicialización del agente

Inicializar el ReplayBuffer para el tamaño máximo dado y la forma de datos del entorno.
if No se introduce entorno **then**
Se establecen el mínimo y el máximo de los valores de las acciones como -1 y 1 respectivamente.
else
Se ajustan los máximos y mínimos de la accion a los del entorno.
end if
Se establecen los valores de alpha, beta, optimizador, tau, gamma, tamaño de batch, dimensiones y ruido.
Inicializar las redes de Actor y Target Actor con el tamaño de las acciones y las dimensiones deseadas para las redes.
Inicializar las redes de Critic y Target Critic con las dimensiones deseadas para las redes.
Se igualan los valores de los pesos de cada par de target-net.

Como se puede ver en el pseudocódigo [8.2] el agente está compuesto de los siguientes atributos que servirán como hiperparámetros para el control del entrenamiento en los entornos:

- **input-dims**: Representa el número de variables de observación que devuelve el entorno.
- **alfa**: Es el ratio de aprendizaje que posee el actor.
- **beta**: Es el ratio de aprendizaje que posee el critico.
- **environment**: Es el entorno de la aplicación.
- **gamma**: Sirve para establecer el peso que representa el target-critic durante el proceso de aprendizaje.
- **n-actions**: Indica al Agente el espacio de acciones que puede hacer el actor.
- **max-size**: Establece el tamaño que debe poseer el replay buffer. es decir, establece en número máximo de iteraciones que debe almacenar.
- **tau**: Establece el porcentaje de actualización de los nuevos pesos sobre los antiguos (target vs net) sirviendo como control de la velocidad de actualización de las redes neuronales con el fin de evitar sobreajustes.
- **batch-size**: Indica al replay buffer cuantas iteraciones ha de devolver cuando se le solicita un lote.
- **noise**: Es el ruido que posee la salida de acción cuyo objetivo es que aumente la exploración.

- **max-action**: Es un array que indica el valor máximo que puede poseer todas las acciones. Un ejemplo de ello seria la velocidad máxima que puede tener un vehículo.

- **min-action**: Es un array que indica el valor mínimo que pueden poseer todas las acciones.

Habiendo establecido los atributos, el agente procede a inicializar sus redes neuronales: el actor, el critico, el target-actor y el target-critic. Los target-actor y target-critic sirven para entrenar el actor y el crítico igualando sus valores iniciales.

8.2.1. Actor

El actor es la red neuronal que se encarga de la toma de decisiones, acciones en este caso, y su objetivo es aprender la política determinista óptima para maximizar su recompensa.

Como se puede ver en la sección , cada vez que el agente actúe con el entorno, este invoca a la red neuronal del actor para que devuelva una acción.

Cuando se inserta el estado actual (s), la red neuronal devuelve una acción (a_t). Posteriormente, el agente se asegura de que el actor no haya sobrepasado los límites del rango de decisiones.

El actor es un modelo de keras secuencial multicapa [6.2] que forma la red neuronal para la toma de decisiones. Durante su inicialización se ensambla la red neuronal que está compuesta por:

- **Una capa de entrada**: Es la capa que recibe el estado actual del actor (s) y las envía a la primera capa oculta.

- **Dos o más capas ocultas**: Que se encargan de transformar los datos aplicando los pesos establecidos y lo envían a la siguiente capa.

- **Una capa de salida**: En esta capa, su número de neuronas es equivalente al espacio de acciones haciendo que cada neurona represente cada una de las variables de acción. Debido a que en el espacio de acciones existe una acción que no afecta al entorno, se ha decidido quitar una neurona que era la encargada de esa acción para no causar problemas en el funcionamiento de la red.

Las primeras capas poseen la función de activación de unidad rectificada uniforme debido a que es la que da mejores resultados en las capas ocultas por lo que es la más utilizada para el control en casos similares al nuestro (Li et al., 2024; Dankwa y Zheng, 2020).

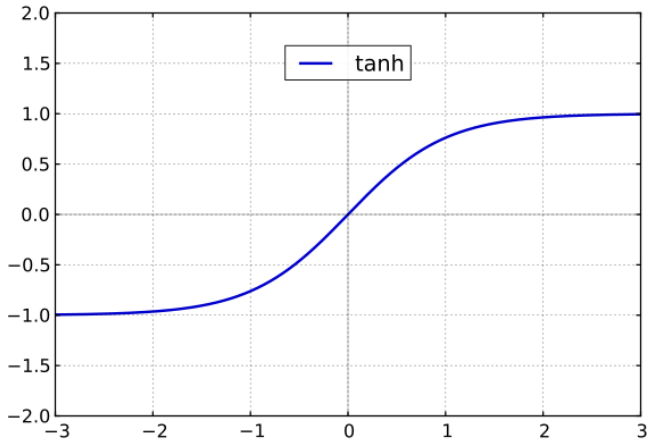

Figura 8.2: Función de tangente hiperbólica, Geek3(2014), CC BY 3.0

A diferencia del resto de las capas, la capa de salida posee una función de activación tangente hiperbólica [8.2][2]. La razón por la que hemos decidido usar esta función de activación es que el dominio del control de todas las acciones que poseen nuestros entornos es de [-1,1] por lo que esta función de activación corresponde con el dominio de la acción de manera exacta y así no será necesario generar código que acote la salida a este intervalo.

8.2.2. Critic

La clase Critic es el otro modelo de keras que al igual que el actor forma una red neuronal cuyo objetivo es predecir el valor de una acción en un estado determinado ($Q(a_t, s_t)$). Debido a que el objetivo del critic es distinto al actor, la estructura de la red neuronal es diferente. La red neuronal está compuesto por una capa de entrada que recibe el estado (s_t)y la acción (a_t), un número de capas ocultas y la capa de salida.

A diferencia de la red neuronal del actor, la capa de salida posee solo una neurona que devuelve el valor del estado y la acción. Además, la neurona de salida, no posee ninguna función de activación en este caso, por lo que devuelve la combinación lineal de sus entradas. Esto es debido a que se desea una salida sin restricciones que imite cualquier clase de recompensa por iteración que establezcamos.

8.2.3. Replay Buffer

Debido a que se trabaja en un entorno continuo, El agente necesita almacenar y recuperar la información para el correcto aprendizaje. Como se puede observar en la figura [8.3]. [3]El Replay Buffer almacena un número fijo de experiencias o transiciones pasadas y devuelve una muestra para que el agente pueda usarlo en su aprendizaje.

[2]CC BY 3.0 `https://creativecommons.org/licenses/by/3.0`, via Wikimedia Commons
[3]`https://creativecommons.org/licenses/by-nc-nd/4.0/`

Estas experiencias consisten en información sobre el estado, la acción, la recompensa, el estado siguiente y si ha terminado.

Este también se encarga de, en el paso de entrenamiento, entregar un número de experiencias aleatorias definidas por el Batch size al agente para que aprenda de estas iteraciones y así evite el sobreajuste.

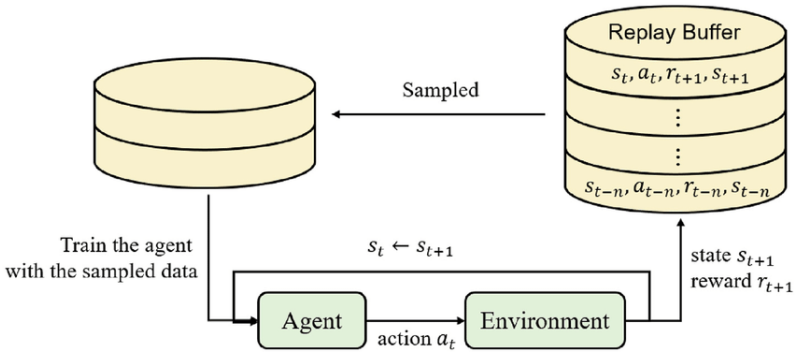

Figura 8.3: *Illustration of the replay buffer in the RL.* como se ve en Lee y Lee (2020) CC BY-NC-ND 4.0

8.2.4. Funcionamiento del Agente

8.2.4.1. Elegir Acción

Algorithm 5 Pseudocódigo de elegir una acción

Recibir el estado actual (s) y transformarlo en un tensor.
Enviar el estado actual al Actor y recibir la acción (a).
if está en la fase de exploración **then**
    se añade el ruido gaussiano a (a).
end if
if Si (a) supera los límites del dominio **then**
    Se sustituye (a) por el valor máximo o mínimo de la acción.
end if

Cuando el agente recibe el estado actual para que tome una decisión, transforma la variable que contiene el estado (observation) en un tensor, para que el Actor pueda recibir el estado y así devolver la acción deseada. Si el agente se encuentra en en el estado de exploración, se le aplica ruido en la acción para agregar variaciones en la toma de decisiones. En este proyecto se ha decidido introducir ruido gaussiano como se puede ver en el *pseudocódigo de elegir una acción* donde se añade el ruido usando una distribución normal. Después de añadir el ruido, el agente se asegura de que la acción se encuentra dentro del dominio de acciones que este puede ejecutar. En caso contrario, se sustituye la acción por la acción máxima o mínima del dominio.

8.2.4.2. Aprendizaje del Agente

Como se explicó en el capitulo de DDPG [6.4.1], la implementación del aprendizaje del agente se desarrolló de la siguiente manera:

Algorithm 6 Pseudocódigo del Aprendizaje del Agente

if El buffer no posee el número de acciones requerido (batch-size) **then**
Se omite el proceso de entrenamiento.
end if
Extraer una muestra aleatoria de B = (s, a, s', r, d) transiciones del buffer.
Obtener los targets:

$$y(r, s', d) = r(s, a) + \gamma * Q_{\phi targ}(s', a') * (1 - d) \tag{8.1}$$

Se obtiene la pérdida del crítico obteniendo el gradiente descendiente a partir del error cuadrático medio entre los críticos y los targets:

$$\Delta_\phi = \frac{1}{|B|} \sum_{(s,a,s',d)\epsilon B} Q_\phi(s, a) - y(r, s', d) \tag{8.2}$$

Se actualizan los pesos del crítico según el gradiente.
Calcular la pérdida del actor calculando el error mediante las acciones predichas y la valoración del crítico.
Se actualizan los pesos del actor según su gradiente de pérdida.
Pasan a actualizarse con una fracción de los cambios las redes target.

Antes de comenzar con el entrenamiento del agente, se supervisa que el agente haya realizado el número de acciones requerido para poder proceder con el proceso de entrenamiento. En caso de que no cumpla con el requisito, se omite el proceso de entrenamiento.

El primer paso para comenzar el proceso de entrenamiento, es extraer la información almacenada del replay-buffer en un grupo de transiciones B. (los estados (s), las acciones(a), los nuevos estados (s'), las recompensas recibidas (r) y si el estado siguiente es el estado terminal (d)) y los convertirlos en tensores. Con estas muestras, se procede a mejorar el Actor y el Crítico. Se ha de tener en cuenta de que se están manejando múltiples iteraciones para poder calcular la pérdida.

En el caso del crítico, se obtiene la pérdida comparando los valores que devuelve el crítico con los targets. Los targets representan los resultados objetivo ($y(r, s', d)$) por las que el critico tiene como objetivo obtener el mismo resultado. Para calcular los targets, se utilizan los target-actor y target-critic para obtener las predicciones de los valores máximos de los nuevos estados ($Q_{\phi targ}(s', a')$) y se suman con las recompensas reales recibidas del replay buffer (r). por lo que se representa de la siguiente manera:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a') * (1 - d) \tag{8.3}$$

Posteriormente, se insertan las acciones y estados en el crítico para predecir los valores Q. Con sus respectivos targets, se obtiene el error cuadrático medio obteniendo así la pérdida del crítico. Finalmente a partir de la pérdida del crítico se extraen los gradientes y se actualiza la red neuronal aplicando los gradientes.

En el caso del actor, se insertan los estados extraídos del replay buffer obteniendo una serie de acciones de cada estado, se evalúan insertando estas nuevas acciones en el crítico para conocer el valor de cada acción. Debido a que en el proyecto se han diseñado rewards que generan recompensas negativas, se cambia de signo a los valores que devuelve el crítico para la correcta representación de la pérdida del actor.

Posteriormente se obtiene el valor medio de todas las acciones obteniendo así la pérdida del actor. Finalmente se extraen los gradientes del actor y usando el optimizador Adam, se optimiza la red neuronal utilizando los gradientes para actualizar los parámetros del actor.

8.2.4.3. Actualizar las Redes

Algorithm 7 Pseudocódigo de elegir una acción

Obtener los pesos del Actor W_a y del target-actor W_{ta} respectivamente
for i = 1 hasta recorrer todos los pesos del target-actor **do**
    Actualizar el peso del target-actor (w_{ta}) con los nuevos pesos del actor (w_a) usando:

$$w_{ta} = w_a * \tau + w_{ta} * (1 - \tau) \tag{8.4}$$

end for
Obtener los pesos del Crítico (W_c) y del target-critic (W_{tc}) respectivamente.
for i = 1 hasta recorrer todos los pesos del target-critic **do**
    Actualizar el peso del target-critic (w_{tc}) con los nuevos pesos del crítico (w_c) usando:

$$w_{tc} = w_c * \tau + w_{tc} * (1 - \tau) \tag{8.5}$$

end for

Cuando se termina de entrenar el actor y el crítico, se actualizan los pesos del target-actor y target-critic mezclando los pesos anteriores con los nuevos pesos del actor y el crítico respectivamente. Con el valor tau, se regula la velocidad de actualización de los targets.

8.3. Péndulo

Para comenzar a crear la aplicación, se decidió empezar con el entorno del péndulo[4], que se puede ver en la figura [8.4]. Las razones de la elección es que es un entorno sencillo para una primera aplicación del algoritmo DDPG.

Figura 8.4: Imagen del entorno del péndulo

El entorno trata de un péndulo donde el agente trata de balancear el péndulo aplicando fuerza a la izquierda o a la derecha. El objetivo del agente es mover y mantener el péndulo en el punto mas alto de la circunferencia. El péndulo posee una fuerza de gravedad que dificulta al agente a elevarla a la posición mas alta y esta fuerza de gravedad tiene un valor por defecto de $g = 10$. En cada episodio, el péndulo aparece en una posición aleatoria de la circunferencia.

Sus variables de observación y el espacio de acciones son las siguientes:

Descripción	Min	Max
$x = cos(\theta)$	-1	1
$y = cos(\theta)$	-1	1
Velocidad Angular	-1	1

Tabla 8.1: Espacio de observación del péndulo

Descripción	Min	Max
Torque	-2	2

Tabla 8.2: Espacio de acciones del péndulo

[4]Documentación del péndulo: `https://gymnasium.farama.org/environments/classic_control/pendulum/#pendulum`

8.3.1. Configuración y Diseño

Debido a el objetivo establecido con esta implementación es la comprobación y prueba del funcionamiento del agente, se decidió utilizar el diseño de la recompensa que otorgaba el entorno.

La recompensa es la siguiente:

$$R_t = -(\theta^2 + 0{,}1 * \theta_{dt}^2 + 0{,}001 * torque^2) \tag{8.6}$$

Esta recompensa es una recompensa negativa, con lo que como el objetivo del agente es que este maximice la recompensa, se considera que la recompensa máxima es cero. Para el diseño del agente hemos establecido los siguientes hiperparámetros [8.3].

Hiperparámetro	Valor
Ratio de aprendizaje del Actor	0.001
Ratio de aprendizaje del Crítico	0.002
Capas ocultas del Actor	50 y 30: 2 Capas
Capas ocultas del Crítico	50 y 30 2 Capas
Batch size	50
Buffer size	10^6 de transiciones
Gamma	0.99
Ruido Gausiano	0.1

Tabla 8.3: Parámetros para el péndulo

Durante el entrenamiento se ha establecido un entrenamiento de 50 episodios con 500 iteraciones cada una. En total en todo el entrenamiento el agente habrá realizado 25.000 iteraciones. La razón por la que cada episodio posea tantas iteraciones es debido a que se desea que el agente pueda comprender la velocidad angular y la fuerza de gravedad.

8.3.2. Resultados

Como se puede ver en la figura [8.5], Se ha conseguido que el agente aprenda durante el entrenamiento. Hay que tener en cuenta de que al ser episodios muy largos, es inevitable que reciba una puntuación muy negativa.

Aunque no haya terminado de converger de manera totalmente estable, se ha conseguido que el agente aprenda la política objetivo por lo que se considera que se ha completado el principal objetivo de su implementación.

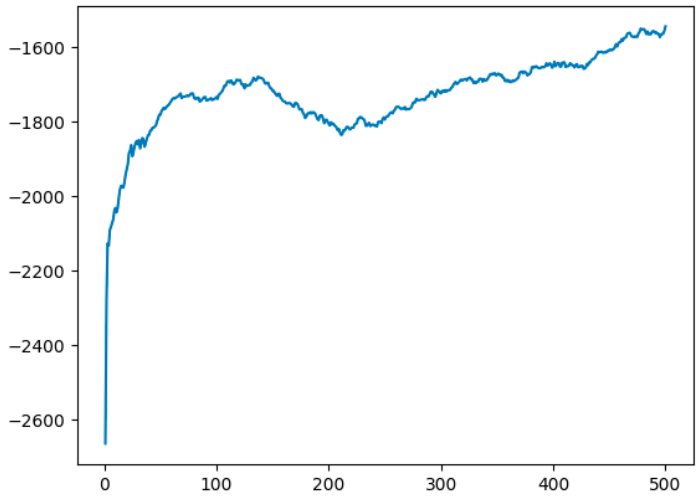

Figura 8.5: Resultado de las medias de entrenamiento en entorno péndulo no normalizadas

Se puede observar un vídeo de su comportamiento de explotación al intentar poner el péndulo recto en el aire en el siguiente enlace [5].

[5]Vídeo del péndulo: `https://youtu.be/xagI9Wn5qoA`

Capítulo 9

Controlando un robot manipulador en un espacio continuo

"Si he logrado ver más lejos ha sido porque he subido a hombros de gigantes"
— Isaac Newton

Durante la duración de este capítulo se estudiará la implementación y entorno concretos del modelo del robot manipulador. En el capítulo se detallarán los puntos concretos sobre su implementación para su posterior estudio de rendimiento y pruebas del controlador.

Una vez se ha terminado de detallar la implementación del sistema para el control del brazo robot es necesario volver a entrenar el modelo en pasos de análisis. Esto se realiza con el objetivo de averiguar cuales son las mejores combinaciones de hiperparámetros y arquitectura de las redes para nuestro controlador.

Con este objetivo se expondrán las variables que se van a estudiar y revelar los resultados encontrados para estas variaciones. Los resultados se dividirán en gráficas de rendimiento en entrenamiento y datos medios de explotación, explicados para cada caso concreto más adelante.

9.1. Robot Manipulador

Una vez se implementó con éxito el anterior entorno, se pasó a la siguiente fase utilizando el robot Manipulador para el entorno *Fetch Reach*.

9.1.1. Mujoco y Fetch

Utilizando la biblioteca de Gymnasium-Robotics, se ha decidido utilizar el entorno Fetch para cumplir con el objetivo del proyecto. Fetch está basado en el brazo

"7-DoF Fetch Mobile Manipulator"[1]. donde su efector final tiene forma de pinza para agarrar objetos.

Se utiliza el motor de físicas MuJoCo para simular el brazo robot. En este entorno se le asigna la tarea llamada "*Fetch-Reach*"que consiste en que el efector final se mueva hasta el punto objetivo. En cada episodio, el punto objetivo cambia de ubicación de forma aleatoria.

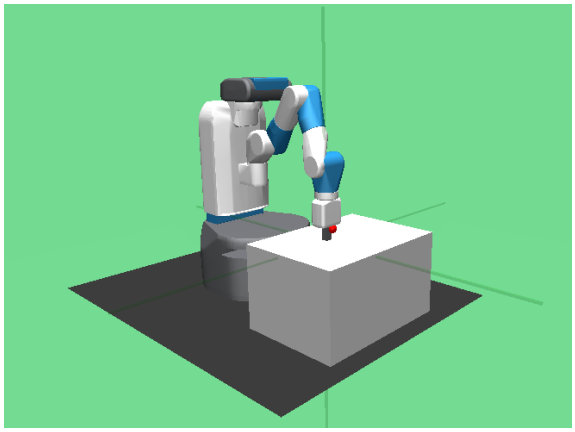

Figura 9.1: Imagen del entorno FetchReach

En cuanto al entorno *Fetch Reach*, este posee el siguiente espacio de acciones para el control del brazo:

Nombre	Descripción	Min	Max
x	Desplazamiento del efector final en la dirección x	-1	1
y	Desplazamiento del efector final en la dirección y	-1	1
z	Desplazamiento del efector final en la dirección z	-1	1

Tabla 9.1: Espacio de Acciones

Respecto al espacio de observación, el entorno devuelve un diccionario que contiene el estado actual del agente, la meta deseada y la meta alcanzada, que será útil para la implementación de HER. El estado actual del agente es representado como un vector que contiene los siguientes datos:

- La posición del efector final en los ejes (x,y,z).
- La velocidad lineal del efector final en las direcciones (x,y,z).
- La velocidad y apertura relativa de la pinza del robot, que no se utilizará en esta implementación

[1]Documentación de fetch: `https://robotics.farama.org/envs/fetch/index.html`

La meta deseada, contiene la posición del objetivo en los ejes (x,y,z) y la meta conseguida contiene la posición del efector final en los ejes (x,y,z). Esta segunda meta existe ya que al convertir esta meta en la meta a tomar es cuando se maximizan las funciones de recompensa, que es el objetivo de la implementación de HER.

9.1.2. Fetch-Reach

En ese entorno se han diseñado varios métodos de entrenamiento para el agente. Estos métodos se dividen en dos técnicas de entrenamiento (DDPG,DDPG+HER) y seis diseños de recompensas que son:dense, dense+ *time to goal*, sparse, sparse + *time to goal*, cuadrática y cuadrática + *time to goal*.

Para el diseño del Agente, entre todas las pruebas que se han hecho, se han establecido los siguientes hiperparámetros como baseline de simulación. Estos fueron los primeros hiperparámetros que se encontraron con una convergencia para todos los modelos:

Hiperparámetro	Valor
Ratio de aprendizaje del Actor	10^{-5}
Ratio de aprendizaje del Crítico	$2 * 10^{-5}$
Capas ocultas del Actor	250, 150 y 50: 3 Capas
Capas ocultas del Crítico	250, 150 y 50: 3 Capas
Batch size	100
Buffer size	10^6 de transiciones
Gamma	0.99
Ruido Gausiano	0.1

Tabla 9.2: Parámetros para el robot manipulador

Durante en cada entrenamiento se ha establecido un límite de 3.000 episodios con 100 iteraciones cada uno. En total en cada entrenamiento el agente habrá realizado 300.000 iteraciones. La razón por la que hemos decidido asignar menos iteraciones por episodio en comparación con el péndulo es para que el agente tome decisiones rápidamente y que se adapte a los cambios de posiciones del punto objetivo.

En cuanto al diseño de la recompensa, se han decidido diseñar 3 tipos de recompensas, la recompensa esparcida, la densa y la cuadrática. A estos diseños, se le puede añadir opcionalmente la recompensa *time to goal*. Esta modificación a la recompensa se explicará posteriormente.

La recompensa densa es la que se ha decidido implementar inicialmente como recompensa. Este diseño calcula la distancia euclídea entre el efector final y el punto objetivo devolviendo su resultado negativo como recompensa. Su ecuación es la siguiente:

$$R_t(p,q) = -1 * (\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}) \tag{9.1}$$

Donde:

- $p(x_1, y_1, z_1)$: Representa la posición del efector final
- $q(x_2, y_2, z_2)$: Representa la posición del punto objetivo
- t: Es el factor temporal, en la implementación reconocido como j, que representa el paso temporal del actual episodio y va desde 0 a el número máximo de iteraciones por episodio menos uno.

En cuanto a la recompensa esparcida, cada vez que el efector final del agente no se encuentra en la misma posición que el punto objetivo, recibirá una recompensa negativa fija en cada iteración hasta que llegue al objetivo siendo:

$$R_t(p, q) = -1 \tag{9.2}$$

En cuanto a la recompensa cuadrática, se eleva al cuadrado la recompensa densa para así penalizar de forma exponencial cuando el efector final se aleja del punto objetivo. por lo que su ecuación seria la siguiente:

$$R_t(p, q) = -1 * (\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2})^2 \tag{9.3}$$

Finalmente, En el caso de que se utilice el *time to goal* y el estado actual no ha sido finalizado, se modifica reward con añadiendo la siguiente recompensa negativa:

$$R_t(p, q) = R_t(p, q) - t * 0{,}01 \tag{9.4}$$

Durante la siguiente sección [9.2] se mostrarán y evaluarán cuales son los resultados de la implementación y el modelo con los parámetros que dan el mejor resultado para el entorno.

9.2. Variables estudiadas

Durante la experimentación y análisis del modelo se ha decidido explorar los efectos de las diferentes configuraciones en las siguientes variables:

- **Uso de diferentes funciones de recompensas:** Se evaluaron diversas funciones de recompensa para determinar su impacto en el rendimiento del controlador. Estas funciones son parte de las expuestas, la función de distancia euclidea negativa, distancia cuadrática negativa, esparcida y sus otras tres variantes con *time to goal*.
- **Uso de HER:** Se ha investigado como la técnica de HER influye en la capacidad del controlador para aprender y generalizar en las funciones con factor de recompensa parcial o totalmente esparcida.
- **Número de capas:** Se ha variado el número de capas en la arquitectura de las redes neuronal para examinar como la profundidad de la red afecta su capacidad para modelar la dinámica del sistema.

- **Cantidad de neuronas por capa:** Se ha ajustado la cantidad de neuronas en cada capa de la red para evaluar como en este caso la complejidad del modelo afecta su rendimiento.
- **Uso de diferentes funciones de activación:** En el capítulo de implementación se han presentado las funciones de activación Relu, que serán la base de la implementación para probar funciones puramente lineales, sigmoides tangentes hiperbólicas y softmax.
- **Uso de diferentes optimizadores:** Además de con Adam se han realizado pruebas con otros optimizadores expuestos anteriormente. También se ha probado con Nadam, que utiliza el momento de Nesterov en vez del momento lineal, introducido en Nesterov (1983). El momento de Nesterov es una variante del momento lineal donde el gradiente se evalúa no en el punto actual, sino en el punto adelantado por el momento.
- **Ruido y otros hiperparámetros:** Se han explorado diferentes configuraciones de hiperparámetros y ruido, como la tasa de aprendizaje, gamma, tau, el ruido y el tamaño de batch. Este estudio se hace para determinar su influencia en la convergencia y estabilidad del entrenamiento.

La fase de análisis a partir de este punto se realizará de manera secuencial, entrenando primeramente para la variación de función de recompensa, luego añadiendo HER y siguiendo con las secciones a continuación.

De esta manera siempre se elegirán los modelos que han mostrado ser más prometedores para nuestra implementación o los modelos más interesantes de ver en otras configuraciones. Este formato de pruebas se utilizará para sacar conclusiones más informadas sobre su funcionamiento.

Tanto el rendimiento en entrenamiento como en explotación se medirá tras normalizarlo, haciendo que el valor 1 pase a ser el nuevo máximo y 0 el mínimo encontrado en las recompensas del agente. Esto principalmente se realiza con el objetivo de que todas las recompensas estén en la misma escala. Como se ha visto anteriormente la recompensa cuadrática se mide en centímetros, la recompensa euclídea en metros y la recompensa esparcida no tiene una unidad concreta. A este nuevo valor normalizado se le nombrará con puntuación R o *R-score*.

El rendimiento del modelo posteriormente se valorará con un escalar que represente la suma de todas las recompensas a lo largo de doscientos episodios de prueba, clasificando así los sistemas en cada apartado. A estos test se les nombrará como test de explotación.

Se ha evaluado el rendimiento mediante una puntuación por episodio normalizada. Una puntuación de 1 en un episodio indica haber conseguido una recompensa de cero, la máxima para cualquier configuración estudiada. Como consecuencia, la puntuación máxima que se puede obtener para los doscientos episodios de simulación de explotación es de doscientos.

Adicionalmente y con el objetivo de que sea más legible se hablará también en términos del porcentaje de acierto. Este será equivalente a la recompensa media obtenida en todos los episodios multiplicada por cien.

9.3. Variación de función de recompensa

Como ya se ha establecido en secciones anteriores, se van a utilizar seis funciones de recompensas para la comparación, inicialmente sin aplicar HER, para delimitar cual o cuales de ellas son candidatas a representar los objetivos y guiar a la red crítico a una convergencia sobre los valores $Q(s, a)$ óptimos.

Las funciones a explorar son la función densa, densa con *time to goal*, esparcida y esparcida con *time to goal* [9.1.2]. A su vez, la densa va a dividirse entre el uso de la distancia estándar y la distancia cuadrática.

Esto nos deja con seis posibles entornos que estudiar para obtener la mejor versión del modelo. Debido a que las diferencias entre valores máximos de recompensa por episodios estos serán normalizados para poder representarse en la misma gráfica y poder compararse fielmente.

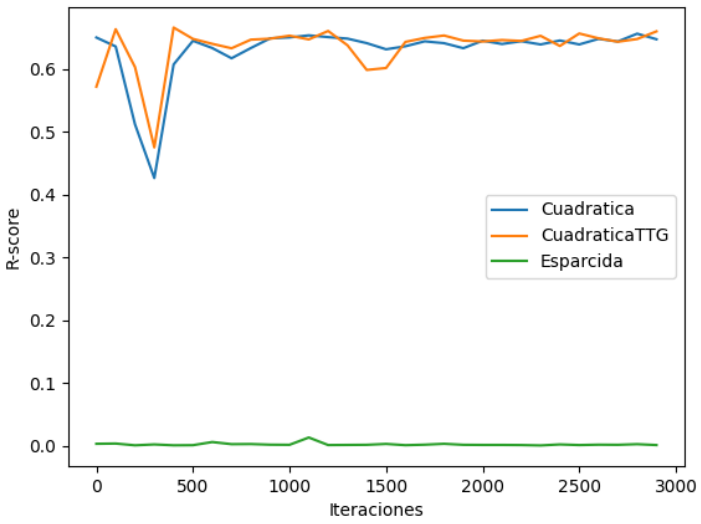

Figura 9.2: Gráfica de resultados: Reward 1

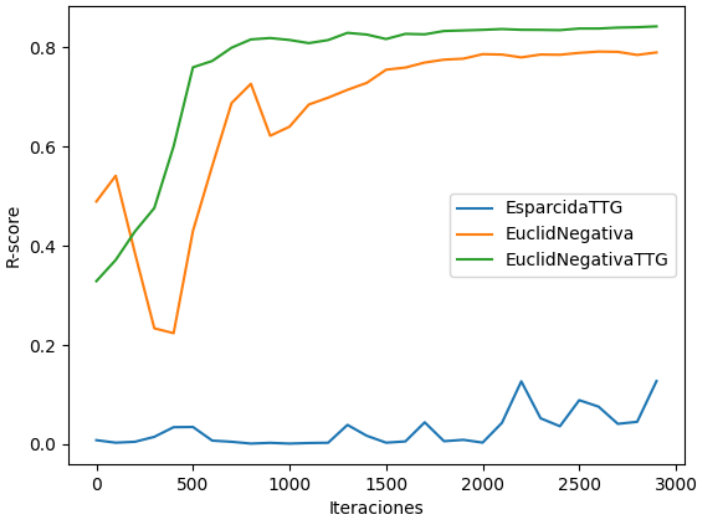

Figura 9.3: Gráfica de resultados: Reward 2

Como podemos observar en las figuras [9.2] y [9.3], la evolución del sistema para diferentes sistemas de recompensa es muy variable. Para las recompensas euclidea y euclidea con TTG presenta una evolución que aunque no es perfecta es muy favorable.

Por otro lado, el modelo se ve incapaz de aprender de los modelos basados en una recompensa esparcida. El modelo basado en el cuadrado de la distancia euclídea converge, pero este lo hace en un valor bastante más bajo que el de la distancia euclídea en cualquiera de sus dos versiones, con o sin TTG.

Función de recompensa	**Explotación**	**Porcentaje**
Cuadrática	77.47	38.7 %
Cuadrática + TTG	119.05	59.52 %
Esparcida	1.50	0.75 %
Esparcida + TTG	14.74	7.39 %
Distancia Negativa	157.19	78.59 %
Distancia Negativa + TTG	**164.38**	**82.18 %**

Tabla 9.3: Resultados de explotación de recompensas

Como podemos ver en la tabla [9.3] en este análisis se ha conseguido llegar a una puntuación bastante alta a falta de modificar más parámetros, siendo que se descartarán las recompensas basadas en la distancia cuadrada en favor de la distancia negativa.

9.4. Variación de uso de HER

Para probar el efecto de HER se van a utilizar las funciones de recompensa esparcidas ya que son para lo que está principalmente diseñado este algoritmo. Adicionalmente se usarán la más prometedora de las anteriores, es decir, que ha llegado a una convergencia en un punto mayor. Con esto vamos ahora a aplicar HER al algoritmo en dos modalidades.

La primera será con un valor de un 0.5 y 0.8 de parámetro de control de HER. Este parámetro medirá el tanto por uno de las iteraciones en las que se aprenderá utilizando HER en el estado.

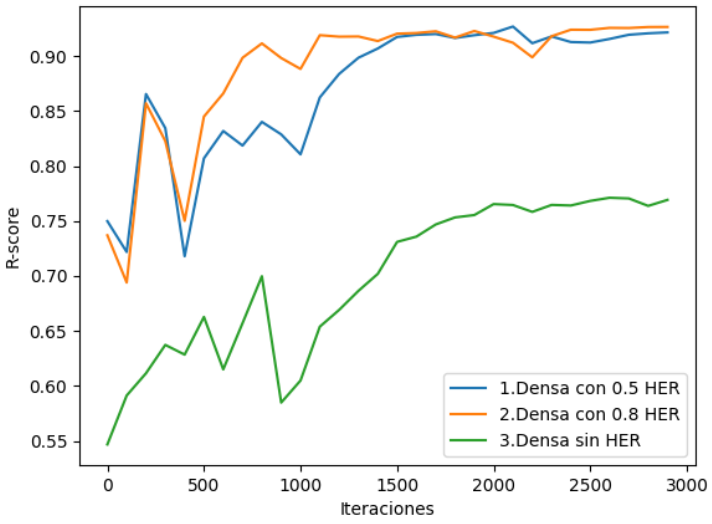

Figura 9.4: Gráfica de resultados: HER A

En esta primera gráfica [9.4] podemos ver el desarrollo del tipo de recompensa densa, distancia euclídea negativa, con sus respectivos valores de HER. Podemos ver como la que presenta una mejor progresión de entrenamiento es la que utiliza un mayor valor de estadística de HER.

Esta recta en la gráfica se mantiene sin converger más tiempo que la que utiliza un parámetro de control de HER menor, pero se ha considerado la mejor. Esta consideración es debida a que presenta una mayor estabilidad en los puntos finales del entrenamiento.

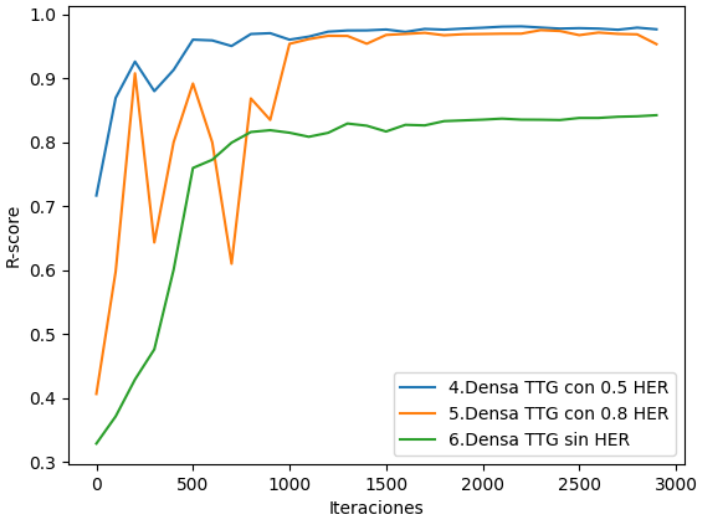

Figura 9.5: Gráfica de resultados: HER B

La segunda gráfica [9.5] nos enseña el desarrollo del tipo de recompensa densa la adición de TTG. Esta vez el mejor entrenamiento lo realiza la que tiene un valor de estadística de her de 0.5 siendo el mejor proceso de entrenamiento visto en esta serie de pruebas.

Este entrenamiento converge rápidamente, siendo la mayor diferencia que estas presentan una gran estabilidad sin caer en el fallo catastrófico, lo que hace que sea la más prometedora hasta el momento.

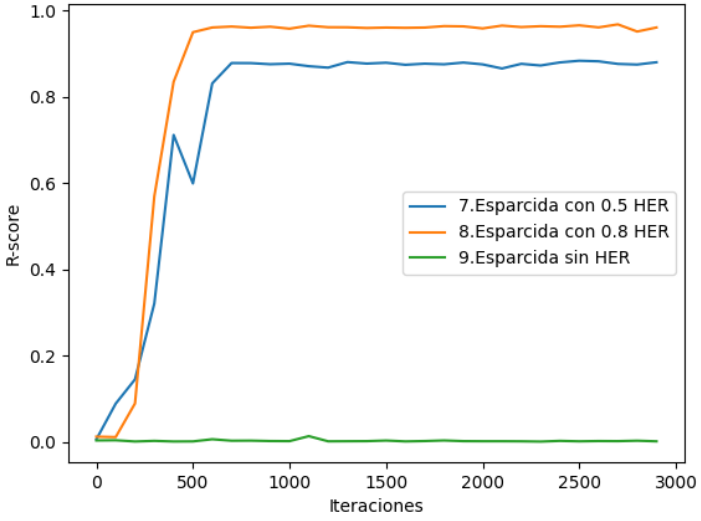

Figura 9.6: Gráfica de resultados: HER C

En la tercera gráfica [9.6] que hemos obtenido se puede ver el desarrollo de las

técnicas esparcidas en este nuevo modelo. A diferencia de en la sección anterior, donde estas daban, y siguen dando si no se utiliza HER, unos resultados que carecían de aprendizaje, ahora obtienen resultados muy prometedores.

Su convergencia, aunque tarda más episodios en estabilizarse que la de recompensa densa y TTG como ya hemos visto, presenta un valor estable muy prometedor de cara a estudiarlo en mayor profundidad.

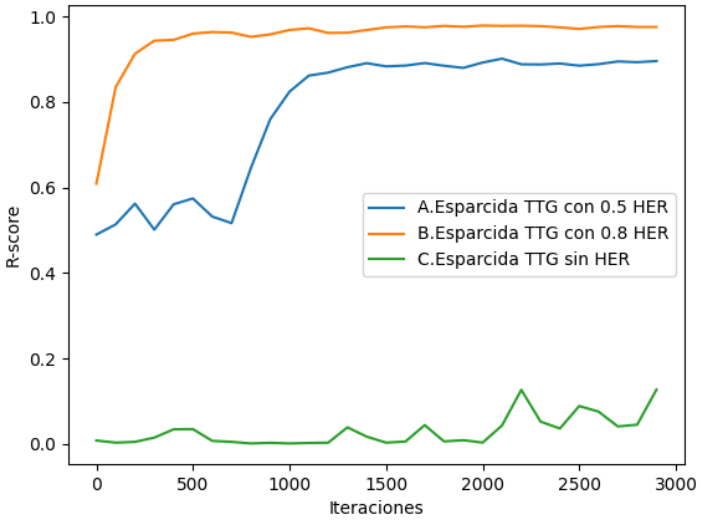

Figura 9.7: Gráfica de resultados: HER D

Por último, la cuarta gráfica [9.7] presenta las recompensas esparcidas con el uso adicional de TTG. En esta podemos observar que una de ellas tiene un desarrollo similar al de el uso de 0.8 de estadística de HER solo con la recompensa esparcida.

Se teoriza que esto es debido a que pese a darse todos los pasos de simulación, la recompensa TTG presenta una mayor utilidad cuando el otro tipo de recompensa con la que se usa es densa al ser un tipo de estimulación diferente para el entrenamiento.

Durante la explotación de estos modelos [9.4] se han evaluado los modelos para comprobar su funcionamiento con el mejor modelo guardado y la misma función normalizada que en el anterior apartado.

Función de recompensa	Explotación	Porcentaje
Densa con 0.5 HER	178.87	89.44 %
Densa con 0.8 HER	175.23	87.62 %
Densa sin HER	147.98	73.99 %
Densa + TTG con 0.5 HER	171.98	85.99 %
Densa + TTG con 0.8 HER	**189.61**	**94.81 %**
Densa + TTG sin HER	159.37	79.69 %
Esparcida con 0.5 HER	125.89	62.94 %
Esparcida con 0.8 HER	142.02	71.01 %
Esparcida sin HER	2.7	1.35 %
Esparcida + TTG con 0.5 HER	146.23	73.11 %
Esparcida + TTG con 0.8 HER	**186.61**	**93.31 %**
Esparcida + TTG sin HER	9.79	4.90 %

Tabla 9.4: Resultados de explotación de cambios en HER

Los resultados de explotación se acercan ya a unas precisiones muy acertadas, con las explotaciones esperadas para la mayoría de sistemas. Destacan entre ellas la que utiliza la recompensa densa con TTG y HER y la esparcida con estas dos mismas modificaciones.

Estas alcanzan puntuaciones superiores al 90 % en ambos test de explotación, llegando a una estabilidad en el punto de convergencia temprana y eficiente.

Igualmente, se van a seguir estudiando los efectos de diferente modificación de parámetros con el fin de refinar este modelo de la mejor manera posible.

9.5. Variación de las capas y número de neuronas

Con los datos obtenidos se obtiene que el mejor modelo para entrenar nuestro sistema es el uso de HER con la recompensa en función del negativo de la distancia euclídea. Utilizando este modelo, se han realizado pruebas para conocer el tamaño de las capas y el número de estos que han generado los siguientes resultados.

Siendo cada separación por comas indicativo del número de neuronas para una capa oculta.

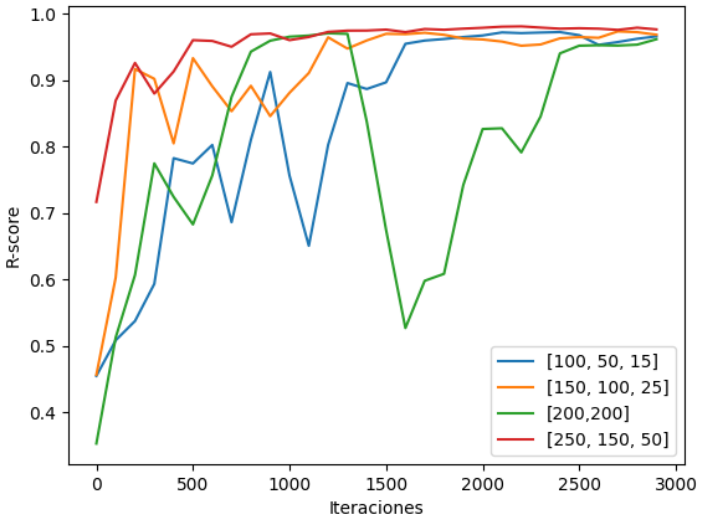

Figura 9.8: Gráfica de resultados: Capas 1

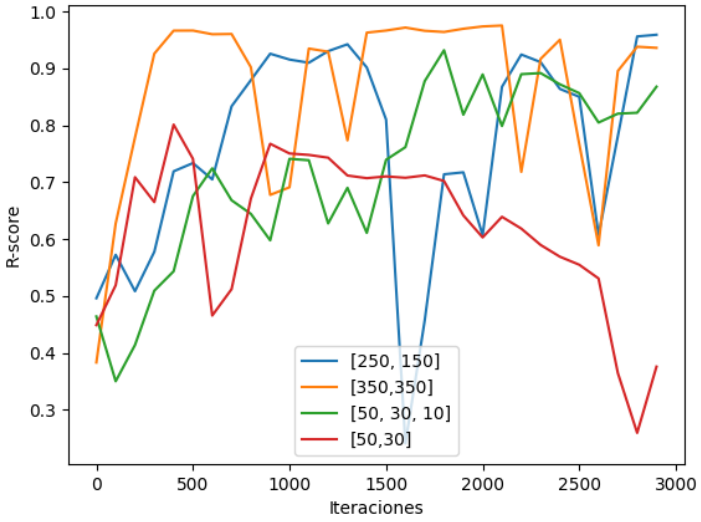

Figura 9.9: Gráfica de resultados: Capas 2

En este caso, en las figuras [9.8] y [9.9] podemos ver que el modelo de tres capas con [250,150,50] neuronas respectivamente, conlleva una convergencia temprana con una estabilización muy clara. Este modelo es el que estábamos ya utilizando para las pruebas anteriores.

Este modelo se estabiliza de manera bastante temprana y no sufre fallos catastróficos como el resto. El resto de modelos parece que se desentrenan con facilidad, especialmente los de menos número de neuronas totales.

También es interesante ver como de bueno es el modelo respecto a su complejidad, es decir, la cantidad de conexiones totales que tienen respecto al desarrollo que demuestran. El modelo con más conexiones es el de 350 en ambas capas, siendo que aún así es superado por el de tres capas pese a tener algo más de la mitad de conexiones.

Es importante también notar la mejora que hay al añadir una tercera capa, que pese a no aumentar demasiado el número de conexiones este otorga una mejora importante a su proceso de entrenamiento y desarrollo posterior.

Neuronas por capa	**Explotación**	**Porcentaje**
50,30	130.95	65.47 %
200,200	182.17	91.0 %
250,150	172.42	86.26 %
350,350	180.97	90.48 %
50,30,10	134.67	65.34 %
100,50,15	168.92	84.46 %
150,100,25	180.45	90.23 %
250,150,50	**187.52**	**93.76** %

Tabla 9.5: Resultados de explotación de neuronas

Este modelo considerado como óptimo genera además una explotación más eficiente en comparación con el resto de modelos como se puede ver en la tabla [9.5]. El margen de mejora de este se considera en el límite dentro de lo que se consideraría del error inevitable. Esto es debido a que al moverse por el entorno para este tipo de recompensa. Inevitablemente, el efector tendrá que acercarse desde el punto inicial al del objetivo, proceso en el que la recompensa va a ser menor a cero de manera casi garantizada[2].

9.6. Variación de las funciones de activación

Durante la fase de pruebas también se ha probado a alterar las funciones de activación de las capas, tanto a funciones lineales como no lineales. Estas funciones solo se han cambiado para las capas internas del sistema, puesto que las capas salida se quiere que sigan teniendo la forma especificada en la sección [8.2].

[2]Puede darse el caso de que el brazo aparezca directamente en el punto objetivo, caso en el que es posible obtener una puntuación final de cero en el episodio

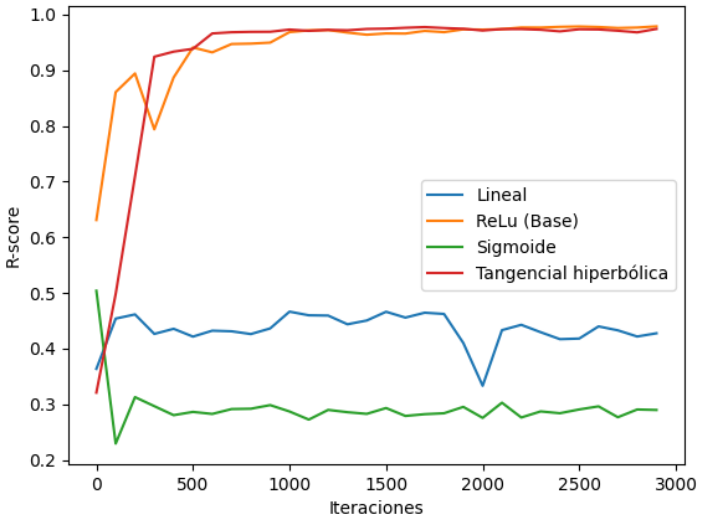

Figura 9.10: Gráfica de resultados: Funciones de activación

Al observar las curvas, se puede notar que las funciones de activación Tangencial hiperbólica y ReLU sobresalen en comparación con las otras. Esto indica que estas funciones permiten al modelo aprender de manera más eficiente y efectiva, logrando un alto desempeño rápidamente.

La estabilidad y alto desempeño de la Tangencial hiperbólica y ReLU puede explicarse por sus características. La Tangencial hiperbólica, al igual que la ReLU, puede manejar de manera más eficiente las no linealidades en los datos. Por otro lado, la ReLU también tiene la ventaja de ser más simple y computacionalmente eficiente.

Las otras funciones, tanto lineal como sigmoide, no parecen ser capaces de capturar la complejidad de las relaciones en los datos en este contexto, por lo que fallan de manera muy clara con sus transformaciones.

Función de activación	**Explotación**	**Porcentaje**
Lineal	70.06	35.03 %
ReLu (Base)	**190.17**	**95.08 %**
Sigmoide	59.82	29.91 %
Tangencial hiperbólica	183.63	91.19 %

Tabla 9.6: Resultados de explotación de diferentes funciones de activación

Las puntuaciones de explotación también confirman estos puntos expuestos. Es por estas razones que se elige como la mejor la función ReLu como la mejor, ya que la explotación de la tangencial hiperbólica tiene una puntuación menor que ReLu siendo que tiene un resto de características similares.

9.7. Variación de los optimizadores

Los optimizadores son una parte muy importante en el contexto del DDPG, ya que controlan el ratio de aprendizaje de los pesos de las redes. Los cambios en factores de aprendizaje y patrones de actualización han resultado tener un peso significativo en la robustez de nuestro controlador. Durante este análisis se utilizarán los optimizadores expuestos en la sección [5.2.3] además de NAdam o Adan, que a diferencia de Adam se basa en el momento de Nesterov para realizar obtener el gradiente de actualización sobre los pesos.

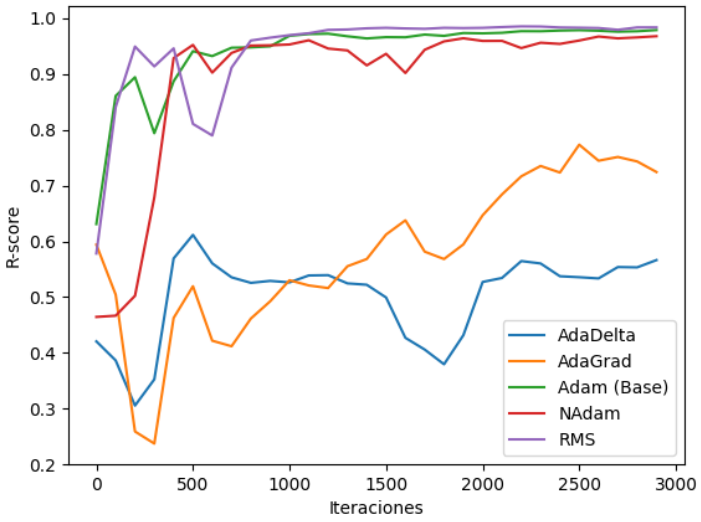

Figura 9.11: Gráfica de resultados: Optimizadores

Adam (utilizado como base) y NAdam destacan por converger rápidamente a valores altos de *R-score*, además de mantenerse estables a lo largo del tiempo. Esto sugiere que ambos son efectivos para este tipo de entrenamiento. RMS también muestra un buen rendimiento, con una rápida subida y estabilidad similar, aunque presenta una ligera fluctuación inicial que se estabiliza rápidamente.

En contraste, AdaGrad presenta una rápida mejora inicial seguida de una caída significativa antes de estabilizarse en un rendimiento intermedio, y aunque finalmente mejora, muestra más variabilidad en comparación con Adam, NAdam y RMSprop. AdaDelta tiene el rendimiento más inestable y bajo durante la mayor parte del entrenamiento, logrando solo mejoras ocasionales y nunca alcanzando niveles altos de R-score de manera consistente. Esto sugiere que AdaGrad y AdaDelta son menos efectivos para este sistema, siendo AdaDelta el optimizador menos recomendable.

Optimizador	**Explotación**	**Porcentaje**
Adadelta	113.45	56.73 %
AdaGrad	146.47	73.23 %
Adam	**190.17**	**95.08 %**
NAdam	175.88	87.94 %
RMS	**191.6**	**95.80 %**

Tabla 9.7: Resultados de explotación de diferentes optimizadores

Con los valores de explotación obtenidos se puede llegar a la conclusión de que el enfoque de uso del momento de Nesterov no es tan bueno en esta implementación como el lineal. También se puede notar que el rendimiento de *RMSprop* es bastante

9.8. Variación de otros hiperparámetros

Una vez tenemos una estructura robusta, tanto en capas como en modelo y técnica, vamos a realizar pruebas al cambiar el ruido y otros hiperparámetros en la fase de exploración.

Se va a reducir la ventana de medias a 50. Esta ventana es la cantidad de elementos que se tienen en cuenta alrededor de un punto concreto de cara a representarlo. Además también se reducirá el número de episodios a 1000 ya que con este número llega a convergencia en el modelo estándar. Con esto se podrá apreciar de una manera más clara el efecto que tienen estos hiperparámetros en el entrenamiento inicial, además de permitir hacer más pruebas en el mismo tiempo.

9.8.1. Variación de los parámetros de aprendizaje

A continuación vamos a variar los parámetros de aprendizaje *alpha* y *beta*. Variando el orden de magnitud de estos parámetros debería alterar la velocidad con la que las redes convergen y su estabilidad.

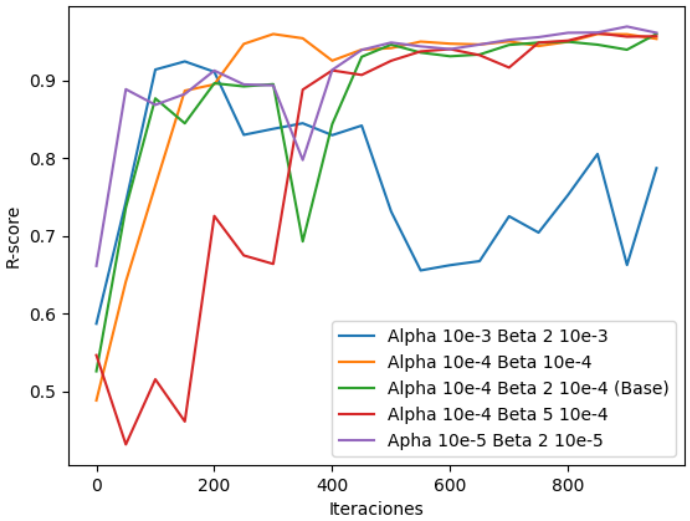

Figura 9.12: Gráfica de resultados: Parámetros de aprendizaje

En la gráfica [9.12] podemos observar que el valor de alpha en relación con beta es importante de cara a mantener una estabilidad. Esto puede deberse a que, al aprender más rápido la red crítica respecto a la de actor, le permite llegar antes a las conclusiones de qué acciones son buenas para los estados. Pese a esto, en el caso de Alpha 0.0001 y Beta 0.0005 esto debe haber funcionado al reves. Esto puede deberse a que ha debido sobrestimar los valores de la función Q para el ratio de aprendizaje.

Además vemos que si estos parámetros son demasiado elevados, puede no alcance nunca la convergencia ya que los saltos en el gradiente son demasiado grandes.

Parámetros	**Explotación**	**Porcentaje**
Alpha 10^{-3} Beta 2*10^{-3}	109.08	54.53 %
Alpha 10^{-4} Beta 10^{-4}	183.55	91.77 %
Alpha 10^{-4} Beta 2*10^{-4} (Base)	190.35	95.18 %
Alpha 10^{-4} Beta 5*10^{-4}	186.65	90.48 %
Alpha 10^{-5} Beta 2*10^{-5}	**190.17**	**95.08 %**

Tabla 9.8: Resultados de explotación de parámetros de aprendizaje

Con estos datos de explotación [9.8], en lo que está a la par con el modelo base en estas pruebas, y debido a su facilidad a la convergencia y relativa estabilidad, consideramos que los valores más óptimos para el modelo son de:

$$Alpha = 10^{-5} Beta = 2 * 10^{-5} \tag{9.5}$$

9.8.2. Variación de Gamma

como ya se ha mencionado en el capítulo de implementación, gamma es el parámetro que establece el peso que representa el target-critic durante el proceso de aprendizaje respecto a la recompensa dada por el episodio. Al bajarlo el sistema actuará teniendo menos en cuenta las predicciones del crítico por mucho que este también aprenda.

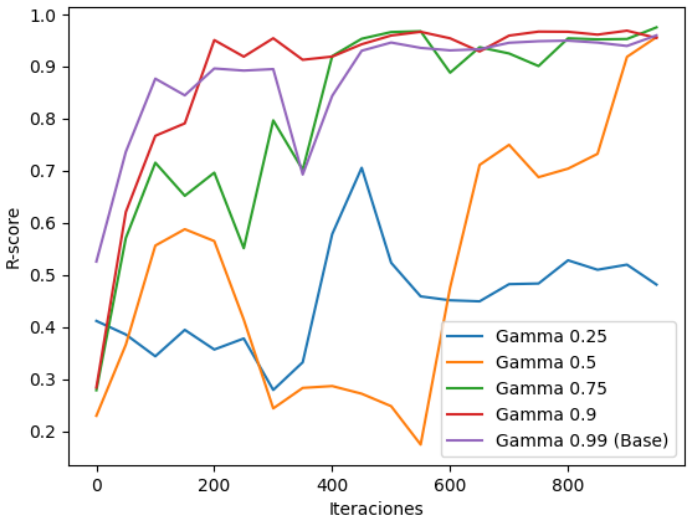

Figura 9.13: Gráfica de resultados: Gamma

Siguiendo la norma de lo explicado teóricamente en la sección sobre el funcionamiento de DDPG [6.4] si el valor de Gamma baja demasiado el proceso se vuelve

inestable. Con esta variación parece volverse más probable que sufra de fallo catastrófico y tarda más en llegar a la convergencia.

Parámetros	Explotación	Porcentaje
Gamma 0.25	123.28	61.64 %
Gamma 0.5	115.79	57.90 %
Gamma 0.75	170.46	85.23 %
Gamma 0.90	**187.96**	**93.98 %**
Gamma 0.99	190.35	95.18 %

Tabla 9.9: Resultados de explotación de Gamma

Según los datos obtenidos, contando tanto los de entrenamiento como de explotación en la tabla [9.9] resulta más interesante tener en cuenta el valor 0.9 para Gamma. Esta consideración es debida a que con este valor no parece sufrir de fallo catastrófico en ningún momento respecto al valor base de 0.99.

9.8.3. Variación de Tau

La variable Tau se encarga de gestionar la actualización de los pesos entre las redes objetivo y las utilizadas como base. Con una puntuación muy alta estas se actualizarían de manera casi simultánea, mientras que si es muy bajo no actualizaría las objetivo.

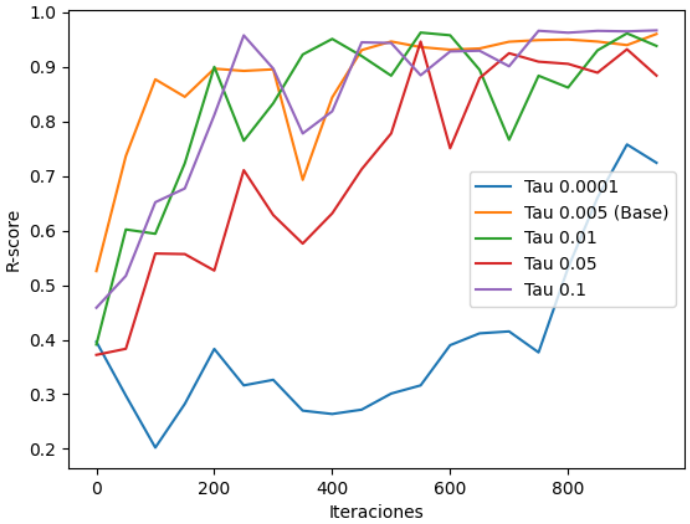

Figura 9.14: Gráfica de resultados: Tau

En la gráfica [9.14] se puede observar como la estabilidad del entrenamiento se pierde para valores de tau más alto, ya que esto aumenta la velocidad de los cambios de las estimaciones.

Parámetros	Explotación	Porcentaje
Tau 0.00001	143.71	71.86 %
Tau 0.005	**190.36**	**95.18 %**
Tau 0.01	165.33	82.67 %
Tau 0.01	165.33	82.58 %
Tau 0.1	184.98	92.49 %

Tabla 9.10: Resultados de explotación de Tau

Los resultados de explotación se acercan más al mejor modelo, pero aun así no son capaces de superarlo. Teniendo en cuenta esto y la estabilidad que su variación lleva al modelo se considerará 0.005 el valor óptimo para Tau.

9.8.4. Variación del tamaño de lote

Creemos que el tamaño de los lotes es algo importante para nuestra implementación, ya que esta altera la cantidad de información que le llega a las redes en cada paso de entrenamiento. Es por esto que un tamaño más grande podría llevar a un sobreajuste, ya que esencialmente estaría aprovechando los mismos datos una y otra vez. Para evitar este tipo de sobreajuste es por lo que elegimos el optimizador Adam entre las otras alternativas [5.2.3].

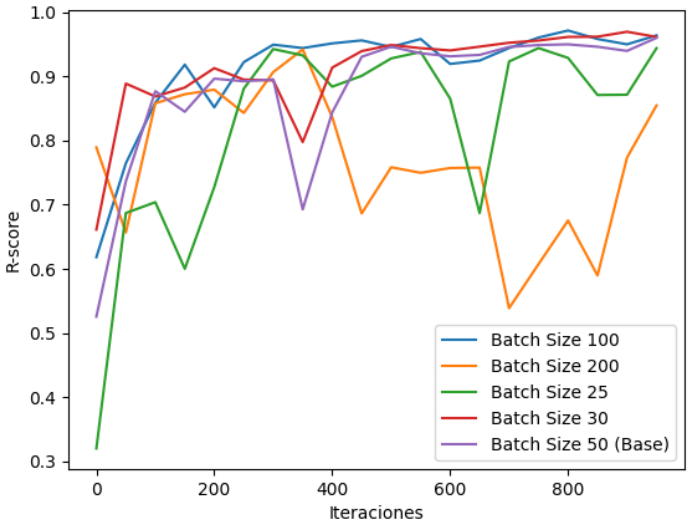

Figura 9.15: Gráfica de resultados: BatchSize

En esta gráfica [9.15] podemos ver como el tamaño de este parámetro no altera en gran medida el entrenamiento, siendo el mejor para este proceso un lote de 100, ya que a partir de este punto parece descender su eficacia, siendo que el de 200 lleva a un fallo catastrófico en el que desaprende la política.

Parámetros	Explotación	Porcentaje
Batch Size = 100	190.98	95.49 %
Batch Size = 200	185.79	92.89 %
Batch Size = 25	187.27	94.64 %
Batch Size = 30	190.18	95.51 %
Batch Size = 50 (Base)	190.35	95.18

Tabla 9.11: Resultados de explotación de tamaño de lote

Sus pruebas de explotación, vistas en la tabla [9.11], nos llevan a pensar que si nuestro objetivo es la precisión, un análisis fino de este elemento no es tan importante. El valor óptimo se considerará que esté entre los valores de 30 a 100 para esta implementación, ya que bajando más se sacrifica la estabilidad del aprendizaje del sistema.

9.8.5. Variación de ruido

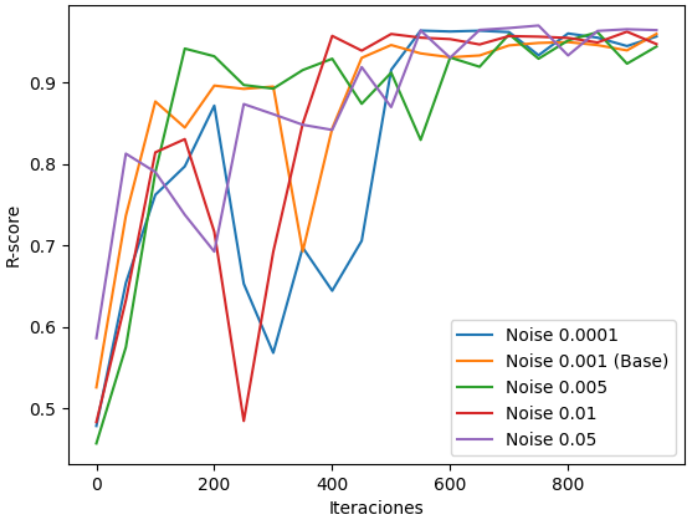

Figura 9.16: Gráfica de resultados: Ruido

Al ver los resultados de esta gráfica [9.16] se puede observar que este parámetro altera principalmente la velocidad de convergencia. Dependiendo de la configuración del ruido, este puede retrasar la convergencia en mayor o en menor medida pero finalmente todos los modelos terminan haciéndolo.

Parámetros	**Explotación**	**Porcentaje**
Noise = 0.0001	190.61	92.89 %
Noise = 0.001	190.35	95.18 %
Noise = 0.005	189.74	94.87 %
Noise = 0.01	**191.02**	**95.51 %**
Noise = 0.05	187.96	93.98 %

Tabla 9.12: Resultados de explotación de ruido

Esta vez los datos de explotación demuestran que para un sistema que priorice la precisión es mejor utilizar un parámetro de ruido mayor aunque este ralentice el entrenamiento del agente al principio, creando un sistema más robusto.

9.9. Mejor modelo encontrado

Después de realizar el estudio de los diferentes parámetros, el mejor modelo encontrado ha terminado teniendo una estructura de tres capas con 250, 150 y 50 neuronas respectivamente, con una función de recompensa densa (distancia euclídea negativa) con *time to goal*.

Este modelo presenta una representación de modelo estable y que converge en solo 1000 episodios a una política óptima. Puede verse su desarrollo de explotación en el vídeo[3] creado para la demostración de sus capacidades.

Este modelo es estable y extremadamente rápido dadas las potencias relativas de los motores, aplicando una muy buena optimización para controlar la dinámica del motor de físicas.

Estas pruebas y análisis nos ha dejado con un modelo que consideramos muy estable y robusto con una puntuación de explotación de **191.3 (95.65 %)**. Tras los ajustes hechos en las fases anteriores los parámetros finales del modelo han sido:

Parámetro	**Valor**
Función de recompensa	Distancia Euclídea Negativa + TTG
Función de activación	ReLu
Optimizador	Adam
Estadística HER	0.8
Ratio de aprendizaje del Actor	10^{-5}
Ratio de aprendizaje del Crítico	$2 * 10^{-5}$
Capas ocultas del Actor	250, 150 y 50: 3 Capas
Capas ocultas del Crítico	250, 150 y 50: 3 Capas
Batch size	100
Buffer size	10^6 de transiciones
Ruido Gausiano	0.01
Gamma	0.90
Tau	0.005

Tabla 9.13: Parámetros para el modelo final óptimo

[3]Link en youtube: `https://youtu.be/53dxIDlTSeE`

Capítulo 10

Conclusiones y Trabajo Futuro

Durante todo este documento se ha establecido la importancia del Aprendizaje por Refuerzo y Deep learning en multitud de aplicaciones, su funcionamiento e intersección entre ellos y con otros tipos de aprendizaje. Es con esto que se llevó a la implementación una de las versiones de AR profundo que identificamos como más interesante para el fin del control robótico, el algoritmo de DDPG.

Este algoritmo que sirve como controlador del brazo robot nos ha permitido obtener un modelo rápido, estable y robusto que se adapta a los objetivos fijados para el mismo. Todo sin utilizar estructuras neuronales más complejas como las convolucionales o las recurrentes, que pueden llegar a requerir una potencia, herramientas y recursos mayores a las secuenciales multicapa.

La adición de HER ha dejado patente la ventaja que su uso provee al utilizarse para solucionar problemas con una recompensa esparcida, pasando de no lograr un entrenamiento que de señales de mejoría a competir con los modelos que han sido prometedores. También se ha visto que este termina mejorando su velocidad de entrenamiento y punto de convergencia a valores más óptimos con la versión que utiliza una formula de recompensa propia con el TTG. Los parámetros finales han sido los que se pueden ver en la tabla [10.1].

Parámetro	**Valor**
Función de recompensa	Distancia Euclídea Negativa + TTG
Función de activación	ReLu
Optimizador	Adam
Estadística HER	0.8
Ratio de aprendizaje del Actor	10^{-5}
Ratio de aprendizaje del Crítico	$2 * 10^{-5}$
Capas ocultas del Actor	250, 150 y 50: 3 Capas
Capas ocultas del Crítico	250, 150 y 50: 3 Capas
Batch size	100
Buffer size	10^6 de transiciones
Ruido Gausiano	0.01
Gamma	0.90
Tau	0.005

Tabla 10.1: Parámetros para el modelo final óptimo. Bis

Con estos datos podemos llegar a la conclusión de que en un sistema DDPG, mejora si se utiliza una recompensa más informada, en este caso con el parámetro temporal. Aunque pueda demostrar cierta inestabilidad en el principio del entrenamiento esta recompensa ha sido capaz de llevar al modelo a precisiones mayores.

Durante este proceso también se ha logrado el objetivo de dar a conocer de qué manera afectan estas variaciones de estructuras, función recompensa e hiper-parámetros al proceso de entrenamiento. Esto se ha vito empíricamente en una implementación con un objetivo final dentro del DDPG que puede servir de base para otros investigadores que se vayan a enfrentar a entornos similares.

Este controlador se podrá utilizar para otras implementaciones reales o simuladas en las que se mantenga en cierto grado la dimensionalidad de las acciones y el estado. Este además no es un modelo estático, ya que aunque el funcionamiento del brazo o sus dimensiones cambiaran, el algoritmo debería ser capaz de adaptarse y modificar el uso de las acciones en consecuencia al continuar el entrenamiento en otro entorno.

En nuevos entornos en los que cambien las dimensiones, grados de libertad o funcionamiento de las tres acciones debería ser posible el uso de este controlador si se realiza un proceso de entrenamiento específico para el entorno. La complejidad del sistema en el control del brazo debería asegurar una capacidad de extrapolación suficiente como para controlar sistemas que no funcionen exactamente igual. Cambios en la dimensionalidad del estado o acciones podrían llegar a funcionar si esta no aumenta demasiado en complejidad, por ejemplo cambiando las acciones del desplazamiento a la fuerza aplicada en las tres dimensiones o directamente a la potencia aplicada a los tres o más motores de movimiento, es decir, grados de libertad que tenga un nuevo brazo robot.

También sería posible realizar otro nuevo tipo de acciones adicionales independien-

tes al movimiento. Dependiendo del modelo del robot manipulador o si esto es más que una acción básica, realizar operaciones diferentes al movimiento podría requerir un controlador. Este podría ser interno al sistema principal o externo, y su objetivo sería controlar como se realiza esta acción. Esto podría ser otra área de investigación a explorar en un futuro. Aplicaciones futuras incluyen casos de uso para robots soldadores o ensambladores para un proceso industrial, que requieren de una precisión de movimiento notable.

Existen también más maneras de definir la recompensa que pueden ser interesantes de estudiar de cara a diferentes problemas. En un entorno con obstáculos o zonas que el efector no debe tocar, como pueden ser paredes, objetos peligrosos o incluso zonas de seguridad, la recompensa deberá ser modificada. Una manera directa de hacer esto es modificar el cálculo de la recompensa para que si el efector está o va a moverse con la acción a una posición dentro de estas áreas reciba una recompensa muy negativa. Con esto se lograría evitar que el sistema entrara en estas zonas al querer maximizar la recompensa para el entorno, evitándolas o pasando alrededor de ellas. Si se necesita que ninguna parte del brazo toque estas áreas se pueden crear puntos adicionales a lo largo de su volumen. Esto se logra para la posición que tiene utilizando matrices de rotación para los valores de sus articulaciones con valores constantes de modificación para sus volúmenes. Esta modificación puede disminuir la velocidad del proceso de entrenamiento inicial ya que se vuelve relevante el tener en cuenta el mayor número de puntos posibles durante el mismo. La mayor consideración de puntos conlleva unos cálculos adicionales, que aunque serían constantes, pueden enlentecerlo si no están bien implementados.

Como punto final, queremos señalar la importancia tanto de la robótica como de aplicaciones de algoritmos avanzados en su control. Con la Inteligencia Artificial cada vez más presente en la sociedad va a ser muy importante centrar nuestros esfuerzos en avanzar en los campos que actúan como puente entre la industria y las versiones teóricas de estos algoritmos.

Especialmente gracias al AR y, sobre todo en los últimos años, el Deep Learning, ahora somos capaces de transmitir el conocimiento de un entorno a agentes flexibles e independientes. Estos, sin conocer nada de antemano, logran tener un control mejor que sistemas PID que requieren de ajustes de grano fino para conseguir un rendimiento similar, lo que se traduce en recursos y dinero invertidos en ello.

Conclusions and Future Work

Throughout this paper, we have established the importance of Reinforcement Learning and Deep learning in a multitude of applications, their operation, and the intersection between them and with other types of learning. With this, one of the versions of deep reinforcement learning that we identified as most interesting for robotic control, the DDPG algorithm, was taken to implementation.

This algorithm, which serves as a controller for the robotic arm, has allowed us to obtain a fast, stable and robust model that adapts to its objectives. All this is done without using more complex neural structures such as convolutional or recurrent neural networks, which may require more power, tools, and resources than sequential multilayer ones.

The addition of HER has shown the advantage that its use provides when used to solve problems with a spread reward, going from not achieving training that shows signs of improvement to competing with models that were already promising. It has also been seen that it ends up improving its training speed and convergence point to more optimal values with the version that uses a dense reward formula with TTG. The final parameters are shown in the table below [10.2].

Parameter	**Value**
Reward function	Negative Euclid distance + TTG
Activation function	ReLu
Optimizer	Adam
HER Statistic	0.8
Actor's Learning Ratio	10^{-5}.
Critic's Learning Ratio	$2 * 10^{-5}$.
Actor's hidden layers	250, 150, and 50: 3 layers.
Critic's hidden layers	250, 150, and 50: 3 layers.
Batch size	100
Buffer size	10^6 of transitions.
Gaussian noise	0.01
Gamma	0.90
Tau	0.005

Table 10.2: Final parameters for the optimal model.

.

With these data, we can conclude that in a DDPG system, it improves if a more informed reward is used, in this case with the temporal parameter. Although it may show some instability at the beginning of training, this reward was able to bring the model to higher accuracies.

During this process, we have also achieved the goal of revealing how these variations of structures, reward functions, optimizers, and hyperparameters affect the training process. This has been seen empirically in an end-goal implementation within DDPG that can serve as a basis for other researchers who will face similar environments.

This controller can be used for other real or simulated implementations where the dimensionality of actions and state is maintained. This is also not a static model, since even if the operation of the arm or its dimensions were to change, the algorithm should be able to adapt and modify the use of the actions accordingly as training continues in other environments.

In new environments where the dimensions, degrees of freedom or operation of the three actions change, it should be possible to use this controller if an environment-specific training process is performed. The complexity of the system in controlling the arm should ensure sufficient extrapolation capability to control systems that do not operate in the same way. Changes in the dimensionality of the state or actions could work if it does not increase too much in complexity, for example by changing the actions from displacement to the force applied in all three dimensions or directly to the power applied to the three or more motion motors, i.e. degrees of freedom that a new robot arm has.

It would also be possible to perform other new types of additional actions inde-

pendent of motion. Depending on the model of the robot manipulator or if this is more than a basic action, performing operations other than motion might require a controller. This could be internal to the main system or external, and its purpose would be to control how this action is performed. This could be another area of research to explore in the future. Future applications include use cases for welding or assembly robots for an industrial process, which require remarkable motion precision.

It would also be possible to perform other new types of additional actions independent of motion. Depending on the model of the robot manipulator or if this is more than a basic action, performing operations other than motion might require a controller. This could be internal to the main system or external, and its purpose would be to control how this action is performed. This could be another area of research to explore in the future. Future applications include use cases for welding or assembly robots for an industrial process, which require remarkable motion precision.

There are also more ways of defining the reward that may be interesting to study for different problems. In an environment with obstacles or areas that the effector should not touch, such as walls, dangerous objects or even safety zones, the reward should be modified. One direct way to do this is to modify the reward calculation so that if the effector is or will move with the action to a position within these areas it receives a very negative reward. This would prevent the system from entering these areas when wanting to maximize the reward for the environment by avoiding them or passing around them. If no part of the arm needs to touch these areas, additional points can be created along its volume. This is accomplished for the position you have by using rotation matrices for your joint values with constant modification values for your volumes. This modification can slow down the speed of the initial training process as it becomes relevant to take into account as many points as possible during training. The increased consideration of points leads to additional calculations, which although they would be constant, may slow it down if they are not well implemented.

As a final point, we would like to point out the importance of both robotics and applications of advanced algorithms in their control. With Artificial Intelligence becoming more and more present in society, it will be very important to focus our efforts on advancing in the fields that act as a bridge between industry and the theoretical versions of these algorithms.

Thanks to reinforcement learning and, especially in recent years, Deep Learning, we are now able to transmit knowledge of an environment to flexible, independent agents. These, without knowing anything in advance, manage to have better control than PID systems that require fine-grained tuning to achieve similar performance, which translates into resources and money invested in it.

Contribuciones Personales

Pablo Pardos Medem

Durante la duración de este trabajo se ha encargado de, en las fases de investigación y documentación previas, recolectar la información relevante para el mismo. Se ocupó de reunir y sintetizar la información encontrada tanto en artículos como en libros recomendados por el director del proyecto para su posterior estudio, y, de ser necesaria, su inclusión en la memoria, siendo el más prevalente de ellos Sutton y Barto (2018) del cual se elaboró un resumen que serviría de base para la redacción de los capítulos cuatro y seis.

Durante los meses posteriores, una vez comprendido el marco teórico en el que se basaría la implementación, este se puso a desarrollar y estudiar el código desde las bases implementadas para otros entornos dentro del marco Gym, entre las que se encontraban frozen lake y taxi.

Se dedicó a adaptar el código aplicando el algoritmo de DDPG al entorno encontrado de péndulo y optimizarlo con varias pruebas y el uso de numpy donde se encontró necesario. Una vez familiarizado con el entorno y tecnologías de este, se dedicó a la implementación de HER para su posterior uso el proyecto del robot manipulador.

Después de realizar las pruebas para el péndulo y confirmar su funcionamiento se pasó a las pruebas del robot manipulador en el que se estuvo trabajando meses para lograr una convergencia dentro de los parámetros considerados como aceptables.

En este proceso fue el que decidió el utilizar un modelo con un objetivo estático sugerido por el director al encontrar problemas en el desarrollo. Fue Pablo el que se encargó de su diferenciación dentro del repositorio y su implementación.

Este también se ha encargado de implementar el código utilizado para realizar todas las pruebas e interpretar los resultados obtenidos con los entrenamientos del brazo robot en el capítulo nueve. Aquí tomó la decisión del optimizador, función de activación, número de capas y de neuronas a utilizar, así como de qué pruebas hacer para el análisis posterior. También este trató de estudiar los rangos delimitados por los

ejemplos de control similares encontrados en la documentación, decidiendo realizar las descritas en el capítulo de control del robot.

En este proceso implementó el código para la generación de gráficas a partir de los datos obtenidos, además de guardar los pesos de las diferentes redes en las pruebas diferenciados por categorías. Incluyó las gráficas y tablas que se obtuvieron con estos datos de las pruebas de explotación para analizarlas en sus secciones correspondientes, así como se dedicó a grabar el vídeo del desarrollo del brazo robot y péndulo. También se ocupó de subirlos a Youtube para incluir sus enlaces en la memoria.

Se ha encargado también de la maquetación de este documento, redactando el resumen, las palabras clave, la introducción con sus tres secciones y la descripción del proyecto, además de ajustar el documento a los requisitos de TFG de la Universidad Complutense de Madrid. También se ha encargado de las traducciones al inglés de las partes relevantes del documento.

De los capítulos principales de este proyecto se ha encargado de la redacción de los capítulos sobre Deep Learning [5], Desafíos en espacio contínuo [6] y de la implementación y análisis del brazo robot [9]. Durante este mismo proceso de redacción se ha ocupado parcialmente de la redacción de los capítulos cuatro, siete y ocho al aportar correcciones, partes del pseudocódigo utilizado y una unificación de estilo en todo el documento.

Tras la corrección de dos tantas de borradores por parte del director del proyecto este se ha encargado de la corrección de erratas y la reescritura de los capítulos cuatro, cinco, seis, ocho y nueve.

Durante esta redacción y posteriores correcciones también se ha encargado de gestionar las referencias bibliográficas, de elementos de consulta, referencia, aplicaciones de los algoritmos, metodologías estudiadas e imágenes, con el objetivo de asegurarse de que estas estén tanto en los puntos como en el formato correctos para su fácil lectura.

Adicionalmente, se ha encargado de componer las imágenes 4.1, 5.1, 5.2, 6.1 y 6.2 en programas de edición de imágenes para ilustrar de una manera clara y sencilla de entender los algoritmos y estructuras tratados.

Como punto final este recogió los datos obtenidos en los puntos anteriores para generar las conclusiones propuestas en su sección. Por parte del trabajo a futuro se encargó de definir y detallar las propuestas de modificaciones para expansión del proyecto. Entre estas modificaciones se encuentran a la función recompensa, uso de diferentes entornos con actuadores complejos y la posibilidad de configuración de puntos adicionales del brazo robot que se encuentran descritas en el capítulo diez del documento.

Carlo Dubini Marqués

Durante la primera parte del trabajo, en los primeros meses del proyecto, se ha encargado de investigar y documentar el funcionamiento de los marcos teóricos del Aprendizaje por Refuerzo y DDPG. Durante la investigación ha utilizado principalmente los artículos y libros que le recomendó el director del proyecto donde Sutton y Barto (2018) es el libro mas relevante entre los que recomendó el director.

Adicionalmente, ha estado practicando para comprender el funcionamiento del lenguaje de programación Python y las diferentes tecnologías como Gym, Keras y TensorFlow hasta conseguir implementar un entorno diseñado para el aprendizaje de estas tecnologías.

Después de haber comprendido las nuevas tecnologías y el marco teórico del proyecto, empezó a utilizar la aplicación de su compañero Pablo, ya que funcionaba mejor en comparación con el que poseía anteriormente, para comenzar con el desarrollo del proyecto.

Él ha sido el encargado de crear el repositorio donde se desarrolló el proyecto y se encargó de adaptar el péndulo en el nuevo entorno de desarrollo (esto es debido a que originalmente se estaba desarrollando en otro entorno) debido a que habían problemas dentro del péndulo para su correcto funcionamiento.

Luego de que el péndulo funcionase dentro del entorno, se ha encargado de que durante la ejecución el entorno muestre la animación del entrenamiento para la visualización directa de como está aprendiendo el agente.

Posteriormente estuvo configurando los parámetros y haciendo pruebas para el correcto funcionamiento del Agente en el péndulo hasta que se consiguió que funcionase.

Durante el desarrollo del brazo robot, este fue el que se dispuso a investigar varios entornos hasta encontrar el entorno Fetch-Reach. El se encargó de adaptar el sistema al nuevo entorno, adaptar las redes neuronales, la administración de los datos que devolvía el nuevo entorno y las funciones auxiliares.

Se ha encargado de la implementación de nuevos diseños de recompensas como la recompensa de la distancia euclídea, la recompensa esparcida y la recompensa cuadrática. En cuanto al modelo de entrenamiento, Ha sido el que ha investigado el HER y junto a su compañero ha implementado el modelo dentro del proyecto.

Además ha estado arreglando varios problemas e implementando nuevas funcionalidades como el almacenamiento de los pesos de las redes neuronales, el diseño de las recompensas, el rediseño de la comunicación entre el entorno y el agente para el correcto funcionamiento del sistema y la optimización del espacio de observación y acciones del actor debido a que el entorno posee acciones que no se utilizan durante

el entrenamiento.

Después de varios meses, ha estado configurando el proyecto y haciendo entrenamientos constantemente para conseguir la convergencia del brazo robot. Él ha estado modificando el diseño de la recompensa para incentivar al agente a explorar en una determinada zona para evitar de que se aleje del objetivo.

En el desarrollo de la memoria, se ha encargado de escribir el capítulo 4 de Aprendizaje por Refuerzo explicando varios conceptos como el Markov Decision Process (MDP), La programación Dinámica y Los algoritmos de Diferencia Temporal (TD) siendo Q-Learning el mas importante de ellos. Utilizando principalmente el libro Sutton y Barto (2018), extrajo los conceptos mas importantes del Aprendizaje por Refuerzo implementados en la memoria.

Ha escrito el capítulo 7 que se encarga de explicar todas las tecnologías que se han utilizado durante el desarrollo de proyecto. Indicando tanto las tecnologías que recomendó el director del proyecto, como Keras, Gym y el lenguaje de programación Python; como tecnologías propias como Gymnasium, los entornos de desarrollo y el motor de físicas MuJoCo.

Adicionalmente se ha encargado de redactar dentro de la memoria el capítulo 8 que contiene toda la Implementación del algoritmo DDPG mas el algoritmo HER en el proyecto, utilizando pseudocódigos para describir cada parte del sistema y diseñar o utilizar varios diagramas para la comprensión de este. Se ha encargado de describir el funcionamiento del programa, el aprendizaje y la toma de decisiones del Agente, el diseño de las recompensas, los parámetros establecidos y la configuración final del agente.

Finalmente en el capítulo 9, Se ha encargado de Introducir el Robot manipulador, explicar su funcionamiento y describir los diferentes diseños de la recompensa. Adicionalmente, Se ha encargado de corregir las el resto del capítulo junto con las correcciones de los resultados.

Bibliografía

Akiake, H. A new look at the statistical model identification. 1974.

Alavizadeh, H., Alavizadeh, H. y Jang-Jaccard, J. Deep q-learning based reinforcement learning approach for network intrusion detection. *Computers*, vol. 11(3), 2022. ISSN 2073-431X.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Pieter Abbeel, O. y Zaremba, W. Hindsight experience replay. vol. 30, 2017.

Ardon, L. Reinforcement learning to solve np-hard problems: an application to the cvrp. 2022.

Belousov, B., Abdulsamad, H., Klink, P., Parisi, S. y Peters, J. *Reinforcement Learning Algorithms: Analysis and Applications*. 2021.

Bengio, Y., Courville, A. y Vincent, P. Representation learning: A review and new perspectives. 2012.

Blischak, J. D., Davenport, E. R. y Wilson, G. A quick introduction to version control with git and github. *PLOS Computational Biology*, vol. 12(1), páginas 1–18, 2016.

Bouhamed, O., Ghazzai, H., Besbes, H. y Massoud, Y. Autonomous uav navigation: A ddpg-based deep reinforcement learning approach. En *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, páginas 1–5. 2020.

Chen, I. Y., Pierson, E., Rose, S., Joshi, S., Ferryman, K. y Ghassemi, M. Ethical machine learning in healthcare. *Annual Review of Biomedical Data Science*, vol. 4(Volume 4, 2021), páginas 123–144, 2021. ISSN 2574-3414.

Creswell, A., White, T., Dumoulin, V., Arulkumaran, K., Sengupta, B. y Bharath, A. A. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, vol. 35(1), páginas 53–65, 2018.

Dankwa, S. y Zheng, W. Twin-delayed ddpg: A deep reinforcement learning technique to model a continuous movement of an intelligent robot agent. En *Proceedings of the 3rd International Conference on Vision, Image and Signal Processing*,

ICVISP 2019. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450376259.

Duchi, J., Hazan, E. y Singer, Y. *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*. Journal of Machine Learning Research, 2011.

Geisslinger, M., Poszler, F., Betz, J., Lütge, C. y Lienkamp, M. Autonomous Driving Ethics: from Trolley Problem to Ethics of Risk. *Philosophy technology*, vol. 34(4), páginas 1033–1055, 2021.

Girosi, F., Jones, M. y Poggio, T. Regularization theory and neural networks architectures. *Neural Computation*, vol. 7(2), páginas 219–269, 1995.

Goodfellow, I., Bengio, Y. y Courville, A. *Deep Learning*. MIT Press, 2016.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P. y Levine, S. Soft Actor-Critic Algorithms and Applications. 2018.

Hasselt, H. Double Q-learning. 2010.

Hung, J.-W., Lin, J.-R. y Zhuang, L.-Y. The evaluation study of the deep learning model transformer in speech translation. En *2021 7th International Conference on Applied System Innovation (ICASI)*, páginas 30–33. 2021.

Hussonnois, M. y Jun, J.-Y. End-to-end autonomous driving using the apex algorithm in carla simulation environment. En *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, páginas 18–23. 2022.

Kemker, R., McClure, M., Abitino, A., Hayes, T. y Kanan, C. Measuring catastrophic forgetting in neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32(1), 2018.

Krizhevsky, A., Sutskever, I. y Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, vol. 25, 2012.

Lee, D. y Lee, S. Motion predictive control for dps using predicted drifted ship position based on deep learning and replay buffer. *International Journal of Naval Architecture and Ocean Engineering*, vol. 12, páginas 768–783, 2020.

Li, C., Zheng, P., Yin, Y., Wang, B. y Wang, L. Deep reinforcement learning in smart manufacturing: A review and prospects. *CIRP Journal of Manufacturing Science and Technology*, vol. 40, páginas 75–101, 2023. ISSN 1755-5817.

Li, X., Xiong, H., Li, X., Wu, X., Zhang, X., Liu, J., Bian, J. y Dou, D. Interpretable deep learning: interpretation, interpretability, trustworthiness, and beyond. *Knowledge and information systems*, vol. 64(12), páginas 3197–3234, 2022.

Li, Z., Peng, X. B., Abbeel, P., Levine, S., Berseth, G. y Sreenath, K. Reinforcement learning for versatile, dynamic, and robust bipedal locomotion control. 2024.

Luo, Z., Liu, H. y Wu, X. Artificial neural network computation on graphic process unit. En *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 1, páginas 622–626 vol. 1. 2005.

Medsker, L. R., Jain, L. R., Lakhmi et al. Recurrent neural networks. *Design and Applications*, vol. 5(64-67), 2001.

Mitchell, T. M. *Machine Learning*. 1997.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. y Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, vol. 518(7540), páginas 529–533, 2015.

Mosavi, A., Faghan, Y., Ghamisi, P., Duan, P., Ardabili, S. F., Salwana, E. y Band, S. S. Comprehensive review of deep reinforcement learning methods and applications in economics. *Mathematics*, vol. 8(10), 2020. ISSN 2227-7390.

Nesterov, Y. A method of solving a convex programming problem with convergence rate o(1/k**2). *Doklady Akademii Nauk SSSR*, 1983.

Noor, F. Advances in Distributed Computing and Artificial Intelligence Journal : 9, Regular Issue 2. 2020.

Panchal, G., Ganatra, A., Kosta, Y. P. y Panchal, D. *Behaviour Analysis of Multilayer Perceptrons with Multiple Hidden Neurons and Hidden Layers*. International Journal of Computer Theory and Engineering, 2011.

Paulus, R., Xiong, C. y Socher, R. A deep reinforced model for abstractive summarization. 2017.

Perera, A. y Kamalaruban, P. Applications of reinforcement learning in energy systems. *Renewable and Sustainable Energy Reviews*, vol. 137, página 110618, 2021. ISSN 1364-0321.

Plappert, M., Andrychowicz, M., Ray, A., McGrew, B., Baker, B., Powell, G., Schneider, J., Tobin, J., Chociej, M., Welinder, P., Kumar, V. y Zaremba, W. Multi-goal reinforcement learning: Challenging robotics environments and request for research. 2018.

Qiu, C., Hu, Y., Chen, Y. y Zeng, B. Deep deterministic policy gradient (ddpg)-based energy harvesting wireless communications. *IEEE Internet of Things Journal*, vol. 6(5), páginas 8577–8588, 2019.

Ramezan, C. A., Warner, T. A., Maxwell, A. E. y Price, B. S. Effects of training set size on supervised machine-learning land-cover classification of large-area high-resolution remotely sensed data. *Remote Sensing*, vol. 13(3), 2021. ISSN 2072-4292.

Rojas, R. *The backpropagation algorithm*. 1996.

Rosenblatt, F. *The Perceptron, a Perceiving and Recognizing Automaton Project Para*. Report: Cornell Aeronautical Laboratory. Cornell Aeronautical Laboratory, 1957.

Russell, S. y Norvig, P., editores. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson Higher Ed, 2016.

Sarkar, D., Bali, R. y Sharma, T. *Practical Machine Learning with Python*. Apress, 2017.

Sebastianelli, A., Tipaldi, M., Ullo, S. L. y Glielmo, L. A Deep Q-Learning based approach applied to the Snake game. *ResearchGate*, 2021.

Shanmugamani, R., Rahman, A. G. A., Moore, S. M. y Koganti, N. *Deep Learning for Computer Vision*. 2018.

Sigaud, O. y Buffet, O. *Markov Decision Processes in Artificial Intelligence*. 2013.

Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning*, vol. 3(1), páginas 9–44, 1988.

Sutton, R. S. y Barto, A. G. *Reinforcement Learning, second edition*. MIT Press, 2018.

Tabor, P. Everything you need to know about deep deterministic policy gradients (ddpg) | tensorflow 2 tutorial. 2020.

Tagesson, D. *A Comparison Between Deep Q-learning and Deep Deterministic Policy Gradient for an Autonomous Drone in a Simulated Environment*. 2021.

Tato, A. y Nkambou, R. Improving adam optimizer. 2018.

Thimm, G. y Fiesler, E. *Neural network initialization*. 1995.

Wang, Y., Gonzalez-Garcia, A., Berga, D., Herranz, L., Khan, F. S. y Weijer, J. v. d. Minegan: Effective knowledge transfer from gans to target domains with few images. En *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

Wilamowski, B. M. *Neural network architectures and learning algorithms*. IEEE Industrial Electronics Magazine, 2009.

Zhang, Q., Lin, M., Yang, L. T., Chen, Z. y Li, P. Energy-efficient scheduling for real-time systems based on deep q-learning model. *IEEE Transactions on Sustainable Computing*, vol. 4(1), páginas 132–141, 2019.