



Deusto

Facultad de Ingeniería
Ingeniaritza Fakultatea

Máster Universitario en Ingeniería Informática

Informatikako Ingeniaritzako Unibertsitate Masterra

Proyecto fin de máster

Master amaierako proiektua

Análisis e implementación de algoritmos de
Aprendizaje por Refuerzo aplicado a entornos con
robots colaborativos

Pablo Martín García



Juan Ignacio Vázquez Gómez

Bilbao, enero de 2021

Resumen

En los últimos años, la Inteligencia Artificial ha logrado un gran desarrollo en todas sus facetas convirtiéndose en uno de los ámbitos de estudio más importantes a día de hoy. Una de sus áreas más relevantes es el Aprendizaje por Refuerzo, con gran utilidad en la automatización y entrenamiento de agentes en ámbitos como los videojuegos o la robótica entre otros muchos.

Este proyecto consiste en el estudio y análisis de algoritmos de Aprendizaje por Refuerzo y, mediante Unity, el modelado de un entorno de trabajo virtual en el que entrenar robots industriales.

Descriptores

Reinforcement Learning, Realidad Virtual, *Machine Learning*, algoritmos, robótica.

Índice de contenidos

1. INTRODUCCIÓN	1
1.1 Objetivos del proyecto	1
1.2 Introducción a RL.....	1
1.2.1 Origen	2
1.2.2 Algoritmos.....	3
1.3 Planificación	6
1.3.1 Fase de investigación	6
1.3.2 Fase de experimentación	7
2. Estado del arte	11
2.1 Algoritmos analizados.....	12
2.1.1 Deep repeated update Q-Network (DRUQN).....	12
2.1.2 Deep loosely coupled Q-Network (DLCQN).....	14
2.1.3 Lenient-DQN (LDQN)	15
2.1.4 Weighted Double Deep Q-Network (WDDQN).....	18
2.1.5 Independent Q-Learning (IQL) + Experience Replay Memory (ERM):	20
2.1.6 Deep Recurrent Q-Network (DRQN) & Enhanced Deep Distributed Recurrent Q-Network (E-DDRQN).....	21
2.1.7 Deep policy inference Q-Network (DPIQN) & Deep recurrent policy inference Q-Network (DRPIQN).....	24
2.1.8 Reinforced inter-agent learning (RIAL) & Differentiable inter-agent learning (DIAL)	28
2.1.9 Multi-task multi-agent RL (MT-MARL):.....	30
2.1.10 Master-slave MARL (MS-MARL):.....	32
2.1.11 Parameter Sharing Trust Region Policy Optimization (PS-TRPO):	34
2.1.12 Action-Specific Deep Recurrent Q-Network (ADRQN):	36
2.1.13 Recurrent MA Deep Deterministic Policy Gradient (R-MADDPG):	38
2.2 Variaciones de esquemas de entrenamiento	41
2.2.1 Agentes modulares.....	41
2.2.2 Curriculum Learning	42

2.2.3	Inyección de ruido.....	42
3.	Experimentación.....	45
3.1	Herramientas.....	45
3.1.1	Entornos y motores.....	45
3.1.2	Librerías.....	46
3.2	Metodología	48
3.2.1	Entrenamiento.....	51
3.2.2	Resultados de la experimentación	57
4.	Conclusiones.....	61
5.	Bibliografía	63

Índice de ilustraciones

Ilustración 1: Diagrama de funcionamiento de RL	2
Ilustración 2: Thesseeus	3
Ilustración 3: Gantt para la fase de investigación	8
Ilustración 4: Gantt fase de experimentación.....	9
Ilustración 5: Algoritmo RUQL.....	13
Ilustración 6: Arquitectura de LDQN	16
Ilustración 7: Reducción de valor temperatura asociado a pares estado-acción en LDQN	18
Ilustración 8: Algoritmo WDDQN.....	19
Ilustración 9: Comparativa de recompensas en WDDQN/LDQN/DDQN	20
Ilustración 10: Comparativas de rendimiento de 5 planteamientos de IQL	21
Ilustración 11: Comparativa de rendimiento tras entrenamiento con observabilidad total en DRQN y DQN	22
Ilustración 12: Comparativa algoritmos DDRQN y E-DDRQN.....	23
Ilustración 13: Comparativa de rendimiento de IQL, DDRQN y E-DDRQN.....	24
Ilustración 14: Comparativa de estabilidad entre DDRQN y E-DDRQN.....	24
Ilustración 15: Algoritmo DPIQN	25
Ilustración 16: Comparativas de rendimiento 1v1 y 2v2	26
Ilustración 17: Arquitectura RIAL y DIAL.....	28
Ilustración 18: Comparativa de desempeño en escenario <i>Switch riddle</i>	29
Ilustración 19: Comparativa de rendimiento en escenario Multi-step MNIST y Colour digit MNIST	30
Ilustración 20: Estructura CERT.....	31
Ilustración 21: Dec-DRQN vs Dec-HDRQN	32
Ilustración 22: Arquitectura MS-MARL.....	33
Ilustración 23: Algoritmo PS-TRPO.....	35
Ilustración 24: Comparativa del rendimiento: <i>Parameter sharing (decentralized), concurrent & centralized</i>	35
Ilustración 25: Rendimiento en escenarios Pursuit y Multi-Walker	36
Ilustración 26: Algoritmo ADRQN.....	37

Ilustración 27: Comparativa de rendimiento en los juegos Pong (1) y Frostbite (2)	37
Ilustración 28: Rendimiento tras generalización POMDP a MDP	38
Ilustración 29: Rendimiento tras generalización MDP a POMDP	38
Ilustración 30: Estructura de modelos <i>actor-critic</i>	39
Ilustración 31: Comparativa de rendimientos en base a la recompensa evaluada y la observabilidad del sistema	40
Ilustración 32: Comparativa de rendimiento con diferentes cantidades de mensajes	40
Ilustración 33: Descomposición de un agente en submódulos	41
Ilustración 34: Comparativa de resultados obtenidos a través del planteamiento modular (Proposed).....	42
Ilustración 35: Comparativa de las curvas de aprendizaje NoisyNet vs DQN, Dueling, A3C	43
Ilustración 36: Escenario 3DBall de ML-Agents	48
Ilustración 37: Escenario de partida para la experimentación	49
Ilustración 38: Componentes de la escena de Unity	49
Ilustración 39: Extracto de código de la clase RobotAgent.cs en la primera aproximación	52
Ilustración 40: Fotograma del entrenamiento bajo la Aproximación 1	53
Ilustración 41: Extracto del código de la clase RobotAgent.cs en la segunda aproximación	53
Ilustración 42: Collider de la pinza del modelo 3D	54
Ilustración 43: Colliders de los objetos <i>FingerA</i> y <i>FingerB</i>	54
Ilustración 44: Asignación de Tag "Finger" para la comprobación de colisiones.....	55
Ilustración 45: Fotograma del entrenamiento bajo la Aproximación 3	55
Ilustración 46: V3 del cubo junto con collider	56
Ilustración 47: <i>Cumulative Reward</i> de los modelos entrenados con las Aprox. 1 y Aprox.2	58
Ilustración 48: <i>Cumulative Reward</i> de los modelos entrenados con las Aprox. 3 y Aprox.4	58
Ilustración 49: <i>Cumulative Reward</i> de los modelos entrenados con las Aprox. 5 y Aprox.6	58
Ilustración 50: <i>Cumulative Reward</i> de los seis modelos entrenados.....	59
Ilustración 51: <i>Policy Loss</i> de los modelos entrenados con las Aprox. 5 y Aprox. 6	59

1. INTRODUCCIÓN

1.1 OBJETIVOS DEL PROYECTO

El proyecto que se presenta en este documento supone un acercamiento al paradigma de *Machine Learning Reinforcement Learning* (RL) a través del análisis de una serie de algoritmos de RL y el entrenamiento de un brazo robótico virtual. Tanto RL como los entornos virtuales son tecnologías vanguardistas y que se encuentran en pleno desarrollo durante estos años. Por ello, y dada la gran compatibilidad entre la Inteligencia Artificial (IA) y la Realidad Virtual (RV), resulta interesante plantear la posibilidad de combinar ambas.

Los objetivos del proyecto contemplaban la implementación de un entorno de trabajo virtual inmersivo para el entrenamiento de personal profesional. Compaginar la actividad laboral con robots puede resultar complejo. Realizar simulaciones del espacio de trabajo permite que las personas aprendan a desempeñar tareas reales en un entorno virtual seguro y controlado. Del mismo modo, es posible entrenar a un agente virtual en la realización de tareas de manera que tenga en cuenta la presencia de un trabajador.

Finalmente, tras revisar los objetivos en varias ocasiones, el producto del proyecto ha resultado en un documento en el que se analizan más de una decena de algoritmos de RL y el entrenamiento de un agente virtual bajo seis aproximaciones diferentes disponible en un repositorio de GitHub¹. El entorno utilizado para el entrenamiento del agente ha sido Unity², en combinación con las librerías de *mlagents*³ y un modelo 3D de un robot diseñado por el equipo de Unity Technologies⁴.

1.2 INTRODUCCIÓN A RL

Ya que prácticamente la totalidad del proyecto está basada en el paradigma RL a continuación se presenta una introducción de cara a entender el contexto en el que surge y su evolución.

Reinforcement Learning es el tercer paradigma dentro de *Machine Learning* (ML) junto con *Supervised Learning* y *Unsupervised Learning*. Cada uno de los tres paradigmas que componen ML son al mismo tiempo una problemática y el campo de estudio que intenta resolverla, pero cada uno de ellos lo hace de manera diferente:

- *Supervised Learning*: algoritmos de aprendizaje basados en ejemplos etiquetados para el entrenamiento de sistemas. La función de sistemas entrenados bajo este enfoque es la de buscar un patrón de comportamiento de los ejemplos analizados para extrapolarlo a ejemplos no estudiados (Caruana, R. et al., 2006).

¹ Repositorio GitHub: <https://github.com/pabsmartin/unity-environment>

² Unity: <https://unity.com/>

³ ML-Agents: <https://github.com/Unity-Technologies/ml-agents>

⁴ Robot 3D: <https://github.com/Unity-Technologies/articulations-robot-demo/>

- *Unsupervised Learning*: utilizado para entrenar sistemas cuya función es la de descubrir una estructura oculta común a los datos analizados (Barlow, H. B., 1989). Este paradigma no utiliza etiquetado en los datos.
- *Reinforcement Learning*: este paradigma basa su aprendizaje en la interacción del agente con el entorno y en la obtención de recompensas o penalizaciones en función de su desempeño. El agente, a través del ensayo-error, perfecciona la toma de decisiones atendiendo a las recompensas que obtiene al ejecutar acciones (Sutton, R. S. et al., 2018).

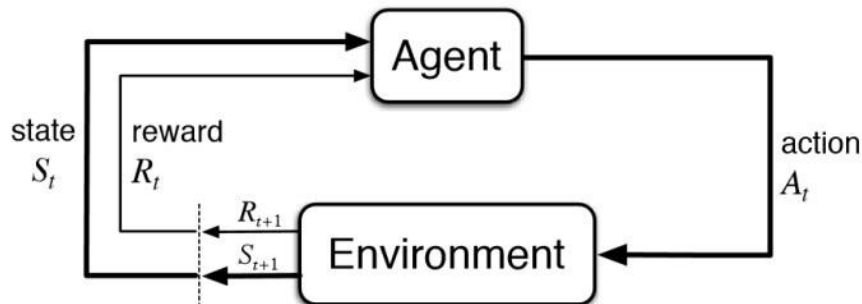


Ilustración 1: Diagrama de funcionamiento de RL

1.2.1 Origen

El inicio de las teorías del aprendizaje por ensayo y error se da en el año 1894 cuando Conway Lloyd Morgan, etólogo y psicólogo, comienza a aplicarlo para describir sus observaciones sobre el comportamiento animal. Edward Thorndike planteó, en relación con esto, la “Ley del Efecto” en la que se analizan cómo la tendencia a tomar ciertas acciones es consecuencia del refuerzo de los eventos derivados de esas acciones (Thorndike, E. L. 1927). Según Ivan Pavlov, el refuerzo debía ser proporcionado con otro estímulo o respuesta en una relación temporal apropiada (Pavlov, I. P. et al., 1941).

Alan Turing utilizó la Ley del Efecto de Thorndike para diseñar un sistema de “placer-dolor” para jugar al ajedrez contra humanos. Sin embargo, no fue posible su implementación en un ordenador real por falta de potencia de cálculo, así que el propio Turing jugó una partida contra un compañero de la universidad simulando las acciones que tomaría la máquina diseñada (Turing, A. M., 1988). En esta partida, en la que el diseño de Turing perdió, se llegaron a tomar plazos de hora y media para calcular la próxima jugada que debería tomar el sistema diseñado por Turing pero supuso un momento destacable por la calidad de la partida desarrollada a pesar de la derrota y una de las razones por la que, de ahí en adelante, la inteligencia artificial volvió a estar en auge.

En 1952 Claude Shannon (Gallager, R. G., 2001) mostró el funcionamiento de un ratón de laberinto llamado Thesaurus que usó el “ensayo y error” para encontrar su camino a través de un recorrido, recordando la ubicación de imanes y relés dispuestos bajo el suelo del propio laberinto.

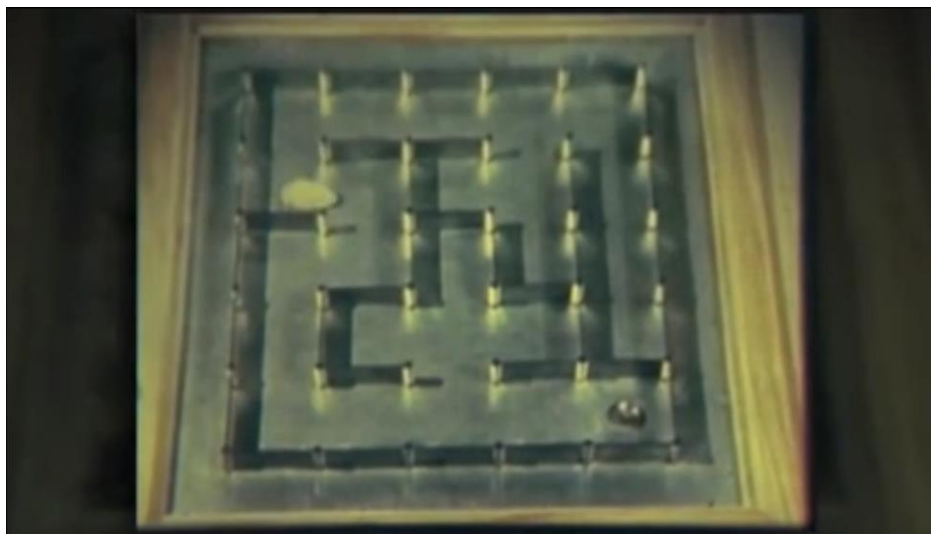


Ilustración 2: Thesseus

La investigación sobre el aprendizaje genuino de prueba y error se convirtió en poco habitual en los años 60 y 70. A pesar de esto, en los años 60 los términos "refuerzo" y "aprendizaje por refuerzo" se utilizaron por primera vez en la literatura relacionada con la ingeniería para describir los usos de la ingeniería en el aprendizaje por ensayo y error. Widrow, Gupta y Maitra (Widrow, B., et al., 1973) produjeron una regla de aprendizaje por refuerzo que podía aprender de las señales de éxito y fracaso en lugar de hacerlo de los ejemplos de entrenamiento. En lugar de aprender de la imitación, aprendía de la interacción.

Tras unos años, el matemático Harry Klopff "revivió" el hilo de la investigación sobre el aprendizaje por refuerzo (Harmon, M. E. et al. 1995). Klopff establecía que faltaban aspectos hedónicos del comportamiento en el aprendizaje de los agentes. Algunos de estos aspectos son el impulso para lograr objetivos, controlar el entorno para lograr resultados deseados y alejarse de los no deseados. Este planteamiento introduce los conceptos de la exploración y la explotación, claves en el aprendizaje por refuerzo y en la teoría detrás de los algoritmos desarrollados.

1.2.2 Algoritmos

A pesar de que en el *Capítulo 2: Estado del arte* se analicen con mayor profundidad una serie de algoritmos de RL, en este apartado se presentan las principales categorías de algoritmos, sus clasificaciones y varias consideraciones que se tienen en cuenta a la hora de plantear un algoritmo.

Los entrenamientos de modelos utilizando algoritmos de RL, se llevan a cabo resolviendo problemas de la forma de un Proceso de Decisión de Markov (MDP) (Feinberg, E. A. et al. 2012). Un MDP representa un proceso que puede describirse a través de la tupla $\{S, A, T, R, \gamma\}$. S representa los estados s en los que se encuentra el entorno a cada instante de tiempo ($s \in S$), A representa el conjunto de acciones a disponibles para el agente ($a \in A$), T representa la probabilidad de transición entre estados en función de la acción escogida por el agente ($P(s' | s, a) = T(s, a, s')$), R hace referencia al espacio de recompensas que el agente puede obtener y γ indica el factor de descuento ($\gamma \in [0, 1]$).

Es posible encontrarse con entornos que reúnan las condiciones idóneas para realizar el entrenamiento. Incluso, es posible que las condiciones favorezcan el aprendizaje y aporten

ventajas. Sin embargo, esto ocurre solamente en algunas ocasiones y, a medida que los planteamientos y los entornos se acercan a situaciones reales, aparecen más condicionantes que complican el aprendizaje del agente (o los agentes).

- Observabilidad parcial

Los agentes no pueden obtener la información completa de los estados s que pertenezcan al espacio de estados S . En estas ocasiones el aprendizaje de los agentes se ve perjudicado y no resulta óptimo al carecer de información. Los MDPs pasan a ser considerados PO-MDPs (*Partially Observable MDPs*). Algunos algoritmos incorporan la recurrencia en sus redes neuronales de aprendizaje de cara a solucionar esta problemática.

- Estocasticidad

Este fenómeno se da en los casos en los que los estados en los que se presenta el entorno son muy inestables. Estas situaciones son características de los sistemas multiagente puesto que las observaciones que realizan los agentes en un instante de tiempo se ven enormemente modificadas en el instante siguiente por las acciones tomadas por el resto de los agentes que actúan en el entorno.

- Coste computacional del entrenamiento

El entrenamiento de un solo agente puede llegar a ser muy caro en términos computacionales por la cantidad de recursos utilizados o el tiempo que toma completarlo. Esto puede ser considerado en ocasiones como un problema y se ve agravado en los entornos en los que es necesario entrenar a múltiples agentes.

En estos casos, existen técnicas de transferencia de conocimiento o de compartición de parámetros mediante las que los agentes facilitan el aprendizaje del resto de componentes del sistema. Algunas de estas técnicas se desarrollan más a fondo en el *Capítulo 2*.

El problema de la estocasticidad del entorno supone una traba significativa en el aprendizaje de los agentes debido a la diferencia entre estados sucesivos y la incapacidad de establecer comportamientos razonables para observaciones tan diferentes. Sin embargo, como se analiza más adelante, en varios algoritmos se plantean alternativas para solucionar este fenómeno. Del mismo modo, de cara a reducir los perjuicios que pueda suponer la observabilidad parcial de los estados del entorno, algunos algoritmos introducen la posibilidad de que los agentes que componen el sistema puedan desarrollar protocolos de comunicación para transmitir sus diferentes observaciones y tomar decisiones con más información, llegando a tener una visión completa del estado en el que se encuentran.

Al final del segundo capítulo, se clasifican los algoritmos analizados en base a los siguientes criterios:

1. **Observabilidad del entorno:** los MDPs pueden tomar la forma de POMDP si los agentes no perciben el estado en el que se encuentra el entorno al completo.
2. **Toma de decisiones centralizada o descentralizada:** bajo ciertos planteamientos los agentes pueden realizar sus acciones en base a decisiones centrales o propias.

3. **Espacio de acciones continuo o discreto.**
4. **Conducta cooperativa o competitiva:** los agentes pueden entrenarse de manera que busquen un mayor rendimiento al realizar una tarea común o buscando obtener éxito frente a otros agentes del entorno (videojuego Pong por ejemplo)
5. **Agentes homogéneos o heterogéneos:** los agentes pueden ser homogéneos y tener un mismo comportamiento o ser diferentes entre sí y cumplir diferentes roles y tareas.
6. **El algoritmo está implementado o no.**

1.2.2.1 Elementos de los algoritmos

- Política

Determina la manera en la que el agente se comporta en un estado determinado. En otras palabras, indica la acción que el agente tomará en el estado s .

$$u = \pi_{\theta}(u|s)$$

Si el entorno es estocástico y no se conoce con certeza la acción que se va a tomar en un estado concreto, la política del algoritmo indicará la probabilidad con la que se tomará una acción para ese estado.

$$p(u|s) = \pi_{\theta}(u|s)$$

- Recompensa

Como se ha mencionado anteriormente, los agentes aprenden qué acciones son mejores en su entrenamiento. Cuánto mayor sea la recompensa, mejor resultado habrá obtenido el agente con la acción tomada. Las recompensas se asignan para una acción en un estado y el objetivo del agente es el de maximizar la recompensa total obtenida para una política.

$$r(s, a)$$

- Función de valor

La función de valor es la recompensa máxima que el agente será capaz de acumular desde el estado en el que se encuentra hasta el final del episodio. La diferencia real entre la recompensa y la función de valor es que la recompensa es inmediata mientras que la función de valor está relacionada con la idoneidad de visitar un estado de cara a conseguir una recompensa óptima.

Un ejemplo de este progreso entre estados se puede identificar en las situaciones las que una acción proporcione una recompensa no-óptima de todas las disponibles en ese estado, pero sirva de punto de partida para llegar a otros estados futuros que otorguen recompensas mayores.

$$V(s)$$

- Factor de descuento
Determina el peso de la recompensa estimada futura.

$$R_{t+1} + \gamma R_{t+2} + \dots + \gamma R_{t+n}$$

- Modelo
Un modelo representa la manera en la que el entorno evolucionará partiendo de un estado y tras tomar una acción. El modelo puede existir previo al entrenamiento o construirse a lo largo del mismo.
Existen algoritmos que no utilizan modelos del entorno, es un elemento opcional de cara a entrenar a un agente. Sin embargo, los algoritmos que sí lo hacen lo utilizan para definir los controles de cara a maximizar las recompensas.

1.3 PLANIFICACIÓN

El proyecto se ha dividido en dos fases que se han desarrollado de manera simultánea a lo largo de varias semanas. A continuación, se describe cada una de ellas y se presentan los diagramas de Gantt correspondientes. La duración total del proyecto es de 3 meses.

1.3.1 Fase de investigación

En esta primera fase, se plantea realizar una aproximación al ámbito del RL a través del análisis y resumen del funcionamiento de una quincena de algoritmos y métodos de entrenamiento.

1. Definición de requisitos
Junto con el tutor del proyecto, introducción al ámbito de ML desde el punto de vista de RL. Planteamiento de los objetivos del proyecto y definición de los requisitos.
2. Selección de los algoritmos
RL cuenta con muchas aproximaciones y enfoques diferentes. Por lo que se han seleccionado alrededor de quince algoritmos que se presentan en el *Capítulo 2* y que componen el estado del arte del RL.
3. Búsqueda de información
A través de Google Scholar, principalmente, y otros repositorios de documentos académicos se han identificado y seleccionado una lista de *papers* publicados por otros investigadores. Todas las referencias utilizadas se encuentran recogidas en la sección correspondiente de este documento.
4. Redacción del documento resumen de los algoritmos
Tras analizar la información recopilada y resumir los documentos, se ha redactado un documento en el que se presentan las mejoras y nuevos enfoques

que propone cada uno de los algoritmos a las diferentes situaciones problemáticas que encuentran.

1.3.2 Fase de experimentación

1. Documentación

De cara a entrenar el modelo, un requisito básico era la búsqueda de plataformas de desarrollo que permitiesen representar un entorno virtual en el que diseñar el escenario en el que entrenar al agente. Por otra parte, se valoraron diferentes librerías de ML que redujesen la complejidad de implementar un modelo desde cero.

2. Pruebas con el entorno

Una vez que se seleccionaron la plataforma (*Unity*) y la librería (*ML-Agents*) que se utilizarían en los diferentes entrenamientos del modelo, se dedicaron varios días a experimentar con los entornos de prueba que ofrecen ambos proyectos. La familiarización con los comandos de entrenamiento y la gestión de ficheros facilitaron el desarrollo posterior.

3. Experimentación con el modelo

Se desarrolló un plan de entrenamiento del modelo dividido en 4 fases en las que se probaron diferentes aproximaciones al problema. Se presentan la metodología aplicada y los resultados obtenidos.

1. INTRODUCCIÓN

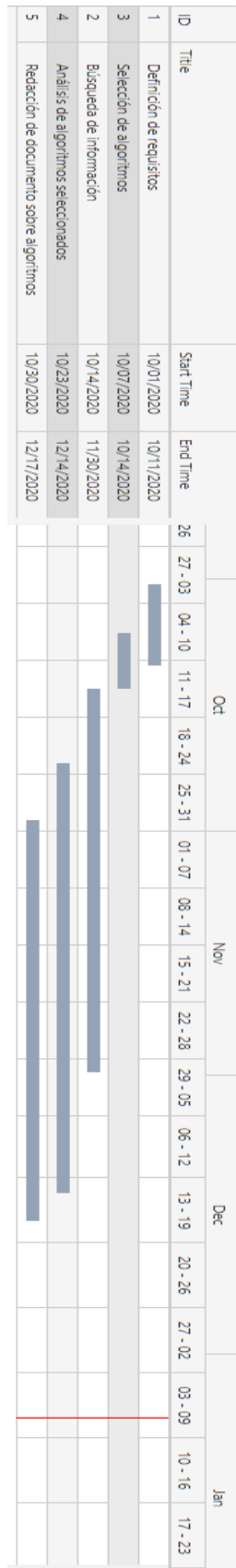


Ilustración 3: Gantt para la fase de investigación

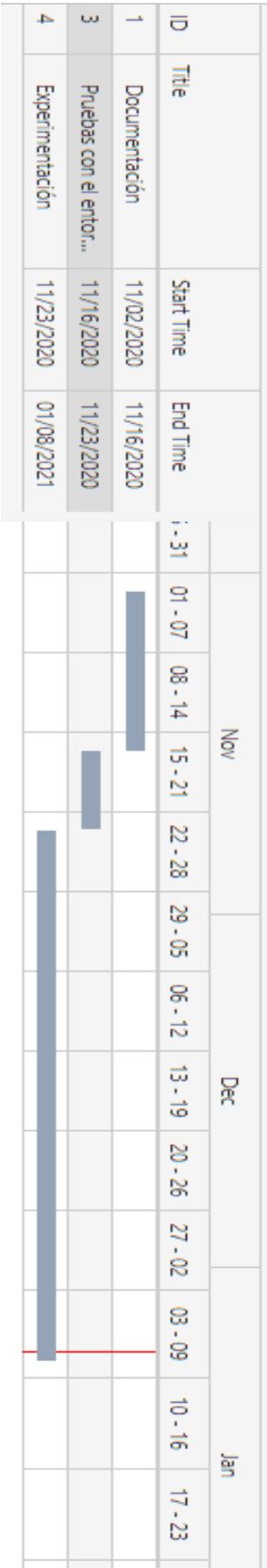


Ilustración 4: Gantt fase de experimentación

2. ESTADO DEL ARTE

Este capítulo muestra el trabajo realizado en la fase de investigación del proyecto.

Como se he comentado anteriormente, los algoritmos de RL están basados en planteamientos con la forma de un MDP (Proceso de Decisión de Markov) y están compuestos por diferentes elementos (política, modelo, recompensa, factor de descuento, ...). En este capítulo se analizan una quincena de algoritmos a partir de investigaciones y experimentación realizada en los últimos años para presentar el estado del arte del RL centrado en sistemas multiagente además de la manera en la que afronta cada uno los problemas derivados de la observabilidad parcial, la estocasticidad de los entornos o la comunicación entre agentes.

Algunos de los algoritmos documentos analizados plantean mejorar aproximaciones anteriores, mientras que otros innovan en la manera en la que solucionan problemáticas no exploradas previamente. Para todo esto se utilizan, para las fases de experimentación, entornos basados en videojuegos de estrategia, entornos personalizados o creados específicamente para probar el algoritmo planteado.

- **Método basado en política**

Uno de los tres métodos a través del que se entrenan los agentes es el basado en política. Una política (s, a) representa la acción a que un agente debe tomar en el estado s . El planteamiento basado en política se encarga de almacenar el mapeo $s \rightarrow a$ a lo largo del aprendizaje.

- **Método basado en valor**

Los métodos basados en valor, al contrario que los basados en política, aprenden valores asociados a la recompensa de las acciones. La política de un agente puede derivarse de estos valores, pero no es explícita. Es un método más estable que el método basado en política, pero tarda más en converger.

- **Método actor-critic**

Este método es mezcla de los dos anteriores en la que dos redes neuronales se encargan de aprender la política y los valores Q al mismo tiempo. Esto favorece los tiempos de ejecución puesto que, en un espacio de acción continuo o muy amplio, se podría utilizar la política (almacenada de manera explícita) en lugar de recorrer la tabla de valores Q .

2.1 ALGORITMOS ANALIZADOS

2.1.1 Deep repeated update Q-Network (DRUQN)

Basado en el algoritmo Q-Learning, Repeated Update Q-Learning (RUQL) propone una variación (o mejora) de este algoritmo a raíz de los problemas que supone el sesgo que realizan las políticas a la hora de actualizar valores Q .

En el algoritmo de Reinforcement Learning (RL) Q-Learning, que pretende identificar la calidad de una acción en un estado de cara a alcanzar la mayor recompensa futura, se parte de una tabla de valores asociados a las recompensas que obtendrá un agente si toma acciones en diferentes estados. Las acciones y estados son finitas y pertenecen respectivamente a un espacio de acciones A y de estados S . A medida que avance en las ejecuciones, el agente actualiza los valores correspondientes a cada par (s, a) . Sin embargo, aunque en entornos con un solo agente no influye mucho, existe un sesgo en las acciones dependiente de la política de un agente que cae en la explotación de acciones en estados ya explorados. La exploración del agente se ve reducida y, en entornos no-estacionarios (entre los que se encuentran los entornos multiagente), genera reacciones en cascada.

La política consiste en la “estrategia” que tiene un agente a la hora de desempeñar una tarea. Dicho de otra manera, se trata de la probabilidad que un agente tiene de tomar una acción en un estado concreto. De esta manera, una vez que se ha encontrado una acción muy prometedora (más que el resto de las acciones), esta se actualiza más a menudo que el resto de acciones para un estado. Este problema se agrava en entornos no estacionarios puesto que acciones que previamente fueron óptimas puede que hayan dejado de serlo en una nueva ejecución, pero se seguirán seleccionando por su estimación positiva.

Para ponerle solución a esta problemática, surgió el algoritmo FAQL (Frequently-Adjusted Q-Learning) que potencia la actualización de acciones menos probables a ser escogidas en un estado. Sin embargo, la ecuación que se plantea para este algoritmo se ve limitada cuando la probabilidad de tomar una acción en un estado es cercana a cero. Por este motivo se debe añadir un parámetro de control que convierte la Ecuación 1 en la Ecuación 2.

$$Q^{t+1}(s, a) = Q^t(s, a) + \alpha (r + \gamma \max_{a'} Q^t(s, a') - Q^t(s, a)) \quad (1)$$

Ecuación 1: Actualización de valores Q en *Q-Learning*

$$Q^{t+1}(s, a) = Q^t(s, a) + \min(1, \frac{\beta}{\pi(s, a)}) \alpha (r + \gamma \max_{a'} Q^t(s', a') - Q^t(s, a)) \quad (2)$$

Ecuación 2: Actualización de valores Q en FAQL

De todas formas, el parámetro β , a pesar de que resuelve una problemática, trae consigo dos propiedades perjudiciales: una relativa al factor de aprendizaje, que previamente era calculado mediante α , pero que ahora depende también de β ($\alpha\beta$), y la otra que supone el mismo problema que pretendía resolver y que ocurre cada vez que ocurre $\pi(s, a) < \beta$.

(Abdallah, S. & Kaisers, M., 2013) proponen inicialmente un enfoque diferente basado en la actualización recursiva en función a la probabilidad de elección de una acción. En lugar de centrarse en el peso de la actualización, este planteamiento se centra en la cantidad de veces que se actualiza un valor. Sin embargo, bajo esta aproximación, cuanto más cercano sea $\pi(s, a)$ a cero, existe el riesgo de que el tiempo de computación sea infinito. Por este motivo, finalmente el algoritmo RUQL plantea una ecuación de actualización de los valores de Q con la siguiente forma:

$$Q^{t+1}(s, a) = [1 - \alpha]^{\frac{1}{\pi(s, a)}} Q^t(s, a) + [1 - (1 - \alpha)^{\frac{1}{\pi(s, a)}}] [r + \gamma Q^t(s', a_{max})] \quad (3)$$

Ecuación 3: Actualización de valores Q en DRUQN

Como se explica en su análisis teórico, RUQL da más peso a los valores obtenidos por la recompensa actual que a los valores esperados para esa acción en ese estado debido a que el valor queda desactualizado por recibir actualizaciones de manera poco frecuente.

Los planteamientos teóricos de RUQL se han puesto a prueba utilizando tres experimentos en los que se ha observado un rendimiento más estable del algoritmo RUQL frente a Q -Learning y FAQL de manera general para situaciones de un único agente.

Algorithm 5 Repeated Update Q-Learning: Intuition

```

1: Initialize to arbitrary  $Q(s, a)$ 
2: Observe current state  $s$ 
3: repeat
4:   Compute policy  $\pi$  using  $Q(s, a)$ 
5:   Select an action  $a$  according to policy  $\pi(s, a)$ 
6:   Execute action  $a$ 
7:   Observe reward  $r$  and next state  $s'$ 
8:   for  $\lfloor \frac{1}{\pi(s, a)} \rfloor$  times do
9:      $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_a Q(s', a') - Q(s, a)]$ 
10:  end for
11:  Set  $s \leftarrow s'$ 
12: until Termination
    
```

Ilustración 5: Algoritmo RUQL

2.1.2 Deep loosely coupled Q-Network (DLCQN)

A la hora de trabajar en entornos cooperativos, es necesario que los agentes que realicen tareas de este carácter lo hagan de manera coordinada y en busca del beneficio del total de los agentes. Se alcanzará un beneficio óptimo en la medida en que las acciones que tomen los agentes sean beneficiosas para el desarrollo adecuado de la actividad. Sin embargo, en los casos en los que las acciones que realicen los agentes no sean óptimas para el correcto funcionamiento del sistema general, será necesario determinar el grado de implicación que debe tener un agente con el resto de los agentes que componen el sistema.

Dicho de otra manera, para cada estado en el que se encuentran los agentes de un sistema, es necesario especificar el grado de independencia de cada agente respecto del resto de agentes. Como expone (Castaneda, A. 2016) en su tesis, *“Las creencias de un agente para actuar de forma independiente en un estado determinado se ajustan en relación con los resultados negativos que recibe. En cada estado en el que se recibe una recompensa negativa, se determina el alcance de la responsabilidad de un estado anterior”*.

Es importante determinar el grado de independencia de un agente, además de para maximizar el beneficio obtenido por las acciones que se tomen de manera conjunta, porque la cantidad de estados y acciones a tener en cuenta, dentro de los espacios de estados S y espacios de acciones A , crecen con cada agente que participe en el proceso de toma de decisiones. De esta manera, cuantos más agentes independientes se identifiquen para un estado concreto, un gran problema distribuido podrá ser gestionado abordando subproblemas más sencillos. Los agentes que componen el sistema aprenden de manera dinámica sus grados de participación en cada estado del proceso.

Para ello se evalúa si los estados que se alcanzan son parte del proceso por el que se recibe recompensas negativas o si, por el contrario, alcanzar esos estados favorecen recompensas positivas. En caso de que no sean beneficiosos, será necesario identificar los estados desfavorables. Estos estados que acercan al sistema a una recompensa negativa pertenecen al conjunto S_i^k . La Ecuación 4 en la que $\zeta(s, s^*)$ representa la similitud entre dos estados, indica el grado en el que un estado s contribuye a alcanzar una recompensa en el estado s^* .

$$f_{s^*}^r(s) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\zeta(s, s^*)^2}$$

Ecuación 4: Grado de contribución de un estado en DLCQN

El grado de cooperación de un agente se aprende de manera dinámica para cada estado en función de su implicación en acciones que resulten en estados $s \in S^C$ a través de la Ecuación 5 en la que se tiene en cuenta su grado de cooperación en el salto de tiempo anterior, su participación en la transición a un estado que encamina al sistema a una recompensa negativa (ε_i^k) y la similitud entre estados.

$$\varepsilon_i^k(t+1) = \begin{cases} \gamma^\lambda \varepsilon_i^k(t) + 1 & \text{if } s_i^k \in S^c \\ \gamma^\lambda \varepsilon_i^k(t) & \text{otherwise} \end{cases}$$

Ecuación 5: Cálculo del parámetro ε

Gracias al grado de independencia de los agentes se pueden plantear enfoques diferentes para resolver situaciones teniendo en cuenta el algoritmo Q-Learning para entornos conjuntos y entornos individuales. Este enfoque de coordinación (Coordinated learning) incluye aprendizajes Q_i y $Q_{i,j}$ que se actualizan en función de si el grado de independencia es alto (Ecuación 6) o si es bajo y el agente es percibido por el resto de los agentes del sistema (Ecuación 7).

$$Q_i(s_i, a_i) \leftarrow Q_i(s_i, a_i) + \alpha [r_i + \gamma \max_{a_i} Q_i(s'_i, a'_i) - Q_i(s_i, a_i)]$$

Ecuación 6: Actualización de los valores Q con grado de independencia alto en DLCQN

$$Q_c(js_i, a_i) \leftarrow Q_c(js_i, a_i) + \alpha [r_i + \gamma \max_{a_i} Q_i(s'_i, a'_i) - Q_c(js_i, a_i)]$$

Ecuación 7: Actualización de los valores Q con grado de independencia bajo en DLCQN

Cuando se lleva este algoritmo a un planteamiento de *Deep Learning*, las funciones que se han definido anteriormente cambian ligeramente de significado. Un agente, en lugar de calcular su responsabilidad sobre el resultado de una acción, el agente calculará la influencia de la información compartida por otro agente. Las variables que definen los comportamientos individuales o cooperativos de los agentes pasan a depender principalmente de la observabilidad respecto de otros agentes. Es por este motivo que Ψ_k (el grado de cooperación de un agente), aumenta en caso de que se reciba una recompensa negativa al mismo tiempo que percibe a otros agentes y disminuye en el caso de que no los perciba.

2.1.3 Lenient-DQN (LDQN)

Leniency se refiere a la capacidad de perdonar o pasar por alto una falta. En el contexto del *Reinforcement Learning*, con *Lenient Learning* los agentes de estos sistemas mapean pares estados-acción a valores de temperatura $[(s, a), t]$ que serán los que determinen la indulgencia que tendrá la política al actualizar valores de recompensa para esos estados. De cara a reducir la indulgencia que aplica el agente a la actualización del valor de la recompensa para cada par (s, a) , cada vez que uno de estos pares es visitado, se reduce su temperatura. Así, el sistema se muestra optimista con las recompensas no exploradas y se acerca a un resultado más realista a medida que visita más un par estado-acción.

Los algoritmos clasificados como *Deep Q-Learning* se basan en información almacenada en una memoria de experiencias previas del agente (*Experience Replay Memory*: ERM) para ser entrenadas. Esta memoria, sin embargo, es contraproducente para la resolución de uno de los desafíos principales de los entornos multiagente. La existencia de múltiples agentes en un entorno colaborativo supone dificultad para la convergencia y la selección de una política óptima para un agente debido a que el resto de las políticas de los demás agentes están cambiando constantemente.

En el planteamiento realizado por (Palmer, G., et al. 2017) se demuestra cómo almacenando el valor de indulgencia en la ERM un sistema multiagente de *Deep Reinforcement Learning* converge para alcanzar una política óptima en entornos cooperativos que necesitan de la coordinación de los agentes que lo constituyen.

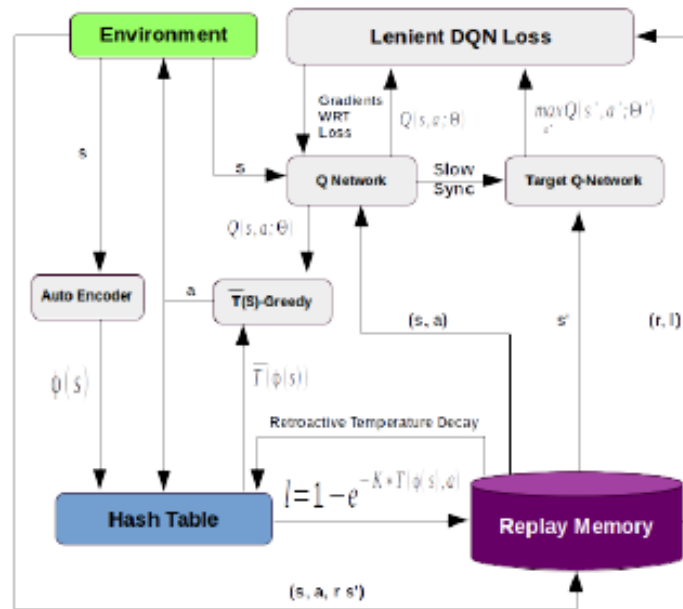


Ilustración 6: Arquitectura de LDQN

Se comprobó la eficacia del algoritmo en entornos cooperativos utilizando un experimento del tipo *Coordinated Multi-Agent Object Transportation Problem* (CMOTP) en contraste con el algoritmo *Hysteretic Q-Learning* (HQL). Este último algoritmo, introduce un parámetro nuevo a la ecuación (β) para tener en cuenta un doble factor de aprendizaje en acciones conjuntas. HQL es un buen algoritmo que aplicar en entornos deterministas. Sin embargo, cuando se incluye un factor estocástico (la recompensa de las acciones, en este caso) se ha demostrado que HQL no obtiene el mismo rendimiento. Esto ocurre debido al parámetro que se comenta anteriormente que introduce una dependencia sobre el resto de los agentes.

Inicialmente planteado por (Potter, M., et al. 1994), la indulgencia de los agentes permite que el sistema converja en políticas óptimas globales, en lugar de políticas óptimas locales. Las temperaturas altas asociadas a pares estado-acción hacen que el agente perdone en mayor medida acciones subóptimas, pero a medida que se van visitando estados más a menudo, se

reduce la temperatura del par estado-acción y el agente es menos permisivo, mientras que se mantiene optimista para los estados no explorados. La Ecuación 8 indica cómo se calcula la indulgencia de un agente y está asociada a un único par estado-valor (s_t, a_t) y a un factor de moderación K .

$$l(s_t, a_t) = 1 - e^{-K * T_t(s_t, a_t)}$$

Ecuación 8: Cálculo de indulgencia de un agente en LDQN

Entornos con espacios de estado continuos no permiten un enfoque en el que se almacenen valores de temperatura o de recompensa para infinitos pares estado-acción. Por este motivo, se plantea un *hashing* de los estados (entendiendo el estado como una serie de datos y tuplas con observaciones) para la agrupación de estados y representado a través de (ϕ_s).

En el trabajo de (Palmer, G., et al. 2018), a la hora de aplicar *Lenient Q-Learning* a *Deep Learning*, se observó que a pesar de utilizar la indulgencia de los agentes para decidir qué transiciones de estados se almacenaban, la exploración inicial y las temperaturas altas de todos los estados terminaron por no ser productivas y se seleccionaron transiciones aleatorias. Por este motivo, se decidió incluir la indulgencia en la tupla almacenada en la ERM. Dado que la indulgencia $l(s_t, a_t)$ depende de la temperatura del par estado-acción, en caso de que no exista temperatura asociada a un estado (ϕ_s), se añade una temperatura máxima.

Otra de las problemáticas que se encuentra este algoritmo es que en el caso de que los agentes vuelvan a la misma situación de partida a cada iteración del proceso, las temperaturas de los primeros estados descenderán de manera significativamente mayor que las temperaturas de los pares estado-acción finales (más cercanos a recompensas globales mayores). Para prevenir esto, se utiliza la aproximación ATF (*Average Temperature Fold*) con la que se dobla la temperatura de las acciones disponibles en un estado previo a otro terminal. Por otra parte, si existe una tarea secundaria compleja en la que los agentes se queden estancados y se reduzcan de manera rápida las temperaturas de las acciones asociadas a esos estados, será necesario controlar el grado de enfriamiento de estas acciones para estabilizar el proceso de aprendizaje de los agentes. Para ello se regula la reducción de temperatura de manera que sea menor en los estados iniciales que en los estados terminales del proceso. Esta aproximación se llama *Retroactive Temperature Decay Schedule* (TDS). En el siguiente gráfico se observa cómo afecta cada planteamiento al enfriamiento de las temperaturas del sistema en el experimento CMOTP. Se puede apreciar cómo TDS reduce el grado de enfriamiento en las fases iniciales a diferencia de ATF.

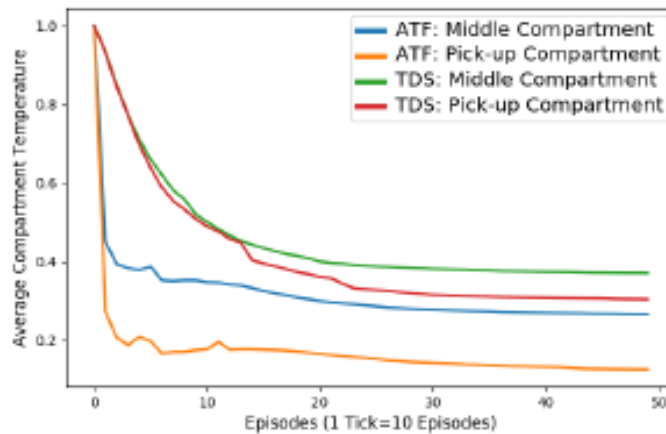


Ilustración 7: Reducción de valor temperatura asociado a pares estado-acción en LDQN

2.1.4 Weighted Double Deep Q-Network (WDDQN)

Este algoritmo basado en dos redes neuronales paralelas incorpora *Lenient Q-Learning* (LQL) y una estrategia llamada *Scheduled Replay Strategy* (SRS) para mejorar el rendimiento de agentes en entornos multiagente estocásticos y cooperativos. (Zheng, Y., et al. 2018) demuestran que WDDQN genera mejores resultados a la hora de alcanzar una política óptima para entornos con las características planteadas mejores que LQL (analizado anteriormente).

WDDQN surge a raíz de dos problemas: la estocasticidad, producida por el ruido que generan los agentes en el sistema y que resulta en sesgos; y la no-estacionalidad, debida al dinamismo de los agentes (cambios en sus políticas constantes) que afecta al *experience replay* por la rápida obsolescencia de experiencias pasadas. Basado en LQL, se procesa el sistema de recompensas de WDDQN utilizando *Lenient Reward Network* (LRN) para atajar la descoordinación de los agentes, puesto que una acción óptima de un agente puede ser subestimada si la recompensa general de todos los agentes se ve reducida por acciones perjudiciales del resto. Por otra parte, SRS se incorpora a este algoritmo para controlar que los estados que se tomen en cuenta de cara al *experience replay* sean estados tardíos y de darles mayor prioridad a medida que se acercan más al estado terminal del episodio. De esta manera acelera la convergencia del sistema hacia una política óptima y mejora su rendimiento. Estas mejoras se pueden apreciar más adelante en los experimentos realizados.

Además, *Weighted Double Deep Q-Network* utiliza dos redes neuronales para estimar valores Q relativos a la acción que se ha de tomar y al valor Q objetivo, respectivamente. Para hacer este cálculo, se procesa la información de la memoria de experiencias, que almacena tuplas del tipo (s, a, r, s', d) , a través de la Ecuación 9. Utilizando dos redes neuronales para la estimación de valores Q , balanceadas por el valor se reduce cualquier sesgo existente y se equilibra la sobreestimación de los valores.

Algorithm 1 WDDQN

```

1: The maximum number of episodes:  $Max_E$ , the maximum number of steps:  $Max_S$ , global memory:  $D^G$ , episodic memory:  $D^E$ , reward network:  $R^N$ , deep Q-networks:  $Q^U$  and  $Q^V$ 
2: for episode = 1 to  $Max_E$  do
3:   Initialize  $D^E$ 
4:   for step = 1 to  $Max_S$  do
5:      $a \leftarrow \max_{a'} \frac{Q^U(s, a') + Q^V(s, a')}{2}$  (with  $\epsilon$ -greedy)
6:     Execute  $a$  and store transitions into  $D^E$ 
7:     Sample mini-batch  $(s, a, r, s')$  of transitions from  $D^G$ 
8:     Update  $Q^U$  or  $Q^V$  randomly
9:     if update  $Q^U$  then
10:       $a^* \leftarrow \arg \max_a Q^U(s', a)$ 
11:       $Q^U_w(s', a^*) \leftarrow \beta Q^U(s', a^*) + (1 - \beta)Q^V(s', a^*)$ 
12:       $Q^{Targat}(s, a) \leftarrow R^N(s, a) + Q^U_w(s', a^*)$ 
13:      Update network  $Q^U$  towards  $Q^{Targat}$ 
14:     else
15:       $a^* \leftarrow \arg \max_a Q^V(s', a)$ 
16:       $Q^V_w(s', a^*) \leftarrow \beta Q^V(s', a^*) + (1 - \beta)Q^U(s', a^*)$ 
17:       $Q^{Targat}(s, a) \leftarrow R^N(s, a) + Q^V_w(s', a^*)$ 
18:      Update network  $Q^V$  towards  $Q^{Targat}$ 
19:     end if
20:     Update  $R^N$  according to transitions in  $D^G$ 
21:   end for
22:   Store  $D^E$  into  $D^G$ 
23: end for

```

Ilustración 8: Algoritmo WDDQN

$$Q(s, a)^{U, WDDQ} = \beta Q^U(s, a^*) + (1 - \beta)Q^V(s, a^*)$$

Ecuación 9: Actualización de valores Q en WDDQN

En los experimentos multiagente realizados en el estudio de (Zheng, Y., et al. 2018) se compara el rendimiento alcanzado por WDDQN en entornos deterministas y estocásticos. En el primero de ellos se compara con dos versiones del propio algoritmo (WDDQN sin SRS y WDDQN sin SRS ni LRN) y en el caso del entorno estocástico se compara con LQL y DDQN. Los resultados de estos experimentos, en los que dos agentes debían encontrar una política de coordinación, muestran que, en entornos deterministas, WDDQN y WDDQN con LRN alcanzan la política óptima, pero que WDDQN lo hace en un número inferior de episodios. Sin embargo, en entornos estocásticos, a pesar de que LQL alcanza una política y se estabiliza de manera más rápida, no alcanza una política óptima. Sin embargo, WDDQN consigue alcanzar una política óptima con una recompensa media del doble de valor que la hallada por LQL.

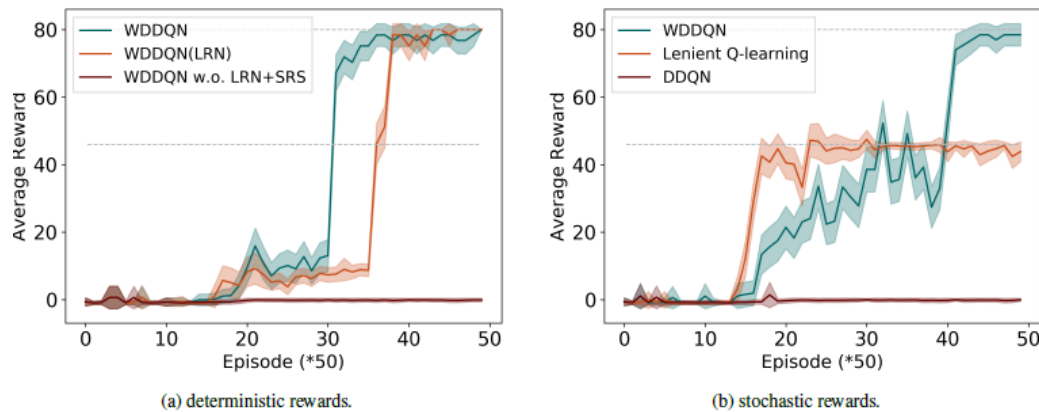


Ilustración 9: Comparativa de recompensas en WDDQN/LDQN/DDQN

2.1.5 Independent Q-Learning (IQL) + Experience Replay Memory (ERM):

Independent Q-Learning (IQL) es uno de los grandes algoritmos de *Reinforcement Learning* en el que cada agente del sistema aprende su propia política mientras que considera al resto de agentes parte del entorno. Este planteamiento, al mismo tiempo que resuelve el problema de la escalabilidad en sistemas de aprendizaje centralizado, introduce la problemática de la no-estacionalidad del entorno debido a las variaciones que realizan el resto de los agentes. Sin embargo, se ha demostrado de manera práctica la efectividad de este algoritmo en el que los agentes pueden realizar un seguimiento del resto de agentes. A pesar de esto, a la hora de trasladar IQL a un entorno de *Deep Reinforcement Learning* (DRL) surge una incompatibilidad entre los dos planteamientos.

DRL se basa en una memoria de experiencias previas del agente para entrenar la red neuronal, de tal manera que si un agente aprende su política en base a lo que captura de un entorno no-estacionario, la presencia de una memoria que almacena experiencias previas confundirá al agente en su aprendizaje con experiencias obsoletas. Para corregir estas carencias de la combinación de IQL y ERM (*Experience Replay Memory* en la que se basa DRL) Jakob Foerster plantea, junto con otros investigadores, dos correcciones (Foerster, J., et al. 2017).

La primera, *importance sampling*, considera ampliar la tupla de información almacenada en la ERM con la probabilidad de una acción conjunta en base a las políticas que estén en uso en ese momento. De esta manera, se descartan las experiencias en la medida en que van quedando obsoletas para no obstaculizar el aprendizaje. La segunda idea planteada, *fingerprint*, consiste en que cada agente aprenda su política apoyado en una estimación del comportamiento del resto de agentes. Al utilizar redes neuronales para construir su política, es imposible que un agente aprenda la configuración de todas las redes neuronales de cada agente. Por eso, se identifican dos parámetros que se añaden a la información con la que se alimenta la función Q y que deben estar relacionadas con el valor real de los pares estado-acción marcados por las políticas del resto de agentes. Foerster y su equipo consideraron que los dos parámetros a tener en cuenta debían ser el número de iteración e y la tasa de exploración ϵ . Estas opciones se consideraron por variar ligeramente a lo largo del entrenamiento y permitir que el modelo se generalice a través de experiencias en las que los demás agentes ejecutan políticas de calidad variable a medida que aprenden.

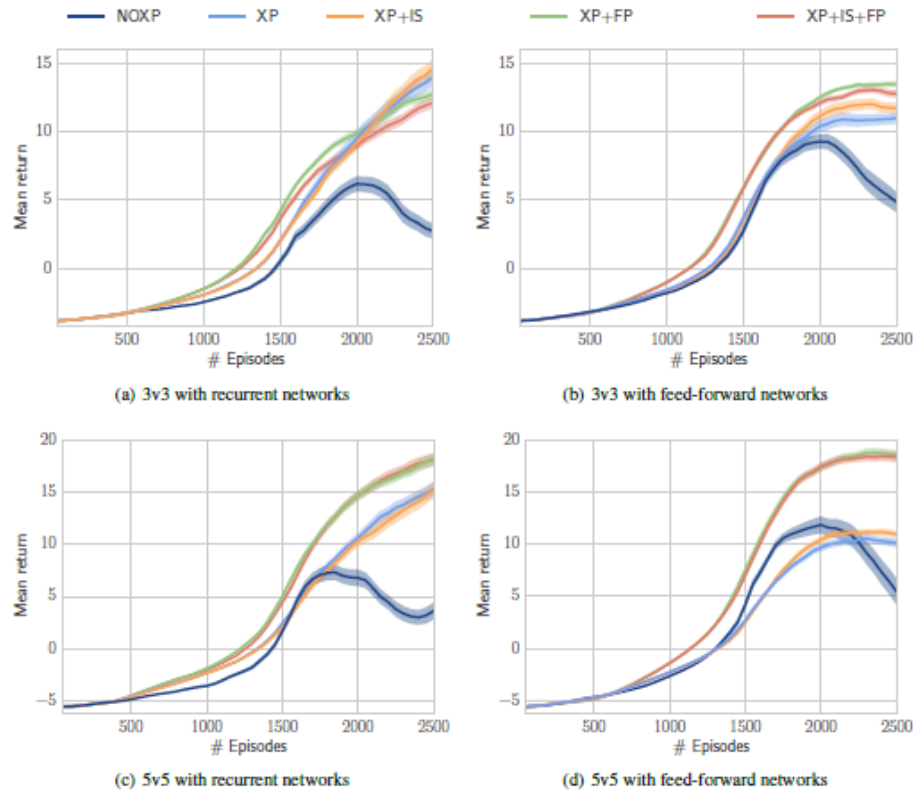


Ilustración 10: Comparativas de rendimiento de 5 planteamientos de IQL

Para realizar los experimentos se utilizó el videojuego StarCraft y un escenario con tres personajes, cada uno controlado por un agente diferente (en un caso de uso normal, el jugador controla todas las unidades al mismo tiempo con una acción conjunta a todos los personajes seleccionados). Las recompensas recibidas se basaron en la suma de daño infligido a unidades enemigas y la salud total del escuadrón de personajes. Los resultados del estudio demuestran cómo la combinación de las técnicas planteadas (XP: *experience replay*, IS: *importance sampling* y FP: *fingerprint*), obtienen un mayor rendimiento que los algoritmos que no las aplican. La diferencia entre los resultados se agranda en entornos de mayores dimensiones.

2.1.6 Deep Recurrent Q-Network (DRQN) & Enhanced Deep Distributed Recurrent Q-Network (E-DDRQN)

Partiendo de la base que aporta DQN, Matthew Hausnecht y Peter Stone introducen un planteamiento nuevo a la escena de los algoritmos de RL para el entrenamiento de agentes en entornos con observabilidad parcial (Hausknecht, M., et al. 2015).

DQN utiliza una red neuronal entrenada mediante las últimas cuatro observaciones del agente. Según Hausknecht y Stone, este algoritmo no será eficaz en entornos en los que el agente necesite recordar más de cuatro estados para tomar decisiones. En esos casos, el agente se enfrentaría a un entorno de observabilidad parcial. Por este motivo, se plantea la combinación

del algoritmo DQN con los avances realizados en el ámbito de las Redes Neuronales Recurrentes a través de una DQN y una memoria de largo plazo (LSTM). Gracias a esta memoria incorporada, *Deep Recurrent Q-Learning* (DRQN) es capaz de gestionar mejor la pérdida de información que sufre DQN. DRQN modifica la arquitectura de DQN sustituyendo la primera capa de la red por una memoria LSTM recurrente.

Los Procesos de Decisión de Markov con Observabilidad Parcial (POMDP) son los planteamientos que mejor representan los problemas del mundo real debido a que la amplia mayoría de estos se dan en circunstancias de observabilidad parcial. Esta afirmación corrobora la siguiente expresión: $Q(o, a | \theta) \neq Q(s, a | \theta)$. La estimación de un valor Q realizada a través de una observación parcial o será diferente de la realizada a través de una observación total del estado del entorno s . Añadiendo recurrencia, se puede aproximar mejor los valores Q tras secuencias de observaciones consiguiendo así mejores políticas para los agentes.

Como estrategia de actualización de la red neuronal se utilizó *Bootstrap Random Update*. En esta estrategia se muestrea la memoria de experiencias de manera aleatoria y se aplican las actualizaciones en puntos aleatorios del episodio. Este planteamiento, además, requiere establecer el estado oculto de la LSTM a cero al inicio de cada iteración, lo que hace que la red tarde más tiempo en entrenarse. Sin embargo, la otra estrategia (*Bootstrap Sequential Update*) viola la política de DQN de muestreo aleatorio de la memoria, por lo que no es una opción adecuada.

Las pruebas de rendimiento de este algoritmo se probaron con juegos de Atari en los que las observaciones de la pantalla realizadas por el agente tenían un 50% de probabilidad de ser totalmente negras. De esta manera, se añade el factor de la observabilidad parcial a este entorno. Un agente lo suficientemente robusto debería ser capaz de obtener resultados satisfactorios bajo estas condiciones. En el siguiente gráfico se muestra cómo DRQN mejora considerablemente el rendimiento y las puntuaciones respecto de DQN en el entorno descrito. Además, se puede observar cómo no sólo supera a DQN en ejecuciones con observabilidad parcial, sino que en situaciones con observabilidad prácticamente total también obtiene mejores resultados.

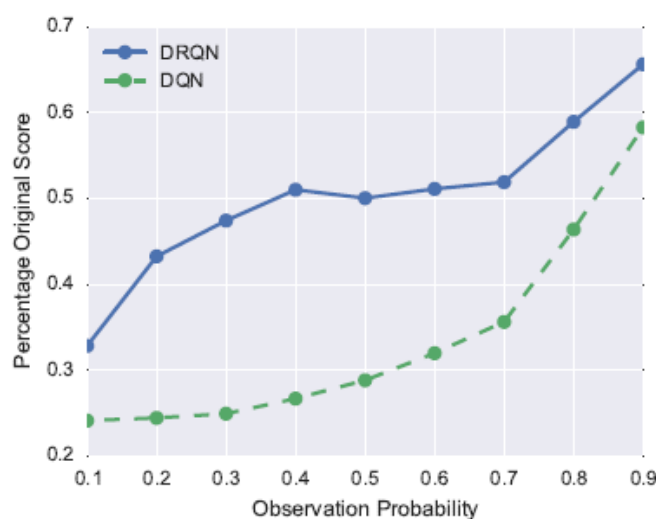


Ilustración 11: Comparativa de rendimiento tras entrenamiento con observabilidad total en DRQN y DQN

A pesar de los resultados obtenidos utilizando DRQN, el planteamiento más cercano a *Deep Reinforcement Learning* en entornos multiagente de observabilidad parcial, es la combinación entre DRQN y IQL (*Independent Q-Learning*). Sin embargo, este método se ve superado por DDRQN (*Deep Distributed Q-Network*) debido a que aporta tres modificaciones:

- Realimentar la red neuronal con la última acción. Los agentes deben asociar historias acción-observación para la construcción de políticas.
- Compartir pesos de la red neuronal. Esta acción reduce significativamente la cantidad de parámetros que deben aprender los agentes. Sin embargo, a pesar de utilizar la misma red neuronal, cada agente utilizará sus observaciones, por lo que se mantendrán comportamientos diferenciados.
- Inhabilitar experience-replay memory (ERM). A pesar de que sea una parte beneficiosa de DQN, en entornos multiagente en los que se pierde la estacionalidad del entorno una característica así haría que experiencias que quedan rápidamente obsoletas dificultasen el aprendizaje de los agentes.

Algorithm 1 DDRQN

```

Initialise  $\theta_1$  and  $\theta_1^-$ 
for each episode  $e$  do
   $h_1^m = 0$  for each agent  $m$ 
   $s_1 = \text{initial state}$ ,  $t = 1$ 
  while  $s_t \neq \text{terminal}$  and  $t < T$  do
    for each agent  $m$  do
      With probability  $\epsilon$  pick random  $a_t^m$ 
      else  $a_t^m = \arg \max_a Q(o_t^m, h_{t-1}^m, m, a_{t-1}^m, a; \theta_i)$ 
    Get reward  $r_t$  and next state  $s_{t+1}$ ,  $t = t + 1$ 
   $\nabla \theta = 0$   $\triangleright$  reset gradient
  for  $j = t - 1$  to  $1$ ,  $-1$  do
    for each agent  $m$  do
       $y_j^m = \begin{cases} r_j, & \text{if } s_j \text{ terminal, else} \\ r_j + \gamma \max_a Q(o_{j+1}^m, h_j^m, m, a_j^m, a; \theta_i^-) \end{cases}$ 
      Accumulate gradients for:
       $(y_j^m - Q(o_j^m, h_{j-1}^m, m, a_{j-1}^m, a_j^m; \theta_i))^2$ 
     $\theta_{i+1} = \theta_i + \alpha \nabla \theta$   $\triangleright$  update parameters
     $\theta_{i+1}^- = \theta_i^- + \alpha^- (\theta_{i+1} - \theta_i^-)$   $\triangleright$  update target network
  
```

Algorithm Enhanced DDRQN

```

initialize  $\theta_1$  and  $\theta_1^-$ 
initialize replay memory D
For each episodes  $e$  do
   $h_1^m = 0$  for each agent  $m$ 
   $\hat{s} = \text{initial state}$ ,  $t = 1$ 
  While  $\hat{s}_t \neq \text{terminal}$  and  $t < T$ 
    For each agent  $m$  do
      With probability  $\epsilon$  pick random  $a_t^m$ 
      Else  $a_t^m = \arg \max_a Q(\tau, a)$ 
    Get reward  $\hat{r}$  and next state  $\hat{s}_{t+1}$ ,  $t = t + 1$ 
    Store transition  $\mathcal{G} = \langle \hat{S}, A, \hat{P}, \hat{r}, \hat{Z}, \hat{O}, m, \gamma \rangle$  in D
    Sample random minibatch of transition  $\mathcal{G}$  from D
    For each agent  $m$  do
      set  $y_i^m = \hat{r}_i^m$  if episode terminates
      =  $\hat{r}_i^m + \gamma \max_a Q(\tau, a)$  otherwise
      Accumulate gradient for  $L(\theta)$ 
    Every C steps let  $\theta_1^- = \theta_1$ 
  End for

```

Ilustración 12: Comparativa algoritmos DDRQN y E-DDRQN

El último de los planteamientos de DDRQN, inhabilitar la ERM supone que la convergencia de la red se ralentiza y, además, resulta en una estrategia de exploración menos eficiente. El planteamiento *Enhanced* DDRQN (E-DDRQN) modifica ligeramente el planteamiento de DDQN para considerar al resto de agentes como información no perteneciente al entorno (Fan, L., et al. 2018). De esta manera, es posible utilizar la ERM para mejorar la velocidad de convergencia de la red.

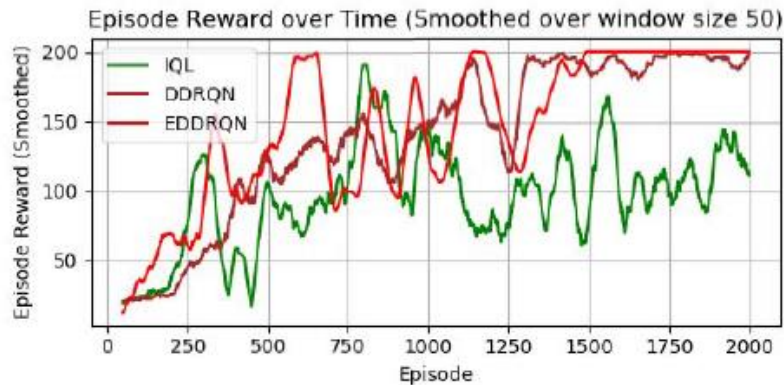


Ilustración 13: Comparativa de rendimiento de IQL, DDRQN y E-DDRQN

Tras realizar experimentos en un entorno SC2LE (*StarCraft 2 Learning Environment*), tanto DDRQN como E-DDRQN mostraron mejores resultados que IQL alcanzando una recompensa mayor como se puede observar en la Ilustración 12. Pero, además, E-DDRQN consiguió una convergencia más rápida y una estabilidad mayor que DDRQN a lo largo de un mismo número de episodios. Como conclusión del estudio se recalca la importancia de la ERM en los algoritmos de RL.

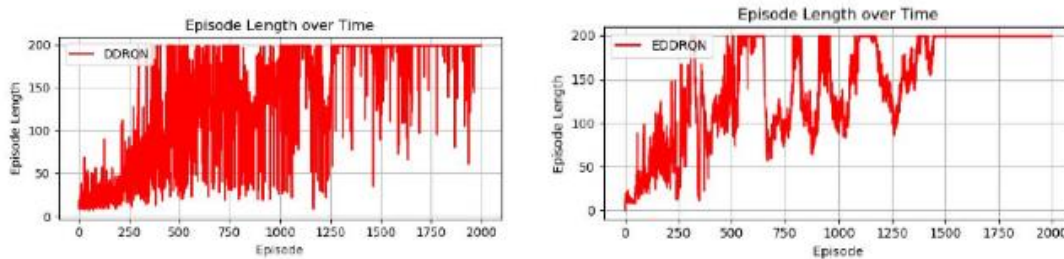


Ilustración 14: Comparativa de estabilidad entre DDRQN y E-DDRQN

2.1.7 Deep policy inference Q-Network (DPIQN) & Deep recurrent policy inference Q-Network (DRPIQN)

En el año 2018 se desarrolla una adaptación del algoritmo *Deep Q-Network* (DQN) con el objetivo de generar una solución para el entrenamiento de agentes con políticas variables. En entornos multiagente en los que existen tanto agentes colaboradores como agentes oponentes, es necesario que los agentes controlados actualicen sus políticas para ajustarse a las estrategias de los demás agentes que componen en el entorno y para tratar estas situaciones surge DPIQN. Añadiendo un vector oculto de información compuesta por observaciones realizadas del resto de

agentes, este algoritmo es capaz de predecir las políticas del resto de agentes y obtener mejores valores Q para la toma de decisiones (Hong, Z., et al. 2018).

Tanto DPIQN como DRPIQN (ampliación de DPIQN para entornos con observabilidad parcial de la misma manera que DRQN amplía DQN) están entrenados mediante módulos adaptativos que separan la inferencia de las políticas del resto de agentes del entrenamiento para la búsqueda de sus propios valores Q . Los dos algoritmos optan por el aprovechamiento de los comportamientos únicos del resto de agentes dando por hecho que no existe conocimiento del dominio previo. Por este último motivo, además, es por lo que DPIQN y DRPIQN son generalizables a diferentes dominios a diferencia de otros algoritmos, como indican los autores de estos algoritmos.

En entornos multiagente, en los que los estados varían con las decisiones de todos los agentes que componen el sistema, la función Q de un agente es dependiente de estas acciones. Sin embargo, para el cálculo de la función Q óptima (Q_M^*) no suponga una explosión computacional se tiene en cuenta las políticas conjuntas del resto de agentes (π_o). El objetivo principal de DPIQN es la mejora en la representación que realiza un agente de los estados en sistemas multiagente. A raíz de esto, se añaden las observaciones sobre las acciones del resto de agentes, como se ha mencionado anteriormente, en forma de vector (h^{PI}) al modelo del agente a entrenar. Al inicio de la fase de entrenamiento, el agente entrenado se centra, principalmente, en el aprendizaje del vector de características de las políticas hasta que la entropía entre la observación y la acción tomada (L^{PI}) se ve reducida en beneficio del factor de aprendizaje λL^Q , gracias a que $\lambda = \frac{1}{\sqrt{L_t^{PI}}}$. A medida que L^Q crece, el aprendizaje se centra más en optimizar los

valores Q . Como se demuestra más adelante, DPIQN realiza un mejor trabajo gracias al factor λ que sin él. El agente aprovecha el conocimiento obtenido sobre las políticas del resto de agentes para realizar mejores decisiones.

Algorithm 1 Training Procedure of DPIQN

- 1: Initialize replay memory Z , environment \mathcal{E} , and observation s
 - 2: Initialize network weights θ
 - 3: Initialize target network weights θ^-
 - 4: **for** timestep $t = 1$ to T **do**
 - 5: Take a with ϵ -greedy based on $Q_M(s, a; \theta)$
 - 6: Execute a in \mathcal{E} and observe μ_o, r, s'
 - 7: Clip r between $[-1, 1]$
 - 8: Store transition (s, a, a_o, r, s') in Z
 - 9: Sample mini-batch $(s^J, a^J, a_o^J, r^J, s^{J'})$ from Z
 - 10: Set $y = \begin{cases} r^J & \text{for terminal } s^{J'} \\ r^J + \gamma \max_{a^{J'}} Q_M(s^{J'}, a^{J'}; \theta^-) & \text{otherwise} \end{cases}$
 - 11: Compute L^Q based on y , and compute L^{PI}
 - 12: Compute $\lambda = \frac{1}{\sqrt{L_t^{PI}}}$
 - 13: Compute gradient G based on L^Q, L^{PI}, λ
 - 14: Perform gradient descent G
 - 15: Update $s = s'$
 - 16: Update $\theta^- = \theta$ for every C steps
 - 17: **end**
-

Ilustración 15: Algoritmo DPIQN

En la medida en que agentes del entorno actualizan sus políticas y cambian las acciones que toman, es necesario reducir el ruido que generan estos cambios en el aprendizaje de los agentes entrenados. Para esto, DRPIQN utiliza una única observación como *input* a cada escalón de tiempo y la almacena en una memoria de largo plazo. De esta manera, DRPIQN es capaz de conseguir una representación más fiable de los cambios de estrategia del resto de agentes.

En los resultados registrados del estudio sobre un entorno en el que se simulaban partidos de fútbol 1v1 y 2v2 los resultados mostraron que generalmente DPIQN y DRPIQN obtuvieron mejor rendimiento que el resto de los algoritmos evaluados.

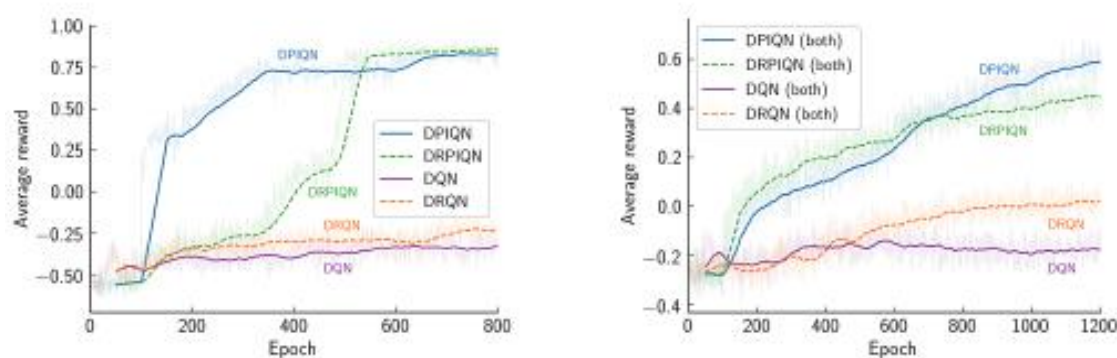


Ilustración 16: Comparativas de rendimiento 1v1 y 2v2

Como se puede observar en los resultados, DPIQN y DRPIQN, además de obtener mejores resultados que DQN y DRQN, lo hacen de manera más rápida. De hecho, DPIQN lo hace más rápido que DRPIQN puesto que este último cuenta con los parámetros de la memoria a largo plazo. Por otra parte, en la tabla que se muestra a continuación, se muestra las recompensas obtenidas por los algoritmos en diferentes entrenamientos. La segunda columna indica el comportamiento del resto de agentes en la etapa de entrenamiento y la tercera columna indica las puntuaciones obtenidas al enfrentarse con agentes con diferentes estrategias. *Hybrid* supone que los agentes toman estrategias tanto ofensivas como defensivas y que esto se establece de manera aleatoria al inicio del episodio.

Se puede observar cómo, a pesar de que DPIQN y DRPIQN obtienen los mejores resultados en todos los casos, en los casos en los que el agente se enfrenta a una estrategia conocida (*Offensive-Offensive*, *Defensive-Defensive*) obtiene buenos resultados (o prácticamente el mismo resultado que DPIQN y DRPIQN).

Model	Training	Hybrid	Offensive	Defensive
DQN	Hybrid	-0.063	-0.850	0.000
	Offensive	0.312	<u>0.658</u>	0.113
	Defensive	-0.081	-1.000	<u>0.959</u>
DRQN	Hybrid	0.028	-0.025	0.168
DPIQN	Hybrid	0.999	0.989	0.986
DRPIQN	Hybrid	0.999	0.981	1.000

Tabla 1: Comparativa de recompensas tras entrenamientos bajo diferentes estrategias conocidas

		Unfamiliar-O	Unfamiliar-C
1 vs. 1 Scenario			
DPIQN		0.909 (90%)	-
DRPIQN		0.947 (94%)	-
2 vs. 2 Scenario (rule-based)			
DPIQN	Both	0.501 (65%)	0.645 (84%)
	O-only	0.488 (75%)	0.535 (82%)
	C-only	0.076 (32%)	0.189 (81%)
DRPIQN	Both	0.534 (76%)	0.565 (81%)
	O-only	0.578 (80%)	0.625 (87%)
	C-only	0.625 (73%)	0.695 (82%)

Tabla 2: Recompensas obtenidas con agentes con estrategias desconocidas

La Tabla 2 muestra que los mejores resultados obtenidos en entornos multiagente se dan en los casos en los que los agentes controlados aprenden las políticas de sus contrincantes (*O-only*), siendo DRPIQN el algoritmo con mejores puntuaciones. Al haberse entrenado enfocado en las estrategias de sus oponentes, es más capaz de plantear estrategias exitosas para oponentes diferentes. Como mencionan Hong et al. en su artículo, el agente tiende a marcar gol por su cuenta debido al desconocimiento de la estrategia de su compañero de equipo.

Por último, haciendo referencia al factor λ que se comenta anteriormente, el siguiente gráfico demuestra cómo gracias a este factor, a pesar de que se termina consiguiendo el mismo resultado en todos los casos, las ejecuciones que no cuentan con el factor λ lo consiguen mucho más tarde que las que sí cuentan con él. De esta manera se demuestra que introducir esta variable a la ecuación favorece en gran medida la velocidad de aprendizaje del agente.

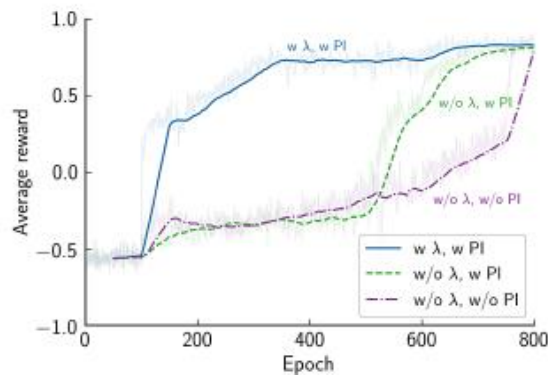


Tabla 3: Diferencia de rendimientos de tres variaciones de DPIQN

2.1.8 Reinforced inter-agent learning (RIAL) & Differentiable inter-agent learning (DIAL)

Otras dos aproximaciones basadas en DQN y DRQN son RIAL y DIAL. En este caso, estos enfoques surgen para abordar problemáticas en entornos multiagente totalmente cooperativos en los que existan condiciones de observabilidad parcial y la toma de decisiones de todos los agentes involucrados sea secuencial. En situaciones de este tipo, surge la necesidad de comunicarse para conseguir un rendimiento mayor y mejores resultados. Sin embargo, encontrar un protocolo de comunicación común es más complicado a medida que el sistema está compuesto de más agentes.

Tanto RIAL como DIAL funcionan con aprendizaje centralizado y ejecución descentralizada, lo que permite que los agentes puedan comunicarse entre ellos de manera ilimitada en la fase de aprendizaje, pero que estén limitados a utilizar un canal de comunicación de capacidad reducida en la fase de ejecución. El protocolo que se debe encontrar debe surgir de las acciones de comunicación que toman los agentes a cada instante de tiempo y que todos pueden observar. Tras observar la evolución del entorno debido a las acciones tomadas y los mensajes emitidos por los agentes, se construye un protocolo tras mapear estas observaciones. La dificultad de este proceso reside en la necesidad de interpretar y coordinar los mensajes emitidos y recibidos (Foerster, J., et al 2016).

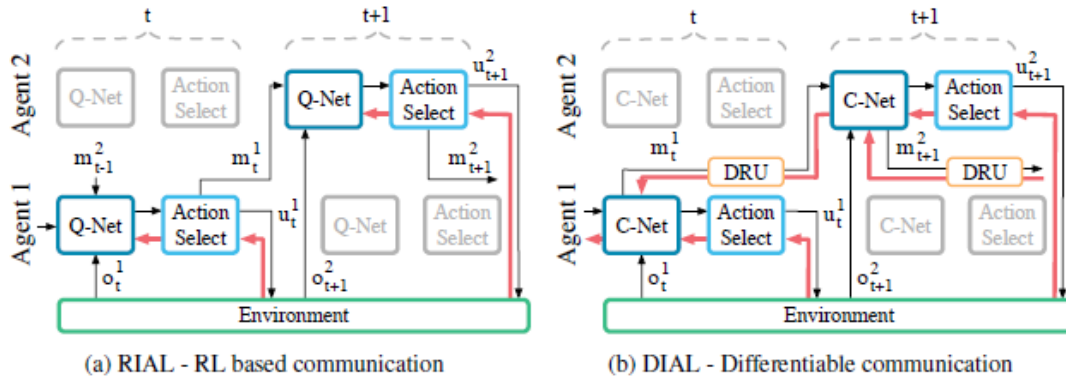


Ilustración 17: Arquitectura RIAL y DIAL

RIAL surge de la combinación de DRQN e IQL para la selección de acciones y acciones de comunicaciones. La red neuronal de cada agente está condicionada por las observaciones del agente (o_t^a), su estado oculto (h_{t-1}^a) y los mensajes del resto de agentes ($m_{t-1}^{a'}$) de manera que la representación de los valores de la red se representa $Q^a(o_t^a, h_{t-1}^a, m_{t-1}^{a'}, u^a)$. Foerster et al. declaran que, dado que son necesarias las elecciones de dos acciones $u \in U$, $m \in M$ (como acciones del agente en el entorno y acción de comunicación), en lugar de utilizar una única red neuronal para ambas, se utilizarán dos redes neuronales independientes Q_u^a y Q_m^a . De este modo solamente será necesario maximizar U y M en lugar de $U \times M$. Además, en RIAL DRQN se aplica sin ERM (*experience replay memory*).

RIAL puede verse beneficiado también gracias a la compartición de parámetros de la red neuronal de todos los agentes. Este método reduce la cantidad de parámetros que se deben

aprender (puesto que son comunes a todos) y hace que todos los agentes sean entrenados a partir de la misma red neuronal. Sin embargo, cada agente mantendrá la independencia de sus acciones debido a que dependen de las observaciones parciales propias, de la misma manera que en E-DDRQN.

A pesar de esto, RIAL carece de ciertas características que no lo hacen óptimo. Una de ellas es que RIAL no utiliza *feedback* en su planteamiento para reforzar el proceso de selección de un protocolo de comunicación, por lo que está limitado a obtenerlo a través del ensayo y error. El funcionamiento de DIAL es diferente en la fase de aprendizaje y en la fase de ejecución. En un primer momento, los mensajes de comunicación se sustituyen por señales directas que enlazan la salida de la red de un agente con la entrada de otro. Esta característica permite que los agentes puedan enviar información completa entre ellos y además, como el funcionamiento es similar a las mecánicas de activación del resto de la red, se permite el *feedback* entre agentes mencionado anteriormente. Como se muestra en el esquema de comunicación de la Fig. 14 la red neuronal C-Net genera dos salidas: *Q-value* y el mensaje para otros agentes m_t^a que es discretizado (en las fases de aprendizaje y ejecución) y regularizado (sólo en la fase de aprendizaje). DIAL es capaz de gestionar espacios de mensajes continuos y escalar a espacios amplios de mensajes discretos.

En todos los experimentos realizados por Foerster et al. se probaron DIAL y RIAL con y sin *parameter sharing* y una línea base de parámetros compartidos sin comunicación. En el primer escenario se resolvió el *Switch riddle*, en el que se testean situaciones con tres y cuatro agentes que intentan maximizar una recompensa común. En este primer experimento se observó cómo, a pesar de obtener una peor puntuación en la situación con cuatro agentes, tanto RIAL como DIAL obtuvieron una puntuación mucho mayor que la línea base de referencia y una puntuación muy similar entre ambos. Sin embargo, en el segundo escenario, en el que se resolvían problemas basados en el conjunto de datos MNIST, se observó cómo DIAL obtenía mejores resultados en comparación con RIAL.

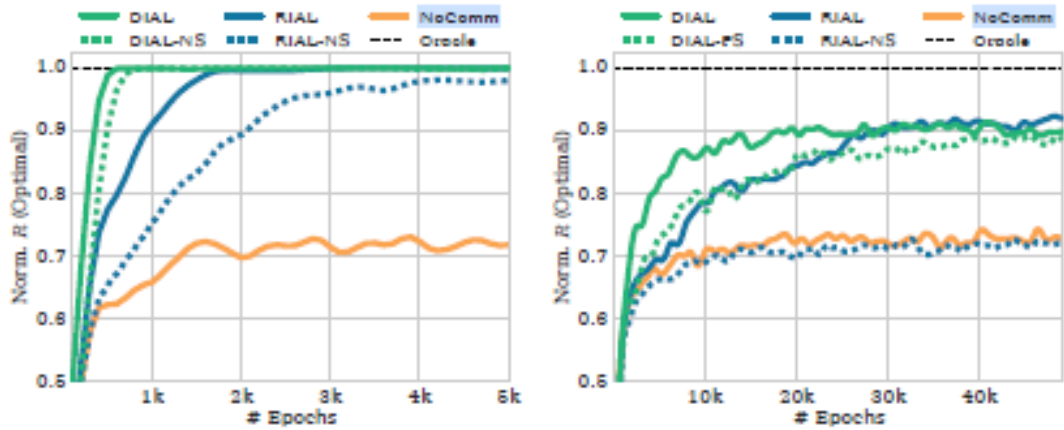


Ilustración 18: Comparativa de desempeño en escenario *Switch riddle*

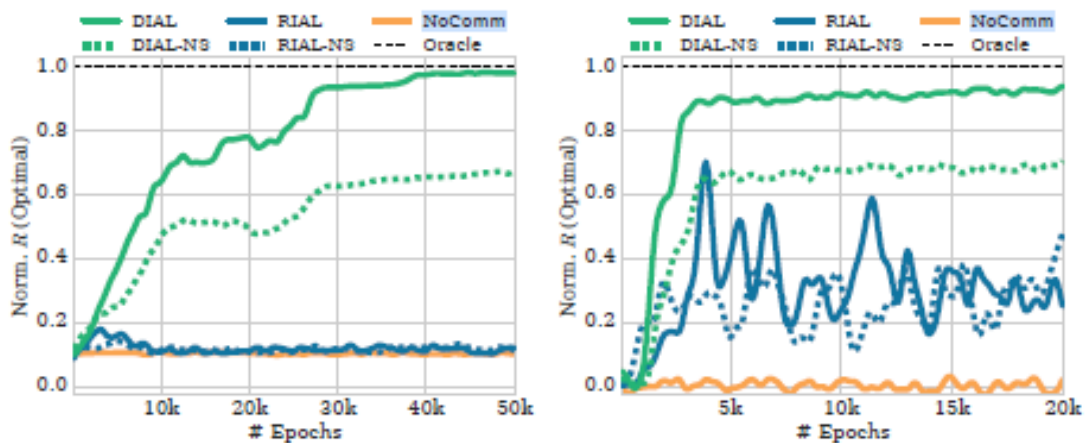


Ilustración 19: Comparativa de rendimiento en escenario Multi-step MNIST y Colour digit MNIST

Observando los gráficos correspondientes a los escenarios MNIST, se puede apreciar la importancia de utilizar *parameter sharing* de cara a obtener un mejor rendimiento y la diferencia de recompensas obtenidas entre los algoritmos que lo aplicaban (DIAL) y los que no (DIAL-NS). Además, es claramente visible cómo DIAL, obtiene resultados prácticamente perfectos mientras que RIAL apenas fluctúa en la línea de base. De esta observación se concluye que el ensayo y error de RIAL resulta ser muy poco eficiente en comparación con la optimización de mensajes de DIAL.

2.1.9 Multi-task multi-agent RL (MT-MARL):

Otra de las aproximaciones realizadas a los algoritmos de RL en entornos multiagente es la estudiada por (Omidshafiei, S., et al. 2017) en la que se presenta un nuevo planteamiento de para entornos colaborativos multiagente, en condiciones de observabilidad parcial, entornos estocásticos y recompensas conjuntas.

Partiendo de la base de que muchos sistemas plantean aprendizajes de políticas especializadas para tareas concretas y de que estos sistemas fallan a la hora de desempeñar tareas en entornos de la vida real, surge la necesidad de encontrar la manera de generalizar conocimientos sobre tareas únicas que permita la intervención de agentes en múltiples tareas relacionadas. En el estudio se expone como ejemplo un entorno de trabajo submarino en el que varios agentes colaboran en la reparación de instalaciones acuáticas y que necesitan coordinarse en condiciones diferentes a las observadas en las fases de entrenamiento. La aproximación planteada por Omidshafiei et al. plantea un sistema descentralizado, tanto en la etapa de aprendizaje como en la de ejecución, que soporta la interacción de varios agentes cooperantes y capaz de destilar una política unificada. Estas características corresponden, según el estudio, a las que tendría un agente eficiente puesto que aceleran su aprendizaje y mejoran la generalización de este.

El algoritmo que se plantea para tratar sistemas MT-MARL se trata de una ampliación del algoritmo DRQN (analizado previamente en este documento). Resolver problemas categorizados como DEC-POMDP (Procesos de Decisión de Markov descentralizados con observabilidad parcial) es realmente complejo, por esta razón MT-MARL se divide en dos fases.

La primera fase trata de realizar un aprendizaje enfocado en una sola tarea facilitando la coordinación y la obtención de los valores Q asociados a cada tarea. Al explorar la calidad de todas las acciones, y no únicamente de las que mayores recompensas reportan, *Q-Learning* se puede someter al proceso de destilación de política. Frente a enfoques optimistas que puedan generar sobreestimaciones a la hora de calcular las recompensas de acciones no óptimas Omidshafiei et al. plantean el uso de *Hysteretic Q-Learning* que acoge valores bajos de Q como consecuencia de la estocasticidad del entorno. Gracias a dos factores (,) la actualización de los valores Q para acciones pasadas satisfactorias fruto de la cooperación de los agentes se protege frente a resultados menos satisfactorios debidos a la exploración de alguno de los agentes. Por este motivo, los buenos resultados de *Hysteretic Q-Learning* en entornos multiagente con observabilidad total y los beneficios que aporta DRQN, el algoritmo final que presentan Omidshafiei et al. es la combinación de estos (Dec-HDRQN).

Este planteamiento utiliza CERTs (*Concurrent Experience Replay Trajectories*) en lugar de ERM. Como se ha comentado en otras ocasiones, los entornos multiagente, debido a la no estacionalidad generada por la toma de decisiones de los agentes que lo componen y el cambio en sus políticas, no encuentran beneficio en el almacenamiento de experiencias porque quedan obsoletas de manera rápida. Por este motivo es importante un registro de experiencias sincronizado entre todos los agentes.

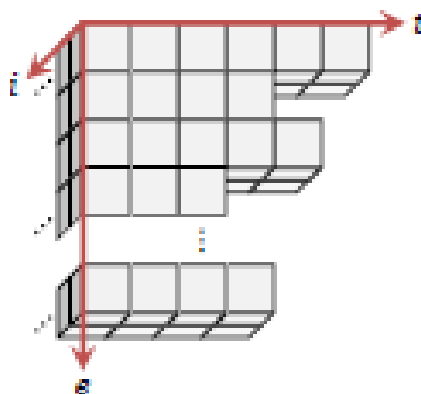


Ilustración 20: Estructura CERT

Utilizando CERTs, es posible almacenar para cada agente en cada episodio una tupla de información (representada como uno de los cubos en la Ilustración 19) a cada instante de tiempo. Este almacenamiento es clave para la generalización de políticas realizado en la segunda fase del proceso de aprendizaje.

La evaluación de la eficacia de este algoritmo se probó utilizando un escenario en el que ciertos agentes debían capturar objetivos en un entorno con forma de cuadrícula. Este planteamiento supone un dominio que facilita retos similares con complejidad variable (ampliación del tamaño de la cuadrícula, objetivos diferentes, presencia de más o menos agentes, ...). Se evaluaron tanto el desempeño en tareas únicas como en tareas múltiples cuyos desafíos eran el aprendizaje de roles para la coordinación entre agentes y la diferenciación de estados para la captura más rápida de los objetivos. Para obtener la característica de observabilidad parcial los agentes solamente fueron conscientes de su posición y, de manera intermitente, de la de su objetivo en la cuadrícula.



Ilustración 21: Dec-DRQN vs Dec-HDRQN

En los resultados de los experimentos se observó la diferencia entre aplicar histéresis al algoritmo y no hacerlo. Dec-DRQN se muestra inestable y no consigue lograr grandes recompensas para entornos más desafiantes debido a la exploración y la falta de coordinación de los agentes. Como consecuencia, los agentes se desvían de una política óptima por la falta de recompensa conjunta. Sin embargo, Dec-HDRQN garantiza la estabilidad en estos casos y facilita conseguir coordinación entre los agentes, por lo que obtiene mayor rendimiento incluso en los planteamientos con mayor complejidad.

2.1.10 Master-slave MARL (MS-MARL):

Según (Kong, X., et al. 2017), gran cantidad de trabajos de investigación previos se han centrado en algoritmos que utilizaban procedimientos centralizados o descentralizados, pero casi siempre se ha hecho por separado. Los sistemas multiagente centralizados requieren la búsqueda de parámetros en un espacio de estados-acciones demasiado grande que, además, crece a medida que el sistema incorpora nuevos agentes. Por otra parte, los sistemas descentralizados, caen en la problemática de la no estacionalidad debido a las observaciones locales y a la toma de decisiones constante del resto de agentes del entorno.

En el estudio del algoritmo MS-MARL se combinan tanto el modelo centralizado como el descentralizado buscando combinar los beneficios que aportan cada uno de ellos en una arquitectura maestro-esclavo (*master-slave*: MS). El enfoque MS combina la planificación global sin prestar demasiada atención a los detalles de las observaciones locales de los agentes esclavos, mientras que estos optimizan las acciones propuestas por el agente maestro en función de los detalles observados.

Kong, X., et al. plantean que el algoritmo desarrollado se encuentra entre los pocos diseñados para el enfoque de arquitecturas maestro-esclavo en entornos MARL a pesar de que esta arquitectura ya se haya utilizado en entornos con varios agentes (Park, K. et al., 2001; Verbeeck, K., et al. 2005). En su estudio se muestra como objetivo alcanzar una política (mapeo de estados-acciones) en un espacio de acciones $a = \{a^1, \dots, a^C\}$ y un espacio de estados $s = \{s^m, s^1, \dots, s^C\}$ en los que C es el número de agentes del sistema y s^m es un estado independiente del agente maestro. Gracias a incluir este estado se obtiene la posibilidad de añadir información independiente al agente maestro.

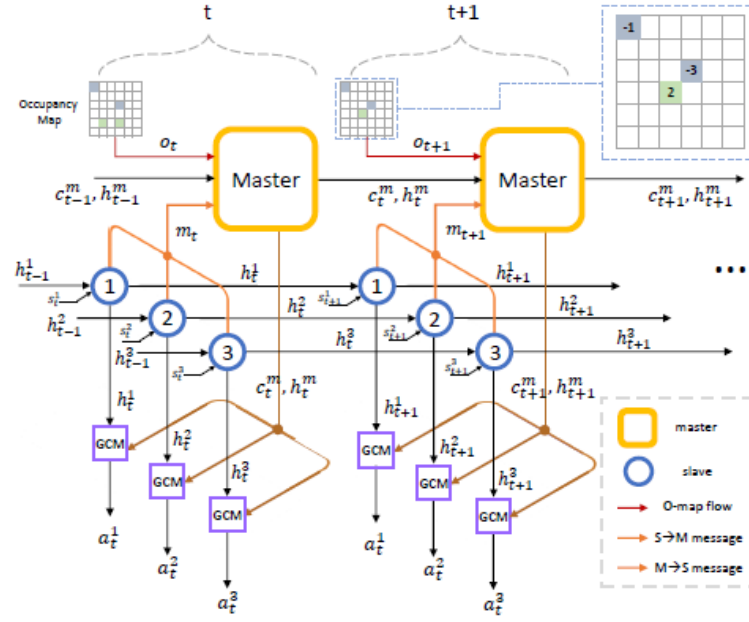


Ilustración 22: Arquitectura MS-MARL

La eficacia del planteamiento de Kong, X., et al. se probó en cinco escenarios diferentes. Dos de ellos ya estudiados previamente basados en entornos con espacios de estado y acciones discretas, proponen un problema de cruces de tráfico y un escenario de combate entre agentes. Los otros tres escenarios están basados en los RTS (*real-time strategy games*), en este caso Star-Craft. Estos tres últimos escenarios cuentan con espacios de acción y estados continuos. En comparación con el algoritmo CommNet (Sukhbaatar, S et al. 2016), empleado originalmente en la resolución de los escenarios planteados anteriormente, MS-MARL demostró ser más eficiente. Además, en los escenarios de acción y estados continuos, a pesar de plantear situaciones desfavorables para el sistema evaluado (inferioridad de unidades respecto del oponente), MS-MARL resultó el algoritmo más eficiente comparado con CommNet, BiCNet y GMEZO. Dada la continuidad de los estados y las acciones, el aprendizaje en estos entornos se realizó utilizando políticas gaussianas y recompensas ($r_t^i, r_{terminal}$) en relación con la diferencia de unidades aliadas respecto de las enemigas, las victorias y las derrotas obtenidas.

$$r_t^i = \Delta n_{j \in N_m(i)}^t - \Delta n_{k \in N_e(i)}^t \quad r_{terminal} = \begin{cases} 1 & \text{if battle won} \\ -0.2 & \text{else} \end{cases}$$

Ecuación 10: Cálculo de recompensas en MS-MARL para escenario StarCraft

Tasks	CommNet	MS-MARL
Traffic Junction	0.94	0.96
Combat	0.31	0.59

Tabla 4: Resultados en tareas discretas CommNet vs MS-MARL

Tasks	GMEZO	CommNet	BiCNet	MS-MARL	MS-MARL + GCM
15M vs. 16M	0.63	0.68	0.71	0.77	0.82
10M vs. 13Z	0.57	0.44	0.64	0.75	0.76
15W vs. 17W	0.42	0.47	0.53	0.61	0.60

Tabla 5: Resultados de los escenarios con tareas continuas

Los resultados obtenidos se muestran superiores para MS-MARL debido a que facilita canales de comunicación constantes con los agentes esclavos e independencia de razonamiento para el agente maestro, apoyándose en los mensajes recibidos de todos los agentes esclavos y en su propio estado s^m . Además, al explorar el espacio de estado-acción de manera jerárquica y explícita, permite al maestro analizar la escena global y dejar a los esclavos centrarse en detalles locales. Este planteamiento permite mantener la estabilidad en la toma de decisiones y la planificación en espacios de estados grandes.

2.1.11 Parameter Sharing Trust Region Policy Optimization (PS-TRPO):

Este algoritmo considera la problemática del aprendizaje de políticas en entornos POMDP cooperativos en los que no existe comunicación explícita entre agentes. En el estudio realizado por (Gupta, J. et al. 2017) se demuestra que, frente a métodos basados en diferencia temporal o *actor-critic*, los basados en *policy-gradient* (gradiente de política) obtienen mejores rendimientos en estos escenarios. Otra de las características destacadas de este algoritmo es que es escalable a tareas complejas en las que toman acciones múltiples agentes en espacios de acción discretos y continuos.

El modelo propuesto por Gupta, J. et al. es una adaptación del trabajo realizado en el año 2000 por Peshkin (Meuleau, N. et al. 2000), pero en lugar de utilizar máquinas de estados hacen uso de redes neuronales profundas (*deep neural-network*: DNN). En los escenarios en los que se evalúa la eficacia del algoritmo PS-TRPO, todos los agentes buscan maximizar la recompensa combinada. Además, los agentes reciben observaciones locales y ninguno tiene acceso a una observación global a diferencia del planteamiento MS-MARL. Por último, como los agentes no se comunican de manera explícita, deben alcanzar comportamientos cooperativos basados en sus observaciones.

Algorithm 1 PS-TRPO

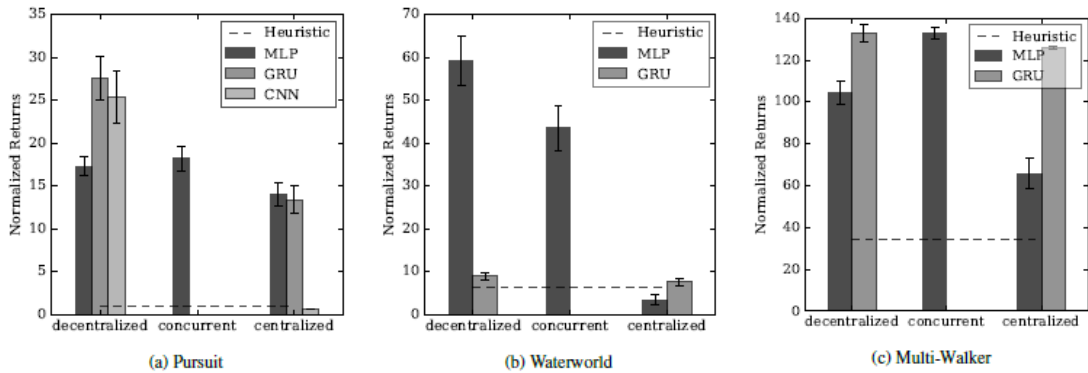
Input: Initial policy parameters Θ_0 , trust region size Δ
for $i \leftarrow 0, 1, \dots$ **do**
 Rollout trajectories for all agents $\tau \sim \pi_{\theta_i}$
 Compute advantage values $A_{\pi_{\theta_i}}(o^m, m, a^m)$ for each agent m 's trajectory element.
 Find $\pi_{\theta_{i+1}}$ maximizing Eq. (1)
 subject to $\overline{D}_{KL}(\pi_{\theta_i} \parallel \pi_{\theta_{i+1}}) \leq \Delta$

Ilustración 23: Algoritmo PS-TRPO

$$L(\theta) = \mathbb{E}_{o \sim \rho_{\theta_k}, a \sim \pi_{\theta_k}} \left[\frac{\pi_{\theta}(a \mid o, m)}{\pi_{\theta_k}(a \mid o, m)} A_{\theta_k}(o, m, a) \right] \quad (1)$$

Ecuación 11: Actualización de parámetros en la red neuronal

Los experimentos realizados comparan tres tipos de cálculo de políticas: centralizado, concurrente y descentralizado (utilizando parámetros compartidos). Además, para cada uno de ellos se prueban tres modelos diferentes de redes neuronales: CNN, tanto en tareas discretas como en tareas continuas y GRU (*gates recurrent units*) y MLP (*multi-layer perceptron*) en tareas continuas. La comparativa de resultados reflejó un mejor rendimiento del modelo descentralizado con compartición de parámetros respecto de los otros dos métodos tanto para tareas discretas como continuas.

Ilustración 24: Comparativa del rendimiento: *Parameter sharing (decentralized), concurrent & centralized*.

Por otra parte, se comparó el desempeño de PS-TRPO en comparación con el algoritmo DDPG (*Deep Deterministic Policy-gradient*) en el escenario *Multi-Walker* y con PS-DQN en el escenario *Pursuit*. Este último es de naturaleza discreta y consiste en una tarea de captura para ciertos agentes y de evasión para otros. Para conseguir su objetivo, los agentes captadores deben rodear

a un objetivo por los cuatro costados. La política desarrollada mediante PS-TRPO demostró ser escalable a entornos con docenas de agentes y dividirlos en grupos de cuatro. El escenario *Multi-Walker* consiste en una serie de agentes que deben mantener en equilibrio sobre ellos una figura mientras avanzan por un terreno. Este escenario plantea un entorno con espacios de acción y estados continuos.

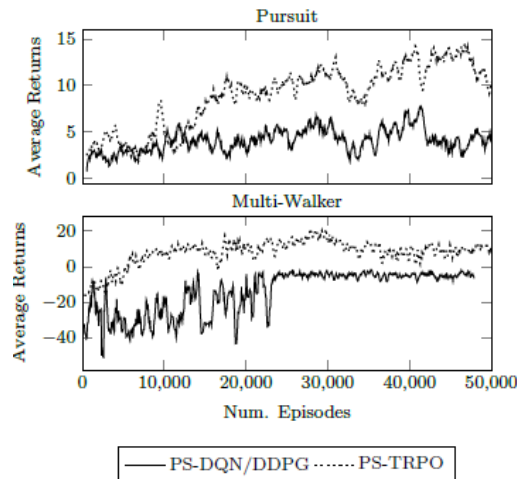


Ilustración 25: Rendimiento en escenarios Pursuit y Multi-Walker

Otros resultados de este estudio muestran cómo un planteamiento de aprendizaje a través de *curriculum learning* resulta ser mucho más eficaz que los planteamientos concurrentes, centralizados o descentralizados. Este enfoque se analiza más adelante en este mismo documento.

2.1.12 Action-Specific Deep Recurrent Q-Network (ADRQN):

Otro algoritmo que aborda situaciones bajo la observabilidad parcial es ADRQN (Zhu, P., et al. 2018). Bajo el planteamiento de que no se han dedicado demasiados esfuerzos al estudio de problemas en condiciones de observabilidad parcial en el ámbito del aprendizaje por refuerzo, se plantea un nuevo algoritmo que pretende extender DQN y DRQN para escenarios en los que es necesario tener en cuenta observaciones de larga duración.

En comparación con DRQN y DDRQN, que desacoplan las tuplas acción-observación o directamente no almacenan las acciones tomadas, ADRQN incorpora a la tupla de información la relación entre las acciones tomadas y las observaciones obtenidas a raíz de estas. La memoria LSTM utilizada recibe una tupla de la forma $\{(a_{t-1}, o_t), a_t, r_t, o_{t+1}\}$ como entrada. En un planteamiento ideal de este enfoque, la memoria almacenaría todas las transacciones de un episodio. Sin embargo, ADRQN utiliza un desdoblamiento de la memoria a lo largo de diez instantes para registrar la información.

Algorithm 1 Action-specific Deep Recurrent Q-Network

```

1: Initialize the replay memory  $D$ , # of iterations  $M$ 
2: Initialize Q-Network and Target-Network with  $\theta$  and  $\theta^-$  respectively
3: for episode = 1 to  $M$  do
4:   Initialize the first action  $a_0 = \text{no operation}$ ,  $h_0 = \mathbf{0}$ 
5:   Init the first obs.  $o_1$  with the preprocessed first screen
6:   while  $o_t \neq \text{terminal}$  do
7:     Select a random action  $a_t$  with the probability  $\epsilon$ 
8:     Else select  $a_t = \arg\max_a Q(h_{t-1}, a_{t-1}, o_t, a; \theta)$ 
9:     Execute action  $a_t$ 
10:    Obtain reward  $r_t$  and resulting observation  $o_{t+1}$ 
11:    Store transition  $\langle \{a_{t-1}, o_t\}, a_t, r_t, o_{t+1} \rangle$  as one record of the current episode in  $D$ 
12:    Randomly sample a minibatch of transition sequences  $\langle a_{j-1}, o_j, a_j, r_j, o_{j+1} \rangle$  from  $D$ 
13:    Compute Q-value of target network

$$y_j = \begin{cases} r_j, & o_{j+1} = \text{terminal} \\ r_j + \gamma \max_a Q(h_j, a_j, o_{j+1}, a; \theta^-), & o_{j+1} \neq \text{terminal} \end{cases}$$

14:    Compute the gradient of  $(y_j - Q(h_{j-1}, a_{j-1}, o_j, a_j; \theta))^2$  to update  $\theta$ 
15:  end while
16: end for

```

Ilustración 26: Algoritmo ADRQN

En las pruebas realizadas por Zhu et al. se plantean como escenarios de prueba juegos de la consola Atari atendiendo, entre otros, a los mismos planteamientos realizado por (Hausknecht, M., et al. 2015), para añadir observabilidad parcial a través de imágenes parpadeantes, y (Bellemare, M., et al. 2013) para omitir *frames* de los juegos y evaluar f_{k+1} como resultado de tomar la acción a_i en el *frame* f_0 . De esta manera, en lugar de tener transiciones de estados de la forma $\{f_0, a_i, f_1, a_i, \dots, f_{k+1}\}$ se obtendría una expresión simplificada $\{f_0, a_i, f_{k+1}\}$. Esta última técnica se utiliza para simular el entorno de manera eficiente.

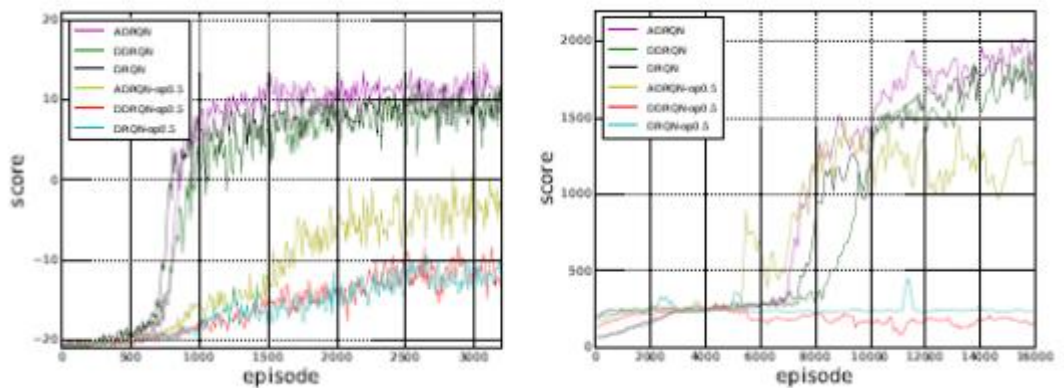


Ilustración 27: Comparativa de rendimiento en los juegos Pong (1) y Frostbite (2)

En los resultados de los experimentos realizados con los algoritmos ADRQN, DRQN y DDRQN se observa cómo los tres obtienen un rendimiento similar con observabilidad total y cómo DRQN y DDRQN obtienen resultados significativamente peores que ADRQN con observabilidad parcial (obs 0.5). DRQN y DDRQN ya aplican recurrencia para obtener mejores resultados en entornos con observabilidad parcial, pero el matiz de no desacoplar las tuplas acción-observación que propone ADRQN resulta ser mucho más eficiente que ambos. Además, el estudio sobre ADRQN demostró mejores resultados a la hora de generalizar el comportamiento del algoritmo a entornos con observabilidad variable. Tras entrenar los tres algoritmos con un 50% de observabilidad y al probarlos sobre un entorno con observabilidad variable (entre 0% y 100%) y tras entrenarlos en entornos totalmente observables y probarlos en entornos con observabilidad variable, ADRQN demostró ser más robusto que los otros dos algoritmos.

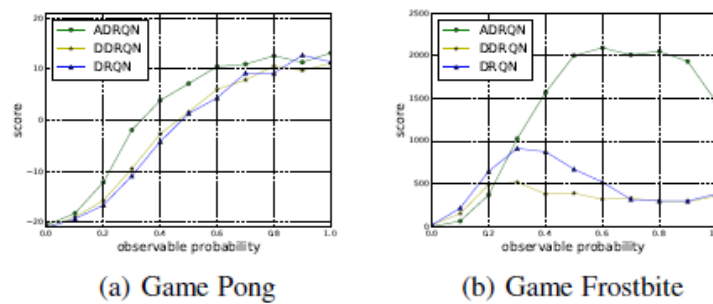


Ilustración 28: Rendimiento tras generalización POMDP a MDP

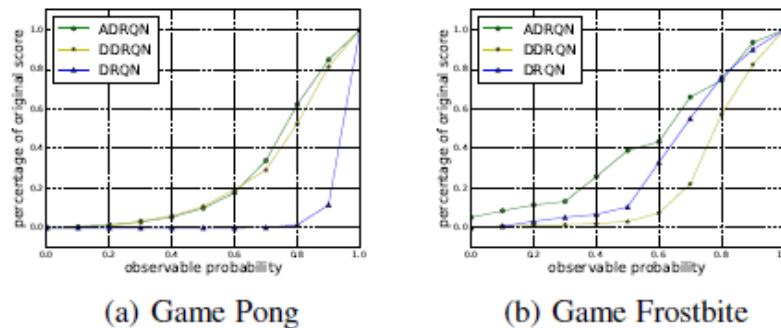


Ilustración 29: Rendimiento tras generalización MDP a POMDP

2.1.13 Recurrent MA Deep Deterministic Policy Gradient (R-MADDPG):

Por último, se presenta un estudio realizado por tres investigadores del MIT (*Massachusetts Institute of Technology*) en el que presentan un nuevo algoritmo multiagente para resolver problemas del mundo real. Un planteamiento que destacan sus autores es que todos los algoritmos de *Reinforcement Learning* deberían plantearse en base a la observabilidad parcial, tener en cuenta entornos no estacionarios y la limitación de comunicación entre agentes y que hacerlo de otra forma sería estar asumiendo planteamientos poco realistas (por ejemplo, conocer el estado oculto de otros agentes como plantea (Singh et al., 2018; Sukhbaatar et al., 2016). El

algoritmo propuesto para abordar estos problemas está basado en el planteamiento *actor-critic*, que consiste en combinar los puntos fuertes de los modelos basados en políticas y los modelos basados en valores. Una red encargada de encontrar la mejor política y, al mismo tiempo, otra red que evalúa las acciones tomadas por la primera para calcular la función de valor $Q(s, a)$.

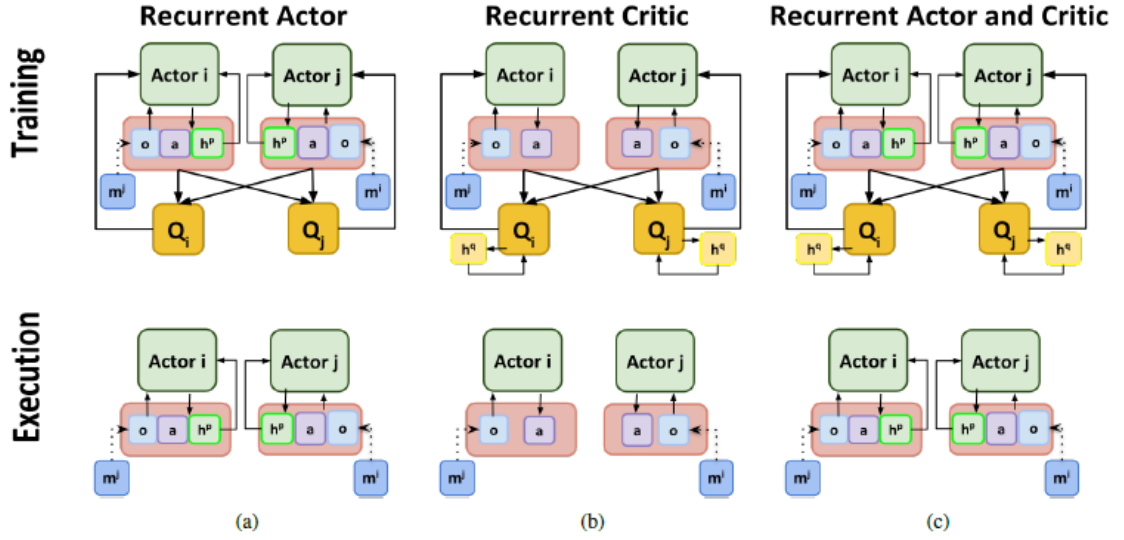


Ilustración 30: Estructura de modelos *actor-critic*

R-MADDPG aprende dos políticas paralelas centradas en las acciones que se llevan a cabo en el escenario y en la obtención de un protocolo de comunicación. Esta segunda tarea viene impulsada por la limitación de comunicación que se supone a los entornos reales, por lo que es necesario que los agentes de un sistema den con el protocolo más conveniente para comunicarse. Además, R-MADDPG utiliza recurrencia en ambas partes de su estructura, el actor y el crítico, a diferencia de planteamientos previos que lo hacen sólo en una de sus partes. Wang, R., et al. plantean que el factor clave de este modelo, y el que marca la diferencia, es contar con un crítico recurrente y que contar con un actor recurrente no es suficiente para resolver problemas de observabilidad parcial. En la Fig. 30 se puede apreciar cómo disponer de un crítico recurrente resulta más eficiente que no disponer de él en entornos parcialmente visibles (Wang, R., et al. 2020). Evaluados en base a dos recompensas diferentes (distancia al objetivo y distancia entre agentes) todos los planteamientos de R-MADDPG rinden prácticamente al mismo nivel con total observabilidad. E incluso el planteamiento con actor y críticos recurrentes rinde de peor manera. Sin embargo, el rendimiento de los planteamientos con críticos recurrentes es notablemente mejor que los que no cuentan con él.

Para las pruebas de rendimiento de este algoritmo, se utilizó un escenario en el que varios agentes deben alcanzar un destino al mismo tiempo. Bajo este planteamiento no es suficiente con que cada agente logre aprender una política que le haga llegar al objetivo lo más rápido posible, sino que debe coordinarse con el resto de los agentes para lograrlo. Esta tarea no sería posible sin la comunicación entre ellos. Por otra parte, de cara a reducir la no-estacionalidad del entorno, los críticos de cada agente reciben como input las observaciones de todos los demás agentes, como se muestra en la Ilustración 29. Previamente se comenta que asumir disponibilidad sobre cierta información de otros agentes es un planteamiento irreal en entornos

reales, pero en el caso de las observaciones, a diferencia de los estados ocultos, siempre que haya comunicación entre agentes es viable y muy útil para el conjunto del sistema. Esto se demuestra en los resultados del experimento Ilustración 31 y cómo MADDPG solamente logra igualar el nivel de rendimiento de R-MADDPG cuando cuenta con diez veces la cantidad de mensajes R-MADDPG.

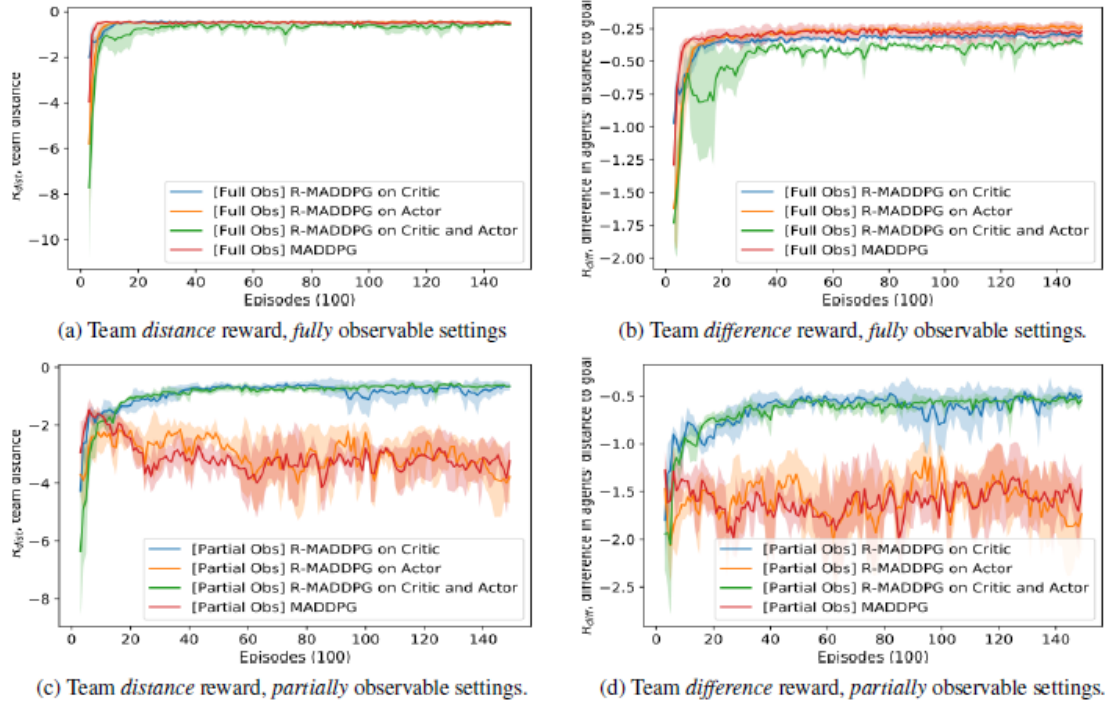


Ilustración 31: Comparativa de rendimientos en base a la recompensa evaluada y la observabilidad del sistema

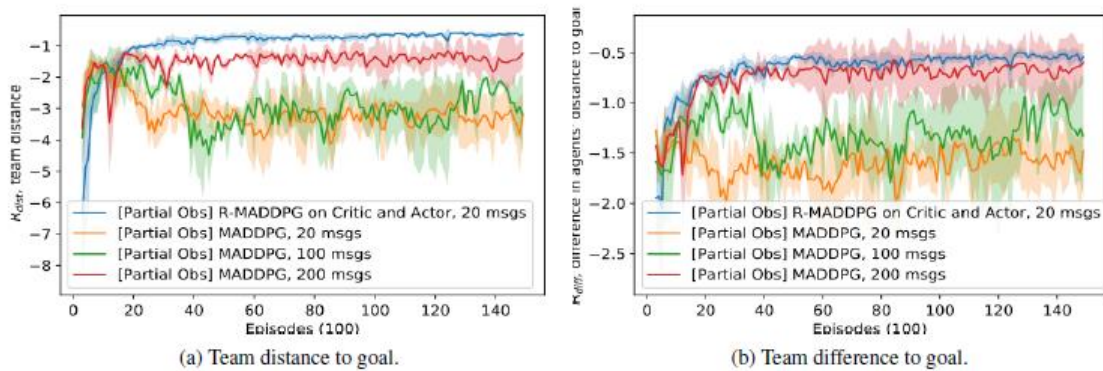


Ilustración 32: Comparativa de rendimiento con diferentes cantidades de mensajes

2.2 VARIACIONES DE ESQUEMAS DE ENTRENAMIENTO

2.2.1 Agentes modulares

En ciertas ocasiones se pueden encontrar agentes en un sistema que pueden ser descompuestos en subagentes homogéneos con el fin de entrenarlos para alcanzar el objetivo del agente inicial. Este fenómeno es fácilmente identificable cuando contamos con brazos robóticos como agentes de un sistema. El movimiento necesario del brazo es fácilmente replicable en los diferentes componentes del robot.

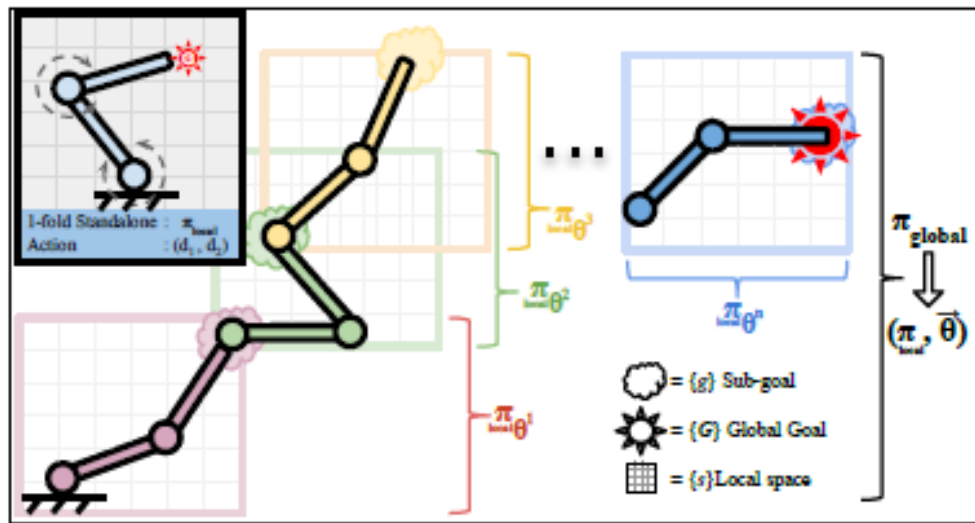


Ilustración 33: Descomposición de un agente en submódulos

En la Ilustración 32 se puede observar cómo los diferentes componentes del agente principal pueden trabajar de manera coordinada para que, a través de objetivos locales, el sistema al completo obtenga el objetivo global. El aprendizaje modular de (Raza, S., et al 2019) está basado en dos capas de una red encargadas, por una parte, de dividir el problema del agente principal teniendo en cuenta que los subagentes están totalmente interconectados y resuelven problemas idénticos y, por otra parte, de aprender una política reutilizable por todos los subagentes. Este planteamiento está pensado para agentes que contemplen espacios de estado y acción continuos y complejos puesto que resulta computacionalmente muy costoso obtener políticas óptimas. Gracias a la modularidad del agente y la reutilización de la política se reduce el esfuerzo computacional y se obtienen mejores resultados a la hora de cumplir un objetivo.

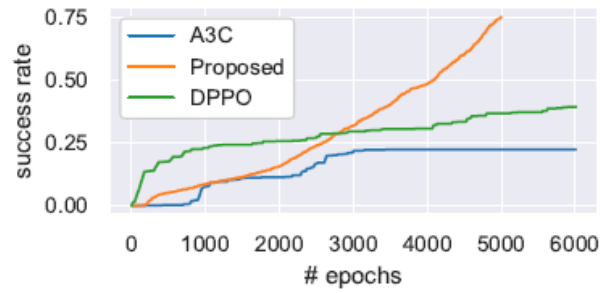


Ilustración 34: Comparativa de resultados obtenidos a través del planteamiento modular (Proposed)

2.2.2 Curriculum Learning

Este método es una extensión de *Transfer Learning* (Zhuang, F., et al. 2020), Aplicar Curriculum Learning en un proceso de aprendizaje implica realizar un diseño en el que las tareas se secuencian de tal manera que el aprendizaje logrado en una tarea previa sirva de precedente para el aprendizaje de la tarea siguiente. Resulta ser una técnica muy útil, al igual que la modularización en subagentes, para situaciones en las que el espacio de estados es demasiado grande o continuo puesto que descompone la tarea en subtareas de complejidad reducida. Mientras que el planteamiento anterior descomponía el agente en subagentes, este hace lo propio con las tareas.

Se plantea también la posibilidad de automatizar el proceso de diseño de currículos o, dicho de otra manera, hallar una *curriculum policy* que mapee el conocimiento actual de un agente con la siguiente tarea a desarrollar. Este planteamiento se desarrolla sobre el trabajo de (Narvekar, S., et al 2017) y está compuesto por dos agentes que se centran en aprender las tareas y en aprender una política de currículum respectivamente. Para el *curriculum agent* las entradas que recibe son las políticas aprendidas por el *learner agent* y cada vez que este la actualiza tras un aprendizaje realimentan al *curriculum agent* para su aprendizaje.

El objetivo de este planteamiento es, al mismo tiempo que se encuentran las diferentes políticas para la realización de las tareas, encontrar una política $\pi^C : S^C \rightarrow A^C$ que permita conocer la siguiente tarea a entrenar dada una política (S^C). Una política será terminal S_f^C en el momento en que obtenga una recompensa superior a un umbral para la tarea objetivo (Narvekar, S., et al. 2018).

2.2.3 Inyección de ruido

Se ha comentado en varios de los análisis previos que la existencia de ruido supone un problema a la hora de entrenar agentes mediante RL y en algunos se plantean soluciones para reducir su incidencia en el aprendizaje. Sin embargo, existen acercamientos a este fenómeno que plantean que el ruido puede ser un factor útil a la hora del entrenamiento.

Uno de ellos es *NoisyNet* (Fortunato, M., et al. 2019) en el que se añade ruido de manera explícita para favorecer la exploración en la fase de aprendizaje. El algoritmo que se presenta utiliza perturbaciones en los pesos de la red neuronal como consecuencia de una función de valores aleatorizada. Otro de los puntos que exponen sobre el algoritmo es que, a diferencia de otros

algoritmos similares (Plappert, M., et al. 2017), no están limitados a la aplicación de ruido Gaussiano, sino que permite introducir diferentes distribuciones de ruido. Además, el algoritmo introduce ruido de manera progresiva sin perjudicar el aprendizaje por exceso de alteración.

	Baseline		NoisyNet		Improvement (On median)
	Mean	Median	Mean	Median	
DQN	319	83	379	123	48%
Dueling	524	132	633	172	30%
A3C	293	80	347	94	18%

Tabla 6: Puntuaciones obtenidas por NoisyNet vs DQN, Dueling & A3C

En los experimentos realizados en la plataforma Atari, *NoisyNet* se compara con DQN, A3C (Chen, M., et al. 2020) y Dueling network (Wang, Z., et al. 2016) y se demuestra cómo el rendimiento de este algoritmo supera al de los otros tres en un 48%, 30% y 18% respectivamente. Estos resultados se ven representados también en las curvas de aprendizaje de cada algoritmo comparados con la aplicación de *NoisyNet*.

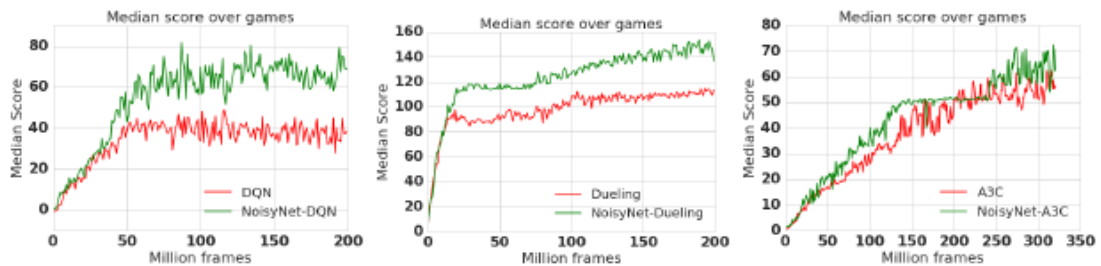


Ilustración 35: Comparativa de las curvas de aprendizaje NoisyNet vs DQN, Dueling, A3C

Por último, otro planteamiento interesante relativo al ruido en los datos es el que plantean (Wang, B., et al. 2019). En el ámbito de los algoritmos de recomendación, mencionan que además de utilizar las recompensas para entrenar a los agentes se utilizan datos que podrían ser de carácter sensible input de la red neuronal del mismo modo que otros algoritmos introducen las observaciones de los agentes. Esta información que se convierte en histórica a medida que el agente se entrena, podría deducirse realizando ingeniería inversa de los algoritmos de aprendizaje. De la misma manera que una función de recompensa describe una tarea, la función de recompensa de un agente de recomendación puede describir los datos que utiliza para su ejecución en sus estados visitados. Por este motivo, Wang, B., et al. introducen el concepto del ruido de manera que la función puede ser evaluada en muchos estados de forma arbitraria, preservando al mismo tiempo la privacidad.

A continuación, se muestra un resumen de los criterios que cumple cada uno de los algoritmos analizados:

	1	2	3	4	5	6
DRUQN	✓	Desc.	DIsc	Coop.	Homo.	-
DLCQN	-	Desc.	-	Coop.	-	-
LDQN		Desc.	Cont	-	-	-
WDDQN	X	Desc.	-	Coop.	Homo.	-
IQL	-	Desc.	-	-	-	-
DDRQN/ EDDRQN	✓	Desc.	-	Coop.	-	-
DPIQN/ DRPIQN	-	Desc.	-	-	-	-
DIAL RIAL	✓	Entrenamiento centralizado Ejecución descentralizada	-	Coop.	-	-
MT-MARL	✓	Desc.	-	Coop.	-	-
MS-MARL	✓	Ambos	Ambos	Coop.	-	-
PS-TRPO	✓	Desc.	Ambos	Coop.	-	-
ADQN	✓	Desc.	Ambos	Comp.	-	-
R-MADDPG	✓	Desc.	Ambos	Coop.	-	-

3. EXPERIMENTACIÓN

Como se ha comentado brevemente en el apartado introductorio, la fase de experimentación del proyecto ha consistido en el entrenamiento de un agente a través de un modelo 3D de un robot industrial articulado.

En este capítulo se analizan más en profundidad las diferentes aproximaciones que se han planteado y se presentan los resultados obtenidos a través de gráficas extraídas de TensorBoard⁵ y referencias a videos referenciados en los apartados correspondientes.

3.1 HERRAMIENTAS

En una primera etapa de la fase de experimentación ha sido necesario documentarse y explorar qué softwares estaban disponibles para la tarea que se quería desarrollar. Tanto a nivel de librerías como de entornos y motores, existe una gran oferta de soluciones, cada una con sus pros y sus contras. A continuación se presentan algunas de ellas con una explicación breve de cada una.

3.1.1 Entornos y motores

3.1.1.1 Gazebo

Esta plataforma de simulación especializada en robótica permite probar algoritmos, diseñar robots y entrenar algoritmos de IA utilizando escenarios realistas. Cuenta con un motor de físicas y gráficos de alta calidad.

Colabora con Ignition Robotics, un *toolbox* de librerías de desarrollo *open source* y servicios *cloud* que permiten iterar de manera rápida por los diseños.

Enlace: <http://gazebosim.org/>



3.1.1.2 Unity

Actualmente una de las plataformas de desarrollo 3D más extendidas. Cuenta con una gran comunidad de desarrolladores y de contenido de acceso gratuito en su Asset Store de manera que se puedan compartir modelos en 3D, ficheros de audio, efectos visuales, texturas, etc.



⁵ TensorBoard: <https://www.tensorflow.org/tensorboard?hl=es-419>

Además, es una herramienta pensada para atender las necesidades de múltiples perfiles profesionales (artistas, diseñadores, programadores, ...).

En la actualidad, es una plataforma que está apostando no sólo por el desarrollo sino también por su implementación como herramienta de trabajo y de simulación de entornos reales.

Por otra parte, cuenta con la integración con Visual Studio como editor de código y sincronización automática de los cambios realizados en los scripts de manera que el entorno se comporte siempre de acuerdo con las modificaciones.

Enlace: <https://unity.com/es>

3.1.1.3 Unreal Engine

Junto con Unity es la herramienta de creación de 3D en tiempo real más extendida, sobre todo gracias a su influencia en el mundo de los videojuegos. Sin embargo, no sólo cumple su propósito original como un motor de juego de última generación, sino que ofrece a los creadores de todas las industrias la libertad y el control para ofrecer contenido de vanguardia, experiencias interactivas y mundos virtuales inmersivos.

Su lenguaje de desarrollo es C++ y cuenta con una API muy robusta que cubre prácticamente cualquier necesidad.

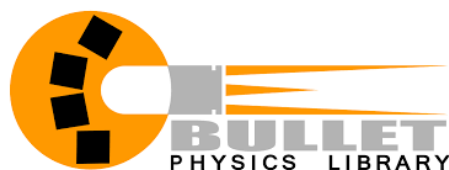
Enlace: <https://www.unrealengine.com/en-US/>



3.1.2 Librerías

3.1.2.1 pybullet

Este módulo de Python de simulación de físicas y robótica está enfocado al *Deep Reinforcement Learning* y está basado en un SDK⁶ desarrollado en C++. Además, pybullet soporta el renderizado, con un renderizador de CPU y visualización OpenGL y soporte para auriculares de realidad virtual.



Enlace: <https://pybullet.org/wordpress/>

⁶ Bullet Physics SDK: <https://github.com/bulletphysics/bullet3>

3.1.2.2 OpenAI Baselines

OpenAI es una empresa de investigación sobre el ámbito de la Inteligencia Artificial cuya misión es la de construir una IA general segura y beneficiosa al servicio de toda la humanidad.

En relación con esto OpenAI desarrolló *Baselines*, un set de implementaciones de algoritmos de aprendizaje por refuerzo.



OpenAI cuenta con un *toolkit* para el desarrollo y comparación de algoritmos llamado Gym⁷ compatible con TensorFlow y Theano. Gym ofrece una serie de entornos de prueba en los que probar los algoritmos que se han desarrollado para comprobar su eficacia en la resolución de diferentes problemas.

Enlace: <https://www.openai.com/>

3.1.2.3 TensorFlow

Es un *framework open-source* que ofrece una base de desarrollo sólida apoyada por diferentes herramientas de visualización e interfaces gráficas que ayudan a la comprensión y análisis de resultados. TensorFlow se centra en la simplicidad y la facilidad de uso. Keras, TensorBoard, TensorForce, ... son algunos de los componentes del *framework* de TensorFlow.



Enlace: <https://www.tensorflow.org/>

3.1.2.4 ML-Agents

El *toolkit* de Unity para realizar entrenamientos de agentes inteligentes. Basado en PyTorch⁸, ML-Agents provee de una serie de agentes preentrenados en modelos de 2D y 3D. Además, facilita el entrenamiento de agentes nuevos gracias a una API sencilla y accesible.

Incorpora más de una decena de escenarios de prueba prediseñados y dos algoritmos (SAC y PPO) que se pueden utilizar para el entrenamiento de modelos. Permite el entrenamiento en múltiples escenarios de manera simultánea y la randomización del espacio para desafiar a agentes robustos.



⁷ OpenAI Gym: <https://gym.openai.com/>

⁸ PyTorch: <https://pytorch.org/>

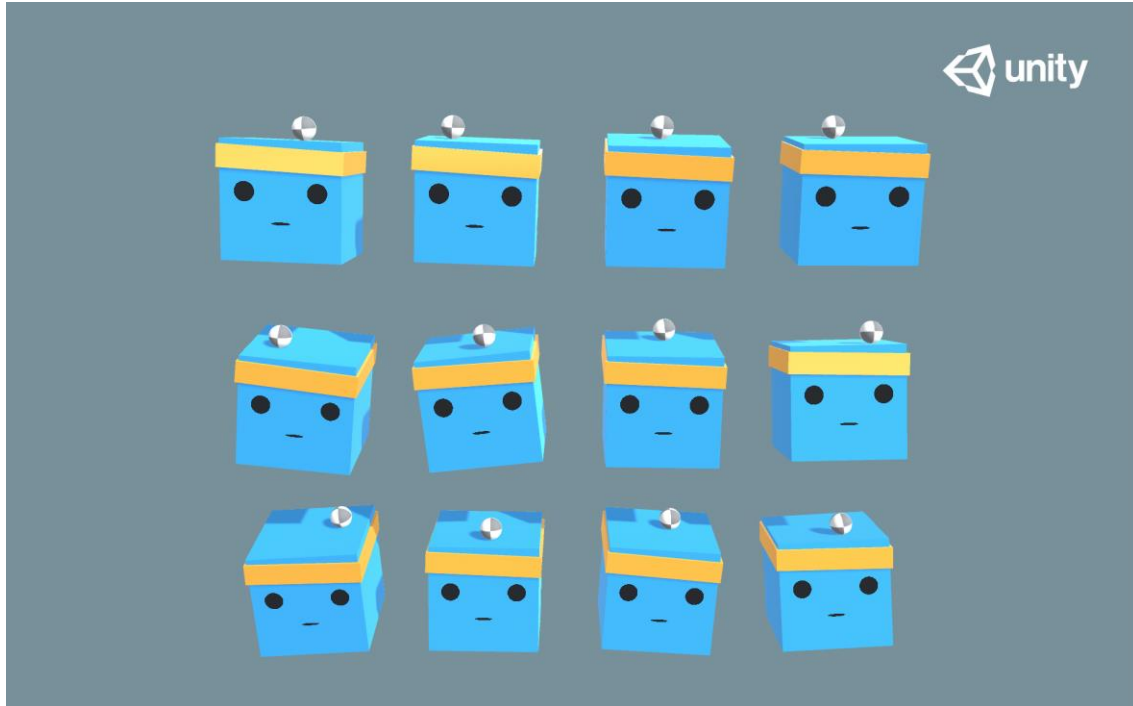


Ilustración 36: Escenario 3DBall de ML-Agents

Enlace: <https://github.com/Unity-Technologies/ml-agents>

Finalmente, se han elegido para el desarrollo de las experimentaciones Unity, como motor de renderizado, y ML-Agents, como librerías de aprendizaje. La gran compatibilidad entre ambas y la experiencia previa con Unity, además de su capacidad para plantear los requisitos que daban forma al proyecto han sido las razones por las que se han elegido estas opciones.

3.2 METODOLOGÍA

Para la fase de experimentación se ha partido de un modelo 3D de un robot articulado prediseñado por parte del equipo de Unity-Technologies que se encuentra disponible en su repositorio de GitHub⁹. El escenario que ofrecen es el que se presenta en la Ilustración 36 que está compuesto por el robot y un cubo de color rojo como elementos principales. El objetivo del robot es el de acercar la pinza del robot a la posición del cubo y se puede visualizar una demostración de esto gracias al entrenamiento que incluye al descargar el entorno.

⁹ Robot articulado de Unity-Technologies: <https://github.com/Unity-Technologies/articulations-robot-demo/tree/mlagents-package>

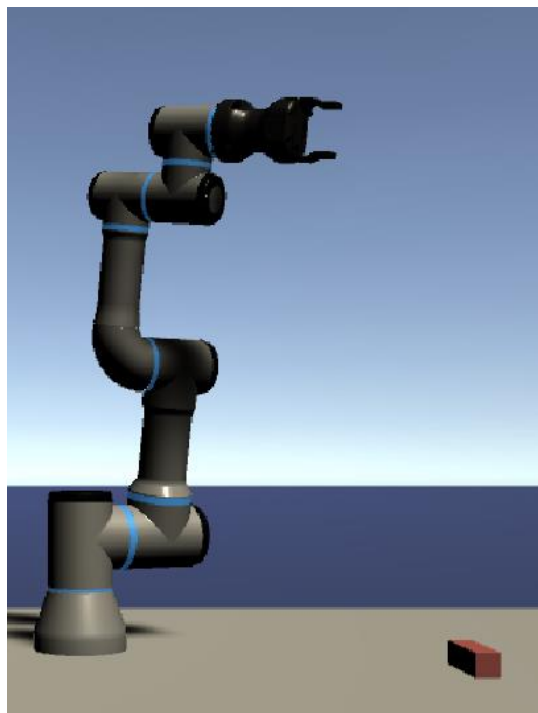


Ilustración 37: Escenario de partida para la experimentación

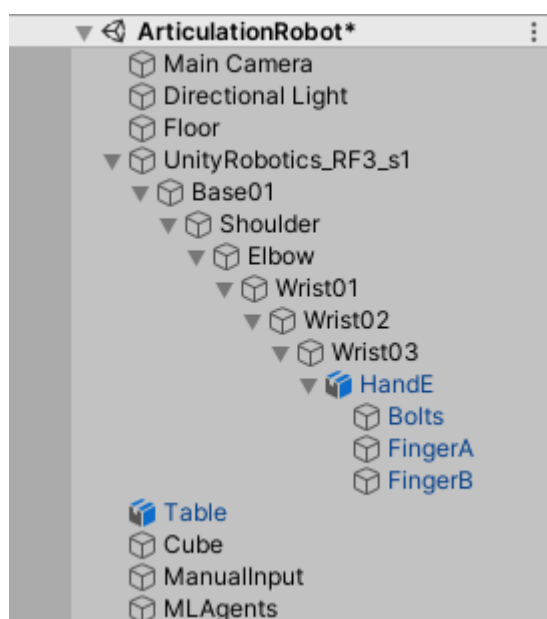


Ilustración 38: Componentes de la escena de Unity

La Ilustración 37 muestra los diferentes elementos que componen la escena de Unity en la que se entrena el agente. Además de tener los elementos por defecto de una escena (como pueden ser la *Main Camera* y la *Directional Light*, cabe destacar el *GameObject* que representa al robot articulado con un desglose de sus diferentes articulaciones, el objeto *ManualInput* y el objeto

MLAgents. En este último se encuentran el script *Behavior Parameters*, en el que se especifica el espacio de acciones, el vector de observaciones, el modelo para la ejecución y el script *RobotAgent* encargado de la gestión de recompensas. Además, si el objeto *ManualInput* está activo en la escena, las articulaciones del robot serán controlables a través de entradas de teclado, mientras que si está inactivo el robot se moverá de manera autónoma en base al modelo cargado en el campo “*Model*” del objeto *MLAgents*.

El espacio de acciones del modelo original se corresponde con las diferentes articulaciones del robot (un total de 7 sin contar el control del cierre de la pinza) y cuenta con una serie de observaciones relativas a la posición del objetivo que tiene que alcanzar (cubo rojo de la imagen), la posición de la pinza del robot en relación con la posición del propio robot y la distancia que separa la pinza del cubo. Con estas tres observaciones y la lógica establecida en los scripts del robot se establece la base para que el agente realice los entrenamientos.

A continuación, se listan los diferentes scripts que permiten el funcionamiento del robot y modelan su comportamiento:

- **ArticulationHandManualInput.cs**
Además de desarrollar un comportamiento automático a través del aprendizaje, también es posible controlar al robot de manera manual a través de entradas de teclado. Este script controla la pinza del robot y se asigna como componente del *GameObject ManualInput*.
- **ArticulationJointController.cs**
Este script se encarga de la gestión de las articulaciones y sus rotaciones estableciendo, a través de una enumeración, las diferentes direcciones de cada una (0 = parada, 1 = sentido horario, 2= sentido antihorario).
- **PincherController.cs**
De la misma manera que el script anterior controla las articulaciones, este script controla el grado de apertura y cierre de la pinza del robot. A través de la posición de los componentes *FingerA* y *FingerB* calcula el *grip* de la pinza y su punto medio.
- **PincherFingerController.cs**
Establece la posición de los objetos *FingerA* y *FingerB* en su eje de desplazamiento.
- **RobotController.cs**
Cuenta con una lista de *Joints*, una estructura compuesta por el eje de acción de la articulación y el objeto correspondiente a cada una de ellas. Controla la estructura general del robot con métodos que bloquean las rotaciones, fuerzan su posición a una concreta o resetean su posición a valores originales.
- **RobotManualInput.cs**
El segundo script, junto con *ArticulationHandManualInput* encargado del control manual del robot cuando el objeto *ManualInput* está activado.
- **TablePositionRandomizer.cs**
 - Gestiona la posición del cubo rojo dentro de la mesa en la que está situado junto con el robot. Controla que la posición del cubo es aleatoria dentro de los límites

de la mesa y que, además, esté comprendida entre la distancia de alcance máxima y mínima del robot.

- **TouchDetector.cs**

Script encargado de identificar que la pinza del robot ha tocado el cubo rojo. El calculo que identifica este método utiliza dos etiquetas que se asignan a los objetos *HandE* y *Cube* en el editor de Unity. En el momento en el que se cumple el evento *OnCollisionEnter* entre los *colliders* (estructuras que identifican impactos en Unity) de ambos objetos, este script devuelve el booleano *true*.

- **RobotAgent.cs**

Este script es el encargado de recoger las observaciones del entorno, reestablecer el escenario en el inicio de cada episodio del entrenamiento y de gestionar las recompensas que se asignaran a las acciones del agente.

Además de estos scripts, los entrenamientos de los agentes inteligentes requieren de un archivo de configuración de hiperparámetros relativos a la duración de los episodios de entrenamiento, la tasa de aprendizaje de la red neuronal o el algoritmo utilizado para el entrenamiento, entre otros muchos.

NOTA: El repositorio de GitHub en el que se encuentra el modelo del robot articulado incluye un archivo de configuración *.yaml* con los hiperparámetros del entrenamiento. Sin embargo, para el proyecto actual ha sido necesario actualizar el archivo a una versión diferente. La información relativa a la migración de versiones está disponible en la documentación del repositorio¹⁰.

En este proyecto no se ha experimentado con la influencia de los hiperparámetros en los resultados de los entrenamientos del agente. Por el contrario, se han hecho modificaciones en los scripts que modelan el comportamiento del agente para intentar cumplir el objetivo del proyecto. Si el objetivo inicial del modelo descargado es el de tocar el cubo, el de este proyecto es conseguir que el robot agarre el cubo entre los *fingers*.

3.2.1 Entrenamiento

Todos los entrenamientos se han realizado bajo el algoritmo SAC de ml-agents.

A pesar de que el modelo descargado desde el repositorio de Unity Technologies contaba con un pre-entrenamiento, en las aproximaciones planteadas en este apartado se ha entrenado al agente desde cero. De esta manera, se percibe el aprendizaje del agente en su totalidad sin partir de una base en estados avanzados.

Cada aproximación cuenta con una explicación de los ajustes que utilizan para el entrenamiento y un enlace a un video en el que se muestra el comportamiento del agente una vez finalizado cada entrenamiento. Los resultados obtenidos se muestran al final del capítulo actual.

¹⁰ Migración documento de configuración: https://github.com/Unity-Technologies/ml-agents/blob/release_3_docs/docs/Migrating.md

3.2.1.1 Aproximación 1

En el primer planteamiento para el entrenamiento se realizan las siguientes modificaciones:

- Nueva clase **PinchDetector.cs**:
Del mismo modo que *TouchDetector.cs* evalúa si se cumple que el robot toca el cubo con su pinza, esta clase comprueba que el punto medio entre los *fingers* de la pinza del robot está en el mismo lugar que el cubo y que ambos *fingers* están impactando con el objeto. Si es así y el *grip* de la pinza es mayor que cero, se considera que el robot está agarrando el cubo.
- Modificación en la clase **PincherController.cs**
Se añaden los métodos **IsOnCube** y **V3Equal**. El primero de los utiliza al segundo, que recibe dos objetos Vector3 y calcula que la distancia entre ambos es menor a 0.0001. De esta manera, se asume que los dos Vector3 ocupan la misma posición. Se utiliza con la posición del cubo objetivo y el punto medio entre los *fingers* de la pinza del robot.
- Modificación en la clase **RobotAgent.cs**
Se modifica para añadir la comprobación de eventos creada en la clase *PinchDetector* y una llamada al método *IsOnCube* de la clase *PincherController*. *IsOnCube* se introduce para fomentar que el agente posicione el punto central de su pinza en la misma posición que el cubo. Sin embargo, como ambos métodos añaden una recompensa al agente y dan por finalizado el episodio cada vez que se cumplen las condiciones que especifican, no se evalúa en ningún momento que la pinza haya agarrado el cubo, puesto que el evento que comprueba *IsOnCube* es un requisito de este objetivo y lo finaliza antes de que se dé.

```

if (pincherController.IsOnCube ())
{
    SetReward(2.0f);
    EndEpisode();
}

if (pinchDetector.hasPinchedTarget)
{
    SetReward(5.0f);
    EndEpisode();
}

```

Ilustración 39: Extracto de código de la clase RobotAgent.cs en la primera aproximación

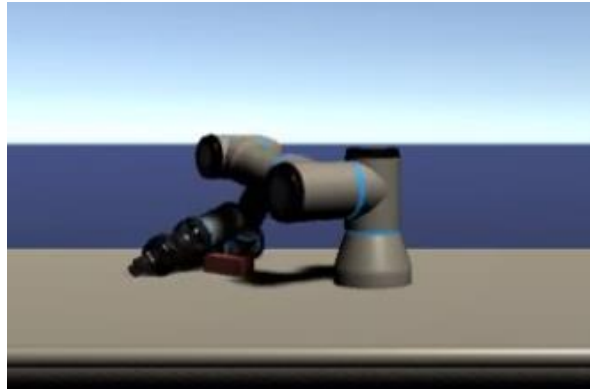


Ilustración 40: Fotograma del entrenamiento bajo la Aproximación 1

En la Ilustración 40 se puede observar cómo el agente establece contacto con el objetivo utilizando la parte posterior de la pinza en lugar de hacerlo con el extremo de la pinza. Este comportamiento se revisa en la Aproximación 3.

Comportamiento del agente tras el entrenamiento bajo la primera aproximación:

<https://www.youtube.com/watch?v=TwEji4Jk8Sk>

3.2.1.2 Aproximación 2

Con la intención de conseguir acciones más directas y sin tanta exploración, en esta aproximación se añade una penalización por cada acción tomada por el agente. De esta manera, y buscando maximizar su recompensa, el agente buscaría reducir el impacto de este cambio y maximizar su recompensa aproximándose al objetivo de manera más eficiente.

De manera general, se mantienen los ajustes realizados en la aproximación anterior pero, además de la penalización comentada en el párrafo anterior, se elimina la llamada al método *EndEpisode* del método *IsOnCube* en la clase *RobotAgent* de manera que el agente pueda seguir tomando acciones de cara a conseguir el evento de pinzado. Este planteamiento es igual para las aproximaciones siguientes. Todas las aproximaciones pares mantienen los ajustes de sus respectivas aproximaciones anteriores y añaden la penalización por acción.

```
//Energy penalization
AddReward(-0.001f);

if (pincherController.IsOnCube())
{
    AddReward(0.1f);
}

if (pinchDetector.hasPinchedTarget)
{
    SetReward(1.0f);
    EndEpisode();
}
```

Ilustración 41: Extracto del código de la clase RobotAgent.cs en la segunda aproximación

Comportamiento del agente tras el entrenamiento bajo la segunda aproximación:
<https://www.youtube.com/watch?v=ixuqgoXhhs0>

3.2.1.3 Aproximación 3

Unity cuenta con la posibilidad de etiquetar los objetos que se crean en una escena. De esta manera es más fácil la gestión y comprobación de ciertos eventos. Por ejemplo, en el entorno del proyecto el robot original cuenta con la etiqueta “*Pincher*”. Dado que las colisiones se comprueban en base a qué etiquetas tienen los objetos que han colisionado y que los objetos tienen un área de impacto que se extiende por todo su componente collider, el evento en el que se valora si la pinza del robot ha tocado el cubo se cumple a pesar de que la colisión se dé con la base o los laterales de la mano del robot en lugar de con los *fingers*. En la Ilustración 40 se muestra de manera más clara la superficie de colisión.

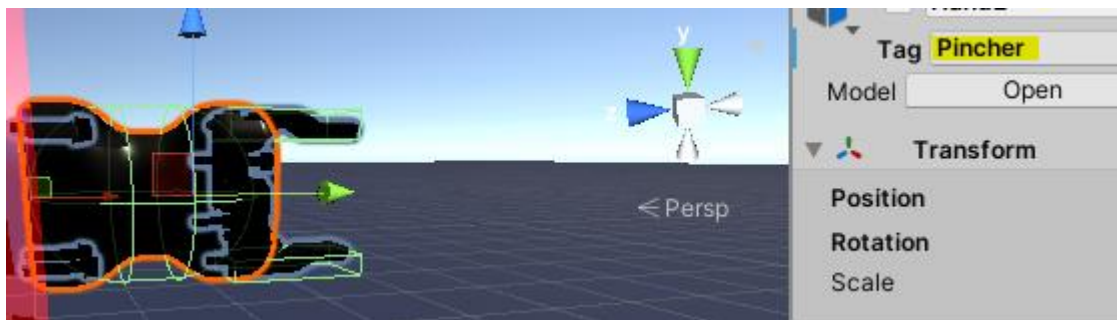


Ilustración 42: Collider de la pinza del modelo 3D

Esta configuración hacía que el robot explotase la recompensa obtenida por el evento de contacto en lugar de explorar la posibilidad de agarrar el cubo, por lo que se optó por asignar una etiqueta diferente a los *fingers* de la pinza del robot y evaluar el éxito de las colisiones en base a esa etiqueta.

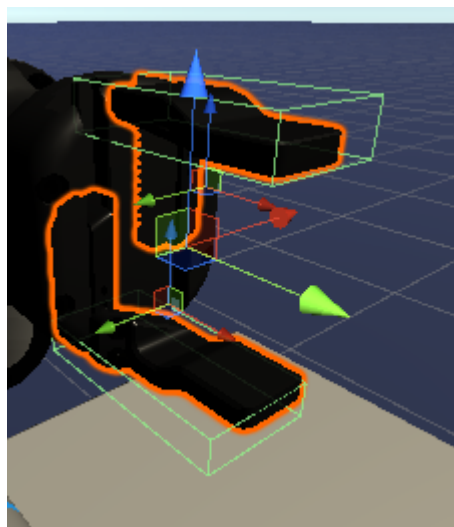


Ilustración 43: Colliders de los objetos *FingerA* y *FingerB*

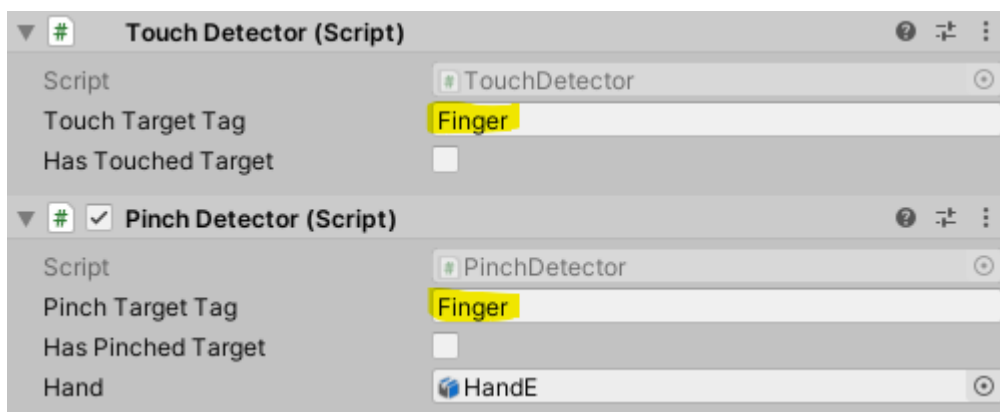


Ilustración 44: Asignación de Tag "Finger" para la comprobación de colisiones

Tanto el script *TouchDetector* como el script *PinchDetector* (ambos componentes del cubo objetivo) utilizan la etiqueta "Finger" para comprobar que el objeto contra el que han colisionado son los *fingers* de la pinza del robot. La diferencia entre ambos es que *TouchDetector* solamente identifica una colisión, mientras que *PinchDetector* comprueba que se realicen dos colisiones. Sólo se considera que la pinza ha agarrado el cubo cuando se identifican dos colisiones con objetos con la etiqueta "Finger" simultáneamente. En el caso de que sólo exista una colisión, a pesar de que el resto de las condiciones se cumplan, no se considerará como agarre.

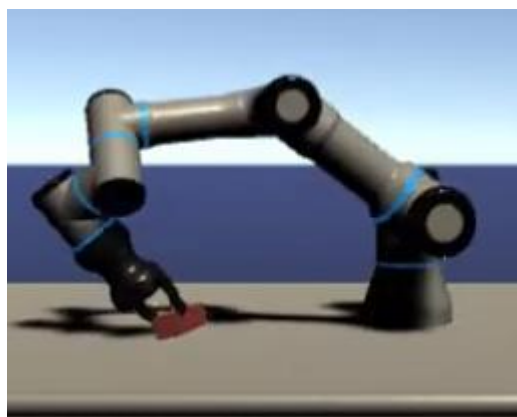


Ilustración 45: Fotograma del entrenamiento bajo la Aproximación 3

Comportamiento del agente tras el entrenamiento bajo la tercera aproximación:
<https://www.youtube.com/watch?v=pDrSf06yOsU>

3.2.1.4 Aproximación 4

Aplica los mismos ajustes que la aproximación anterior añadiendo el factor de penalización por cada acción tomada.

Comportamiento del agente tras el entrenamiento bajo la cuarta aproximación:

<https://www.youtube.com/watch?v=Sozlp4FwROg>

3.2.1.5 Aproximación 5

En las aproximaciones anteriores el control sobre las acciones de la pinza era intuitivo y se buscaba perfeccionar la aproximación del robot al cubo, de manera que pudiese situar la pinza en la misma posición que el objetivo. En esta aproximación, se añade una nueva rama al vector de acciones discreto del agente de manera que aprenda a manejar, no solo las articulaciones del cuerpo, sino también el *grip* de la pinza.

Además, se añaden estos cambios en los *scripts*:

- En la clase *PincherController* se añade un método para controlar que la pinza deje de abrirse o cerrarse y se mantenga en estado *Fixed*.
En la clase *RobotAgent* se ha añadido la gestión de las decisiones del agente sobre el comportamiento de la pinza.
- El método *V3Equal* que calcula si dos objetos están en la misma posición se modifica para que sea más permisivo. No era realista establecer un valor demasiado pequeño puesto que la diferencia de posiciones puede ser grande a pesar de que los objetos puedan estar tocándose. Es importante dar margen para el grosor de los cuerpos.

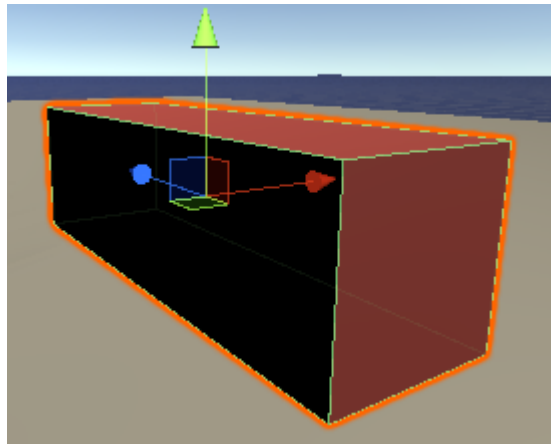


Ilustración 46: V3 del cubo junto con collider

Comportamiento del agente tras el entrenamiento bajo la quinta aproximación:

<https://www.youtube.com/watch?v=8l3YVWqt-T0>

3.2.1.6 Aproximación 6

Aplica los mismos ajustes que la aproximación anterior añadiendo el factor de penalización por cada acción tomada.




Comportamiento del agente tras el entrenamiento bajo la sexta aproximación:

<https://www.youtube.com/watch?v=OeVWQn4ntPE>

3.2.2 Resultados de la experimentación

Tras completar los seis entrenamientos e introducir diferentes variantes en el planteamiento, no se ha conseguido el objetivo inicial. El robot no logra pinzar el objeto de manera clara. Sin embargo, se pueden observar comportamientos claros bajo ciertas aproximaciones, principalmente la tercera y la cuarta, en la que el acercamiento del robot al cubo objetivo es claro y preciso. Estos dos entrenamientos son los que utilizan la etiqueta “*Finger*” para evaluar las colisiones con el cubo por lo que buscan tocar el objetivo con la punta de la pinza en lugar de con cualquier parte de ella.

La información del desempeño de los modelos entrenados se ha analizado utilizando las gráficas que genera TensorFlow. En la tabla siguiente se muestran los códigos de colores mediante los que se representan los diferentes entrenamientos en los gráficos

Aproximación 1	
Aproximación 2	
Aproximación 3	
Aproximación 4	
Aproximación 5	
Aproximación 6	

En el análisis realizado de los resultados se ha observado que, de manera general, los modelos se estabilizan en fases tempranas de los entrenamientos. Los modelos entrenados bajo las aproximaciones 5 y 6 (en las que se añade una rama más al vector de acciones) distan más del resto en las gráficas. Al tener control sobre más acciones del agente y tomar decisiones sobre el grado de apertura de la pinza del robot, obtiene recompensas numéricas mayores. Una vez que el agente acerca la pinza al cubo, las acciones realizadas con los *fingers* de la pinza del robot no lo alejan del objetivo, pero sigue recibiendo recompensas positivas por la distancia que le separa del objetivo.

Todos los entrenamientos se han realizado hasta alcanzar los 500.000 pasos, pero se puede apreciar como a partir de los 200.000 pasos, las recompensas de los agentes no varían de manera significativa. En las ilustraciones 44, 45, 46 y 47 se muestra como los modelos mejoran rápidamente sus recompensas y se mantienen en niveles parecidos a lo largo del entrenamiento llegando a hacerlo incluso antes de los 100.000 pasos. Por lo general todos los modelos que aplican penalización por cada acción tomada (Aprox. 2, 4, 6) tienen recompensas ligeramente más bajas que sus aproximaciones sin penalización (Aprox. 1, 3, 5) pero llegan a realizar acercamientos al objetivo más eficientes en el entorno virtual.

3. EXPERIMENTACIÓN



Ilustración 47: *Cumulative Reward* de los modelos entrenados con las Aprox. 1 y Aprox.2



Ilustración 48: *Cumulative Reward* de los modelos entrenados con las Aprox. 3 y Aprox.4

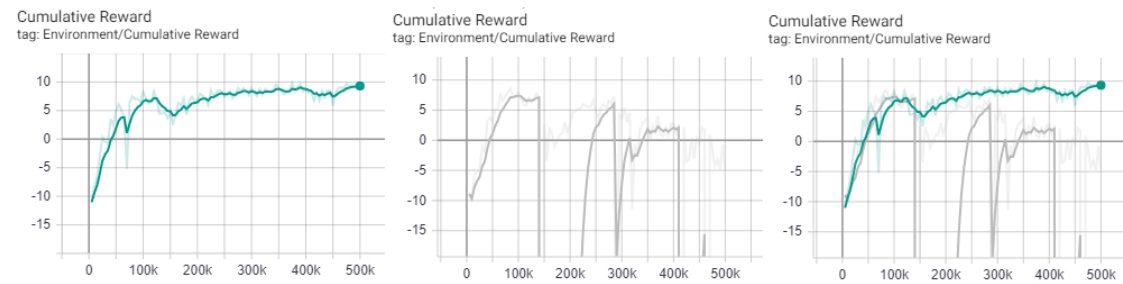


Ilustración 49: *Cumulative Reward* de los modelos entrenados con las Aprox. 5 y Aprox.6

Ilustración 50: *Cumulative Reward* de los seis modelos entrenados

Se puede observar cómo, de manera repentina, el modelo de la sexta aproximación tiene la misma tendencia que el modelo de la aproximación 5 hasta el paso 140.000, pero desciende en picado de golpe y que lo vuelve a hacer varias veces más sin lograr alcanzar el nivel de la aproximación anterior. Este fenómeno es conocido como *Catastrophic Forgetting* (Cahill, A. 2001) y ocurre cuando el agente olvida todo el aprendizaje previo como si nunca lo hubiese realizado. También podemos visualizar el mal desarrollo del entrenamiento del modelo observando su *Policy Loss*. Este indicador se corresponde con lo que cambia una política con el paso del tiempo, por lo que debería reducirse a medida que el agente aprende. Sin embargo, vemos que a diferencia del modelo de la aproximación 5 (verde), el modelo de la aproximación 6 (gris) crece a medida que avanza el entrenamiento.

Ilustración 51: *Policy Loss* de los modelos entrenados con las Aprox. 5 y Aprox. 6

4. CONCLUSIONES

Tras analizar los resultados de los entrenamientos, se puede observar cómo numéricamente ninguno supone una mejora considerable salvo el realizado añadiendo una rama al vector acción del agente. De todas formas, esto no supone que el agente haya desarrollado un comportamiento exitoso respecto del objetivo que debe cumplir. Al comparar los comportamientos en el entorno virtual no existe mucha diferencia entre los entrenamientos realizados bajo esta aproximación.

Una reflexión extraída a lo largo de la experimentación es que centrarse en modificar el código que modela el comportamiento del agente, añadiendo comprobaciones para afinar los acercamientos o métodos auxiliares, deja cada vez menos margen para que el agente desarrolle un comportamiento inteligente a través de la exploración. Es posible que cambios en el entorno o en los scripts ayuden a que el agente aprenda más rápidamente o identifique eventos que le proporcionan recompensas mejores. Pero añadir demasiados cambios de este carácter acerca cada vez más el sistema a un proyecto de programación clásica en lugar de a uno de aprendizaje por refuerzo.

En experimentaciones posteriores sería interesante comparar resultados y evoluciones de comportamientos bajo la incidencia de diferentes algoritmos (como los planteados en el Capítulo 2). Es posible que entrenar al agente en la misma tarea, pero utilizando otros algoritmos, proporcione modelos más eficientes, más efectivos y en definitiva mejores. De la misma manera, también sería interesante considerar la modificación de los hiperparámetros que se utilizan para entrenar los modelos de cara a valorar su influencia en los resultados finales. El factor del tiempo no ha marcado demasiado la diferencia entre los aprendizajes, pero es posible que, bajo otros planteamientos, entrenar los modelos durante más tiempo ayude a desarrollar comportamientos más eficaces.

Además, como se ha comentado anteriormente, diferentes algoritmos plantean enfoques diferentes para problemáticas similares. Teniendo en cuenta esto, podría ampliarse el alcance del proyecto o modificarlo para plantear situaciones de visibilidad parcial del agente, comunicación y coordinación con otros agentes o priorización de objetivos, por ejemplo.

Este proyecto ha sido muy útil para adquirir las nociones básicas del paradigma *Reinforcement Learning*, sus algoritmos principales y experimentar de manera superficial con entornos virtuales. Ha resultado ser un ámbito muy interesante y en el que existe gran margen de investigación aún.

5. BIBLIOGRAFÍA

- Abdallah, S., & Kaisers, M. (2013, May). Addressing the policy-bias of Q-learning by repeating updates. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems* (pp. 1045-1052).
- Barlow, H. B. (1989). Unsupervised learning. *Neural computation*, 1(3), 295-311.
- Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 253-279.
- Cahill, A. (2011). *Catastrophic Forgetting in Reinforcement-Learning Environments* (Thesis, Master of Science). University of Otago. Retrieved from <http://hdl.handle.net/10523/1765>
- Caruana, R., & Niculescu-Mizil, A. (2006, June). An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning* (pp. 161-168).
- Castaneda, A. O. (2016). *Deep reinforcement learning variants of multi-agent learning algorithms*. Master's thesis, School of Informatics, University of Edinburgh.
- Chen, M., Wang, T., Ota, K., Dong, M., Zhao, M., & Liu, A. (2020). Intelligent resource allocation management for vehicles network: An A3C learning approach. *Computer Communications*, 151, 485-494.
- Fan, L., Liu, Y. Y., & Zhang, S. (2018, July). Partially Observable Multi-Agent RL with Enhanced Deep Distributed Recurrent Q-Network. In *2018 5th International Conference on Information Science and Control Engineering (ICISCE)* (pp. 375-379). IEEE.
- Feinberg, E. A., & Shwartz, A. (Eds.). (2012). *Handbook of Markov decision processes: methods and applications* (Vol. 40). Springer Science & Business Media.
- Foerster, J., Assael, I. A., De Freitas, N., & Whiteson, S. (2016). Learning to communicate with deep multi-agent reinforcement learning. In *Advances in neural information processing systems* (pp. 2137-2145).
- Foerster, J., Nardelli, N., Farquhar, G., Afouras, T., Torr, P. H., Kohli, P., & Whiteson, S. (2017). Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*.
- Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., ... & Blundell, C. (2017). Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*.
- Gallager, R. G., "Claude E. Shannon: a retrospective on his life, work, and impact," in *IEEE Transactions on Information Theory*, vol. 47, no. 7, pp. 2681-2695, Nov. 2001, doi: 10.1109/18.959253.
- Gupta, J. K., Egorov, M., & Kochenderfer, M. (2017, May). Cooperative multi-agent control using deep reinforcement learning. In *International Conference on Autonomous Agents and Multiagent Systems* (pp. 66-83). Springer, Cham.

- Harmon, M. E., Baird, L. C., & Klopff, A. H. (1995). Reinforcement Learning Applied to a Differential Game. *Adaptive Behavior*, 4(1), 3–28. <https://doi.org/10.1177/105971239500400102>
- Hausknecht, M., & Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*.
- Hong, Z. W., Su, S. Y., Shann, T. Y., Chang, Y. H., & Lee, C. Y. (2017). A deep policy inference q-network for multi-agent systems. *arXiv preprint arXiv:1712.07893*.
- Kong, X., Xin, B., Liu, F., & Wang, Y. (2017). Revisiting the master-slave architecture in multi-agent deep reinforcement learning. *arXiv preprint arXiv:1712.07305*.
- Meuleau, N., Peshkin, L., Kaelbling, L. P., & Kim, K. E. (2000). Off-policy policy search. MIT Artificial Intelligence Laboratory.
- Narvekar, S., Sinapov, J., & Stone, P. (2017, August). Autonomous Task Sequencing for customized Curriculum Design in Reinforcement Learning. In *IJCAI* (pp. 2536-2542).
- Narvekar, S., & Stone, P. (2018). Learning curriculum policies for reinforcement learning. *arXiv preprint arXiv:1812.00285*.
- Omidshafiei, S., Pazis, J., Amato, C., How, J. P., & Vian, J. (2017). Deep decentralized multi-task multi-agent reinforcement learning under partial observability. *arXiv preprint arXiv:1703.06182*.
- Palmer, G., Tuyls, K., Bloembergen, D., & Savani, R. (2017). Lenient multi-agent deep reinforcement learning. *arXiv preprint arXiv:1707.04402*.
- Park, K. H., Kim, Y. J., & Kim, J. H. (2001). Modular Q-learning based multi-agent cooperation for robot soccer. *Robotics and Autonomous Systems*, 35(2), 109-122.
- Pavlov, I. P., & Gantt, W. H. (1941). *Conditioned reflexes and psychiatry* (Vol. 2). New York: International Publishers.
- Plappert, M., Houthoofd, R., Dhariwal, P., Sidor, S., Chen, R. Y., Chen, X., ... & Andrychowicz, M. (2017). Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*.
- Potter, M. A., & De Jong, K. A. (1994, October). A cooperative coevolutionary approach to function optimization. In *International Conference on Parallel Problem Solving from Nature* (pp. 249-257). Springer, Berlin, Heidelberg.
- Singh, A., Jain, T., and Sukhbaatar, S. Learning when to communicate at scale in multiagent cooperative and competitive tasks. *arXiv preprint arXiv:1812.09755*, 2018.
- Sukhbaatar, S., & Fergus, R. (2016). Learning multiagent communication with backpropagation. *Advances in neural information processing systems*, 29, 2244-2252.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
- Thorndike, E. L. (1927). The law of effect. *The American journal of psychology*, 39(1/4), 212-222.
- Turing, A. M.. (1988). Chess. In *Computer chess compendium*. New York, NY: Springer.
- Verbeeck, K., Nowe, A., & Tuyls, K. (2003). Coordinated exploration in stochastic common interest games. In *Proceedings of Third Symposium on Adaptive Agents and Multi-agent Systems*.

- Wang, B., & Hegde, N. (2019). Privacy-preserving q-learning with functional noise in continuous spaces. In *Advances in Neural Information Processing Systems* (pp. 11327-11337).
- Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995-2003). PMLR.
- Wang, R. E., Everett, M., & How, J. P. (2020). R-maddpg for partially observable environments and limited communication. *arXiv preprint arXiv:2002.06684*.
- Widrow, B., Gupta, N. K., & Maitra, S. (1973). Punish/reward: Learning with a critic in adaptive threshold systems. *IEEE Transactions on Systems, Man, and Cybernetics*, (5), 455-465.
- Zheng, Y., Meng, Z., Hao, J., & Zhang, Z. (2018, August). Weighted double deep multiagent reinforcement learning in stochastic cooperative environments. In *Pacific Rim international conference on artificial intelligence* (pp. 421-429). Springer, Cham.
- Zhu, P., Li, X., Poupart, P., & Miao, G. (2017). On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978*.
- Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., ... & He, Q. (2020). A comprehensive survey on transfer learning. *Proceedings of the IEEE*.