GRADO EN INGENIERÍA EN TECNOLOGÍA DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

APRENDIZAJE PROFUNDO POR REFUERZO PARA AUTÓMATAS CELULARES MULTI-POBLACIONALES: DISEÑO DE POLÍTICAS LOCALES, GRUPALES Y GLOBALES

Alumno: Fernández López-Areal, Mateo

Director: Del Ser Lorente, Javier

Curso: 2019-2020

Fecha: Viernes, 10 de Julio de 2020

PAGINA EN BLANCO

Resumen trilingüe

Resumen

Este Trabajo de Fin de Grado consiste en la implementación de estrategias de aprendizaje con Machine Learning para adaptarlas a un entorno en específico. El entorno tiene varios agentes, cada uno haciendo uso de un modelo de Deep Reinforcement Learning. El objetivo es, mediante este entorno, simular las políticas intrínsecas y extrínsecas de una civilización, y conseguir ver que comportamiento tienen los agentes con estas. Al agente, se le aplicarán políticas locales, grupales y globales.

Palabras clave: Inteligencia Artificial, Redes neuronales, Aprendizaje Automático, Aprendizaje por Refuerzo, Aprendizaje Profundo, Multiagente.

Laburpena

Gradu Amaierako Lan hau Machine Learning-ren bitartez ingurune espezifiko batera egokitzeko ikasteko estrategiak ezartzean datza. Inguruak hainbat eragile ditu, bakoitzak Deep Reinforcement Learning eredua erabiltzen du. Inguru honen bitartez zibilizazio baten barruko eta kanpoko politikak simulatzea eta agenteek hauekiko duten jarrera ikustea du helburu. Tokiko, taldeko eta mundu osoko politikak aplikatuko zaizkio eragileari.

Hitz gakoak: Adimen artifiziala, Neurona-sare, Ikasketa automatiko, Ikasketa errefortzu bidez, Ikaskuntza sakon, Multiagentea.

Abstract

This Final Degree Project consists of the implementation of Machine Learning strategies to adapt them to a specific environment. The environment has various agents, each one making use of a Deep Reinforcement Learning model. The goal is to simulate, by using this environment, the intrinsic and extrinsic policies of a civilization, and see how the agents interact with these. Local, group and global policies will be applied to the agent.

Keywords: Artificial Intelligence, Neural Networks, Machine Learning, Reinforcement Learning, Deep Learning, Multiagent.

Tabla de contenido

Lista de ilustraciones

Lista de ecuaciones

Lista de algoritmos

Lista de tablas

Lista de gráficas

Lista de acrónimos

IA – Inteligencia Artificial

ML – Machine Learning

DL – Deep Learning

RL – Reinforcement Learning

DNN – Deep Neural Network

TRL – Technology readiness level

DQN – Deep Q-Network

DDQN – Double Deep Q-Network

PG – Policy Gradient

TRPO – Trust Region Policy Optimization

PPO – Proximal Policy Optimization

A3C – Asynchronous Advantage Actor-Critic

DDPG – Deep Deterministic Policy Gradient

SAC – Soft Actor-Critic

API – Application Programming Interface

GAIL – Generative Adversarial Imitation Learning

C# – C Sharp

1 Introducción

La finalidad de este proyecto es explorar cómo controlar las dinámicas comportamentales de sistemas multiagente multipoblacionales. Para ello se hace uso de herramientas de Inteligencia Artificial con el fin de lograr este objetivo. Se emplean técnicas de **Machine Learning** para buscar la configuración de aprendizaje más óptima.

Hoy en día el término inteligencia artificial está muy extendido, pero nadie conoce los límites que hay en el campo. Para cumplir el objetivo de este proyecto se emplean técnicas de Machine Learning, una implementación de la inteligencia artificial. Machine Learning, o ML, es una forma de asimilar los datos o estados, y proporcionar una solución o respuesta aprendida por un equipo. Un sistema completo que implemente inteligencia artificial incorpora tanto herramientas de machine learning como algoritmos más sencillos como pueden ser los árboles de comportamiento.

Ahora que ya se ha introducido el concepto de Machine Learning, se presentan los tres tipos de entrenamientos que se suelen ver en ML:

- **Unsupervised Learning**: Para este tipo de aprendizaje se examina un conjunto de datos y se realiza una clasificación de estos. Es utilizado para la agrupación (clustering), el aprendizaje de características, o la estimación de la densidad.
- **Supervised Learning**: Este es el método empleado para realizar predicciones y clasificar datos. Los datos de entrada están etiquetados, consiguiendo así salidas etiquetadas. Para estos entrenamientos se necesita de un conjunto amplio de datos etiquetados a la entrada para poder crear un modelo.
- **Reinforcement Learning**: Para el Reinforcement Learning, o RL, no se necesita modelar ni etiquetar la entrada. En RL se dispone de un entorno y un agente. El agente decide que acciones tomar sobre el entorno. El entorno es todo lo que no sea el agente, todo con lo que pueda interactuar el agente de forma directa o indirecta. El entorno cambia cada vez que el agente realiza una acción, este cambio se denomina transición de estado. Los agentes trabajan en el entorno y reciben recompensas basadas en sus acciones. Los agentes también pueden tomar observaciones para simplificar la tarea de aprendizaje.

Para este proyecto se emplea **Deep RL**, es decir Reinforcement Learning junto con Deep Learning. El Deep Learning, es un subconjunto del ML, que utiliza las redes neuronales para analizar diferentes factores con una estructura similar a la del sistema neuronal humano. Generan modelos de datos que dan una salida apropiada cuando se les da una entrada de algún tipo con un cierto valor. El más popular y el método que se utiliza en este proyecto son las redes neuronales profundas, o Deep Neural Networks.

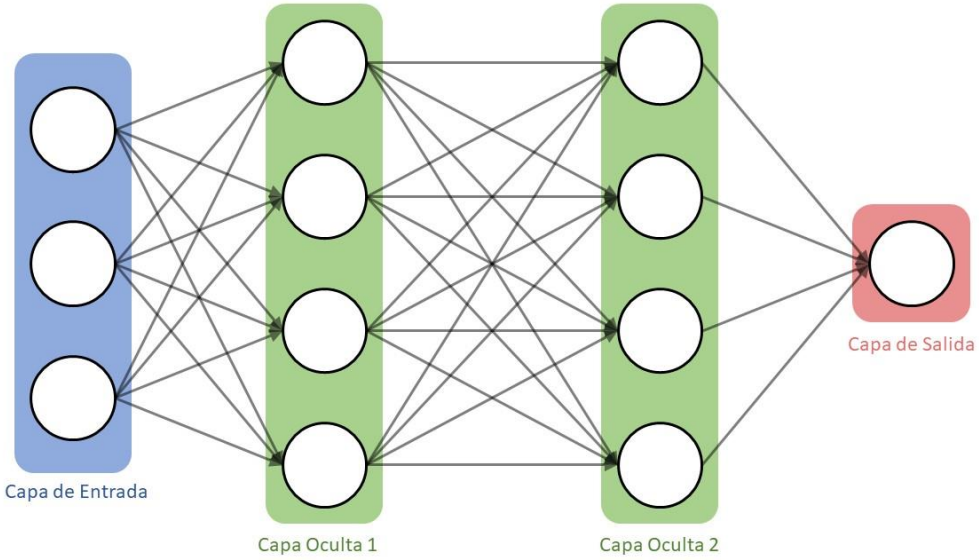


Ilustración 1: Deep Neural Network

Las **redes neuronales profundas**, o DNNs, están compuestas por una serie de nodos unidos entre sí. Se pueden catalogar en tres partes: capa de entrada o input layer, capas ocultas o hidden layers y capas de salida u output layers. En la capa de entrada se utilizan los valores necesarios para tomar una decisión. Estos valores se transforman en las capas ocultas y luego los valores deseados aparecen en la capa de salida. Los valores se transforman utilizando los números que se obtienen a través de las sesiones de entrenamiento. Cada nodo contiene una serie de valores (pesos) utilizados para modificar la entrada y proporcionar una salida procesada. El valor original entra a través de la capa de entrada y viaja hasta la capa de salida. Cuando los valores atraviesan los diferentes nodos, se transforman hasta que los valores finales llegan a la última capa. Los valores se utilizan para tomar una decisión.

Gracias al uso de esta computación neuronal, se permite desarrollar un modelo de agente de RL para el estudio de su comportamiento en un sistema multipoblacional con diferentes políticas.

2 Contexto

En esta última década, la Inteligencia Artificial se ha desarrollado tanto, que se encuentra en todos los sectores posibles a nivel global. Y gracias a su desarrollo, está consiguiendo arraigarse en la sociedad, siendo empleada en cualquier tipo de producto. Este proyecto estudia, una parte ya mencionada de la Inteligencia Artificial, el Machine Learning enfocándose, sobre todo, en la toma de decisiones del agente. Normalmente, los proyectos que implementan Machine Learning, trabajan para la optimización y predicción de datos. La aproximación tradicional es formular la toma de decisiones como un problema de optimización, y resuelto como tal.

Sin embargo, esto supone que, si cambian las condiciones del problema, hay que volver a ejecutar el algoritmo de optimización, lo que en contextos dinámicos o en sistemas complejos puede dar lugar a latencias de cómputo no asumibles. Para ello, en este proyecto se analiza la predicción de estos datos en diferentes escenarios. Aquí es donde entra el aprendizaje multiagente, y en particular Reinforcement Learning. El RL soluciona los problemas de implementar ML en escenarios complejos, y permite la opción de utilizar sistemas multiagente los cuales permiten resolver problemas que son difíciles o imposibles de resolver para un agente individual. El estudio de estos escenarios busca medir la sensibilidad del algoritmo y la interacción de múltiples agentes en un sistema multipoblacional.

3 Objetivos

3.1 Objetivo principal

El principal objetivo del proyecto es explorar cómo controlar las dinámicas comportamentales de sistemas multiagente multipoblacionales empleando aprendizaje por refuerzo profundo. Para ello se simulan las políticas intrínsecas y extrínsecas de una civilización, y se consigue ver que comportamiento tienen los agentes con estas.

3.2 Objetivos secundarios

Para la realización del proyecto se han ido marcando objetivos, con el fin de establecer distintas etapas con objetivos en cada una de ellas. El objetivo de estas etapas es contribuir al alcance del objetivo principal antes definido.

3.2.1 Diseño de un entorno multipoblacional

El objetivo de diseñar un entorno multipoblacional, en el que haya interacción entre las poblaciones y que permita validar el objetivo principal del proyecto, es la base de todo el proyecto. El entorno tiene que soportar el aprendizaje de múltiples agentes a la vez, con el fin de estudiar sus comportamientos en el entorno.

3.2.2 Creación de una red neuronal

La creación de un sistema mediante aprendizaje profundo, implica la creación de una red neuronal y el entrenamiento de la misma empleando un algoritmo de aprendizaje por refuerzo. Es necesario buscar un algoritmo que se ajuste a los requerimientos del proyecto.

3.2.3 Análisis del algoritmo con diferentes configuraciones

Analizar la sensibilidad paramétrica del algoritmo de Deep RL multiagente, e inspeccionar las diferentes dinámicas de comportamiento observadas en el entorno ante diferentes configuraciones de dicho algoritmo. Este objetivo es el que otorga los resultados de cómo controlar las dinámicas comportamentales.

4 Beneficios que aporta el trabajo

En cuanto a los beneficios, se debe tener en cuenta que este proyecto, se trata de un estudio de investigación en TRLS [5], o niveles de madurez tecnológica, bajos, por lo que los beneficios inmediatos de este no son evidentes. Aun así, sí es un estudio necesario para ver cómo implantar funcionalidades complejas en sistemas multiagente reales cuyas poblaciones compiten entre sí.

4.1 Beneficios Técnicos

Este proyecto sirve para intentar entender los límites del aprendizaje por refuerzo en un caso en concreto, por lo que como beneficio técnico se tiene el resultado logrado de la implementación desarrollada, y el análisis de alternativas de algoritmos y entornos de simulación a emplear. Sin embargo, el mayor beneficio técnico es la posibilidad de **extrapolar estos resultados obtenidos** en otro tipo de aplicaciones en el que existen multiagentes multipoblacionales, es decir en cualquier escenario donde se exija una coordinación entre diferentes grupos. Por ejemplo, en enjambres de drones para misiones de exploración, donde es crucial la coordinación y la agrupación de los drones, o computación en malla, donde se necesita distribuir el uso de los recursos debido a que no están sujetos a un control centralizado. Es decir, el conocimiento en las dinámicas de autoorganización de los autómatas celulares multipoblacionales puede ser extrapolado a los escenarios anteriores, ya que los agentes están llamados a colaborar unos con otros.

4.2 Beneficios Económicos

Apoyándonos en el beneficio anterior de la extrapolación a otras aplicaciones de autoorganización de autómatas multipoblacionales, se puede llegar también a la conclusión de que, si estos drones o el mantenimiento de la computación mallada, se realiza todo dentro del mismo equipo, no es necesario un piloto para el dron o dinamizar la computación distribuida. Los beneficios económicos se derivan de una **menor intervención humana** al proporcionar esta inteligencia, ya que la Inteligencia Artificial aprende a organizarse sin necesidad de que un humano le diga cómo se tiene que comportar. Gracias a esto, se puede conseguir reducir los costes de una empresa en cuanto a personal técnico. El hecho de dotar a una máquina de conocimiento para que pueda llevar a cabo estas acciones, al igual de bien que la podría hacer un ser humano, puede otorgar una mayor rentabilidad en cuanto a productividad y costes, a consecuencia de que, por ejemplo, una maquina no suele necesitar descansos o vacaciones.

4.3 Beneficios Sociales

En el ámbito social este estudio de investigación consigue promover un conocimiento innovador. Puede conseguir despertar la curiosidad sobre este campo y desarrollar el pensamiento crítico para una actitud científica que favorece la objetividad y la tolerancia. Pero volviendo al tema de emplear este estudio de investigación para otros campos, el hecho de

sustituir a las maquinas por humanas puede perjudicar mucho al sector laboral. Este hecho es uno de los dilemas más discutidos del Machine Learning, donde se pregunta si es moralmente correcto sustituir a maquinas por humanos. La incorporación de la automatización origina que detrás de cada máquina, pueda existir una pérdida de trabajo.

Dejando de lado el problema moral, el hecho de emplear este tipo de equipos, al tener un margen de error de menos nivel suelen ser más fiables. Incluso se pueden llegar a emplear maquinas organizadas entre ellas mediante aprendizaje profundo para explorar lugares hostiles o imposibles de acceder para el ser humano.

5 Análisis del estado del arte

5.1 Algoritmos de Reinforcement Learning

El Reinforcement Learning, o RL, está ligado a la psicología del comportamiento, o la ciencia de toma de decisiones. Es usado en muchos ámbitos del ML, y, en este caso, para la optimización en videojuegos y simulaciones. Para poder desarrollar RL, se necesita un agente y un entorno.

Un agente es una entidad la cual puede observar el entorno, y usando estas observaciones decidir acciones y ejecutarlas sobre el entorno. Es el modelo que se intenta entrenar. El entorno es todo lo que no sea el agente, todo con lo que pueda interactuar el agente de forma directa o indirecta. Es la demostración del problema a resolver.

El modelo más sencillo de RL, stateless RL, es una agente actuando en un entorno dado, consiguiendo recompensas negativas o positivas dependiendo de sus actos. A este modelo se le añade un término más, el estado. Un **estado** es cada escenario en la que el agente se encuentra en el entorno, es la situación actual. Este es el diagrama de RL:

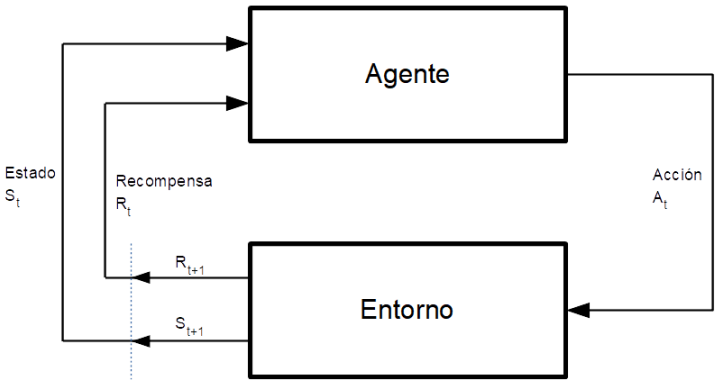


Ilustración 2: Diagrama de bloques RL

El estado, antes definido, está referenciado mediante S_t, indicando que es el estado en el que se encuentra el agente en el momento t. El estado es lo que el agente sabe del entorno, por ejemplo, donde se encuentra. Teniendo el estado en cuenta, el agente decide que acción, referenciada con A_t, tomar sobre el entorno en el momento t. Una acción es lo que puede hacer un agente en cada estado. De esta acción tomada el agente recibe una recompensa. Como se puede ver, R_t hace referencia a la recompensa que recibe un agente dependiendo de la acción tomada en el entorno. Una recompensa es una señal numérica que representa el grado de éxito de la acción en el momento t. El principal objetivo del agente es maximizar el valor de la recompensa acumulativa. Aunque la palabra recompensar signifique, premiar un beneficio o mérito; la recompensa en RL no siempre se define así, ya que, si es negativa, la recompensa se puede denominar castigo.

Dentro del agente existen más términos importantes que mencionar, como son la **política** y el **algoritmo** que sigue este. Todo algoritmo de RL tiene que seguir alguna política para que decida

qué acciones ejecuta en cada estado. La política se puede definir como el mapeo estado-acción. El principal objetivo del agente es maximizar la recompensa. Una de las estrategias para conseguir esto es intentar entender el sistema y predecir los resultados de las acciones y su recompensa futura, o, otra estrategia es que agente intente todo tipo de acciones sobre muchos estados y guarde los resultados. Cualquiera de estas dos estrategias consigue que el agente calcule una buena política, y una vez calculado, el agente solo debe mirar que acción realizar para cada estado.

El uso de esta política tiene dos fases en cada algoritmo de RL: la fase de aprendizaje o entrenamiento, y la fase de comportamiento o de post-aprendizaje. En la fase de aprendizaje, o etapa de entrenamiento, el agente necesita aprender una estimación de la política óptima. Teniendo en cuenta que el agente no conoce esta función óptima para la política, el agente necesita explorar por todos los estados posibles para llegar a esta. Durante el entrenamiento, el agente debe decidir si es mejor explorar o explotar las opciones que tiene, esto es el dilema **exploración-explotación**. Explotación es la ejecución de una acción, teniendo ya algo de conocimiento obtenido, no óptima. Exploración es la selección de una acción óptima conforme con el conocimiento obtenido del agente. Para explorar estados no vistos por el agente, este tiene que tomar decisiones subóptimas. La fase de comportamiento, en cambio, es cuando el agente tiene ya la política óptima, por lo que su labor es realizar las acciones más óptimas sobre el entorno.

Según el comportamiento dentro de la fase de aprendizaje se diferencian dos tipos de algoritmos de RL: **off-policy** y **on-policy**. Los algoritmos on-policy estiman el valor de la política mientras la usan para el control. Se trata de mejorar la política que se usa para tomar decisiones, es decir, la misma política que se evalúa y se mejora también se usa para seleccionar acciones. En cambio, los algoritmos off-policy, utilizan la política y el control de forma separada. La política de comportamiento está separada de la política que se quiere mejorar, es decir, la política que se evalúa y se mejora es diferente de la política que se usa para seleccionar acciones. Las dos clases de algoritmos más populares son Q-Learning, off-policy, y policy gradient (método de gradiente de política), on-policy.

A continuación, se puede ver donde se encuentran estos tipos de algoritmo dentro de la clasificación de algoritmos basados en modelo. En RL no basados en modelos, o model-free, se ignora el modelo, es decir, se depende completamente del muestreo y la simulación para estimar las recompensas, así que no se necesita conocer el funcionamiento interno del sistema. En la RL basado en modelo, si se puede definir una función de coste y calcular las acciones óptimas usando el modelo directamente.

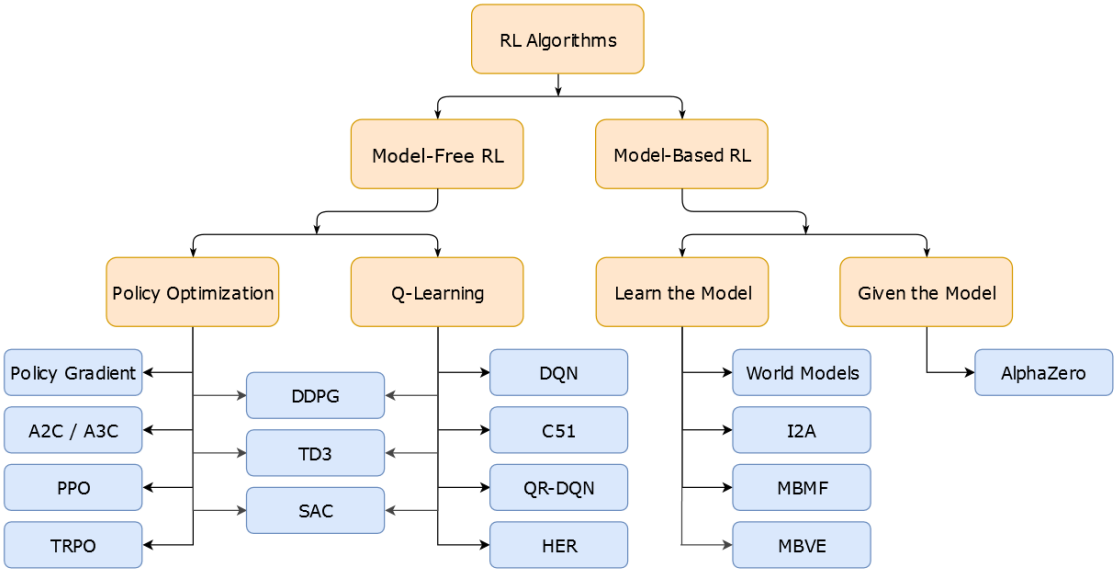


Ilustración 3: Algoritmos RL, extraído de [3]

5.1.1 Q-Learning

Q-Learning es uno de los tipos de algoritmo basados en la optimización del valor dentro de una función $Q(s_t, a_t)$. El nombre Q viene de **quality**, en este caso, la calidad representa cuan útil es una acción en conseguir una recompensa futura. $Q(s_t, a_t)$ es una función dependiente de s_t y a_t, es decir del estado y la acción en un momento t. El objetivo de los algoritmos de RL siempre es llegar a la política óptima. Con las funciones de valor óptimas, se puede llegar a la política óptima escogiendo las acciones que maximizan estas funciones. La función $Q(s_t, a_t)$ muchas veces se puede encontrar en forma de tabla, donde las filas son las acciones y las columnas son los estados. Dentro de la tabla se guarda la recompensa máxima futura esperada si realiza una acción en un estado. La expresión de la función es:

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * [R(s_t, a_t) + \gamma * maxQ(s_{t+1}, a_{t+1})]$$

Ecuación 1: Función Q

De una forma sencilla se puede decir que, el nuevo valor Q para el estado y acción en el momento t es igual al valor actual de Q más el valor del nuevo aprendizaje.

■El **nuevo valor** de Q para ese estado y acción

■El **valor actual** de Q

■**Learning rate** o ratio de aprendizaje, es el parámetro que ajusta la importancia de la recompensa adquirida. Un valor cercano a 0 otorga importancia al valor de Q actual, es decir, lo ya sabido; y un valor cercano a 1 da más importancia al nuevo aprendizaje adquirido.

■**Recompensa** por tomar esa acción en ese estado

■**Discount factor** o factor de descuento, es el parámetro que nivela la importancia de las recompensas futuras con respecto a las recompensas inmediatas. Un valor cercano a 0 otorga más importancia a R, es decir, a la recompensa inmediata; y un valor cercano a 1 da más importancia a las recompensas que se pueden adquirir en el futuro.

■La **recompensa futura máxima estimada** teniendo en cuenta el nuevo estado y todas las posibles acciones en ese estado

Una vez entendidos los términos y la fórmula, se puede ver cómo y en qué orden funciona este algoritmo:

Algoritmo Q-Learning
Se inicializa la función $Q(s, a)$ para todas las estados y acciones (p.e. a 0)
Se obtiene el estado inicial s
Se realizan iteraciones k=1, 2, … hasta llegar a convergencia o límite de tiempo
    Se realiza una acción a, y se obtiene un nuevo estado s'
        Se comprueba si s' es un estado terminal:
            Solo se deja la parte de la recompensa, $maxQ(s', a')$ = 0
            Se muestrea el nuevo estado inicial a s'
        Si no lo es:
            Se calcula el máximo de la función Q para s'
            $Q(s, a) = (1 - \alpha) * Q(s, a) + \alpha * [R(s, a) + \gamma * maxQ(s', a')]$
            Se sobrepone el valor de s' en s

Cuadro 1: Funcionamiento Q-Learning

Este algoritmo se implementa sobre entornos donde el agente tiene un número de estados y acciones escaso. Para implementaciones de mayor tamaño, debido a que la función Q queda muy extensa, no merece la pena emplear Q-Learning normal. Sin embargo, existe otro tipo de Q-Learning que arregla este problema, el Deep Q-Learning. Para desarrollar este algoritmo se hace uso de redes neuronales que permiten calcular a partir de un estado, un array de valores Q, el cual representa el valor obtenido para cada acción en ese estado, sin necesidad de guardarlo en ninguna tabla o función.

Deep Q-Learning, o DQN, hace uso de políticas ε-greedy, la cual permite a los agentes realizar acciones que no tienen por qué ser las más óptimas de forma aleatoria con el fin de acabar con el dilema de exploración-explotación. DQN también emplea Experience Replay Memory-s las cuales guardan un número N de experiencias pasadas para optimizar el rendimiento. Sin embargo, este algoritmo tiene un problema, puesto que al usar la misma red con los mismos pesos para el cálculo el valor actual de Q y del valor estimado de Q-target, aparecen los problemas de sobreestimación. De una manera muy simple, DQN funciona así:

Algoritmo DQN
Se inicializa la memoria de experiencias a una capacidad N
Se inicializa la red neuronal con pesos aleatorios
Por cada episodio
Se inicializa el estado inicial
Por cada paso (step)
Se realiza una acción a, mediante la política ε-greedy
Se ejecuta la acción y se observa la recompensa y el estado siguiente
Se almacena en la memoria de reproducción
Se actualiza la función de error:
$Q_{target} = [\, R(s,a) + \gamma * maxQ(s',a')]$ y $Loss = Q_{target} - Q_{actual}$

Cuadro 2: Funcionamiento Deep Q-Learning

Aclarar que los pesos son coeficientes que se adaptan en la red que determinan la intensidad de la señal de entrada registrada por la neurona artificial.

Finalmente mencionar también que existe otro algoritmo de Q-Learning, el cual emplea dos redes neuronales, una para el cálculo del valor de Q-actual y otra para el cálculo del valor estimado de Q. De esta forma se resuelve el problema de sobreestimación que surgía en DQN. Este algoritmo se denomina **Double Deep Q-Learning**, o DDQN.

5.1.2 Policy Gradient

El aprendizaje por refuerzo es probablemente el marco más general en el que se pueden enunciar los problemas de aprendizaje de los animales, los humanos o las máquinas relacionados con la recompensa. Sin embargo, la mayoría de los métodos propuestos en la comunidad de aprendizaje de refuerzo no son todavía aplicables a muchos problemas como en el mundo de la robótica. Esta inaplicabilidad puede ser el resultado de problemas con la información de estado incierto. La mayoría de los métodos tradicionales de aprendizaje de refuerzo no tienen garantías de convergencia. Los estados y acciones continuos en espacios de altas dimensiones no pueden ser tratados por la mayoría de los enfoques de aprendizaje de refuerzo que existen en el mercado.

Los métodos de **gradientes de política**, o **PG**, difieren significativamente, ya que no sufren estos problemas de la misma manera. Los métodos PG funcionan computando un estimador del gradiente de política y conectándolo a un algoritmo de ascenso del gradiente. En la mayoría de implementaciones este estimador se convierte en la función objetivo de tal forma que:

$$L_t^{PG}(\theta) = \hat{\mathrm{E}}_t[\nabla_\theta(\log \pi_\theta(a_t|s_t)) * \hat{\mathrm{A}}_t\,]$$

Ecuación 2: Función objetivo PG

Se puede decir que esta función tiene dos términos, el término donde se encuentra π_θ, la política; y el término de la ventaja o advantage $\hat{\mathrm{A}}_t$, que lo que busca es estimar el valor relativo

de la acción tomada. Para el cálculo de la ventaja se enuncian otros dos términos, el return (que es todas las recompensas que ha tenido el agente cada timestep durante un episodio, con un factor de descuento gamma) y el baseline o función valor (que estima el valor de las recompensas descontadas a partir del estado actual). El valor de la ventaja es el más importante, en vista de que, si este es positivo significa que la acción tomada es la correcta aumentando la probabilidad de tomar esa misma acción cuando vuelva a pasar por este estado; pero, si es negativo, reduce la probabilidad de esta acción en la política. El método PG canónico tiene esta forma:

Algoritmo canónico PG
Se inicializa los parámetros de la política θ_0 y la baseline b
Por cada episodio
Se recogen un conjunto de trayectorias parciales en la política $\pi_k = \pi(\theta_k)$
Se computa el return $R_t = \sum_{t'=t}^{T-1} \gamma^{t'-t} * r_{t'}$
Se estima la ventaja $\hat{\mathrm{A}}_t = R_t - b(s_t)$
Reajustar el baseline minimizando $\|b(s_t) - R_t\|^2$
Se actualiza la política usando la función del estimador:

$$L_t^{PG}(\theta) = \hat{\mathrm{E}}_t[\nabla_\theta(\log \pi_\theta(a_t|s_t)) * \hat{\mathrm{A}}_t]$$

Cuadro 3: Funcionamiento Policy Gradient

Los métodos PG no utilizan un buffer de repeticiones para guardar experiencias pasadas, si no que aprende directamente de lo que el agente se encuentra en el entorno. Cuando el lote de experiencia es usado para realizar una actualización del gradiente, ese lote de experiencia es descartado y la política sigue avanzando.

Dentro de esta familia de algoritmos está el algoritmo Proximal Policy Optimization, o PPO, pero para poder definir este algoritmo hay que definir el algoritmo en el que está basado, Trust Region Policy Optimization, o TRPO. **TRPO** actualiza las políticas dando el mayor paso posible para mejorar el rendimiento, a la vez que satisface una restricción especial en cuanto a la cercanía que se permite a las políticas nuevas y antiguas. La restricción se expresa en términos de KL, una medida de entre distribuciones de probabilidad.

El PG en su versión primigenia produce políticas nuevas muy similares a las viejas. Pero incluso, diferencias aparentemente pequeñas en los parámetros pueden tener diferencias muy grandes en el rendimiento, por lo que un solo paso en falso puede colapsar el rendimiento de la política. Esto hace que sea peligroso utilizar grandes stepsizes con PG canónico, perjudicando así su eficiencia. TRPO evita este tipo de colapsos, y tiende a mejorar rápida y monótonamente el rendimiento. Para ello la función a maximizar se convierte en esta:

$$L_t^{CPI}(\theta) = \hat{\mathrm{E}}_t[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} * \hat{\mathrm{A}}_t - \beta * \mathrm{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]$$

Ecuación 3: Función objetivo TRPO

PPO tiene algunos de los beneficios de TRPO, pero es mucho más sencillos de implementar, más general y tiene una mejor complejidad de muestra (es el número de muestras de entrenamiento que necesita para aprender con éxito una función objetivo). Hay dos variantes primarias de PPO: PPO-Penalty y PPO-Clip.

- PPO-Penalty resuelve aproximadamente una actualización de la restricción KL como la TRPO, pero penaliza la divergencia KL en la función objetivo.
- PPO-Clip no tiene un término de divergencia KL en el objetivo y no tiene ninguna limitación. En cambio, se basa en recortes, o clips, especializados en la función objetivo.

En este caso, se habla sólo de **PPO-Clip**. Para definir la función objetivo de PPO se necesita definir un término denominado el ratio de probabilidad, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. El ratio de probabilidad es mayor que uno si la acción es más probable en la política actual que en la política antigua, y menor que uno si es menos probable. La función central de PPO, es el mínimo entre la función objetivo de normal PG (ya que vuelca a la política a tomar acciones que ofrecen valores altos de $\hat{A}_t$) y la función objetivo de normal PG acotada entre 1-ε y 1+ε.

$$L_t^{CLIP}(\theta) = \hat{E}_t[\min(r_t(\theta) * \hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) * \hat{A}_t)]$$

Ecuación 4: Función acotada objetivo PPO

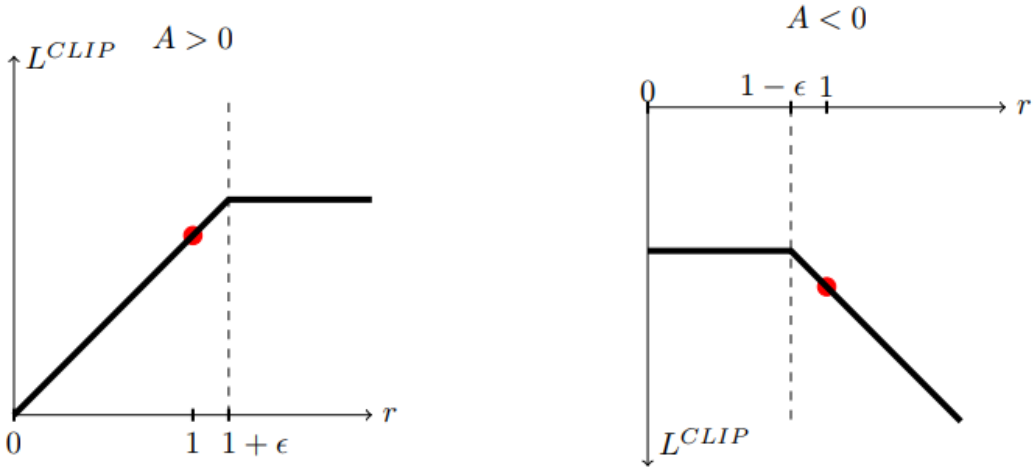


Ilustración 4: PPO-Clip, extraída de [16]

En la Ilustración 4 se pude ver qué forma tiene la función acotada objetivo de PPO para valores positivos y negativos de $\hat{A}_t$. Esta acotación consigue que no ocurran cambios bruscos en la política, mediante un método simple en la función objetivo. Es decir, el objetivo es el mismo que TRPO, pero con una implementación más sencilla. Como se puede ver, para valores de $r_t(\theta)$ altos, si la ventaja es positiva se acota a 1+ ϵ, en cambio si la ventaja es negativa y los valores de $r_t(\theta)$ son más bajos, también se acota en 1- ϵ. Cuando la ventaja es negativa y el ratio de probabilidad es alto significa que la acción en la política actual es más probable que la política antigua, por lo que se revierte esto (aquí es donde aparece el término min de la función, ya que si $r_t(\theta) * \hat{A}_t$es menor que 1- ϵ, se necesita revertir el cambio de política). Con todo ya definido, se puede ver cómo funciona el algoritmo:

Algoritmo PPO

Se inicializa los parámetros de la política θ_0

Por cada episodio

Se recogen un conjunto de trayectorias parciales en la política $\pi_k = \pi(\theta_k)$

Se estima la ventaja $\hat{A}_t^{\pi_k}$

Se computa la actualización de política:

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

tomando K pasos de minibatch SGD, donde

$$L_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau\sim\pi_k}[\sum_{t=0}^{T}[\min(r_t(\theta) * \hat{A}_t^{\pi_k}, \text{clip}\,(r_t(\theta), 1-\epsilon, 1+\epsilon) * \hat{A}_t^{\pi_k})]\,]$$

Cuadro 4: Funcionamiento Proximal Policy Optimization

5.1.3 Actor-Critic

Finalmente, existe otra familia de algoritmos que hacen uso de los algoritmos anteriormente mencionados, los algoritmos actor-critic. Los métodos de actor-critic constan de dos partes:

- El **critic**, que estima la función de valor. Esto podría ser el valor de acción (el valor Q) o el valor de estado (el valor V).
- El **actor**, el cual actualiza la distribución de políticas en la dirección sugerida por el critic (por ejemplo, con policy gradient).

Es decir, se tiene al actor, el cual recoge el estado y determina la mejor acción para el agente, y al critic, el cual recibe el estado y la acción tomada y devuelve un valor que referencia cuan buena es la acción para ese estado. Tanto la función del critic como la del actor están parametrizadas con redes neuronales. De una forma simple un algoritmo actor-critic funciona de esta forma:

Algoritmo actor-critic simple

Se inicializa los parámetros de la política θ_0 y el estado s; se muestrea $a \sim \pi_\theta(s_t, a_t)$

Por cada episodio

Se muestrea la recompensa $r_t \sim R(s_t, a_t)$ y el siguiente estado $s_{t+1} \sim P(s_{t+1} | s_t, a_t)$

Se muestrea la siguiente acción $a_{t+1} \sim \pi_\theta(a_{t+1}|s_{t+1})$

Se actualiza los parámetros de la política $\theta \leftarrow \theta + \alpha_\theta Q_\omega(s_t, a_t)\nabla_\theta \ln \pi_\theta(a_t|s_t)$

Se computa la corrección en el momento t:

$$\delta_t = r_t + \gamma Q_\omega(a_{t+1}|s_{t+1}) - Q_\omega(s_t, a_t)$$

Y se usa para actualizar los parámetros de la función valor-acción:

$$\omega \leftarrow \omega + \alpha_\omega \delta_t Q_\omega(s_t, a_t)\nabla_\omega$$

Se actualiza la acción $a \leftarrow a_{t+1}$ y el estado $s_t \leftarrow s_{t+1}$

Cuadro 5: Funcionamiento Actor-Critic

Como se puede ver se tiene dos ratios de aprendizaje α_θ y α_ω predefinidos para los parámetros de actualización de las funciones (política y valor respectivamente).

Dentro de estos tipos de algoritmos están el Asynchronous Advantage Actor-Critic, o A3C, que se explica más adelante, y el Soft Actor-Critic, o **SAC**. SAC es un algoritmo off-policy el cual optimiza una política estocástica formando un puente entre la optimización de la política estocástica y los enfoques de estilo DDPG. DDPG, abreviatura de Deep Deterministic Policy Gradient, es un algoritmo model-free, autor-critic, que combina el Deterministic policy gradient (DPG) con el Deep Q-Learning (DQN).

La característica central de la SAC es la regularización de la entropía, la cual evita que la política converja prematuramente en un mal óptimo local. La política se ocupa de maximizar el equilibrio entre el rendimiento esperado y la entropía, una medida de aleatoriedad en la política. La entropía está relacionada con la compensación entre exploración y explotación: el aumento de la entropía da lugar a una mayor exploración. En el aprendizaje regulado por entropía, el agente obtiene una recompensa extra proporcional a la entropía de la política. La entropía es el grado incertidumbre remanente sobre un conjunto de información. En otras palabras, es la cantidad que refleja lo aleatoria que es una variable. Por lo que, por ejemplo, si se tiene una moneda trucada la cual siempre sale cara, la entropía es baja; si en cambio la moneda es normal, al ser equiprobables las salidas posibles, la entropía es máxima (es decir, uno).

SAC utiliza al mismo tiempo una política π_θy dos funciones Q_{ϕ_1}, Q_{ϕ_2}. Hay dos variantes de SAC que son actualmente estándar: una que utiliza un coeficiente fijo de regularización de entropía α y otra que variando α en el curso del entrenamiento hace cumplir una restricción de entropía. Sin meterse en las ecuaciones complejas dentro del cálculo de las funciones, SAC sigue estas pautas:

Algoritmo SAC
Se inicializa los parámetros de la política θ y los parámetros de la función Q ϕ_1, ϕ_2 ; se vacía el buffer de repeticiones $\mathcal{D}$
Por cada episodio
- Se observa el estado s y se selecciona y ejecuta la acción $a_t \sim \pi_\theta(\cdot|s_t)$
- Se observa el siguiente estado s_{t+1}, la recompensa r_ty si s_{t+1} es terminal
- Se guarda (s_t, a_t, r_t, s_{t+1}) en el buffer $\mathcal{D}$
- Si el estado s_{t+1}es terminal se resetea el entorno
- Si es momento de actualizar entonces
- Aleatoriamente se muestrea $B = (s_t, a_t, r_t, s_{t+1})$, lote de transiciones de $\mathcal{D}$
- Se computa los objetivos para la función Q
- Se actualizan las funciones Q por un peso de gradient descent
- Se actualiza la política por un paso de gradient ascent
- Se actualiza las redes:

$$\phi_{targ,i} \leftarrow \rho\phi_{targ,i} + (1-\rho)\phi_i \qquad \text{para } i = 1{,}2$$

Cuadro 6: Funcionamiento Soft Actor-Critic

5.2 Entornos de simulación

En los últimos años se han desarrollado múltiples plataformas de simulación con el fin de proporcionar desafíos y puntos de referencia para los algoritmos de aprendizaje de refuerzo. Muchas de estas plataformas se basan en juegos o motores de juego existentes y llevan consigo ventajas y desventajas específicas. Aunque la lista de todas las plataformas disponibles actualmente no es exhaustiva, a continuación, se examinan algunos de los simuladores teniendo en cuento su popularidad.

El aprendizaje de refuerzo requiere un volumen muy alto de episodios, o interacciones con un entorno, para aprender una buena política. Por lo tanto, se necesitan simuladores para lograr resultados de manera rentable y oportuna. Los simuladores permiten que estos episodios sucedan en un mundo digital, entrenando a un agente para que alcance su máximo potencial mientras se ahorra tiempo y dinero.

5.2.1 AirSim

AirSim es un simulador de código-abierto para drones y coches principalmente, construido sobre Unreal Engine. También es posible implementar AirSim en Unity, aunque con versiones experimentales. El principal objetivo de AirSim como plataforma para el desarrollo de Inteligencia Artificial es experimentar con algoritmos de deep reinforcement learning. AirSim pone a disposición del usuario sus APIs para la recolección de datos y control de los vehículos.

En cuanto al motor de juego, Unreal Engine, fue desarrollado por Epic Games inicialmente para el desarrollo de juegos de disparo en primera persona, pero hoy en día es usado para todo tipo de géneros y plataformas. Gracias al uso de Unreal Engine y el plugin de AirSim es posible desarrollar cualquier juego que haga uso de vehículos controlados por Inteligencia Artificial. Por ejemplo, se pueden desarrollar juegos, donde cada vez el agente es más inteligente para comprobar hasta qué nivel es capaz de llegar. También se puede utilizar para experimentar con simulaciones, como entrenar a un vehículo a moverse por la calle haciendo uso solo de su visión.

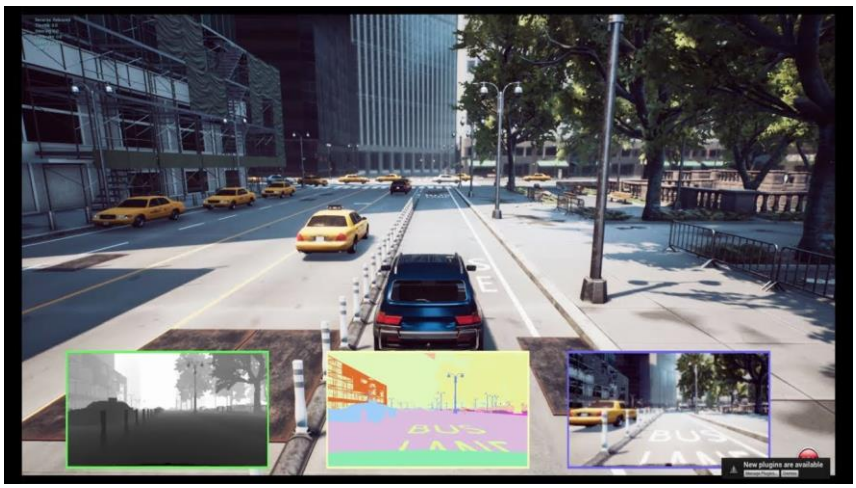


Ilustración 5: Entorno de simulación Unreal + AirSim, imagen extraída de [18]

5.2.2 Ml-Agents

Ml-Agents, o Unity Machine Learning Agents Toolkit, es un proyecto de código-abierto que permite a juegos o simulaciones usar entornos para entrenar agentes inteligentes. Los agentes se pueden entrenar a través de reinforcement learning, imitation learning u otros métodos de machine learning mediante el uso de una API de Python. Unity es una plataforma de desarrollo 3D en tiempo real que consiste en un motor de renderización, así como una interfaz gráfica de usuario llamada Unity Editor. Unity permite desarrollar todos estos algoritmos en entornos de videojuegos en 2D, 3D y videojuegos de realidad aumentada o realidad virtual. Ml-Agents simplifica el trabajo, debido a que se centra en la implementación del algoritmo de machine learning permitiendo así al desarrollador del proyecto centrarse más en desarrollar el entorno y el agente.

Para el entrenamiento usando algoritmos de Deep Reinforcement Learning se tiene dos algoritmos, Proximal Policy Optimization, o PPO y Soft Actor-Critic, o SAC. En caso de usar imitation learning se puede entrenar tanto como por clonación de comportamiento, o Behavioral Cloning, o Generative Adversarial Imitation Learning, o GAIL. También se pueden usar mecanismos de self-play para entrenar agentes en escenarios adversarios (es decir unos equipos contra otros).

Ilustración 6: Logo Unity y logo TensorFlow, imagen extraída de [19]

La implementación de los algoritmos está construida encima de la librería de código abierto TensorFlow, la cual permite el despliegue de computación a través de CPU y GPS. El entrenamiento realizado por TensorFlow es ejecutado en un proceso de Python separado (comunicado con Unity mediante sockets), el cual genera un modelo en un fichero una vez finalizado el entrenamiento para poder entregar este conocimiento adquirido al agente. En la imagen siguiente se pueden ver todas las formas posibles de interacción del agente con el entorno.

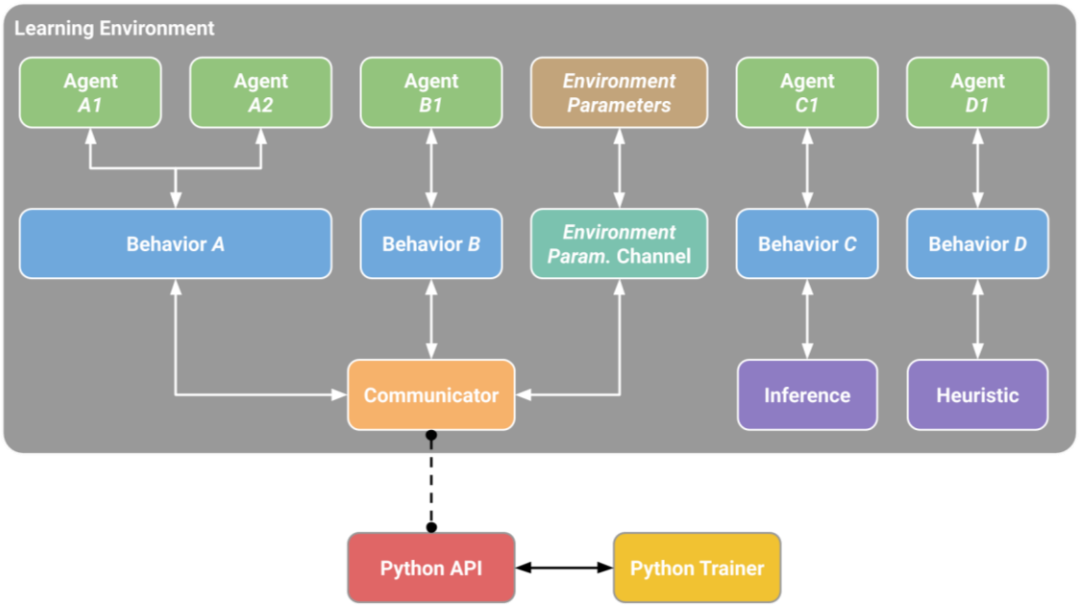


Ilustración 7: Entorno de aprendizaje, extraída de [20]

Primero, se tiene en caso en el que dos agentes tengan el mismo comportamiento A, agentes A1 y A2; después un agente B1 que entrena con otro comportamiento B. Los parámetros del entorno también estarán conectados mediante un canal a este proceso de Python. Por último, los dos agentes C1 y D1, los cuales no estarán conectados a Python debido el agente C1 usa un modelo de red neuronal pre-entrenado y el agente D1 es controlado manualmente.

5.2.3 DeepMind Lab

Desarrollado desde el motor id Tech 3, para el juego Quake III, DeepMind Lab fue creada en 2016 como la versión externa de la plataforma de investigación usada por DeepMind. Con el uso del motor de videojuegos 3D, se pueden crear y estudiar ejercicios de navegación compleja similares a los estudiados por robótica o psicología animal. La posibilidad de crear un set especifico de tareas hace que DeepMind Lab sea una plataforma de dominio especifico, ya que permite la creación del grupo de tareas únicamente para la navegación en primera persona.

El algoritmo empleado por DeepMind Lab se denomina IMPALA, y tiene como objetivo maximizar el data throughput haciendo uso de una eficiente arquitectura distribuida con TensorFlow. Este algoritmo está inspirado en A3C (Asynchronous Advantage Actor-Critic), como ya se ha definido antes, un tipo de algoritmo que se separa en dos modelos, uno que tiene como salida las acciones que toma el agente y otro que calcula los valores de Q. A3C usa múltiples actores distribuidos para que aprendan los parámetros de la política del agente. En este algoritmo, cada actor usa una copia de los parámetros de la política actual para actuar en el entorno. Periódicamente, los actores pausan la exploración para enviar los gradientes calculados a un servidor de parámetros central. En cambio, en IMPALA, los actores no se usan para calcular el gradiente. Estos solo recolectan experiencias que pasan a un central learner que

computa gradientes, resultando en un modelo que diferencia actores de learners. En la siguiente imagen se puede observar esta diferencia:

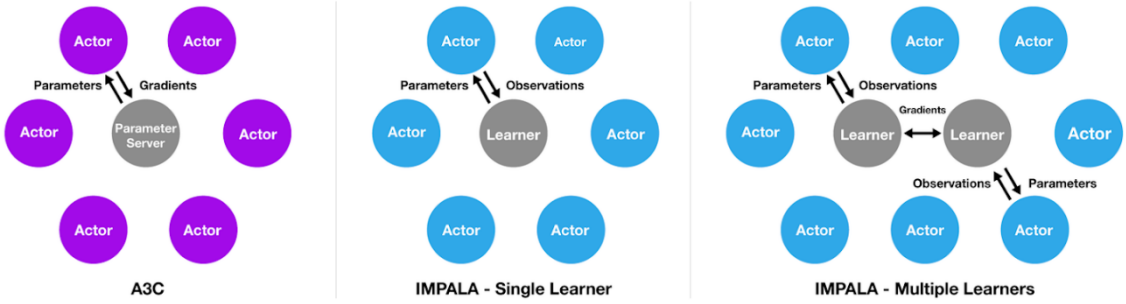


Ilustración 8: Importance Weighted Actor-Learner Architecture, extraída de [23]

5.2.4 OpenAI Gym

OpenAI Gym es un toolkit para desarrollar y comparar algoritmos de aprendizaje de refuerzo. No hace suposiciones sobre la estructura del agente, y es compatible con cualquier biblioteca de cálculo numérico, como TensorFlow o Theano. La biblioteca Gym es una colección de problemas para testear, en forma de entornos de RL, algoritmos creados por el usuario, puesto que estos entornos estandarizados tienen una interfaz compartida que permite escribir algoritmos generales. OpenAI Gym cuenta con un conjunto diverso de entornos que van de lo fácil a lo difícil e involucran muchos tipos diferentes de datos, desde agentes que completan tareas sencillas hasta juegos de Atari o robots en 3 dimensiones.

Ilustración 9: OpenAI Gym Logo, extraído de [24]

6 Análisis de alternativas

En el apartado anterior se han propuesto varios algoritmos de aprendizaje por refuerzo, junto con los posibles entornos de simulación a utilizar. Durante este apartado se realiza un estudio de que algoritmo junto a que entorno de simulación se utiliza para el desarrollo del proyecto.

El principal objetivo del algoritmo es conseguir llegar a la convergencia lo antes posibles, sin embargo, también se debe tener en cuenta cuan eficiente es y el riesgo de no llegar nunca a la convergencia. Para el entorno de desarrollo se busca que sea capaz de cumplir con el objetivo del proyecto permitiendo implementar el entorno y los agentes necesarios, y permitiendo interaccionar de manera sencilla con librerías de despliegue de modelos de aprendizaje por refuerzo.

6.1 Algoritmos de Reinforcement Learning

Por lo que se ha visto, hay tres formas populares de abordar el problema en aprendizaje por refuerzo, los métodos de política, los métodos de valores y los actor-critic. En los métodos de política se intenta optimizar directamente la política, en cambio, en los métodos de valor, se intenta evaluar el rendimiento futuro esperado (aprender una función de valor), y deducir la política a partir de ahí. Ahora se pasa a evaluar sus ventajas y desventajas y escoger una para la implementación de este proyecto.

6.1.1 Ventajas y desventajas

6.1.1.1 Q-Learning vs Policy Gradient

Para las aplicaciones en el mundo real, los métodos PG constan de numerosas ventajas. Los métodos PG tienen mejores propiedades de convergencia, considerando que, con el gradiente de la política, se realiza una actualización suave de la política en cada paso. De esta forma, siempre se converge, ya sea a un máximo local, lo cual sería el peor caso de convergencia, o a un máximo global, el mejor caso. Normalmente se utilizan menos parámetros en el proceso de aprendizaje del agente, comparado con los métodos basados en la aproximación por valores (como es Q-Learning). Aunque tanto los algoritmos PG como los algoritmos Q-Learning son model-free, es decir, maximiza la recompensa esperada sólo a partir de la experiencia real, por ejemplo, actualizando su conocimiento basado en prueba y error; Policy Gradient también puede ser basado en un modelo, es decir, utiliza un modelo predictivo del entorno. Los métodos PG se han vuelto particularmente interesantes para las aplicaciones de la robótica, y esto se debe a que éstas soportan acciones tanto continuas como discretas. Q-Learning, en cambio, solo soporta acciones discretas.

Otra ventaja es que PG puede aprender políticas estocásticas, mientras que las funciones de valor no pueden. Una política estocástica permite a nuestro agente explorar el entorno sin tomar siempre la misma acción, pues produce una distribución de probabilidad sobre las acciones.

Por supuesto, los gradientes de política no son la salvación de todos los problemas, también tienen problemas significativos. Los algoritmos PG son on-policy y necesitan olvidar los datos muy rápidamente para evitar la introducción de un sesgo en el estimador de gradientes. En las representaciones tabulares, se garantiza que los métodos de la función de valor convergen en un máximo global, mientras que los métodos PG convergen en un máximo local y, en problemas discretos, puede haber muchos de ellos.

En cambio, los métodos Q-learning tienen la ventaja de ser sustancialmente más sample efficient cuando funcionan, porque pueden reutilizar los datos más eficazmente que las técnicas de optimización de políticas, como es PG. Un algoritmo es sample efficient, o eficiente en cuanto al muestreo del entorno a resolver (entendiendo muestreo como una trayectoria del entorno por parte del agente), si puede hacer buen uso de cada una de las experiencias que genera y mejora rápidamente su política. Un algoritmo tiene una eficiencia en cuanto al muestreo pobre, si no aprende nada útil de muchas muestras de experiencia y no mejora rápidamente. Aunque sea más sample efficient, Q-Learning al ser un método basado en valor, puede tener una gran oscilación durante el entrenamiento. Esto se debe a que la elección de la acción puede cambiar drásticamente por un cambio arbitrariamente pequeño en los valores de acción estimados. Haciendo de Q-Learning un algoritmo poco estable en comparación.

6.1.1.2 Actor-Critic

Los algoritmos que emplean PG tienen características muy similares, aunque luego obtengan resultados distantes. Ocurre lo mismo con Q-Learning. Esto con Actor-Critic es muy diferentes. Cada algoritmo dentro de este tipo de algoritmos tiene sus propias características, debido a que dependen de que método se implementen en el actor y el critic. Por lo que para este caso se estudia el algoritmo, ya definido, Soft Actor-Critic o SAC.

SAC soporta tanto espacios de acción continuos como discretos. Es un algoritmo eficiente en cuanto al muestreo, y a su vez estable. Requiere de menos pasos para obtener la misma recompensa que algoritmos como PPO, aunque, por otro lado, es un algoritmo que tarda mucho en realizar cada paso.

6.1.2 Selección de alternativas

Dentro de cada método de aprendizaje se escoge un algoritmo en concreto: para Q-Learning, Double Deep Q-Learning; para Policy Gradient, Proximal Policy Optimization; y para Actor-Critic, Soft Actor-Critic. Los campos que se han tenido en cuenta para decidir sobre que algoritmo trabajar son:

- **Estabilidad,** hace referencia a como el algoritmo de aprendizaje es perturbado por pequeños cambios en sus entradas. Un algoritmo de aprendizaje estable es aquel cuya predicción no cambia mucho cuando los datos de entrenamiento se modifican ligeramente.

- **Tiempo de aprendizaje,** hace referencia al tiempo que tarda el agente en realizar el entrenamiento del agente para un mismo número de steps.
- **Versatilidad,** hace referencia a la capacidad del algoritmo a emplearse en todo tipo de escenarios y entornos, como pueden ser agentes con acciones discretas y continuas.
- **Eficiente,** hace referencia a la cantidad de experiencia que un agente necesita generar en un entorno (por ejemplo, el número de acciones que realiza y el número de estados y recompensas que observa) durante el entrenamiento para alcanzar un determinado nivel de aprendizaje o rendimiento.

Por lo que para decidir qué método emplear se utiliza la siguiente tabla:

	Q-Learning (DDQN)	Policy Gradient (PPO)	Actor-Critic (SAC)
Estabilidad	Aceptable	Muy bueno	Bueno
Tiempo de aprendizaje	Bueno	Excelente	Deficiente
Versatilidad	Regular	Excelente	Bueno
Eficiente	Excelente	Aceptable	Muy bueno
Total	Bueno	Muy bueno	Aceptable

Tabla 1: Análisis de alternativas de algoritmos

El algoritmo con mejor resultados es PPO, el cual proporciona una estabilidad en el aprendizaje sacrificando sample efficiency, y consiguiendo un tiempo de entrenamiento alto. A parte, PPO es un algoritmo que se puede implementar en cualquier tipo de entorno gracias a su versatilidad.

6.2 Entornos de simulación

Toda la evolución del ser humano para aprender habilidades complejas, ha sido gracias a la capacidad de mantener relaciones dentro de los grupos sociales. Gracias a la interacción entre personas formando grupos, se ha podido desarrollar una herramienta de comunicación en entornos sociales, el lenguaje. El desarrollo del comportamiento social entre grupo de agentes es de particular interés para muchos investigadores en el campo de la IA. Se estudia el comportamiento complejo que sólo puede llevarse a cabo a nivel de la población, como la coordinación necesaria para construir. Un entorno de simulación, debe permitir el estudio de la comunicación y el comportamiento social. Por lo tanto, debe proporcionar un marco robusto de múltiples agentes que permita la interacción entre ellos, tanto para la misma población como para la interacción entre grupos de agentes de diferentes poblaciones.

6.2.1 Ventajas y desventajas

6.2.1.1 AirSim

AirSim es una de las herramientas más potentes en la actualidad en cuanto a la simulación de vehículos. AirSim gracias al uso del motor gráfico de Unreal Engine y de Mo-cap (motion capture) obtiene un gran nivel de realismo. También es posible cambiar de motor gráfico y emplear Unity, siendo así un simulador multiplataforma. Tiene una gran cantidad de trabajos y documentos

sobre diferentes tipos de proyectos, garantizando así obtener más información de ayuda a la hora de llevar a cabo el proyecto. Tiene soporte a entornos multiagente y a los algoritmos que se deseen implementar

Sin embargo, es un sistema preparado únicamente para la implementación de vehículos, como son drones o automóviles. Esto implica que, a la hora de implementar el objetivo del proyecto, es complicado reflejar las dinámicas comportamentales de los agentes en el entorno.

6.2.1.2 ML-Agents

La naturaleza del lenguaje de scripts en Unity y su sistema de componentes hace que el planteamiento de escenarios de múltiples agentes sea simple y directo. De hecho, dado que la plataforma fue diseñada para apoyar el desarrollo de videojuegos multijugador, ya se proporcionan de fábrica varias abstracciones útiles. ML-Agents no tiene soporte a cualquier tipo de algoritmo, pero, como ya se ha mencionado en el estado del arte, si soporta el uso de PPO para el aprendizaje de los agentes, ya que este algoritmo esta ya dentro de las librerías del programa. Esto hace que lo único que se necesite para implementar este algoritmo sea modificar un parámetro de los ficheros de configuración con el nombre ppo.

6.2.1.3 DeepMind Lab

Las limitaciones de esta plataforma están en gran parte ligadas a la naturaleza anticuada del motor de representación, que fue construido con una tecnología de décadas de antigüedad. De tal forma que, la brecha de calidad entre el mundo real y la simulación a través del DeepMind Lab es relativamente grande. Y lo más importante, el motor sólo fue diseñado para permitir juegos de disparos en primera persona, por lo que los entornos construidos por DeepMind Lab se limitan a agentes con perspectiva en primera persona. Esto también implica que no permite el aprendizaje de múltiples agentes a la vez.

Aparte de esto, esta plataforma solo permite el uso de su algoritmo propio IMPALA, el cual no cumple con los objetivos del proyecto como algoritmo. Sin embargo, DeepMind Lab es un entorno sencillo de utilizar y cómodo con el que trabajar, sobre todo si se quiere comenzar con el aprendizaje con refuerzo.

6.2.1.4 OpenAI Gym

OpenAI Gym cuenta con un gran número de entornos creados para poder realizar simulaciones sobre ellos con diferentes algoritmos. Pero este también permite al usuario crear sus propios entornos en los que implementar estos algoritmos. Por lo que, de esta forma, aunque OpenAI Gym ofrece ya algún entorno multiagente, es posible crear uno personalizado. Sin embargo, la creación de estos entornos no es tan sencillos, sobre todo si se comparan con la creación de entornos con Unity o Unreal.

OpenAI Gym soporta todo tipo de algoritmos, siendo este su principal característica, aunque esto también implica que se debe crear el algoritmo para implementar, en este caso, PPO.

6.2.2 Selección de alternativas

Los campos que se han tenido en cuenta para decidir qué entorno de simulación emplear son:

- **Multi-Agente**, es decir, si se tiene soporte a sistemas multiagente
- **Sencillez**, es decir, si es sencillo implementar el código del agente, el entorno, el algoritmo y sus hiperparámetros, etc.
- **Algoritmos**, es decir, si tiene soporte a varios algoritmos, y, sobre todo, si tiene soporte para el algoritmo PPO.

Por lo que para decidir sobre que plataforma trabajar se utiliza la siguiente tabla:

	AirSim	ML-Agents	DeepMind Lab	OpenAI Gym
Multi-Agente	Excelente	Excelente	Nulo	Excelente
Sencillez	Regular	Excelente	Excelente	Aceptable
Algoritmos	Excelente	Muy bueno	Deficiente	Excelente
Total	Muy bueno	Excelente	Deficiente	Muy bueno

Tabla 2: Análisis de alternativas de simuladores

El entorno con mejor puntuación es ML-Agents, donde implementar sistemas multiagente es algo sencillo, y aunque no tenga un soporte a cualquier tipo de algoritmo, proporciona soporte de algoritmos como PPO y SAC, donde PPO, como se ha visto antes, es el algoritmo que se quiere utilizar.

7 Análisis de riesgos

Los riesgos existentes en un proyecto deben ser analizados con el fin de poder solucionar los problemas que estos puedan causar. Para ello, se ha realizado un estudio de los posibles riesgos que pueden retrasar o incluso invalidar el proyecto. Existen infinidad de causas y eventos no deseados que tener en cuenta, pero para este análisis sólo se tienen en cuenta los más evidentes o los que más daños pueden causar.

Primero se identifican los riesgos y luego se muestran de forma gráfica a través de una matriz de riesgos, donde se puede ver el nivel de peligrosidad del riesgo sobre la probabilidad de ocurrencia del riesgo con respecto a su impacto. Para cada riesgo se crea un plan de contingencia, para poder evitar o resolver estos casos.

7.1 Errores en el desarrollo software

El error en el código del agente o errores por no entender bien el algoritmo empleado, son situaciones que ocurren bastante a menudo durante el desarrollo de este tipo de proyectos. Si se codifica mal el agente, se puede estar pensando que los resultados conseguidos de las simulaciones son los indicados cuando no debería ser así. Esto se debe a que es muy difícil predecir el comportamiento de un agente de este nivel de complejidad. También el error se puede deber a haber introducido una serie de hiperparámetros que no coinciden con el objetivo del agente (por ejemplo, hacer que actualice la política muy pronto). Es por eso que, de una escala de probabilidad de ocurrencia de rara a muy probable, este suceso es muy probable; y el impacto que tiene sobre el proyecto, en una escala de consecuencias despreciables hasta consecuencias mayores, es de impacto moderado.

Aun así, es muy sencillo elaborar un plan de contingencia para este riesgo. Lo primero es realizar un buen estudio del algoritmo a emplear y del simulador a emplear, para así entender que se busca y como se debe implementar. Una vez entendido lo anterior, se debe comenzar a realizar varios entrenamientos cuanto antes, de tal forma que se obtenga un amplio margen para corregir posibles errores de código. Y finalmente, la mejor solución es realizar pruebas unitarias en el proyecto de vez en cuando para asegurarse el correcto funcionamiento del código.

7.2 Pérdida de datos y avería de equipos

Un riesgo a tener en cuenta es que el proyecto donde se trabaja con el agente, el código del agente, o el fichero de configuración del algoritmo, se pierda o borre. Esto conlleva a realizar de cero el proyecto, y su causa puede ser por avería de equipos o infortunios. Siguiendo con la misma valoración de antes, la probabilidad de ocurrencia de algo así es rara, mientras que el impacto que tiene es moderado.

Para que no se pierda toda la información se pueden hacer copias cada poco tiempo en una nube local del proyecto, para que en caso de pérdida se pueda recuperar.

7.3 Superación fecha límite o falta de tiempo

En proyectos de este tipo, como es un trabajo que emplea algoritmos de aprendizaje por refuerzo profundo, el tiempo de cada entrenamiento suele ser largo. Es por eso que, lo primero es saber cuánto se puede llegar a tardar en conseguir los resultados esperados y en estudiarlos y compararlos.

Si la fecha limite está cerca y el proyecto no ha conseguido ningún resultado evidente, se suele impulsar a intentar realizar todo lo más rápido posible, y que, por consiguiente, no salga como se desee. La probabilidad de ocurrencia de un caso así, al tratarse de aprendizaje por refuerzo, es probable, pero el impacto es menor, pues en la mayoría de casos se puede acordar nuevas fechas.

Para el plan de contingencia lo mejor es realizar una planificación más ordenada y poner como fecha final de entrega, una fecha anterior a la fecha límite, para dejar así los últimos retoques para el final.

7.4 Presupuesto insuficiente

A la hora de empezar con un proyecto no se tiene bien claro cuanto será el presupuesto final del proyecto. Puede ocurrir que los costes que ha tenido el proyecto hasta cierto momento, han cubierto ya el presupuesto y todavía no se ha acabado el proyecto, siendo imposible continuar con este. La probabilidad de ocurrencia de algo así es rara, ya que normalmente se suele plantear correctamente el presupuesto del proyecto, y las consecuencias de algo así serían mayores, no pudiendo así continuar con el proyecto.

Con aumentar de primeras el presupuesto, aumentado el dinero dedicado a imprevistos valdría para que no ocurran estos improvistos.

7.5 Incapacidad del agente

En el Reinforcement Learning tiene un riesgo claro, y sobre todo para las personas que comienzan con estas tecnologías, es el riesgo a crear un escenario tan complejo que el agente no es capaz de interpretar la información otorgada. Esto puede provocar que el agente necesite, para llegar a la política óptima, un número de pasos excesivo, o que incluso que nunca llegue a la estabilidad.

Las razones principales que causan un nivel de complejidad alto para el agente pueden ser: otorgar la recompensa de formas muy rebuscadas o no otorgar observaciones de acuerdo a las recompensas otorgadas, asignar valores erróneos a los hiperparámetros del algoritmo, o utilizar un espacio de acciones y observaciones muy grande. La probabilidad de ocurrencia de estos casos, es de poco probable, y el impacto es de moderado.

Para ahorrarse problemas de este estilo lo mejor es realizar una buena investigación sobre el aprendizaje por refuerzo y algoritmo antes de comenzar a desarrollar el escenario.

7.6 Matriz de riesgos

La matriz de riesgos, o matriz probabilidad-impacto permite establecer prioridades en cuanto a los posibles riesgos del proyecto, en función tanto de la probabilidad de que ocurran como de las repercusiones que podrían tener sobre el proyecto en caso de que ocurrieran.

A continuación, se presenta la matriz de riesgos, la cual identifica el nivel del riesgo sobre el proyecto desde un riesgo muy bajo hasta riesgos muy altos.

Probabilidad / Impacto	Raro	Poco Probable	Probable	Muy Probable
Despreciable	Muy bajo	Bajo	Bajo	Medio
Menor	Bajo	Bajo	Medio Superación fecha límite	Alto
Moderado	Medio Pérdida de datos	Medio Incapacidad del agente	Medio	Alto Error desarrollo software
Mayor	Medio Presupuesto insuficiente	Alto	Alto	Muy Alto

Tabla 3: Matriz probabilidad-Impacto

8 Descripción de la solución propuesta

En este apartado se explica cómo se ha desarrollado la solución para cumplir con el objetivo del proyecto. Pero antes de esto, hay que definir las especificaciones empleadas para poder desarrollar este proyecto:

- Unity 2018.4.17f1
- ML-Agents 1.0
- TensorFlow 2.2.0
- CUDA 10.1 y cuDNN 7.4
- Visual Studio 15.9.24
- Python 3.8.3

Unity es empleado para la creación del escenario, para que el agente lo utilice este escenario como el entorno donde realizar el entrenamiento. Para añadir la inteligencia artífica al proyecto de Unity se añade ML-Agents junto a TensorFlow para poder realizar las pruebas de aprendizaje. A los agentes creados en Unity, se le añade el aprendizaje profundo y se programan mediante Visual Studio y C#. Finalmente, se tiene CUDA y cuDNN para que la tarjeta gráfica NVIDIA se pueda emplear para el aprendizaje de los agentes.

8.1 Construcción del escenario

Se recuerda el objetivo de este proyecto: se busca demostrar dinámicas comportamentales de sistemas multi agente basados en algoritmos de aprendizaje por refuerzo. Una vez realizado el análisis de alternativas algorítmicas, se comienza con el desarrollo del entorno mediante la herramienta Ml-Agents de Unity. Lo primero es analizar cómo implementar las dinámicas de aprendizaje comportamental sobre los agentes. Para desarrollar estas dinámicas es necesario establecer qué recompensa, o reward, recibe el agente del entorno ante una acción. Así, se distinguen: un comportamiento local, que busca lo mejor para el propio agente viendo lo que tiene a su alrededor (supervivencia); un comportamiento poblacional, que se centra en la prosperidad de la población a la que pertenece el agente (por ejemplo, no fomentando la dispersión física de los agentes que pertenecen a una determinada población); y por último, un comportamiento global, que busca un equilibrio entre todas las poblaciones del escenario que evite, por ejemplo, que el escenario quede poblado por agentes de una única población (dominancia).

Para implementar el agente en el proyecto se estudia todas las posibilidades existentes. El agente puede ser estacionario y que su única acción sea, por ejemplo, cambiar de color o forma. También puede ser un agente en movimiento el cual busca juntarse o alegarse de los demás agentes dependiendo de qué población sean. El agente puede recibir una observación entera del estado del entorno, o únicamente recibir observaciones de los agentes que puede ver en un eje. A la hora de desarrollar los agentes hay que estudiar cómo se quiere que el agente interactúe con el entorno. Hay que tener en cuenta todo tipo de factores a la hora de diseñar

este agente, como pueden ser, el número de pasos máximo que realiza un agente en un entrenamiento, con cuenta frecuencia realiza acciones aleatorias para explorar el entorno, elección de modelos 2D o 3D, etc.

Finalmente, se decidió, para optimizar el consumo de recursos, modelos 2D de cubos estacionarios apilados en una matriz de doce filas y doce columnas, donde cada acción es el color que toma, y como observación recibe, o los cubos que tiene a su alrededor o toda la malla, o grid, de cubos. Existen tres colores, cada uno referenciando una población, y un cuarto color que marca la muerte. De tal forma que cada vez que se inicia el entorno solo un pequeño número de agentes vive, mientras los otros esperan muertos hasta que tengan a su alrededor algo de vida. Lo único que quedaría una vez elegido el modelo que va a tener el agente es, programarlo para que pase las observaciones a la red neuronal, recoja las acciones recibidas por la política, se inicialice y reinicie adecuadamente y, uno de los objetivos más difíciles al programar un agente, programarlo con un reparto adecuado de recompensas. Ml-Agents tiene funciones en su librería en la clase agente para cada una de los métodos que hay que implementar. Para las observaciones se hace uso de un método llamado, **CollectObservations(VectorSensor sensor)**, el cual es un void method (que no devuelve nada) a sobrescribir donde al VectorSensor sensor se le añaden observaciones con el método sensor.AddObservation(int observacion); para recoger las acciones se hace uso del método **OnActionReceived(float[] vectorAction)**, el cual es un método void que sobrescribir donde el array de decimales vectorAction contiene en la primera posición la acción a realizar (0→muere, 1→población verde, 2→ población amarilla…); para inicializar y resetear el agente se usa, **Initialize()** y **OnEpisodeBegin()** respectivamente, los cuales son métodos vacíos a sobrescribir, e inicializan el agente a cierto estado; y por último para las recompensas se utiliza, **AddReward(float r)**, el cual que no devuelve ningún valor, pero entrega al agente la señal de recompensa. En Unity el entorno queda tal que así:

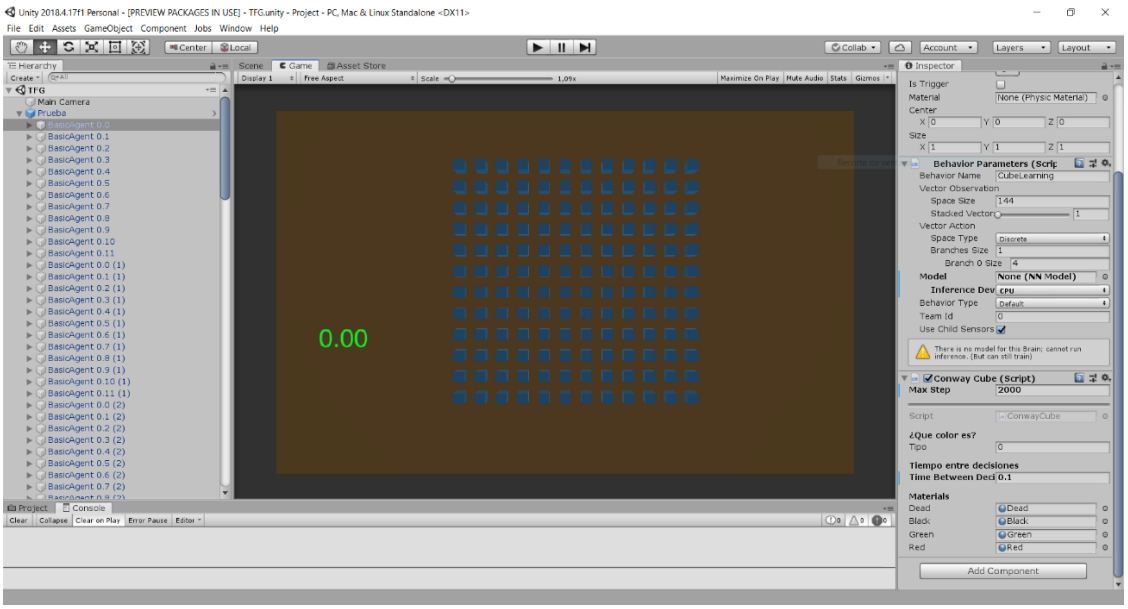


Ilustración 10: Entorno de simulación en Unity

Antes de explicar lo que se ve en la imagen, hay que definir algunos términos de Unity para entender mejor cómo funciona el entorno de simulación. Unity tiene un sistema de Prefabs, el cual te permite crear configurar y guardar un GameObject completo con todos sus componentes, propiedades e hijos GameObject. Un **GameObject** es el objeto fundamental en Unity que representa personajes, accesorios o paisajes, es decir es el objeto básico dentro del entorno de Unity. Son utilizados como contenderos a los cuales añadir Componentes que implementan la verdadera funcionalidad. Por ejemplo, para crear un cubo se debe crear un GameObject y añadirle los componentes necesarios para darle forma. Finalmente definir que todos ellos están dentro de una escena, acompañada de una o varias cámaras para ver hacia donde se apunte con ella.

De la Ilustración 10, de la parte superior izquierda de la imagen, se obtiene la Ilustración 11: Escena TFG en Unity, que es la escena TFG. La escena denominada TFG consta con una cámara que mira desde arriba a la matriz para que se vea en dos dimensiones. Luego está el prefab Prueba, referenciado con un cubo azul, que contiene los agentes BasicAgent. Estos agentes están dispuestos en forma de matriz 12x12 separados 0.5 de distancia uno del otro.

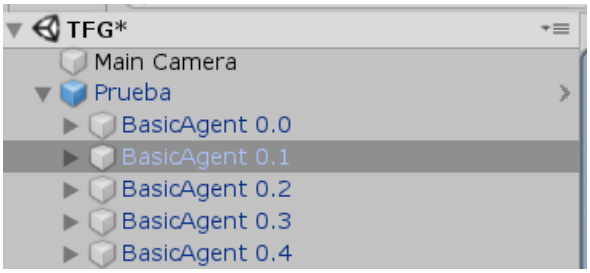


Ilustración 11: Escena TFG en Unity

De la Ilustración 10, también se obtiene la Ilustración 12, donde aparece el inspector del agente el cual muestra todos los componentes que tiene BasicAgent. Mencionar los componentes Behavior Parameters y Conway cube, los cuales hacen que el agente deje de ser un GameObject y se convierta en un agente de Ml-Agents.

En Behavior Parameters se ajustan algunos parámetros importantes del agente como son: el tamaño exacto del vector de observaciones, el número de acciones que puede tomar cada agente o la opción de otorgar un modelo ya entrenado. El componente Conway Cube es un script creado en C#, donde se encuentran todos los métodos del agente antes definidos, y en el cual se necesitan algunos valores que recibir de Unity como son el material o, a la hora de realizar entrenamientos, el número máximo de steps hasta que se reinicie.

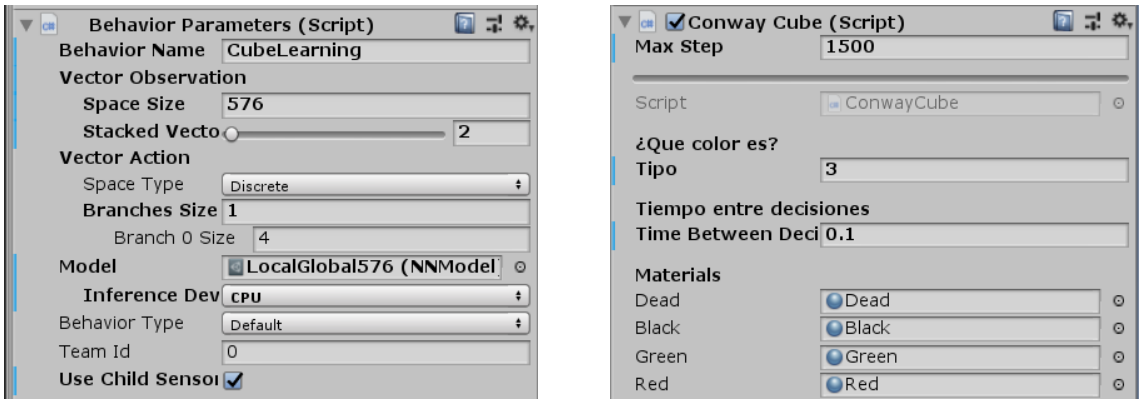


Ilustración 12: Componentes Behavior Parameters y Conway Cube en Unity

Una vez con todo esto montado, y los métodos del agente completos, el agente estaría listo para empezar a entrenar con Ml-Agents. Para el aprendizaje todos los agentes tienen un orden que seguir definido por la academia. La Academia es un singleton que organiza a los agentes y sus procesos de toma de decisiones. Se ha realizado un **diagrama de actividades UML** para modelar el comportamiento del agente, haciendo énfasis en el proceso que se lleva a cabo.

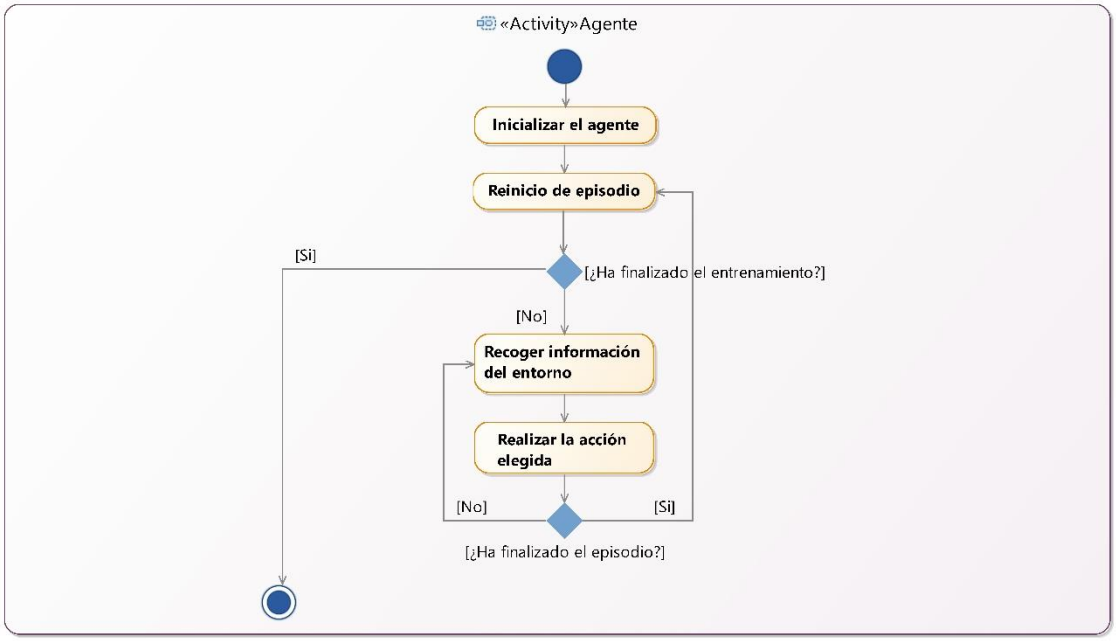


Ilustración 13: Diagrama de actividades del agente en UMLDesigner

La primera actividad que realiza el agente, es la inicialización de sí mismo, donde se llama al método `Initialize()`, el cual inicia el objeto y componentes en Unity. Una vez inicializado el agente, se pasa al inicio de episodios para el aprendizaje de este. Antes de comenzar el episodio, se llama al método `OnEpisodeBegin()`, el cual escoge un color de forma aleatoria (con un 1/24 de que no esté muerto, de tal forma que de los 144 cubos al iniciar haya una media de 6 vivos por episodio). Antes de comenzar el episodio se mira si el entrenamiento ha llegado a su fin.

Esto puede ocurrir por interrupción del usuario o por llegar al máximo número de pasos en el entorno. Si el entrenamiento ha finalizado se crea un modelo con lo aprendido hasta ese momento y se cierra el proceso de aprendizaje del agente. Si el entrenamiento continúa, se da comienzo al bucle del episodio del agente. En este bucle lo primero que se hace es recoger la información de entorno mediante observaciones. Para ello existe el método `CollectObservations(VectorSensor sensor)`, que se encarga de mediante el objeto sensor, introducir las observaciones al agente llamando al método `sensor.AddObservation(x)`. En este caso, cada agente recibe como observaciones, o el estado de todos los agentes a su alrededor, o el estado de todos los bloques de la malla. Para pasar esta información al agente se emplea codificación One-hot para los tipos de estados del agente. De esta forma si lo que se pasa al agente son los vecinos, se le otorga al agente el estado de ocho agentes más el suyo, nueve; y al emplear codificación One-hot con cuatro posibles casos, se tiene un espacio de observaciones de $4 \times 9 = 36$. En el caso de dar toda la malla es $4 \times 144 = 576$. Además de esto, el agente deberá guardar la observación pasada, teniendo en cada step el doble de observaciones en cuenta, las observaciones anteriores más las actuales (para ello se debe ajustar en el componente Behavior Parameters el campo Stacked Vectors a 2). El agente después de recolectar toda la información del entorno, usa la política para decidir la acción que debe tomar. Por lo que la siguiente acción es recoger y realizar la acción que escoja la política. La acción se recoge en un array de float vectorAction por el agente en un método llamado `OnActionReceived(float[] vectorAction)`. Dentro de este método se realizan las acciones dependiendo del valor que devuelva la política, en este caso el valor se encuentra en la primera posición del array y este valor varía entre 0 y 3. Las acciones que se realizan son únicamente para cambiar el color del agente. Una vez realizada la acción se añade la recompensa asociada a la acción tomada y se comprueba si se ha llegado a un estado terminal. Para llegar a este estado el agente tiene que llamar al método `Done()` o llegar al máximo número de steps del agente. Si el estado no es terminal, el episodio no ha finalizado por lo que se sigue el bucle colectar observaciones-realizar acciones. En cambio, si se llega hasta este estado, el episodio se da por finalizado y se pasa a realizar un nuevo episodio por lo que se llama al método `OnEpisodeBegin()`.

Ahora que ya está claro, como y que orden siguen los agentes durante su entrenamiento, se puede dar paso al algoritmo empleado en Ml-Agents y como ha sido implementado en este caso. El algoritmo utilizado ha sido Proximal Policy Optimization, donde la función a optimizar es:

$$L_t^{CLIP+VF+S}(\theta) = E_t[L_t^{CLIP}(\theta) - c_1 * L_t^{VF}(\theta) + c_2 * S[\pi_\theta](s_t)]$$

Ecuación 5: Fórmula PPO

8.2 Algoritmo

Proximal Policy Optimization, o PPO, es un algoritmo on-policy que se adapta muy bien a cualquier entorno. El algoritmo tiene una serie de **hiperparámetros** que configurar para poder realizar un buen entrenamiento. Para la configuración de estos parámetros se ha seguido la

documentación sobre el fichero de configuración subido en GitHub[29]. Según esta documentación los parámetros a modificar se encuentran en la carpeta de ML-Agents, en un fichero llamado ***trainer_config.yaml***. Dentro de este documento se pueden ver varios ejemplos de proyectos de prueba que otorga ML-Agents, y donde se añade la configuración de nuestro proyecto. Ahora se pasa a explicar los campos más relevantes a la hora de entrenar nuestros agentes:

- `trainer`: (Por defecto= ppo) Es el campo que selecciona que algoritmo a emplear para el entrenamiento: `ppo` o `sac`. En nuestro caso ppo.
- `time_horizon`: (Por defecto= 64) Cuantos steps de experiencia guardar por cada agente antes de añadirlo al buffer de experiencia. En nuestro caso 128.
- `max_steps`: (Por defecto= 500000) Número total de pasos que tienen que ocurrir en el entorno antes de acabar el proceso de entrenamiento. En nuestro caso 10000000.
- `learning_rate`: (Por defecto= 3e-4) Valor inicial para el ratio de aprendizaje del descenso de gradiente. Corresponde a la fuerza de cada actualización de gradiente. En nuestro caso 2.0e-4.
- `batch_size`: (Por defecto= 1024) Numero de experiencias en cada iteración del descenso de gradiente. En nuestro caso 1024.
- `buffer_size`: (Por defecto= 10240) Número de experiencias que recolectar antes de actualizar la política. Este valor tiene que ser múltiples veces más grande que el `batch_size`. En nuestro caso 10000.
- `num_layers`: (Por defecto= 2) Número de capas ocultas en la red neuronal. En nuestro caso 2.
- `hidden_units`: (Por defecto= 128) Número de unidades en las capas ocultas de la red neuronal. Junto con el campo `num_layers` son los campos que se ocupan de crear la red neuronal a utilizar. En nuestro caso 512.
- `beta`: (Por defecto= 5.0e-3) Fuerza de la regularización de la entropía, la cual hace la política más aleatoria. Su objetivo es regularizar la entropía, lo que asegura que el agente explore adecuadamente todas las acciones posibles durante el entrenamiento. En nuestro caso 2.0e-1.
- `reward_signals`: Es la sección donde se especifican los ajustes de las señales de reward tanto excéntricas (basadas en el entorno) como intrínsecas (curiosidad).
    - `extrinsic`: Es el campo que habilita los ajustes para asegurar que el entrenamiento incorpora señales de recompensa basadas en el entorno. Dentro de este campo existen otros dos llamados `strength` y `gamma`, para modelar el impacto de la señal.
    - `curiosity`: Es el campo que asegura la incorporación de señales de curiosidad, las cuales permiten al agente explorar su entorno y aprender habilidades que podrían ser útiles más adelante. Dentro de este campo,

también, existen otros dos llamados `strength` y `gamma`, para modelar el impacto de la señal.

- `summary_freq`: (Por defecto= `50000`) Número de experiencias que se necesitan colectar antes de generar las estadísticas del entrenamiento. En nuestro caso `250000`, debido a que se necesita que todos los agentes realicen un episodio completo. Al tener 144 agentes con un máximo de pasos de 1500, necesitan 216.000 pasos (144x1500).

Una vez definidos los hiperparámetros para el aprendizaje del agente, se pasa a explicar cómo se plasman las políticas locales, grupales y globales como recompensas al agente.

Para la **política local**, el agente busca la supervivencia de él y busca estar rodeado de agentes de su población. Para ello al agente siempre que esté muerto, se le otorga una recompensa negativa de tal forma que nunca busque estar muerto. Aun así, un agente no puede vivir por que sí, primero debe interaccionar con otro agente que esté vivo para que se reproduzca. De tal forma que los agentes que están muertos, están muertos hasta que hagan contacto con un agente vivo. Cuando se realice el contacto, el agente antes muerto, vive con el color, o población, de ese agente (si le tocan varios agentes a la vez al mismo momento, la decisión de qué color tomar la toma la red neuronal). Para que se rodee de agentes de su población, se le otorga una recompensa múltiplo del número de agentes de su mismo color a su alrededor. Para que el agente reciba la información de los objetos de su alrededor se hace uso de Ray Casting. En este caso se emplea para determinar el primer objeto interceptado por un rayo, y a partir del objeto recoger su estado o color. El agente emite ocho rayos con direcciones desde los cero grados hasta los trescientos sesenta con diferencia de cuarenta y cinco grados entre cada rayo, y de esta manera recoge a todos los agentes de su alrededor.

Para la **política grupal**, lo que se busca es que la población se agrupe entre ella en bloques grandes con el fin de poder ver bloques diferenciados de poblaciones en la simulación. Para ello al agente se le otorga una recompensa equivalente al inverso de la media de las distancias que hay entre este agente y los otros agentes de la misma población. De esta manera esta recompensa depende de todos los agentes de la misma población. Para el cálculo de la distancia media entre agentes del mismo color se utiliza una clase que referencia el entorno, el cual tiene la posición de todos los cubos en el grid. Se realiza la diferencia de distancia Manhattan entre los agentes del mismo color, se suman todas estas y se dividen entre el número de agentes de ese color.

Para la **política global**, lo que se busca es que todas las poblaciones perduren el tiempo, es decir, que no haya una sola población viva en el entorno. Para ello al agente se le otorga una recompensa igual a la entropía de las poblaciones. Se realiza un histograma de las poblaciones en cada momento con la cual se calcula la entropía. Para el cálculo de esta entropía se vuelve hacer uso de la clase del entorno que cuenta con todos los agentes, y realiza el cálculo después de haber realizado la función para obtener el histograma de la matriz.

Sin embargo, faltaría una de las tareas más difíciles del aprendizaje por refuerzo, la calibración de las recompensas. Depende de cuál sea el objetivo del entrenamiento se debe dar más valor a un tipo de recompensa que a otros. En este caso al tratarse de un estudio, se han realizado numerosas pruebas buscando diferentes objetivos. Primero se estudió cómo reaccionan los agentes si la única recompensa que obtienen es la local. El resultado de esta primera prueba era de esperar, ya que el resultado es que una población sobreviva sobre las demás. Si el único objetivo del agente es perdurar en el tiempo sin tener los colores más allá de los de su alrededor, los agentes acabarán todos del mismo color. Se hicieron pruebas dando al agente como observación los agentes que tiene alrededor únicamente, y dando al agente como observación todos los agentes de la matriz; el resultado obtenido fue el mismo para ambos casos.

En el siguiente experimento se estudió cómo reaccionan estos agentes si se tiene en cuenta las políticas locales y las globales. De esta manera se intentaba solucionar el problema de que únicamente sobreviviese una única población. Como ya se ha definido antes, para la política global se le otorga la entropía total de los agentes en la malla. La imagen final de la malla varía, puesto que el resultado para observaciones de los vecinos del agente y observaciones de todos los agentes fue distinto. Aun así, no se consiguió una agrupación ordenada entre los agentes sobre el grid.

Con el fin de buscar la agrupación de poblaciones, sin que estas acaben dominando sobre el entorno, se añadió al entrenamiento anterior la política grupal. Se añadió como recompensa el inverso de la media de las distancias entre los agentes de la misma población. Remarcar que cada vez que se añade una nueva recompensa, hay que calibrar tanto el reward introducido como los rewards que ya estaban antes. Con las tres recompensas, hubo resultados diferentes si se otorgaba como observación el alrededor del agente o si se otorgaba el conjunto de todos los agentes del entorno.

9 Análisis de los resultados

Para realizar el análisis de los entrenamientos realizados para cada caso antes definido, se debe mirar como avanza y con qué política se queda el estado, y las gráficas de Tensorboard que ofrece TensorFlow sobre el aprendizaje del agente.

Para lo primero basta con implementar el **fichero .nn**, es decir la red neuronal (neural network) creada a partir del entrenamiento, sobre el entorno donde se ha realizado las pruebas y pulsar el play en Unity. De esta forma se puede observar que decisiones toma cada agente en cada caso, es decir se puede observar con que política ha acabado el agente. Aquí se observa si los agentes tienden a juntarse o a que población tienden a elegir.

Después se tienen las **gráficas** ofrecidas por **Tensorboard**, las cuales se instalan al instalar ML-Agents. Hay varias gráficas que definen el entrenamiento ejercido por el agente durante los episodios, pero las más relevantes son:

- **Cumulative Reward**, o recompensa acumulativa, muestra la recompensa acumulativa media de todos los agentes que participan en el entrenamiento. Debe aumentar si el entrenamiento ha sido exitoso, ya que el objetivo es hacer que el agente encuentre la mayor recompensa posible. En este caso, se tienen valores altos de recompensa, dado que cada episodio consta de muchos pasos, para dejar a los agentes encontrar el equilibrio.
- **Entropy**, o entropía, es la aleatoriedad de las decisiones del modelo. En otras palabras, cuanto mayor sea la entropía, más difícil es sacar conclusiones sobre la información dada. Por lo tanto, un buen comportamiento para un entrenamiento exitoso es ver la entropía disminuyendo lentamente, porque las decisiones tienen que ser más precisas cuanto más avance el entrenamiento.
- **Episode Length**, o longitud del episodio, sólo informa de la duración media de cada episodio en el entorno para todos los agentes. Está relacionado con los reajustes de la Academia y los agentes.
- **Learning Rate**, o ratio de aprendizaje, debe disminuir con el tiempo. Notifica cuán grande es el paso que da el algoritmo de entrenamiento al buscar la política óptima. Al principio de cada entrenamiento, el modelo externo trata de hacer acciones aleatorias para aprender de los nuevos estados. A medida que el entrenamiento progresa, trata de elegir la mejor acción y hace que la tasa de aprendizaje disminuya.
- **Policy Loss**, pérdidas de la política, oscilan durante el entrenamiento. Se refiere a los cambios en el proceso de decisión de una acción.
- **Value Estimate**, o valor estimado, debe aumentar durante entrenamientos exitosos, pues esta gráfica muestra el valor promedio estimado para todos los estados ya visitados.

- **Value Loss**, o pérdida de valor, debe disminuir durante un entrenamiento exitoso (cuando la recompensa se vuelve estable). Esta gráfica dice lo bueno que es el modelo para predecir el valor de cada estado.

Para cada caso antes definido se tiene un conjunto de redes neuronales y gráficas (se muestran únicamente las gráficas de la recompensa acumulada y la entropía, ya que son las más problemáticas).

9.1 Local

El estado final del entorno y las gráficas obtenidas para si el agente está observando su alrededor o si está observando el entorno entero, son **iguales**, excepto la gráfica de longitud del episodio, a consecuencia de que al dar un espacio de observaciones mayor el tiempo de cálculo es mayor.

En el caso de estar únicamente aplicando políticas locales se tiene:

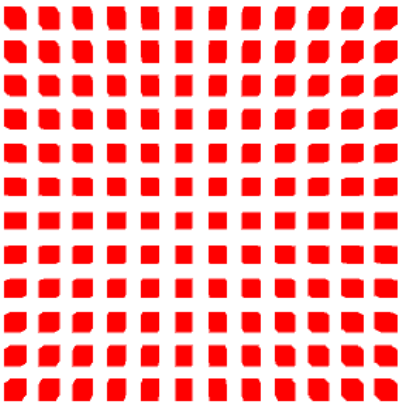
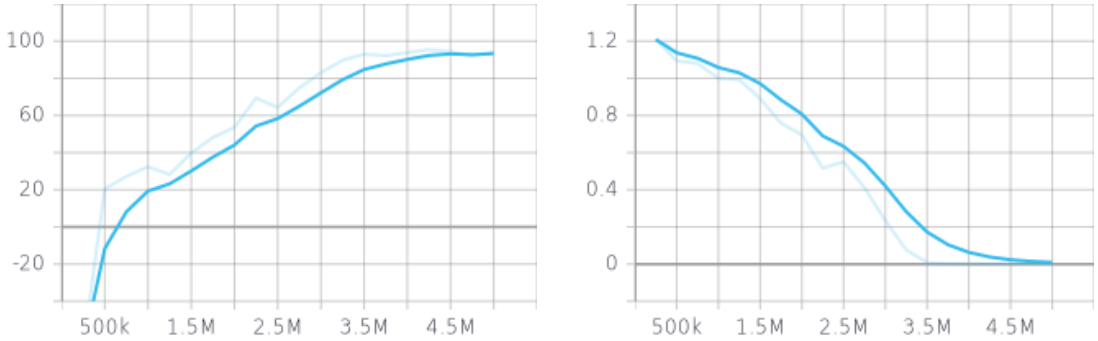


Ilustración 14: Entorno con políticas locales



Gráficas 1: Cumulative Reward y Entropy de políticas locales

Como se puede observar si el agente sólo tiene en cuenta su propia supervivencia, el entorno acaba siendo dominado por una sola población, en razón de que los agentes solo buscan el bien individual. En la supervivencia, por selección natural, es normal que sólo sobreviva una raza, es la ley del más fuerte.

En las gráficas se pueden ver que todos los entrenamientos han sido estables. Se puede observar como la función de recompensa avanza adecuadamente, la entropía disminuye de forma estable

y constante. Esto indica, respectivamente, que los valores del reward están correctamente equilibrados (en este caso al solo haber una recompensa no es tan relevante, pero si es relevante el hecho de que crezca de forma estable), y que los valores asignados en el fichero de configuración del algoritmo son los correctos.

9.2 Local y global

El estado final del entorno y las gráficas obtenidas para si el agente está observando los agentes que le rodean o si está observando el los agentes que forman la matriz completa son **diferentes**, es por eso que en este caso se explican de forma separada, vecinos como observación y, grid como observación.

9.2.1 Vecinos como observación

Aplicando políticas locales y globales se tiene:

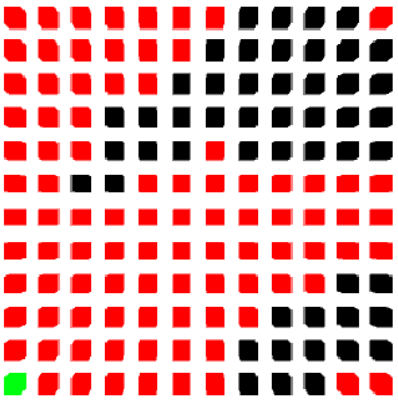
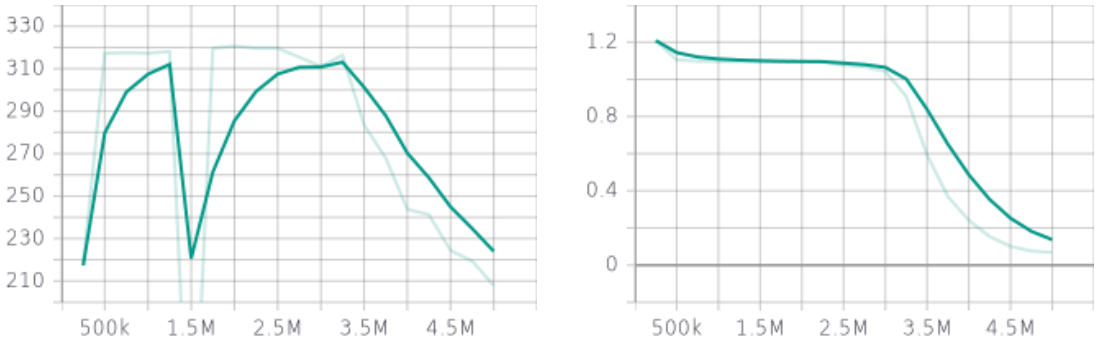


Ilustración 15: Entorno con políticas locales y globales y observaciones de vecinos



Gráficas 2: Cumulative Reward y Entropy de políticas locales y globales y observaciones de vecinos:

El comportamiento de los agentes para este caso es un entorno donde dos poblaciones luchan por la supervivencia, pero de una forma bastante peculiar. Los agentes están constantemente cambiando de color cada paso entre rojo y negro de tal forma que el entorno cambia a cada step. Se mantienen dos bandos, aunque los agentes que los componen varían completamente por cada paso.

Observando las gráficas se aprecia como durante el entrenamiento el agente ha ido aprendiendo con el paso del tiempo. Aprende al millón de pasos, y a los tres millones de pasos. Después de haber aprendido a los tres millones la entropía disminuye drásticamente, haciendo así que el reward también disminuya. El pico del millón de pasos se puede deber a que en ese episodio no ha llegado a existir ninguna población, ya que el estado inicial del entorno es aleatorio.

Este escenario incita al estudio de una nueva línea donde se modelen los agentes mediante su estabilidad de comportamiento a lo largo del tiempo. Por ejemplo, forzando que una población no varíe en más de un porcentaje a lo largo de un cierto número de pasos.

9.2.2 Malla como observación

Aplicando políticas locales y globales se tiene:

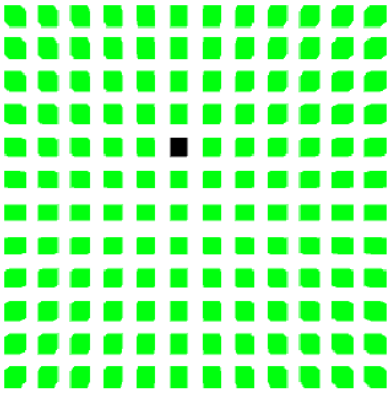
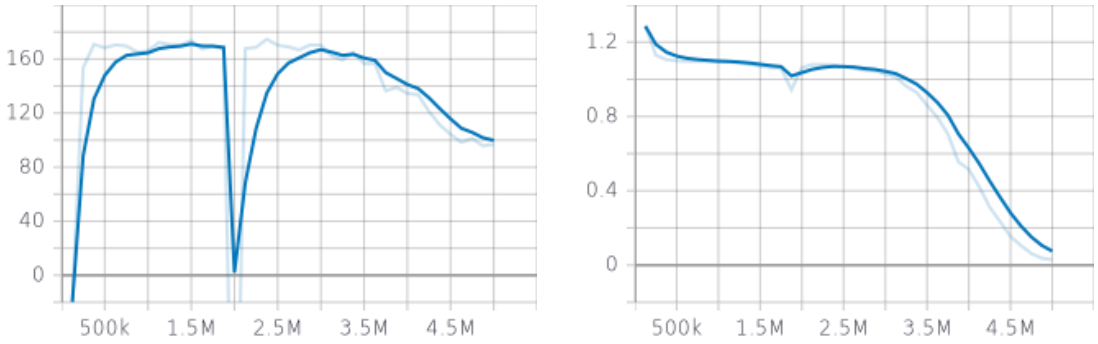


Ilustración 16: Entorno con políticas locales y globales y observaciones de la malla



Gráficas 3: Cumulative Reward y Entropy de políticas locales y globales y observaciones de la malla

El agente, al tener en cuenta la política local y global, tiende a la inestabilidad. Para este caso, cambiando levemente los hiperparámetros del algoritmo, obtienen varios tipos de resultado. En la mayoría de escenarios el resultado obtenido es el mostrado en la Ilustración 16, donde una población domina sobre las demás, ya que sólo encuentra la relación de la recompensa local. Otro caso que se puede obtener es cuando el agente encuentra la relación con la recompensa global, y busca maximizar está creando un entorno donde la aparición de las poblaciones es equiprobable (para maximizar la entropía). Para ese escenario los agentes cambian constantemente de color sin agruparse entre poblaciones. En el caso de únicamente otorgar una recompensa local el entorno se queda del mismo color durante todos loes episodios, sin

embargo, para local y global con observaciones del entorno completo, de vez en cuando hay un agente que cambia el color, esto se debe a la política global.

En las gráficas se pueden encontrar un entrenamiento en el cual los agentes han ido aprendiendo de forma estable, hasta que se tiene un pico a los dos millones de pasos. Este pico como antes ya se ha dicho, se puede deber a que el episodio no ha tenido ningún agente con vida desde el primer estado, haciendo así que durante todo el episodio todos los agentes estuviesen muertos. Este episodio no es un problema ya que como se puede ver el agente en el episodio siguiente sigue aprendiendo. La entropía disminuye de forma estable por lo que se puede decir que este enteramiento ha sido correcto,

9.3 Local, grupal y global

Si el agente recibe como observación los agentes de su alrededor y si recibe los agentes de toda la malla, en caso de aplicar políticas locales, grupales y globales, el resultado es **diferente**. Por lo tanto, se explica este apartado también está separado en dos apartados.

9.3.1 Vecinos como observación

Aplicando políticas locales, grupales y globales y dando como observación los agentes de alrededor al agente se tiene:

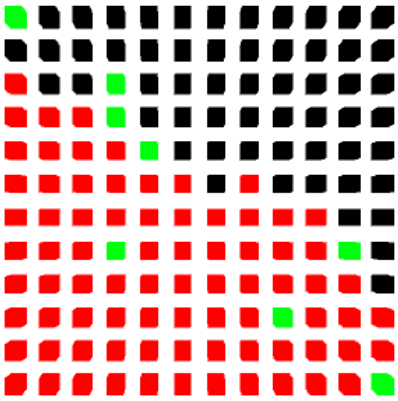
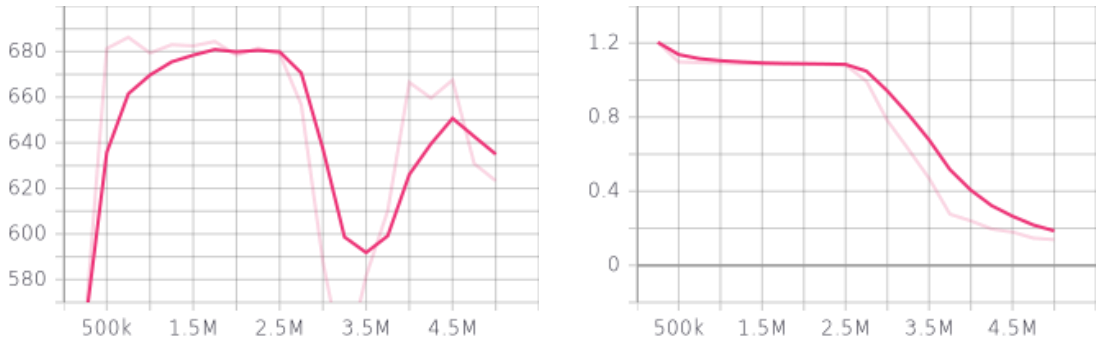


Ilustración 17: Entorno con políticas locales, grupales y globales y observaciones de vecinos



Gráficas 4: Cumulative Reward y Entropy de políticas locales, grupales y globales y observaciones de vecinos

Como se puede observar si el agente se comporta teniendo en cuenta políticas locales, grupales y globales, se obtiene un entorno donde los agentes de dos poblaciones distintas consiguen dividirse el entorno para conseguir una dominancia conjunta. La población de color verde es la población dominada la cual sobrevive en el entorno gracias a la política global. Esta distribución sobre la malla se debe a que el agente al solo darse cuenta de lo que tiene a su alrededor, únicamente conoce que si se junta con agentes de su misma población aumenta su recompensa, esto se debe a la política local y a la grupal. Pero como se puede observar en la malla el verde se mantiene vivo y como se ha dicho esto es gracias a la política global. La política global en este caso actúa de forma independiente funcionando como observación de la distribución en el grid.

Esta vez al tener tres tipos de recompensa, se distingue una clara irregularidad en la gráfica a los tres millones de pasos. Esta irregularidad no implica que los resultados de este entrenamiento no sean válidos, lo que ha ocurrido es que el agente ha comenzado a explorar hasta el punto de llegar a una política que en principio favorece menos. Pero finalmente, como se ve en las Gráficas 4, el agente acaba encontrando la estabilidad debido a que la recompensa acumulativa acaba aumentando, mientras la entropía sigue disminuyendo. Por lo tanto, estos resultados indican que la calibración de la recompensa debería ser más precisa, pero, aun así, se consigue llevar a la política óptima. También indican que los hiperparámetros del algoritmo PPO están bien dados.

9.3.2 Malla como observación

Aplicando políticas locales, grupales y globales y dando como observación los agentes de toda malla al agente se tiene:

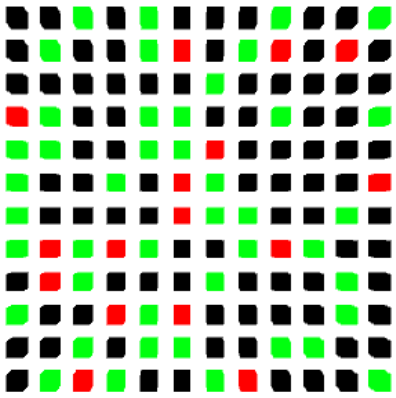
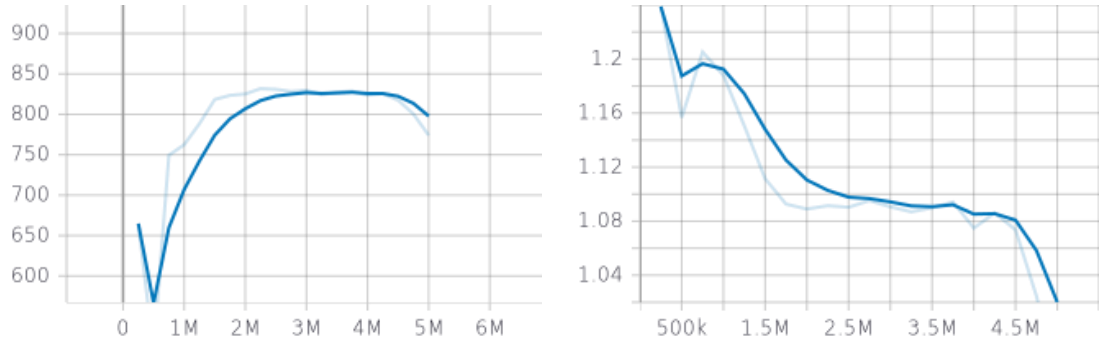


Ilustración 18: Entorno con políticas locales, grupales y globales y observaciones de la malla



Gráficas 5: Cumulative Reward y Entropy de políticas locales, grupales y globales y observaciones de la malla

Para comportamientos locales, grupales y globales observando el entorno completo, los agentes no son capaces de llegar a ningún momento de estabilidad. Los agentes únicamente cambian de población de forma aleatoria intentando recorrer las posibilidades del vector de observaciones unto a las recompensas. Con un número tan alto de observaciones y tres señales de recompensa al mismo tiempo, el agente necesita un número elevado de pasos para igual llegar a algo.

Como se puede ver en las gráficas el entrenamiento se realizó para cinco millones de pasos, pero también se realizó para quince millones y el resultado fue el mismo. Es un escenario demasiado complejo para el agente y demasiado complejo como para entrar a estudiarlo en un proyecto de nivel de Trabajo de Fin de Grado (es más un Trabajo de Fin de Master). Como se puede ver el agente si aprende mientras los pasos aumentan, dado que la recompensa va en aumento, pero, por otro lado, la gráfica de la entropía disminuye muy poco para haber transcurrido cinco millones de episodios. Una solución para esto es hacer la beta para que así la entropía disminuya más rápido. Pero el objetivo no es que disminuya de forma más rápida, sino que con esa beta sea capaz de aprender, por lo que la verdadera solución, como bien se menciona antes, es dejarlo más tiempo entrenando. La razón por la que la entropía en este caso tenga esta forma se debe a que con observaciones más sencillas el agente puede descubrir estados con más recompensa con mayor facilidad, por lo que se vuelve menos aleatorio con el fin de visitar esos estados. Es decir, es más fácil para el agente asociar la recompensa con los estados si las observaciones son más simples, por lo que la política se vuelve menos aleatoria más rápido.

10 Planificación

La planificación es el plan general metódicamente organizado para obtener el objetivo del proyecto. En esta sección se define esta planificación seguida para la realización del proyecto. Se define primero el ciclo de vida, seguido del Diagrama de Gantt.

10.1 Ciclo de vida

El ciclo de vida de un proyecto software sirve para explicar y entender los pasos seguidos durante el desarrollo del proyecto. Dentro de este ciclo se tienen cuatro fases: el planteamiento inicial del proyecto, la preparación del proyecto, el desarrollo del proyecto, y la documentación del proyecto.

10.1.1 Planteamiento inicial del proyecto

En este apartado se explica la elección del objetivo del proyecto junto con el director del proyecto. Es la parte donde se establecen las condiciones y objetivos a cumplir por parte del alumno. Para esta primera fase, se tuvieron varias reuniones donde finalmente se acordó el objetivo del proyecto, y los recursos, tecnologías y herramientas necesarios para su desarrollo.

10.1.2 Preparación del proyecto

Cuando ya se tiene claro el objetivo del proyecto, el siguiente paso es ver cómo realizarlo o implementarlo. Por ello, el siguiente paso fue realizar un análisis de alternativas sobre que algoritmo y que simulador emplear para los entrenamientos. Para realizar este análisis se realizó una investigación profunda sobre blogs, libros electrónicos, estudios y proyectos con el fin de entender todo a la perfección. Una vez elegidas las alternativas, se estudió cómo funcionan y se buscó con que se podía empezar.

10.1.3 Desarrollo del proyecto

Esta es la parte más extensa del proyecto, debido a que refleja la parte donde se crea el entorno del agente, el código del agente y se realizan las pruebas para después estudiarlas. El comienzo es lento, ya que son los primeros pasos sobre este entorno nuevo, pero finalmente, después de muchos entrenamientos, se obtienen los resultados esperados.

10.1.4 Documentación del proyecto

Esta es la parte final, donde se recoge todo lo anterior y se plasma sobre la memoria del proyecto.

10.2 Diagrama de Gantt

Gracias a esta herramienta, diagrama de Gantt, se pueden planificar y programar tareas a lo largo de la duración del proyecto. Antes de mostrar el diagrama se muestra una tabla para que se pueda ver las fases explicadas en el apartado anterior y su periodo de duración. El diagrama de Gantt representa el flujo de las fases en el periodo de tiempo.

Id	Task Name	Duración	Comienzo	Fin
1	**Trabajo de Fin de Grado**	**110 días**	**jue 13/02/20**	**jue 16/07/20**
2	**Planteamiento inicial del proyecto**	**12 días**	**jue 13/02/20**	**lun 02/03/20**
3	Elección de tema para el proyecto	11 días	jue 13/02/20	jue 27/02/20
4	Definición de objetivos y recursos	2 días	vie 28/02/20	dom 01/03/20
5	Inicio del proyecto	0 días	lun 02/03/20	lun 02/03/20
6	**Preparación del proyecto**	**10 días**	**lun 02/03/20**	**dom 15/03/20**
7	Investigación sobre el tema	5 días	lun 02/03/20	vie 06/03/20
8	Realización del análisis de alternativas	4 días	lun 09/03/20	jue 12/03/20
9	Familiarización con las alternativas seleccionadas	2 días	vie 13/03/20	dom 15/03/20
10	**Desarrollo del proyecto**	**83 días**	**mié 18/03/20**	**dom 12/07/20**
11	Creación del entorno y los agentes	14 días	mié 18/03/20	sáb 04/04/20
12	Entrenamientos y pruebas	70 días	lun 06/04/20	vie 10/07/20
13	Estudio de los resultados	35 días	mar 26/05/20	dom 12/07/20
14	**Documentación del proyecto**	**34 días**	**vie 29/05/20**	**mié 15/07/20**
15	Estructuración de la memoria	2 días	vie 29/05/20	sáb 30/05/20
16	Redacción de la memoria	33 días	lun 01/06/20	mié 15/07/20
17	Fin del proyecto	0 días	jue 16/07/20	jue 16/07/20

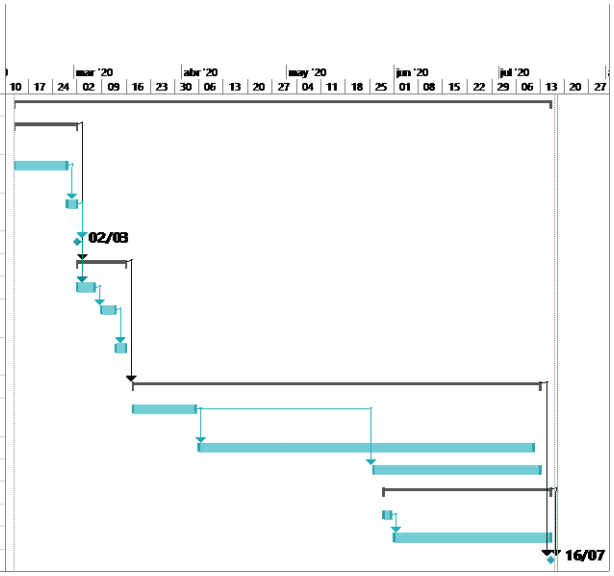


Tabla 4: Tareas a realizar y diagrama de Gantt

11 Costes del proyecto

En esta sección se presentan los costes y recursos empleados durante el proyecto. Para el cálculo de los costes se han tenido en cuenta el coste de los recursos humanos y el coste de los recursos materiales empleados.

11.1 Recursos humanos

El personal necesario para el desarrollo de este proyecto es un **ingeniero junior** y un **director de proyecto**. El ingeniero junior se encarga de la realización del proyecto y su documentación mientras el director de proyecto se encarga de la coordinación. Los costes de cada uno se recogen en la siguiente tabla:

ID	Personal	Coste unitario (€/h)
P1	Director del proyecto	110
P2	Ingeniero Junior	30

Tabla 5: Coste unitario del grupo de trabajo

Teniendo en cuenta el número de horas dedicadas, mostrados en el Diagrama de Gantt, se calcula el coste total.

ID	Horas	Coste unitario (€/h)	Coste total (€)
P1	50	110	5.500
P2	300	30	9.000
Subtotal			**14.500**

Tabla 6: Coste total de recursos humanos

11.2 Recursos materiales

Respecto a los recursos empleados para el proyecto, hay algunos que son de uso libre, sin embargo, se ha tenido que pagar por otros de ellos.

11.2.1 Materiales amortizables

Para el desarrollo de hizo uno de un ordenador de escritorio, y luego de un portátil (a causa de que por el virus no se podía acceder al ordenador de mesa). La siguiente tabla muestra el coste de los materiales amortizables:

Concepto	Coste de adquisición (€)	Vida útil (meses)	Tiempo de uso (meses)	Coste (€)
PC HP Z400 Workstation	900	36	3	75
Asus ROG Strix G15 G512LW-HN038	1.500	60	4	100
Subtotal				**175**

Tabla 7: Coste total de los materiales amortizables

11.2.2 Materiales fungibles

Para los materiales fungibles utilizados se tiene la siguiente tabla:

Concepto	Coste (€)
Material de oficina	30
Documentación	60
Subtotal	**90**

Tabla 8: Coste total de materiales fungibles

11.3 Coste total del proyecto

Concepto	Coste (€)
Recursos humanos	14.500
Materiales amortizables	175
Documentación	90
Subtotal	14.765

Tabla 9: Coste total del proyecto

12 Conclusiones

El principal objetivo del proyecto ha sido siempre la obtención de un sistema capaz de estudiar la convivencia de múltiples agentes multipoblacionales, para así en un futuro extrapolarse a cualquier proyecto de autómatas multiagente. Este objetivo ha llevado a varios escenarios y pruebas no satisfactorias, las cuales han servido para poder desarrollar los resultados alcanzados. Tras su análisis del sistema final para cada caso, se han llegado a las siguientes conclusiones.

Del Análisis de los resultados, para todos los casos, se deduce que el agente siempre va a tender a evolucionar y sobrevivir. En ocasiones de formas sencillas como manteniéndose de un color si la única política activa es la local, o de formas complejas cambiando de color constantemente como ocurre con el caso de tener políticas locales y globales como recompensa. También se puede observar cómo cuanto más complejo se hace el sistema, ya sea por añadir nuevas políticas como recompensa o añadiendo un espacio mayor de observaciones, más le cuesta al agente el aprendizaje, y por lo tanto más preciso habrá que ser con los valores de los hiperparámetros y la calibración de la recompensa. Esta precesión y un mayor número de pasos es lo necesario para continuar con este proyecto, para poder obtener resultados estables con las tres políticas y espacios de observaciones grandes. Se puede observar también la sensibilidad del algoritmo de RL empleado a los cambios en la parametrización. Realizando pequeños cambios en el balance entre los tres objetivos del agente o cambiando ligeramente los hiperparámetros del fichero de configuración, los agentes obtienen resultados muy distintitos, y, en la mayoría de casos, inestables. Otra continuación de este proyecto es ver como implementar estas redes en diferentes escenarios, como pueden ser los enjambres de drones.

La conclusión final de este proyecto es que, aunque para la elaboración de este tipo de estudios se requiera un conocimiento avanzado sobre sistemas complejos de RL y mucho tiempo debido al tiempo de cada entrenamiento, si las soluciones técnicas empleadas tanto para la definición de la función recompensa, como para la política de selección de acciones son las adecuadas, el conocimiento adquirido por parte del agente tendera a ser óptimo. Es por eso que, es fundamental realizar previos análisis para definir y estructurar los objetivos a cumplir, siendo posible así definir correctamente las bases del aprendizaje del agente.

13 Bibliografía

[1] Seita, D. (2018, 12 diciembre). Scaling Multi-Agent Reinforcement Learning. Recuperado de https://bair.berkeley.edu/blog/2018/12/12/rllib/

[2] Fumo, D. (2017, 15 junio). Types of Machine Learning Algorithms You Should Know. Recuperado de https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861

[3] OpenAI. (2018, noviembre 11). RL Intro. Recuperado de https://github.com/openai/spinningup/blob/master/docs/spinningup/rl_intro2.rst

[4] Lapan, M. (2018). Deep Reinforcement Learning Hands-On. Zaltbommel, Países Bajos: Van Haren Publishing.

[5] Wikipedia contributors. (2020, 29 mayo). Technology readiness level. Recuperado de https://en.wikipedia.org/wiki/Technology_readiness_level

[6] Cybenko, George & Jiang, Guofei & Bilar, Daniel. (1999). Machine Learning Applications in Grid Computing.

[7] Schilling, F., Lecoeur, J., Schiano, F., & Floreano, D. (2018, 3 septiembre). Learning Vision-based Cohesive Flight in Drone Swarms. Recuperado de https://arxiv.org/abs/1809.00543

[8] OpenAI. (2018, noviembre 8). Algorithms. Recuperado de https://github.com/openai/spinningup/blob/master/docs/user/algorithms.rst

[9] freeCodeCamp.org. (2018, 9 agosto). An introduction to Q-Learning: reinforcement learning. Recuperado de https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/

[10] Choudhary, A. (2020, 27 abril). A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python. Recuperado de https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[11] Simonini, T. (2018, 29 marzo). An introduction to Policy Gradients with Cartpole and Doom. Recuperado de https://www.freecodecamp.org/news/an-introduction-to-policy-gradients-with-cartpole-and-doom-495b5ef2207f/

[12] Kapoor, S. (2018, 2 junio). Policy Gradients in a Nutshell. Recuperado de https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d

[13] Peters, J. (2010, 26 noviembre). Policy gradient methods - Scholarpedia. Recuperado de http://www.scholarpedia.org/article/Policy_gradient_methods

[14] Weng, L. (2018, 8 abril). Policy Gradient Algorithms. Recuperado de https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[15] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., Abbeel, P. (2015, 19 febrero). Trust Region Policy Optimization. Recuperado de https://arxiv.org/abs/1502.05477v5

[16] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, 20 julio). Proximal Policy Optimization Algorithms. Recuperado de https://arxiv.org/abs/1707.06347

[17] OpenAI. (2018, noviembre 8). Soft Actor-Critic. Recuperado de https://github.com/openai/spinningup/blob/master/docs/algorithms/sac.rst

[18] Shital Shah. (2017, 13 noviembre). AirSim Car Demo. Recuperado de https://www.youtube.com/watch?v=gnz1X3UNM5Y

[19] Contributors to Wikimedia projects. (2020, 17 mayo). Main Page. Recuperado de https://commons.wikimedia.org/wiki/Main_Page

[20] Çelik, F. (s. f.). docs/ML-Agents-Overview.md · 码云极速下载/Unity-ML-Agents - 码云 Gitee.com. Recuperado de https://gitee.com/mirrors/Unity-ML-Agents/blob/master/docs/ML-Agents-Overview.md

[21] Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, Amir Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). Deepmind lab. arXiv:1612.03801. Recuperado de https://arxiv.org/abs/1612.03801

[22] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). IMPALA: Scalable distributed deep-rl with importance weighted actor-learner architectures. Recuperado de https://arxiv.org/abs/1802.01561

[23] Vishakha , J. (2018, 24 febrero). IMPALA distributed agent in DMLab-30. Recuperado de https://www.techleer.com/articles/488-impala-distributed-agent-in-dmlab-30/

[24] OpenAI. (s. f.). Gym: A toolkit for developing and comparing reinforcement learning algorithms. Recuperado de https://gym.openai.com/

[25] Yu, F. (2017, 12 octubre). Deep Q Network vs Policy Gradients - An Experiment on VizDoom with Keras | Felix Yu. Recuperado de https://flyyufelix.github.io/2017/10/12/dqn-vs-pg.html

[26] Casas, N. (2016, 27 julio). What are the advantages / disadvantages of off-policy RL vs on-policy RL? Recuperado de https://datascience.stackexchange.com/questions/13029/what-are-the-advantages-disadvantages-of-off-policy-rl-vs-on-policy-rl

[27] Conceptual differences - A2C & PPO (reinforcement learning). (2017, 19 agosto). Recuperado de https://www.reddit.com/r/MachineLearning/comments/6unujm/d_conceptual_differences_a2c_ppo_reinforcement/

[28] Vasudev. (2018, 3 agosto). What is One Hot Encoding? Why And When do you have to use it? | Hacker Noon. Recuperado de https://hackernoon.com/what-is-one-hot-encoding-why-and-when-do-you-have-to-use-it-e3c6186d008f

[29] Ervin, T., Mattar, M., Berges, V.-P., Elion, C., & Coh, A. (2020). Unity-Technologies/ml-agents. Recuperado de https://github.com/Unity-Technologies/ml-agents/blob/release_3_docs/docs/Training-Configuration-File.md

[30] Lanham, M. (2018). Learn Unity ML-Agents – Fundamentals of Unity Machine Learning. Zaltbommel, Países Bajos: Van Haren Publishing.

14 ANEXO I: Código

El código del trabajo esta subido a un repositorio en GitHub con el proyecto de Unity entero. Para acceder basta con entrar en este link: https://github.com/mateolopezareal/TFG.git

El proyecto concretamente consta de 4 carpetas: Brains, Materials, Scene, Prefab, Scripts.

- La carpeta Brains contiene las redes neuronales entrenadas para cada caso descrito en el documento, y posteriormente analizados.
- La carpeta Materials guarda los materiales que coge el agente, es decir los colores con los que referencia de que población es.
- La carpeta Scene tiene dos escenas, uno para la realización de pruebas con un entorno más pequeño, y la escena para el entrenamiento final del proyecto.
- La carpeta Prefab únicamente consta del entorno del proyecto, donde se encuentran todos los agentes.
- La carpeta Scripts es donde se encuentra el código para el agente y el entorno, escritos en C#.

15 ANEXO II: Guía de instalación de ML-Agents

Antes de pasar a explicar cómo instalar ML-Agents y todos los paquetes necesarios para poder entrenar un agente, se explica que contiene ML-Agents y que programas se tienen que descargar antes. ML-Agents Toolkit contiene varios componentes:

- El paquete Unity (`com.unity.ml-agents`) que contiene el SDK de Unity en C# que se integra en la escena de Unity.
- Tres paquetes Python:
  - `mlagents`, el cual contiene los algoritmos de Machine Learning que le permite entrenar comportamientos en la escena de Unity. La mayoría de los usuarios de ML-Agents sólo necesitarán instalar directamente `mlagents`.
  - `mlagents_envs` contiene una API de Python para interactuar con una escena de Unity. Es una capa fundamental que facilita el intercambio de datos entre una escena de Unity y los algoritmos de ML en Python. En consecuencia, `mlagents` depende de `mlagents_envs`.
  - `gym_unity` proporciona una wrapper de Python para una escena Unity que soporta la interfaz de OpenAI Gym.
- Proyecto de Unity que contiene varios entornos de ejemplo que resaltan las diversas características del conjunto de herramientas para ayudar a empezar con ML-Agents.

Y para instalar y utilizar el ML-Agents Toolkit se necesita:

- Instalar Unity (2018.4 o posterior)
- Instalar Anaconda
- Instalar Python (3.6.1 o superior)
- Instalar el paquete de Unity `com.unity.ml-agents`
- Instalar el paquete Python mlagents
- Instalar TensorFlow
- Instalar CUDA y cuDNN (sólo si se tiene una GPU Nvidia)

Por lo que lo primero se instala Unity. Para ello sólo hace falta acceder a este link en el cual están todas las versiones de Unity disponibles. Solo habría que descargar una versión posterior o igual a 2018.4, en este caso la 2018.4.17. Es posible descargarlo tanto para Windows como para Linux y Mac.

Una vez instalado Unity, se procede a instalar Anaconda para crear entornos virtuales. Los entornos virtuales permiten aislar los paquetes que se usan para un proyecto en específico, y de esta forma, evitar problemas como son los bugs entre las versiones de Python 2 y 3. Sobre este entorno virtual se realiza toda la instalación necesaria de paquetes. Para instalar Anaconda solo basta con descargarse el paquete desde aquí, y seguir los pasos del ejecutable. Con Anaconda ya instalado, para comprobar que esta todo correcto se crea un entorno virtual. Para ello en el

terminal del equipo se escribe `conda create -n prueba`, esto crea un entorno virtual con el nombre prueba. Si todo ha ido correctamente, se podrá ejecutar el comando `conda activate prueba` el cual activa el nuevo entorno virtual prueba.

Antes de instalar lo referente a ML-Agents se debe integrar Python 3 en el sistema, para ello sólo hace falta, en el caso de Windows, descargar un ejecutable de aquí y ejecutarlo; y en caso de Linux, hacer un `sudo apt install python3.x` (siendo x la versión que se quiera escoger).

El siguiente paso es clonar el repositorio de github con el comando `git clone git@github.com/Unity-Technologies/ml-agents.git.` Esto instala el paquete de mlagents el cual contiene `com.unity.ml-agents.` Cuando se tenga ya el paquete `com.unity.ml-agents` se añade a Unity mediante el Packet Manager. Para ello se abre Unity, en la ventana Window se selecciona Packet Manager y se clica en el más o plus para añadir el paquete de ML-Agents. Este paquete se encuentra dentro de `com.unity.ml-agents` con el nombre de package.json. Se selecciona este archivo y ya está añadido ML-Agents a Unity.

Sin embargo, todavía no existe el paquete que permite entrenar los agentes sobre Unity, por lo que, para ello, se instala a través de Python el paquete mlagents con el comando `pip3 install mlagents`. Este paquete descarga la versión necesaria de TensorFlow para la versión de ML-Agents que se va a emplear. Una vez instalado, si no ha dado ningún tipo de error, se podrá ejecutar el comando `mlagents-learn` sin ningún tipo de problema. Si salta un error de CUDA al ejecutarlo, es porque se dispone de una gráfica de Nvidia. Por lo que para quitarlo únicamente hace falta descargar la versión correcta de CUDA y cuDNN. Dependiendo de la versión de TensorFlow instalada al realizar el comando de instalación anterior se descarga unas versiones u otras para. Siguiendo la tabla de este link, como en este caso, se he instalado Tensorflow 2.0.0, se descarga CUDA 10.0 y cuDNN 7.4.

Con todo esto ya instalado, es posible realizar entrenamientos de agentes con ML-Agents. Para comprobarlo, como en el paquete de ML-Agents clonado se tiene un proyecto con varios ejemplos, se puede abrir el proyecto desde Unity y entrenar a estos agentes con el comando `mlagents-learn` desde la terminal sobre nuestro entorno virtual.