GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE TÉCNICAS DE APRENDIZAJE POR REFUERZO EN UN DRON AUTÓNOMO

Autor: Javier Carrera Fresneda

Director: Miguel Ángel Sanz Bobi

Madrid

Julio 2021

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

APLICACIÓN DE TÉCNICAS DE APRENDIZAJE POR REFUERZO EN UN DRON AUTÓNOMO

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2020/2021 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Javier Carrera Fresneda Fecha: 06/07/2021

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado por SANZ BOBI MIGUEL ANGEL - 25965998X el día 08/07/2021 con un certificado emitido por AC FNMT Usuarios

Fdo.: Miguel Ángel Sanz Bobi Fecha: ..8.../ ..7.../ 2021

RESUMEN

APLICACIÓN DE TÉCNICAS DE APRENDIZAJE POR REFUERZO EN UN DRON AUTÓNOMO

AUTOR: Javier Carrera Fresneda
DIRECTOR: Miguel Ángel Sanz Bobi
ENTIDAD COLABORADORA: ICAI – Universidad Pontificia de Comillas

RESUMEN DEL PROYECTO

El principal objetivo de este proyecto ha sido el desarrollo de un entorno de simulación de un dron autónomo mediante algoritmos de aprendizaje por refuerzo, con la misión de completar una trayectoria fijada por su inicio y fin, evitando colisionar con obstáculos.

1. INTRODUCCIÓN

Se presenta a continuación, el esquema general que define el comportamiento del algoritmo del aprendizaje por refuerzo, según describe la Figura 1.

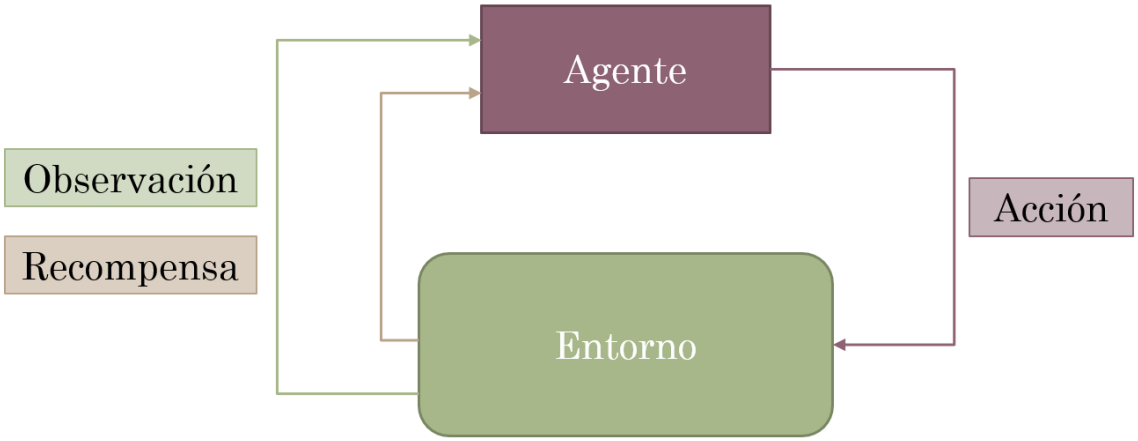


Figura 1: Esquema del aprendizaje por refuerzo

Componentes

El esquema general del algoritmo se define mediante los siguientes componentes:

- **Agente:** es el elemento del entorno de aprendizaje por refuerzo que va a aprender en base a la interacción que realiza con el entono a través de la ejecución de la acción que considere más conveniente en cada momento para conseguir alcanzar el objetivo fijado de antemano. El agente aprenderá de sus errores y aciertos a través de las acciones que ejecuta y la respuesta que a ellas le da el entorno.

- **Entorno:** es el ambiente que rodea al agente y sobre el que éste puede interactuar con acciones, dando lugar a distintos resultados que se verán reflejados en las observaciones.
- **Acción:** decisión del agente dentro de una lista de posibilidades, que causará una variación en el entorno.
- **Recompensa:** función escalar que define lo buena o mala que resulta una acción para alcanzar un objetivo prefijado.
- **Observación o estado:** representación de la tupla compuesta por el estado actual del agente, la acción que realiza y la recompensa obtenida conjunto de acciones.

Comportamiento

En primer lugar, el agente realiza una observación del entorno o estado del mismo y toma una decisión sobre la acción a realizar en el entorno, y la realiza. El entorno entonces cambia como consecuencia del acción del agente, y cambia a otro estado devolviendo además un valor escalar denominado recompensa que representa lo buena o mala que ha sido esa decisión de cara a conseguir el objetivo dado el estado observado previo a la decisión realizada A partir de aquí, el bucle se repite.

La misión del agente será entonces buscar una **relación óptima** entre el espacio vectorial de las observaciones y el de las acciones, **maximizando la recompensa** obtenida en cada paso para conseguir el objetivo buscado. Es por ello por lo que, tanto la estructura que tomará el papel del agente, como la función de recompensa serán decisivas en una correcta solución al problema.

2. DESCRIPCIÓN DEL SISTEMA

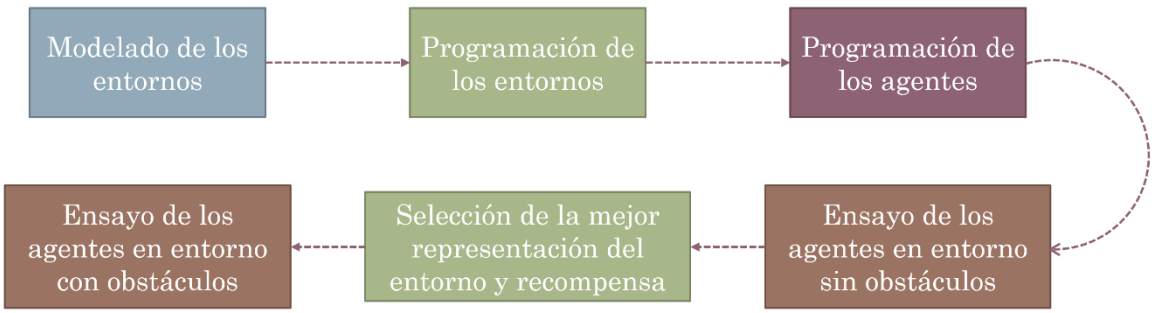


Figura 2: Flujo de trabajo empleado

En respuesta al esquema general del aprendizaje por refuerzo, se ha creado cada uno de los elementos que lo conforman, ajustándose al problema presentado, en distintas configuraciones, con el objetivo de encontrar aquella que mejor se ajusta al problema propuesto.

Se exponen a continuación la estructura del entorno que mejores resultados ha obtenido en los entornos simplificados (sin obstáculos), que posteriormente se ha implementado en los entornos finales (con obstáculos).

“AirGym – Sensors” y recompensa “Delta”:

Observaciones: un vector de 32 sensores de distancia con un alcance entre 0m y 40m, que varían su ángulo de guiñada con respecto al sistema de referencia del dron, y distribuidos uniformemente alrededor del mismo, indica en la Figura 3. A este vector de distancias se le añade al final el ángulo entre el eje x del dron y el objetivo como indica la Figura 4.

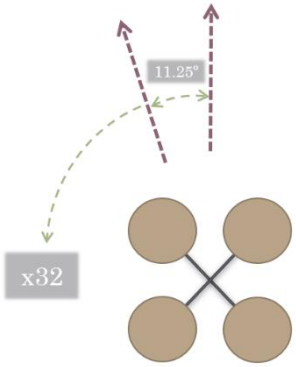


Figura 3: Disposición de los sensores de distancia en el entorno "AiGym - Sensors"

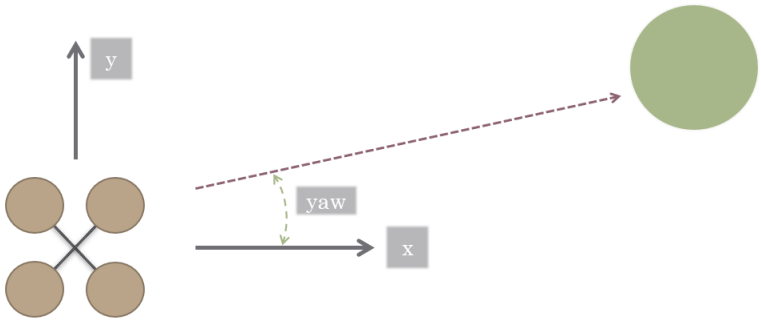


Figura 4: Sistema de referencia angular en el entorno "AirGym - Sensors"

Acciones: el agente puede moverse en los cuatro sentidos solidarios a los ejes del dron (siempre en un movimiento paralelo al plano horizontal), a saber: delante, atrás, izquierda y derecha.

Recompensa:

- Si el dron colisiona con un obstáculo, el agente pierde 1000 puntos.
- El agente recibe un valor proporcional al incremento de distancia con el objetivo en cada paso. En caso de alejarse recibe una recompensa negativa, mientras que, en el caso de acercarse, la recompensa será positiva.

Agente: Agente basado en estructura de red neuronal densa

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 33)                0
_________________________________________________________________
dense (Dense)                (None, 256)               8704
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 1028
=================================================================
Total params: 75,524
Trainable params: 75,524
Non-trainable params: 0
_________________________________________________________________
```

Figura 5: Descripción del modelo de red neuronal densa

Parámetro	Valor
Épsilon inicial	1.0
Épsilon final	0.1
Gamma	0.99
Ratio de aprendizaje	2.5e-4
Número de pasos	500.000

Entornos con obstáculos

Una vez obtenida la estructura del entorno que mejor describe el problema a resolver, se entrena al agente en los tres siguientes entornos, que simulan situaciones de interés para la implementación de drones autónomos.

Entorno "Forest": Cuenta con obstáculos de bajo grosor que tratan de simular la presencia de árboles.

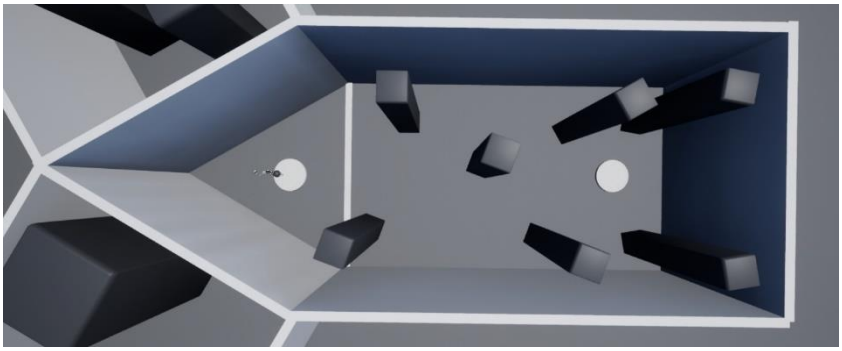


Figura 6: Planta del entorno "Forest"

Entorno "City": Presenta obstáculos de tamaño mediano simulando la presencia de edificios.

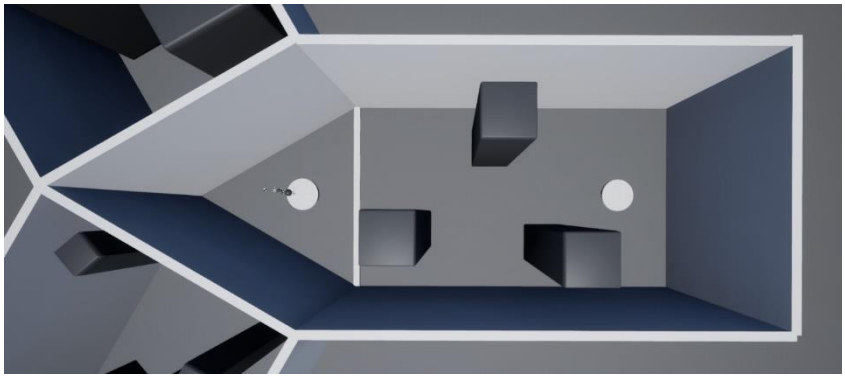


Figura 7: Planta del entorno "City"

Entorno "Mountains": Recoge los obstáculos de mayor tamaño recordando a generaciones naturales grandes como montañas

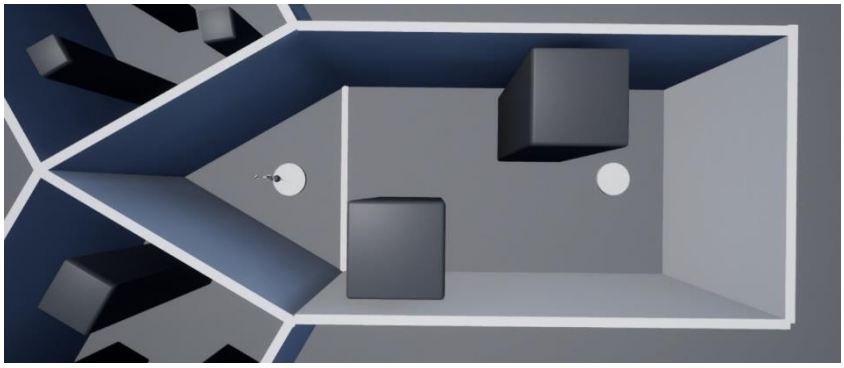


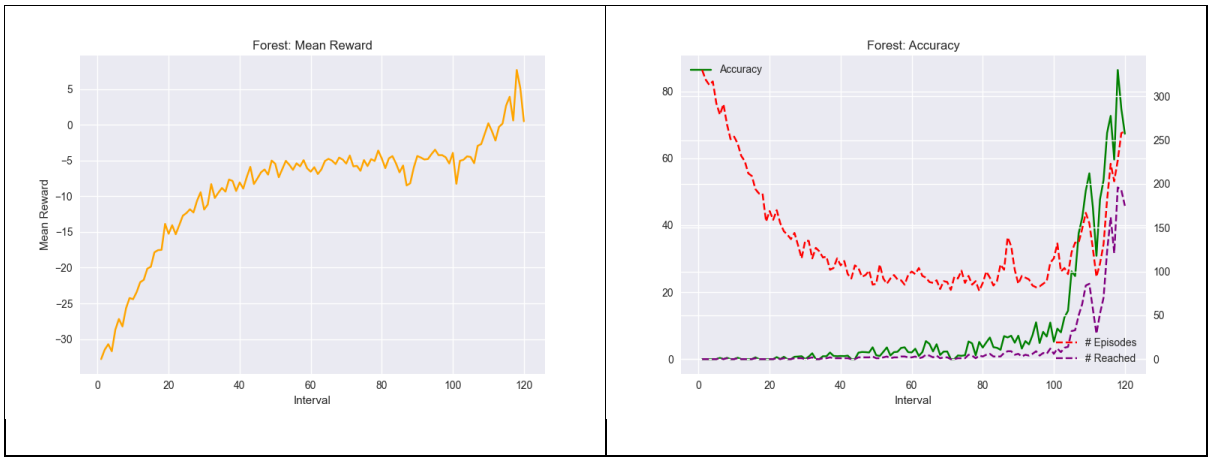
Figura 8: Planta del entorno "Mountains"

3. RESULTADOS

Se escogen como variables más representativas de un aprendizaje favorable por parte del agente la **recompensa media** obtenida por paso, y la **precisión** a la hora de alcanzar el objetivo, que se define como el número de episodios en los que se alcanza el objetivo sobre el total de episodios.

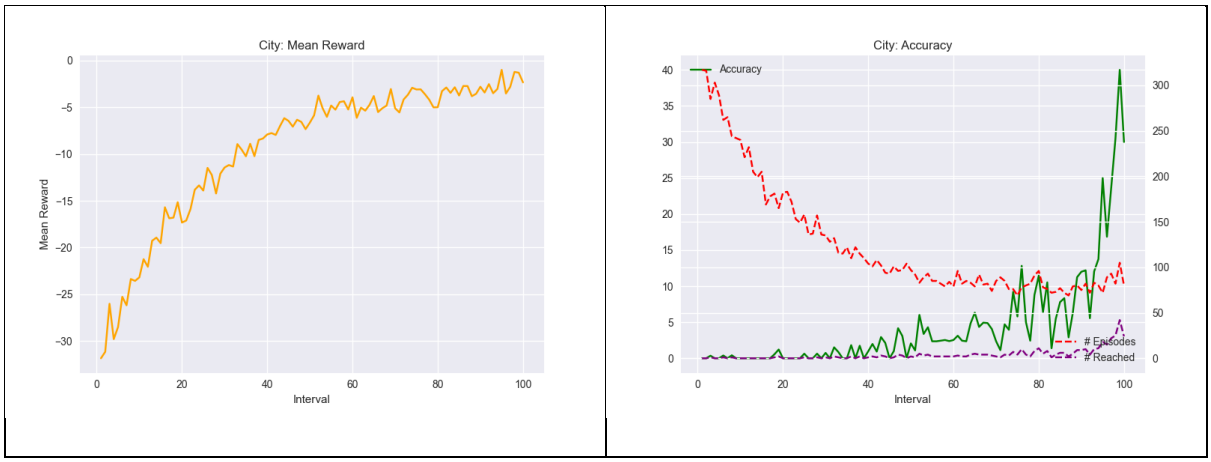
Para los tres escenarios propuestos, se exponen estas dos variables.

Entorno "Forest":



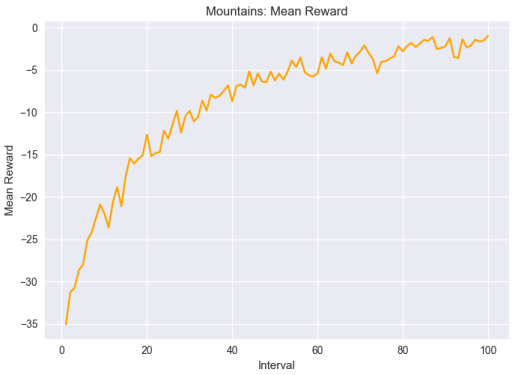
Las variables sujetas a estudio indican que el agente entrenado en el entorno "Forest" ha aprendido satisfactoriamente a esquivar obstáculos y a alcanzar el objetivo indicado, con una recompensa media final de 0.4920, y una precisión final de 98%.

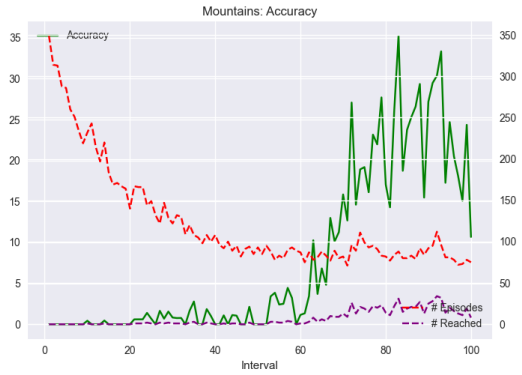
Entorno "City":



Se aprecia un aprendizaje en la tendencia al alza de la recompensa media por paso, sin embargo, esta no llega a alcanzar valores positivos, lo que indica que no está compensando todo lo que podría el efecto negativo de las colisiones con un avance predominante hacia delante. Además, la precisión plantea un dilema a la hora de considerar aptos o no los resultados del agente entrenado en el entorno "City", ya que el dron solo alcanza el objetivo un 30% de las veces.

Entorno “Mountains”:





El crecimiento final descrito por la precisión no alcanza un valor suficientemente elevado y estable como para indicar que el agente haya aprendido a llegar al objetivo y esquivar obstáculos. Se considera por lo tanto que el agente ensayado en el entorno “Mountains” no es apto para la resolución del problema propuesto.

Comparativa de los agentes:

Para evaluar la aptitud de los agentes a la hora de resolver el problema propuesto en su entorno de entrenamiento, se ha establecido un umbral de éxito a partir del 50% de precisión.

	Modelo entrenado en el entorno		
	Forest	**City**	**Mountains**
Recompensa media	0.4920	-2.3366	-0.9589
Precisión (test) [%]	98	30	23
Aptitud	Apto	No apto	No apto

4. CONCLUSIONES

Se considera que el objetivo del proyecto ha sido cumplido al obtener un modelo de agente capaz de trazar una ruta en un entorno con obstáculos sin colisionar, con una precisión del 98%. Sin embargo, ha de cuestionarse la gran diferencia entre la precisión obtenida en dicho agente y los otros dos agentes ensayados, que obtienen precisiones de 30% y 23%.

5. REFERENCIAS

[1] Richard S. Sutton and Andrew G. Barto. "*Reinforcement Learning: An Introduction"*, (2014, 2015), Bradford Books.

[2] David Silver. *"Reinforcement Learning Course, DeepMind"*, https://www.youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ, (2015), (visitado 10/2020).

[3] Andrew Ng. *"Deep Learning Specialization, Course 1: Neural Networks and Deep Learning"*, (visitado 12/2020).

[4] Sandro Skansi. *"Introduction to Deep Learning"*, (2018), SPRINGER.

[5] Harrison Kinsley and Daniel Kukiela. *"Neural Networks From Scratch in Python"*, (2020), Sentdex.

ABSTRACT

TECNICAL APLICATIONS OF REINFORMENT LEARNING IN AN AUTONOMOUS DRONE

AUTHOR: Javier Carrera Fresneda
DIRECTOR: Miguel Ángel Sanz Bobi
IN COLLABORATION WITH: ICAI – Universidad Pontificia de Comillas

SUMMARY OF THE PROYECT

The main objective of this project was the development within the realm of simulation of an environment for an autonomous drone. Through the usage of algorithms where the drone receives feedback and consequently is able to learn, the end goal was the completion of an obstacle course from beginning to end without any collisions.

1. INTRODUCTION

Figure 1 shows the diagram of the general scheme followed within the project, where the behavioral algorithm used in the apprenticeship of the drone is described.

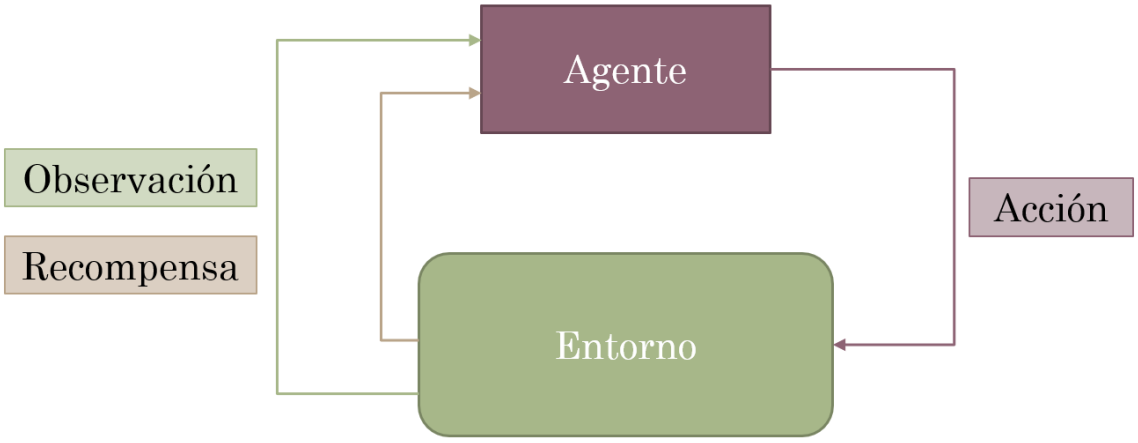


Figure 1: General Scheme of apprentenship

COMPONENTS:

The general scheme of the algorithm used is defined through a series of components which will be further described below.

- **Agent:** The agent is the component with capacity to learn within the environment through the execution of the algorithm. This apprenticeship is a byproduct of the interaction it has with its surroundings, where at all times, the agent performs the

actions believed to be best and which are later changed through the response it obtains from its settings.

- **Environment:** The environment consists of the setting that surrounds the agent at all times and where the interaction takes place. Through the different actions and its consequences, the results will vary and are reflected in the observations of each step.
- **Action:** This component consists of the decision taken by the agent within a list of possibilities, creating as a result a change within the environment.
- **Reward:** Scalar function that defines how good or how bad an action results in obtaining a predefined objective.
- **Observation or state:** Representation of the tuple composed of the initial state of the agent, the action it executes, and the reward obtained in the sum of actions taken.

Behavior:

To begin with, the agent performs an initial observation of the environment and its conditions, only to then make a decision of the action it will perform. Once this action is executed, the surroundings change, and the environment gives back a scalar known as the reward that will represent how good or bad the decision was in obtaining the objective through an analysis of the previous state and the one currently inhabited by the agent. From there, the loop repeats itself.

As a result, the main objective of the agent is the finding of an **optimal relationship** between the observations within the vectorial space and the actions taken, **maximizing the reward** obtained in every step taken to achieve the desired objective. Consequently, the role played by the agent and the rewards obtained play a pivotal place in the solution of the problem.

2. DESCRIPTION OF THE SYSTEM:

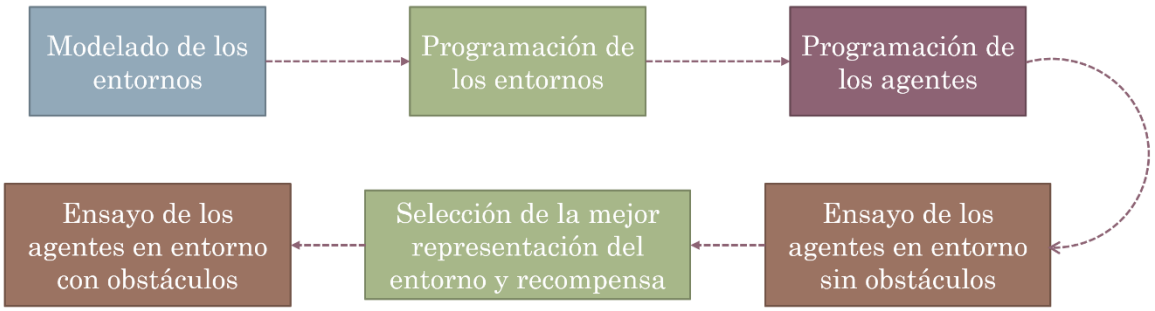


Figure 2: Work Flow used in the project

In response to the general scheme of apprenticeship through rewards, all the elements that make up the system have been created. Each one is adjusted to the problem at hand through different configurations that have as the objective, finding the one that best adjusts itself to the problem at hand.

Below, the structure that provided the best results in simplified environments (those without obstacles) are presented. This structure was later implemented in the most elaborate of settings (those with obstacles).

“AirGym – Sensors” and “Delta” rewards:

Observations: In Figure 3, you can observe a vector with 32 different sensors with a maximum reach of 40 meters. These vary their angle with respect to the reference system of the drone and are distributed uniformly around it. As indicated in Figure 4, the vector of distance obtained is then modified to include the angle between the x axis of the drone and the objective it wishes to reach.

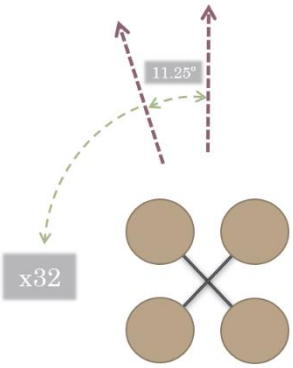


Figure 3: Placement of the distance sensors placed within the environment of "AiGym - Sensors"

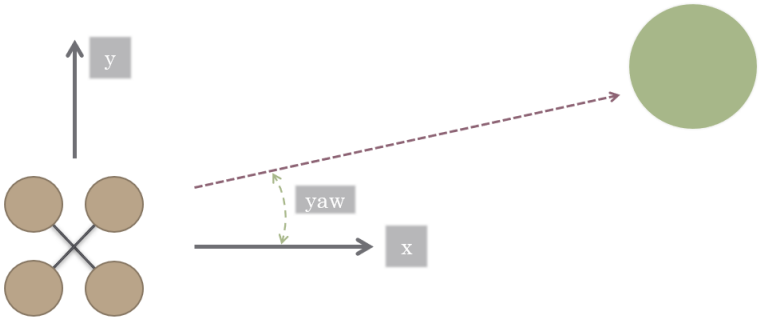


Figure 4: Angular reference system within the environment of "AirGym - Sensors"

Actions: The agent is free to move in the four paths solidary to its axis but always maintaining the movement parallel to the horizontal plane. These include moving: forwards, backwards, left and right.

Reward:

- If the drone collides with an obstacle, the agent loses 1000 points.
- The agent also receives a reward that is proportional to the incrementation of the distance with the objective. If its action results in getting closer to the objective, the drone receives a reward with positive value. If, however, the drone´s actions result in gaining distance to the objective, the reward takes a negative value.

Agent: The agent is created based on the dense neural network

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 33)                0
_________________________________________________________________
dense (Dense)                (None, 256)               8704
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 1028
=================================================================
Total params: 75,524
Trainable params: 75,524
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: Description of the dense neural network

Parameter	Value
Initial Epsilon	1.0
Final Epsilon	0.1
Gamma	0.99
Learning ratio	2.5e-4
Number of actions	500.000

Environments with obstacles:

Once we have obtained the structure of the environment that is best in tackling the problem at hand, the agent is obtained within the following three settings. All environments simulate situations where the implementation of autonomous drones is of special interest.

"Forest" Environment: This environment implements obstacles of small width with the objective of simulating the presence of trees.

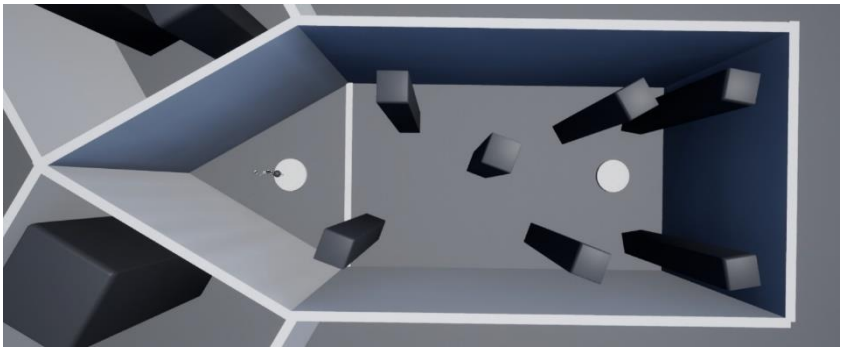


Figure 6: Birds eye view of the "Forest" environment

"City" Environment: This environment implements obstacles of medium width with the objective of simulating the presence of buildings.

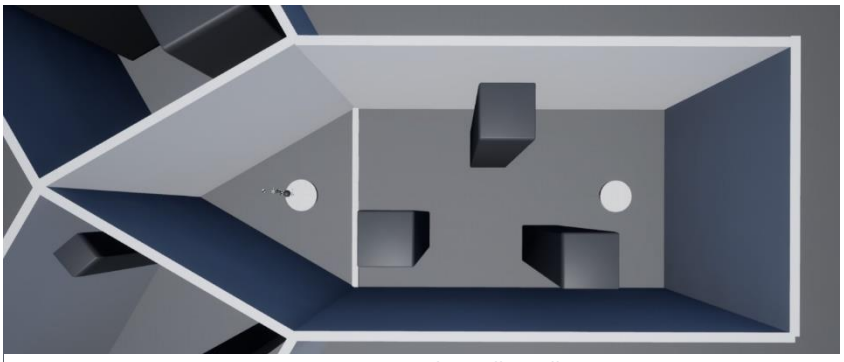


Figure 7: Birds eye view of the "City" environment

"Mountain" Environment: This environment implements obstacles of considerable width with the objective of simulating the presence of mountains.

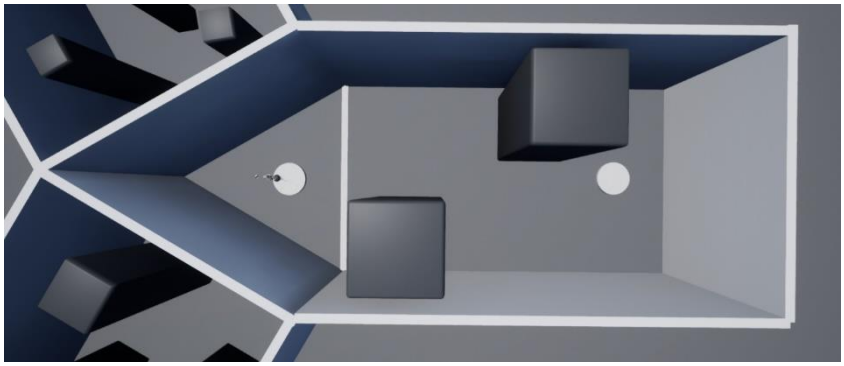


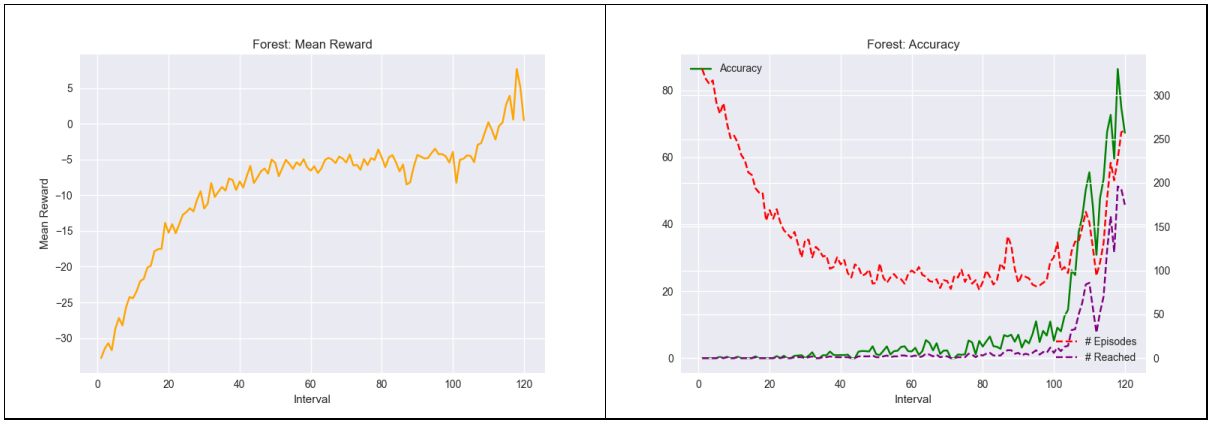
Figure 8: Birds eye view of the "Mountain" environment

3. RESULTS:

When it comes to the analysis of the actions taken by the drone, the two variables chosen as the most representative are as follows --The average reward obtained per action taken and the precision in reaching the objective. These are defined through the usage of episodes and the comparison between the number of episodes the drone reached the objective and the total number of episodes.

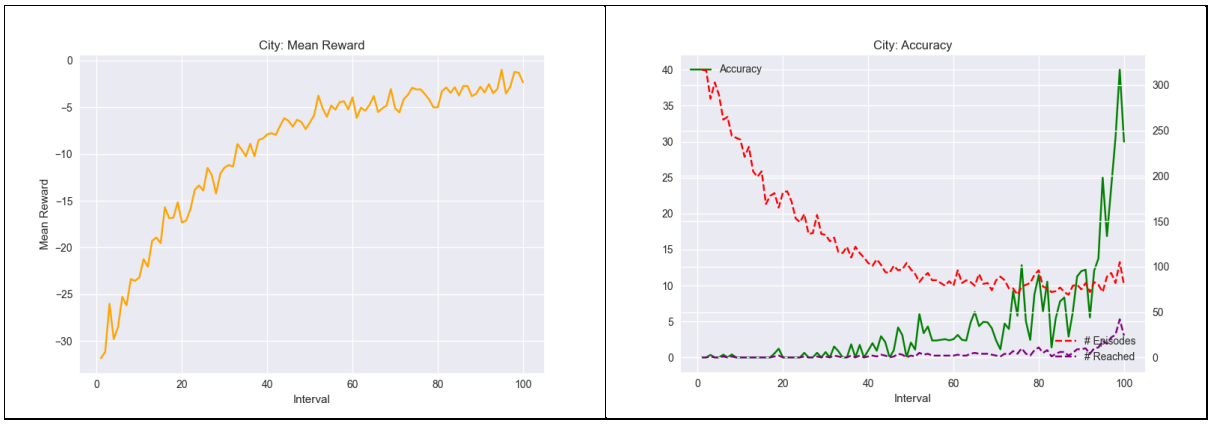
For the three scenarios previously described the results of the two most representative variables are presented.

"Forest" Environment:



The variables that undergo the analysis of the drone dictate that the agent trained within the "forest" environment was able to proficiently learn to avoid the obstacles and reach the desired objective. This was obtained through a final average reward of 0.4920 per action and a final precision of 98%.

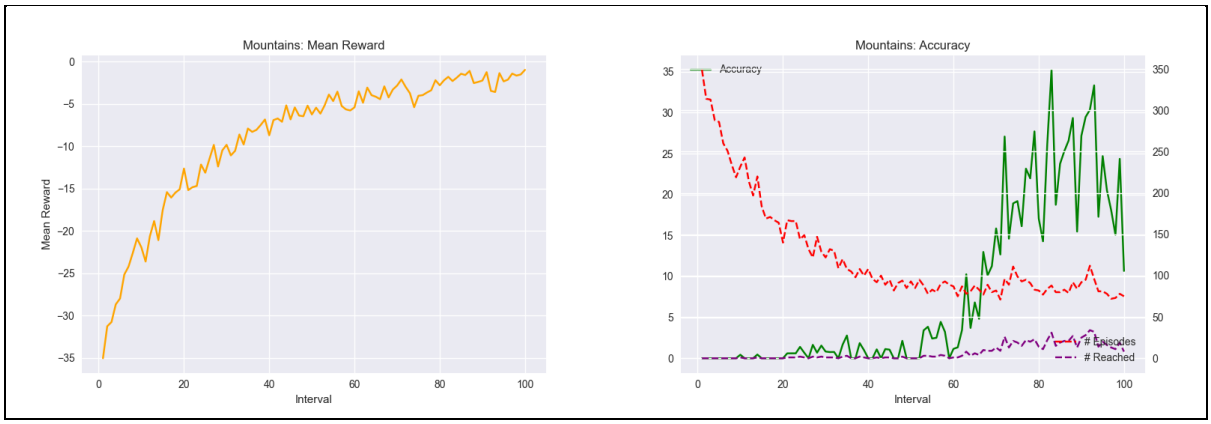
"City" Environment:



These graphs show an upwards trend in the average reward obtained per action. Nevertheless, the fact that these do not reach positive values is an indication that, although the drone appears to be learning, the positive actions taken towards reaching the goal are not compensated by the considerable deduction of reward every time it crashes into an obstacle. Also, when dictating

if the drone was able to learn or not, we reach an impasse, as the drone only managed to reach the objective 30% of the times.

"Mountain" Environment:



The upwards trend described by the precision of the drone is not able to reach a value large enough to indicate that the drone has learned. As a result, the final verdict reached is that the drone did not manage to pass the Mountain environment and therefore the solution to this particular problem was not reached.

Comparative between agents:

To evaluate whether the drone was able to learn within its training environment, a minimum of 50% precision is dictated as the lowest possible to conclude that the drone passed the problem at hand.

	Environment		
	Forest	**City**	**Mountains**
Average Reward	0.4920	-2.3366	-0.9589
Precision (test) [%]	98	30	23
Verdict	Passed	Failed	Failed

4. CONCLUSIONS

It is considered that the main objective of the project was reached. This is due to the obtention of an agent that was able to calculate the route it had to take within the environment and reach a goal without colliding with any obstacles in its path. This was achieved in the forest environment with 98% precision. However, it is essential that we also question the differences between said agent and the other two agents simulated who in return obtained precisions of 30% and 23%.

5. REFERENCES

[1] Richard S. Sutton and Andrew G. Barto. "*Reinforcement Learning: An Introduction"*, (2014, 2015), Bradford Books.

[2] David Silver. *"Reinforcement Learning Course, DeepMind"*, https://www.youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ, (2015), (visited 10/2020).

[3] Andrew Ng. *"Deep Learning Specialization, Course 1: Neural Networks and Deep Learning"*, (visited 12/2020).

[4] Sandro Skansi. *"Introduction to Deep Learning"*, (2018), SPRINGER.

[5] Harrison Kinsley and Daniel Kukiela. *"Neural Networks From Scratch in Python"*, (2020), Sentdex.

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

APLICACIÓN DE TÉCNICAS DE APRENDIZAJE POR REFUERZO EN UN DRON AUTÓNOMO

Autor: Javier Carrera Fresneda

Director: Miguel Ángel Sanz Bobi

Madrid

Julio 2021

AGRADECIMIENTOS

A mi padre.

A mi madre.

A Miguel Ángel, Álvaro y Lucía.

A Sergio.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO 1: CONTEXTO

En la búsqueda de aquello que hace al ser humano único entre el resto de los animales, destacan su inteligencia y su conciencia. Ambas características se han visto siempre fuertemente ligadas al aprendizaje, favoreciéndolo, y viéndose beneficiadas por él. A mayores niveles de conciencia dados, mayor es la capacidad de aprendizaje en una situación dada, que a su vez desemboca en un mayor desarrollo de la conciencia e inteligencia. Este bucle evolutivo es el que nos ha posicionado actualmente donde estamos, siendo ahora capaces de cuestionar nuestra propia conciencia.

De esta cuestión nace la intención de sintetizar nuestra capacidad de toma de decisiones en el entorno del aprendizaje, buscando modelos matemáticos capaces de crear esa relación entre la causa y el efecto. El acercamiento más desarrollado a este problema es el **aprendizaje por refuerzo.**

1.1 Historia del aprendizaje por refuerzo

Echando la vista atrás, a donde nace esta área de investigación, se definen dos principales ramas independientes:

- La primera, originaria del estudio de la psicología del aprendizaje animal, se centra en la compresión de un aprendizaje básico, mediante prueba y error. Edward Thorndike, psicólogo y pedagogo estadounidense, y precursor de la psicología conductista, concluye en una de sus publicaciones lo que él denomina como "Ley del Efecto":

  > "De un conjunto de respuestas asociadas a una situación, aquellas acompañadas de satisfacción hacia el animal se verán, dadas constantes el resto de las variables, más ligadas a la situación, tal que, cuando ésta vuelva a ocurrir, serán más propensas a ocurrir; aquellas acompañadas de incomodidad hacia el animal, verán, dadas constantes el resto de las variables, su ligadura con la situación debilitada, tal que, cuando ésta vuelva a ocurrir, serán menos propensas a ocurrir. Cuanto mayor sea dicha satisfacción o incomodidad, mayor será el refuerzo o la debilitación del lazo. (Thorndike, 1911, p. 244)"

- La segunda se construye en base al área de investigación del control. Ésta se centra en la búsqueda del **control óptimo**. El término "control óptimo" toma sentido a finales de la década de los 50, con el objetivo de aproximar lo máximo posible una variable de salida de un sistema dinámico a una referencia deseada a lo largo del tiempo, es decir, minimizar el error de una variable en su forma deseada y la real. Este problema de optimización (carente de aprendizaje) pone en marcha las herramientas necesarias para comenzar a desarrollar el aprendizaje por refuerzo.

Estas dos líneas convergen finalmente a finales de la década de los 80 para dar lugar a las técnicas de aprendizaje por refuerzo como las conocemos hoy en día.

En la actualidad, existen innumerables trabajos relacionados con el aprendizaje por refuerzo, pero cabe destacar aquellos realizados por las empresas DeepMind y OpenAI. Entre sus muchos proyectos, se pueden encontrar:

Empresa	Proyecto	Descripción
DeepMind	AlphaGo	En octubre de 2015 una aplicación de inteligencia artificial basada en aprendizaje por refuerzo y desarrollada por la empresa, venció al campeón europeo de Go, un juego de mesa para dos personas originario de China, computacionalmente más complicado que el ajedrez a causa de su amplio abanico de posibilidades.
	AlphaZero	En diciembre de 2017, el motor de ajedrez desarrollado mediante técnicas de aprendizaje por refuerzo adversario (la inteligencia artificial aprende jugando contra sí misma) derrotó al entonces actual mejor motor de ajedrez "Stockfish 8".
OpenAI	OpenAI Gym	La empresa ha publicado y puesto a disposición libre de los desarrolladores, la mayor cantidad al área de entornos de aprendizaje por refuerzo con su herramienta de código abierto "OpenAI Gym", que facilita la creación de entornos personalizados para la aplicación de algoritmos de aprendizaje por refuerzo.

Tabla 1: Trabajos relevantes en el área del aprendizaje por refuerzo

1.2 Esquema general del aprendizaje por refuerzo

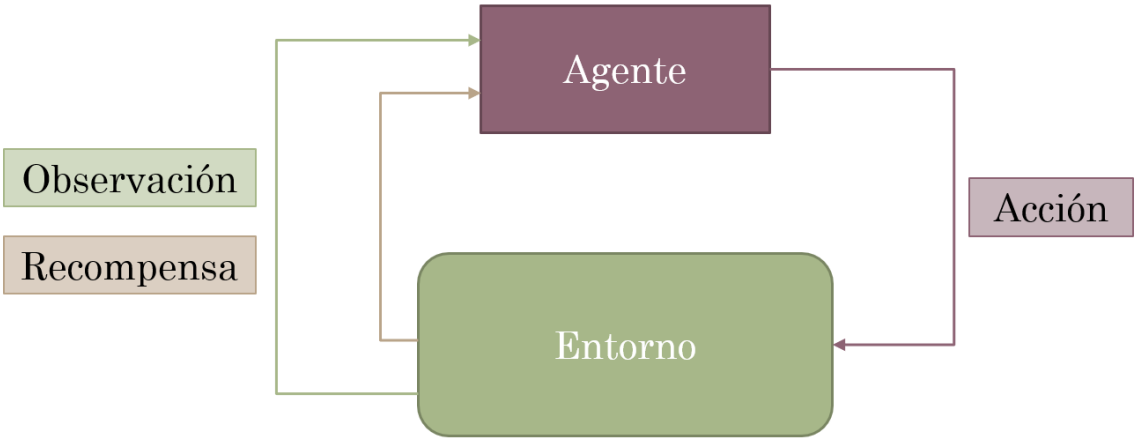


Figura 1: Esquema del aprendizaje por refuerzo

1.2.1 Componentes

El esquema general de un algoritmo de aprendizaje por refuerzo se representa en la Figura 1. A continuación, se describen sus componen principales:

- **Agente:** es el elemento del entorno de aprendizaje por refuerzo que va a aprender en base a la interacción que realiza con el entono a través de la ejecución de la acción que considere más conveniente en cada momento para conseguir alcanzar el objetivo fijado de antemano. El agente aprenderá de sus errores y aciertos a través de las acciones que ejecuta y la respuesta que a ellas le da el entorno.
- **Entorno:** es el ambiente que rodea al agente y sobre el que éste puede interactuar con acciones, dando lugar a distintos resultados que se verán reflejados en las observaciones.
- **Acción:** decisión del agente dentro de una lista de posibilidades, que causará una variación en el entorno.
- **Recompensa:** función escalar que define lo buena o mala que resulta una acción para alcanzar un objetivo prefijado.
- **Observación o estado:** representación de la tupla compuesta por el estado actual del agente, la acción que realiza y la recompensa obtenida conjunto de acciones.

1.2.2 Comportamiento

En primer lugar, el agente realiza una observación del entorno o estado del mismo y toma una decisión sobre la acción a realizar en el entorno, y la realiza. El entorno entonces cambia como consecuencia del acción del agente, y cambia a otro estado devolviendo además un valor escalar denominado recompensa que representa lo buena o mala que ha sido esa decisión de cara a conseguir el objetivo dado el estado observado previo a la decisión realizada A partir de aquí, el bucle se repite.

La misión del agente será entonces buscar una **relación óptima** entre el espacio vectorial de las observaciones y el de las acciones, **maximizando la recompensa** obtenida en cada paso

para conseguir el objetivo buscado. Es por ello por lo que, tanto la estructura que tomará el papel del agente, como la función de recompensa serán decisivas en una correcta solución al problema.

1.2.3 Procesos de decisión de Markov (MDPs)

Para poder aplicar los algoritmos de aprendizaje por refuerzo a un entorno, éste debe cumplir la **propiedad de Markov**, según la cual, todos los estados sucesivos a uno previo son dependientes únicamente del mismo, y de la acción tomada. De esta forma, al conocer un estado no es necesario conocer el historial previo de estados para tomar la decisión óptima. Esta propiedad permite también la estimación del próximo estado, dado un estado previo y una acción dada. Para ser el agente capaz de tomar la decisión óptima, se define una **función de valores**, que estima lo favorable que resulta encontrarse en un estado. Esta función de valores trata de recoger las recompensas susceptibles de suceder al estar en un estado y seguir tomando la decisión óptima a partir de él. Con el objetivo de dar más importancia a recompensas cercanas en el tiempo al estado actual, se introduce un **factor de descuento** (γ), que reduce el valor de recompensas futuras. De esta forma, la función de valores de estado viene definida por la Ecuación 1.1.

$$v_\pi(s) = \mathrm{E}\left[\sum_{k=0}^{\infty} \gamma\, R_{t+k+1} \mid S_t = \mathrm{s}\right] \tag{1.1}$$

El aprendizaje del agente reside en una correcta estimación de la función de valores. Para ello, debe existir un compromiso entre el porcentaje de acciones que el agente toma con el objetivo de **explorar** el entorno, y el que toma con el objetivo de **explotar** lo que ya ha aprendido. Para ello, se introduce la política de comportamiento "**ϵ-Greedy**", que controla mediante el parámetro "ϵ", cuando el agente toma la mejor decisión estimada según lo ya aprendido (explotación), o realiza una acción aleatoria. Se define dicha política con la Ecuación 1.2.

$$a(s_t) = \begin{cases} argmax_a\, Q(S_t, A) & P = 1 - \epsilon \\ random(A) & P = \epsilon \end{cases} \tag{1.2}$$

1.2.4 Principales algoritmos de aprendizaje por refuerzo

Respondiendo a este esquema general, los distintos algoritmos de aprendizaje por refuerzo se distinguen por la estructura que conforma al agente. A continuación, se presentan los principales métodos más utilizados:

- **"Q-Learning":**

Al comienzo del ensayo, el agente inicializa una tabla o matriz con todos sus valores a cero de dimensiones (S, A), siendo S el número de estados, y A el número de acciones. Dicha tabla contiene en cada una de sus celdas el valor que tendría tomar la acción 'a', dado el estado 's', denominados **valores Q** Este valor recoge una combinación entre la recompensa inmediata esperada por realizar dicha acción, y el valor de estar en el siguiente estado ('s+1') al que esta acción le llevaría. Dado que en un principio estos valores son desconocidos, en cada iteración del bucle, la tabla es actualizada como se indica en la Ecuación 1.3.

$$Q(s_t, a_t) = R_t + \gamma * \max Q(s_{t+1}, a') \quad (1.3)$$

Donde:

s_t: estado en el instante t.

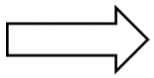
a_t: acción tomada en el instante t.

$Q(s_t, a_t)$: valor Q actual para la acción a en el estado s.

R_t: Recompensa obtenida por realizar dicha acción.

γ: Factor de descuento que atenúa el valor de la recompensa de acciones futuras frente a acciones tomadas en el instante t. $\gamma \in [0, 1]$, 1 teniendo en cuenta todas las acciones futuras por igual, y 0 únicamente la acción del instante t.

Estados	Acciones		
	0	0	0
	0	0	0
	0	0	0
	0	0	0



Estados	Acciones		
	2.4	-5.3	0.4
	-4.4	7.8	5.1
	9.2	0.6	-3.7
	1.9	3.0	2.7

Figura 2: Ejemplo de aprendizaje en "Q-Learning"

El "Q-Learning" es el algoritmo que introduce al resto de métodos más utilizados, sin embargo, cuenta con una serie de problemas que lo hacen apto únicamente para problemas reducidos:

En primer lugar, las dimensiones de la tabla crecen linealmente con la dimensión del espacio vectorial de las observaciones, y con el de las acciones, lo que resulta en tiempos de procesamiento realmente altos a poco que se amplíen dichas dimensiones. Además, las dimensiones de la tabla son discretas, por lo que solo pueden resolverse aquellos problemas con un número discreto de estados y de acciones, o que estén sujetos a discretizaciones en ambos espacios. Finalmente, en caso de encontrarse en un estado nunca explorado, el valor nunca habrá sido actualizado, por lo que el agente tomará una acción completamente aleatoria, a pesar de haber explorado estados cercanos de los que podría haber extrapolado la mejor decisión.

De esta forma, este algoritmo es recomendable para problemas donde la tabla de valores Q se pueda expresar de manera finita y con una dimensión manejable.

- **“Deep Q-Learning” (DQN):**

Mediante el algoritmo de “Deep Q-Learning” se intenta solucionar los principales problemas de dimensionalidad finita del “Q-Learning”. Para ello, en lugar de una tabla discreta se hace uso de una red neuronal como función de aproximación a los valores Q. Esta red es capaz de resolver problemas en espacios continuos, su dimensión no se ve tan afectada por incrementos en las dimensiones de las observaciones o acciones, y es capaz de extrapolar una acción óptima en estados inexplorados.

Existen diversas estructuras de redes neuronales, en función principalmente de los datos de entrada y de su posterior tratamiento. A continuación, se presentan las dos estructuras más utilizadas para tratamiento de información en forma vectorial y tratamiento de imágenes.

- **Redes Neuronales Densas:**

Las redes neuronales densas están compuestas por capas de **neuronas**. La primera capa, la **capa de entrada**, es donde se introducen los valores de los datos que se tratan de analizar, en el caso del aprendizaje por refuerzo, las observaciones, en forma de vector n-dimensional. La información se trata a través de una serie de **capas ocultas**, llegando finalmente, a la **capa de salida**, que tendrá tantas neuronas como acciones podamos realizar en el entorno. El tratamiento de la información de entrada ocurre de la siguiente forma:

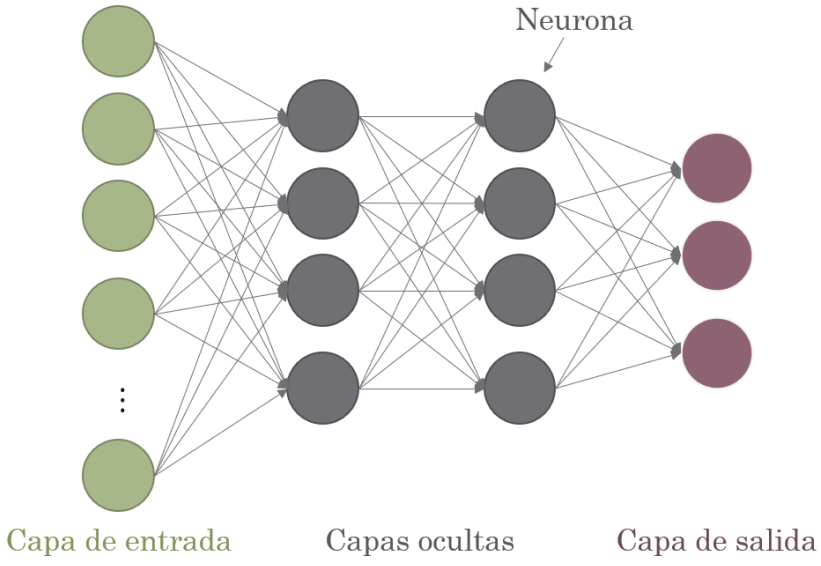


Figura 3: Esquema de red neuronal densa

La activación de cada neurona de una capa densa o de salida resulta de una combinación lineal de todas las activaciones de las neuronas de la capa anterior, a la que posteriormente se le aplica una función no lineal. De esta forma, la activación de la neurona ‘n’ de la capa ‘m’ se calculará según la Ecuación 1.4.

$$a_{m,n} = \sigma\left(\sum_{i=0}^{P} {w_{n,i}}^{m} + {b_n}^{m}\right) \tag{1.4}$$

Donde:

m: el número de la capa.

n: el número de la neurona en dicha capa.

P: el número total de neuronas de la capa anterior.

${w_{n,i}}^{m}$: el peso asociado a la neurona n de la capa m con la neurona i de la capa $m-1$.

${b_n}^{m}$: el sesgo asociado a la neurona n de la capa m.

σ: una función no lineal, denominada función de activación.

Sin embargo, no es necesario calcular cada activación una a una. La naturaleza de las operaciones permite que sean realizadas mediante **cálculos matriciales**. De esta forma, cada capa quedaría representada por una matriz de pesos, de tantas filas como neuronas haya en dicha capa, y tantas columnas como neuronas haya en la capa anterior, y un vector de sesgos de tantas dimensiones como neuronas haya en dicha capa.

Así, la capa m, con 'N' neuronas, y 'P' neuronas en la capa anterior quedaría representada como:

$$W_m = \begin{bmatrix} {w_{1,1}}^{m} & \cdots & {w_{1,P}}^{m} \\ \vdots & \ddots & \vdots \\ {w_{N,1}}^{m} & \cdots & {w_{N,P}}^{m} \end{bmatrix} \qquad B_m = \begin{bmatrix} {b_1}^{m} \\ \vdots \\ {b_N}^{m} \end{bmatrix}$$

Dando lugar al cálculo del vector de activaciones de la capa 'm+1' según la Ecuación 1.5.

$$A_m = \sigma(\, W_m \times A_{m-1} + \, B_m) \tag{1.5}$$

De esta forma, cada vector de salida de una capa se alimenta al siguiente hasta llegar a la capa de salida. La función no lineal aplicada a la combinación lineal del vector de activación cobra más sentido ahora, ya que, de no ser por ella, una sucesión de multiplicaciones matriciales se vería reducida a una única multiplicación por una nueva matriz, lo que equivaldría a tener una red neuronal de una sola capa. Existen distintas funciones de activación, se nombran y dibujan a continuación las dos más comunes. El uso de las distintas funciones de activación es un **hiperparámetro** que se deberá ajustar en los ensayos, al igual que el número de capas ocultas, o el número de neuronas que conforman cada una de ellas.

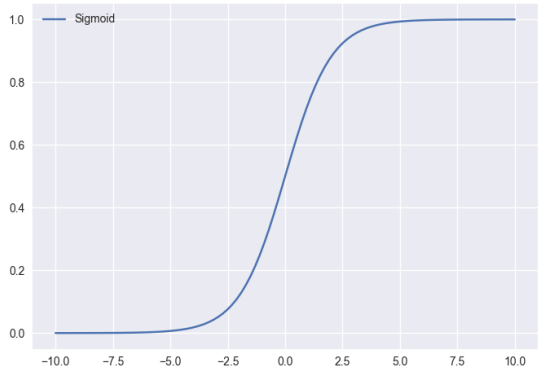
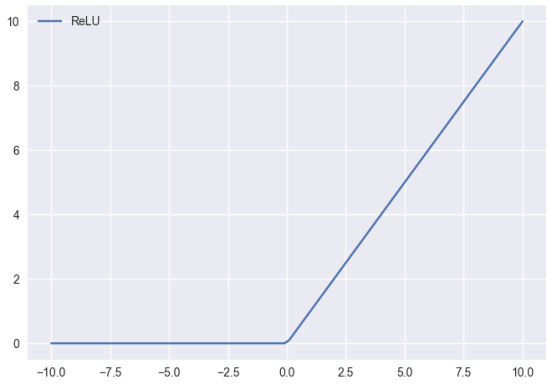
Sigmoide	ReLU (Rectified Linear Unit)
	
$y = \frac{1}{1 + e^{-x}}$	$y = \max(0, x)$

Tabla 2: Principales funciones de activación

Se ha expuesto como trata una red neuronal densa un vector de estímulos de entrada, logrando el paso hacia delante de la información. Sin embargo, al crearla, los pesos y los sesgos de la red son iniciados aleatoriamente, y por lo tanto también lo son las salidas. El aprendizaje de la red reside en lograr encontrar unos pesos (y sesgos, pero a partir de ahora se considerará a ambos como pesos) óptimos para el problema a resolver. Esta optimización es posible gracias al algoritmo de **retropropagación** (o "Backpropagation") del error.

Hasta ahora se ha dicho que la salida de la red corresponde a las acciones disponibles en el entorno. Sin embargo, en realidad corresponde al valor Q esperado para cada acción, y la red escoge aquella acción con un valor Q esperado más alto. La diferencia entre aquel valor esperado y el realmente obtenido (o una función que resemble dicha divergencia), se considera el **error** de la red, y es función de todos aquellos parámetros variables de la red, los pesos. Como cualquier función multivariable, contará con un mínimo (global), y alcanzarlo significa obtener un **error mínimo** entre lo esperado y lo obtenido, y por lo tanto una **red óptima** para el problema a resolver.

En el ámbito del aprendizaje por refuerzo, el nuevo valor Q de salida de la red, que posteriormente se utiliza para calcular los nuevos pesos se calcula según la Ecuación 1.6.

$$\hat{Q}_{(s,a)} = Q_{(s,a)} + \alpha \left[R + \left(\gamma \max Q_{(s',)}\right) - Q_{(s,a)}\right] \qquad (1.6)$$

Donde:

$Q_{(s,a)}$: valor actual

α: ratio de aprendizaje (Learning rate)

R: recompensa obtenida

γ: tasa de descuento (da mayor importancia a recompensas cercanas en el tiempo)

$\max Q_{(s',)}$: valor óptimo esperado

- **Redes Neuronales Convolucionales:**

Hasta ahora se ha visto como una red neuronal es capaz de tratar un vector de entrada y aprender del error entre el valor esperado y el valor real de la salida. Sin embargo, hay arquitecturas de redes neuronales que han mostrado ser más eficaces en el ámbito del tratamiento de imágenes que as redes neuronales densas. como es el caso de las redes neuronales con capas convolucionales.

En este caso, la entrada de la red tiene la forma de una matriz de dimensiones (alto, ancho, número de canales), correspondiendo dicha matriz a un fragmento de la imagen bien con valores de pixel binarios, escalas de grises o color con los tres canales básicos R (rojo), G (verde) o B (azul).

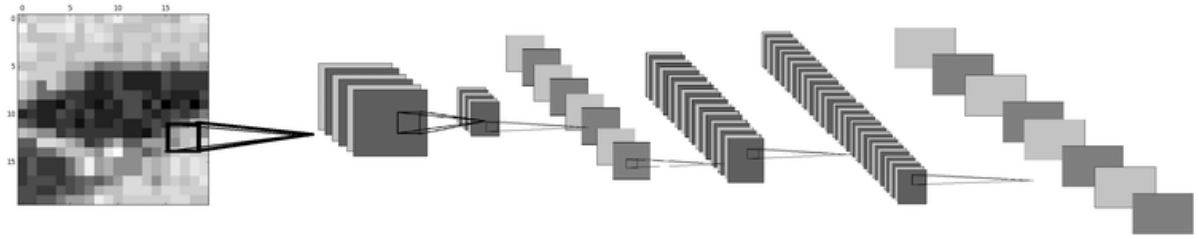


Figura 4: Esquema de red neuronal convolucional

Una capa convolucional consiste en una serie de **filtros**, o matrices de pesos. Cada filtro se superpone con una sección de la imagen y la recorren de izquierda a derecha y de arriba abajo. De esta forma, cada peso se multiplica por cada valor del píxel con el que corresponda en cada instante, y luego se suman los valores resultantes, creando una combinación lineal de la sección en ese instante. El resultado se registra como un nuevo píxel de la imagen de salida de la capa, que contará con tantos canales, como filtros tenga dicha capa.

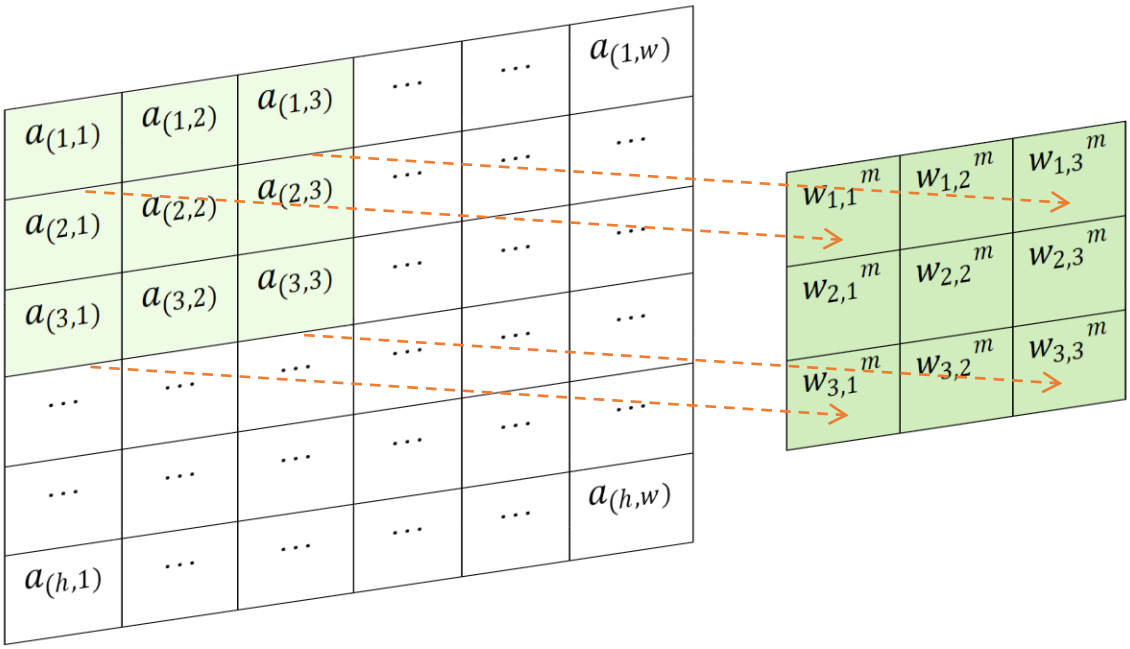


Figura 5: Algoritmo de funcionamiento de red neuronal convolucional

El valor de cada píxel correspondiente a la imagen de salida se calcula como se indica en la Ecuación 1.7.

$$\hat{a}_{(1,1)} = \sum_{j=1}^{3}\sum_{i=1}^{3} {w_{(i,j)}}^{m} \cdot a_{(i,j)} \tag{1.7}$$

Finalmente, tras las capas de convolución, se añade un clasificador formado por capas densas que desemboca en una capa de salida correspondiente a los valores Q de las acciones disponibles en el entorno. De la misma forma que en las redes neuronales completamente densas, existirá una función de error que cuantifica la divergencia entre los valores esperados y los obtenidos. La **minimización** de dicho error con respecto a los pesos de los filtros y del clasificador dará lugar a una **red óptima** para el problema que se desee resolver.

Normalmente, dos o tres capas convolucionales son suficientes para resolver la mayoría de los problemas y sintetizar cualquier imagen. Si se analizan los filtros de la primera capa, bien entrenada, se aprecia como ésta es capaz de detectar **bordes**, mientras que la segunda suele ser capaz de detectar **formas geométricas** más complejas, como **polígonos**.

Para más documentación sobre redes neuronales y la propagación hacia atrás del error se referencia al curso [3] y a los libros [4] y [5].

Normalmente, el algoritmo de "Deep Q-Learning" es capaz de obtener buenos resultados en aplicaciones de aprendizaje por refuerzo. Sin embargo, hay ocasiones en las que el algoritmo de "Backpropagation" queda atrapado en un **mínimo local** de la función, y no es capaz de alcanzar el mínimo global, resultando en una red que no es la óptima para la resolución del problema. En otros casos, al estar actualizando los valores de la red de forma tan frecuente, el aprendizaje se vuelve **inestable** dado que la red intenta ajustarse a demasiados puntos a la vez. Para resolver estos dos últimos problemas, se presenta el algoritmo de "Double Deep Q-Learning".

- **"Double Deep Q-Learning" (DDQN):**

El DDQN introduce dos redes neuronales inicialmente idénticas, es decir, comparten la misma estructura, y son inicializadas con los mismos pesos, aunque de forma aleatoria. La primera de ellas se utiliza para **predecir** la acción de salida en función de la observación obtenida, mientras que es en la otra en la que se aplica el algoritmo de propagación hacia atrás del error, y por lo tanto donde se produce el efecto del **aprendizaje**. Cada cierto intervalo de pasos establecidos, ambas redes se igualan, quedando dos copias de la red que ha aprendido durante ese intervalo, y el proceso vuelve a empezar.

Este método carece de las inestabilidades comentadas en el DQN, ya que la red que toma las decisiones no está cambiando constantemente de política de decisión, y hace más difícil que se quede atascada en mínimos locales, y por lo tanto sea capaz de llegar a una solución óptima.

Para mayor profundización sobre el aprendizaje por refuerzo, se referencia al libro [1], y al curso [2].

1.3 Aplicaciones de los UAV

Un UAV es un **vehículo aéreo no tripulado** originario de la industria aeronáutica – militar. A pesar de su origen inicialmente exclusivo, en la actualidad son utilizados en numerosos campos con el objetivo de facilitar o hacer posibles ciertas tareas:

- Ampliamente utilizados por la policía para tareas de vigilancia y reconocimiento.
- Asistiendo a bomberos con el objetivo de controlar incendios.
- Utilizados por empresas de paquetería, tanto para realizar repartos, como tareas de inventario.
- En control de obras, o instalaciones eléctricas, como herramientas de localización de fallos antes de que puedan resultar fatales.
- En inspección de terreno, para optimizar el cultivo.
- En aplicaciones lúdicas, como drones de carreras.

A pesar de no estar tripulado, un dron necesita de un piloto en tierra con una emisora para volar. Este piloto ha de estar debidamente cualificado para ser capaz de maniobrar con destreza el dron esquivando obstáculos como árboles, edificios, o el mobiliario del interior de un edificio. Sin embargo, esto está comenzando a cambiar, ya que se está apostando por la búsqueda de la autonomía total en los drones, lo que les otorgaría la capacidad de tomar decisiones, tanto a alto nivel (decidir hacia dónde moverse), como a bajo nivel (decidir la alimentación de sus motores), en función de la aplicación para la que estuviesen pensados. De esta forma, el dron quedaría exento de la necesidad de supervisión humana, siendo así capaz de realizar todas las tareas que conformen una misión: despegue, vuelo con evasión de obstáculos, aterrizaje, vuelta a casa en caso de emergencia, y carga de baterías.

En el momento en el que aparece la necesidad de autonomía, y dados los recientes avances en el área del aprendizaje por refuerzo, resulta natural hacer uso de esta herramienta para favorecer el desarrollo de aplicaciones autónomas en vehículos no tripulados. A continuación, se aplica el esquema general del aprendizaje por refuerzo al problema del dron autónomo, manteniendo

las generalidades en cuanto a la variabilidad de los distintos experimentos que podrían llevarse a cabo.

1.4 Esquema del aprendizaje por refuerzo aplicado al dron autónomo

Agente: Al contrario de lo que podría pensarse en primera instancia, el dron en sí mismo no es el agente, si no el algoritmo que subyace bajo las decisiones que son tomadas en cada paso. De los tres algoritmos presentados anteriormente, es común encontrar ensayos que aplican el "DDQN". Podrían obtenerse buenos resultados con el "DQN", sin embargo, el espacio de los estados sería demasiado grande para plantearse utilizar "Q-Learning" en ensayos con aplicaciones reales.

Entorno: El entorno será todo aquello externo al agente que describa el problema a resolver. En una tarea de evasión de obstáculos, el entorno podría vendría por la pose del propio dron (posición y orientación), los obstáculos, etc. En una aplicación de este estilo, el entorno resulta parcialmente observable, a diferencia del entorno que podría representar una partida de ajedrez. En el caso del ajedrez, el agente conoce la posición de todas las piezas en el tablero en todo momento, mientras que, al tratar de representar el estado de un dron, será demasiado complicado albergar toda la información en un espacio vectorial de tamaño razonable.

Acción: todas aquellas posibilidades que tenga el agente de realizar dentro del entorno. Pueden ser acciones a alto nivel, como moverse o girar en alguna dirección, o ir a unas coordenadas en concreto; o a bajo nivel, como controlar las tensiones de alimentación de cada uno de los motores con los que cuente. En el caso de querer entrenar un dron para que sea capaz de volar de forma estable se tendrían en cuenta acciones a bajo nivel, mientras que en el caso de que la ruta del dron sea el problema principal por resolver, las acciones serían tomadas a alto nivel, dejando el problema de la estabilidad en el vuelo a los algoritmos clásicos de control.

Recompensa: es una medida de lo buena que ha resultado la ejecución de una acción para conseguir el objetivo deseado. Con la recompensa se castigan comportamientos indeseados y premiando aquellos deseados. Si quisiésemos que un dron volase de forma estable controlando los PWM (Pulse Witdh Modulation) de sus motores, una posible función de recompensa sería penalizar con puntos negativos si alguno de los dos ángulos de Euler que definen la horizontalidad del dron ("pitch" o "roll") incrementasen su valor por encima de un límite establecido. Es importante que la función de recompensa sea completamente asociable con la información aportada por la observación (expuesta a continuación), es decir, que si no damos al agente ninguna información sobre la distancia que tiene con el suelo, no podemos penalizarle por volar demasiado bajo, por ejemplo, ya que el agente será incapaz de establecer esta relación.

Observación: representación parcial del entorno conformada por las lecturas de aquellos sensores que se haya dotado al dron: sensores de distancia, cámaras de profundidad, LIDAR, IMU, barómetros, etc. A mayor variedad de sensores, más compleja será la representación del entorno. Es preciso encontrar un compromiso entre complejidad y sencillez en función del problema que el agente deba resolver.

1.5 Estado del arte

Existe una gran variedad de experimentos llevados a cabo recientemente en el campo del aprendizaje por refuerzo aplicado a la robótica, y en concreto al problema del dron autónomo. Las principales características que los distinguen son las representaciones de las observaciones, y las acciones que pueden ser llevadas a cabo en el entorno descrito.

El abanico de la representación de observaciones viene descrito por el tipo de sensores utilizados para la detección y representación del entorno, siendo los más comunes: cámaras de profundidad, sensores de distancia por ultrasonidos, sensores de distancia por láser y LIDARs.

En cuanto a las acciones, la principal diferencia viene descrita por el nivel al que el agente es capaz de actuar, como se describe en el apartado de **"Esquema del aprendizaje por refuerzo aplicado al dron autónomo"**.

Entre dichos trabajos se han considerado influencias para el objetivo de este proyecto, dada su relación con el ámbito del aprendizaje por refuerzo en general, o su aplicación al área de investigación del dron autónomo los siguientes:

- En [10] se formaliza un entorno de simulación de un dron basado en la herramienta AirSim de Microsoft, haciendo uso de la librería de Python "OpenAI Gym"

- En [11] se desarrolla una aplicación para la implementación de algoritmos de aprendizaje por refuerzo y aprendizaje por refuerzo profundo, haciendo uso de las lecturas de cámaras de profundidad como observaciones del entorno. Para ello, se hace uso de la herramienta de ROS y Gazebo, y se implementa el código necesario para el desarrollo de los algoritmos en Python.

- En [12] se aplican algoritmos de aprendizaje por refuerzo profundo en un agente capaz de completar los videojuegos de la consola Atari, recibiendo como espacio de observaciones, imágenes de la pantalla.

CAPÍTULO 2: METODOLOGÍA

En este capítulo se describe el flujo de trabajo seguido en este proyecto. En primer lugar, se detallan aquellas tecnologías (software y hardware) empleadas. Posteriormente, se detalla la estructura general de los distintos entornos, su modelado y programación, describiendo las distintas variables a las que está sujeto dicho esquema: el diseño de la recompensa y de las observaciones. Finalmente, se presentan los distintos agentes que se ajustan a dichas observaciones, se discute el mejor resultado en cuanto a recompensa y observación.

2.1 Tecnologías empleadas

En este apartado se exponen aquellas tecnologías utilizadas en el proyecto para llevar a cabo la simulación y el aprendizaje del dron. Para la primera tarea, la de simulación del dron, se ha utilizado el simulador de vuelo AirSim basado en Unreal Engine, mientras que para implementar y desarrollar el algoritmo de aprendizaje por refuerzo se ha hecho uso de las librerías de Python de TensorFlow, Keras-RL y OpenAI Gym.

En la Tabla 3 se muestran las especificaciones de los componentes de software del sistema.

Componente	Versión
Sistema Operativo	Windows 10
Unreal Engine	4.25.4
AirSim	1.2
TensorFlow	2.4.1
Keras-RL	0.4.2
OpenAI Gym	0.18.0

Tabla 3: Especificación de las versiones del software utilizado

AirSim – Unreal Engine

Unreal Engine es el motor gráfico de videojuegos y físicas realistas en el que está basado AirSim, el simulador de vuelo desarrollado por Microsoft. De la amplia oferta de simuladores de vuelo se ha escogido esta configuración por dos principales razones. La primera, por el realismo de las físicas calculadas por Unreal, que, al estar programado con su propia librería de C++, permite una gran agilidad de cálculo. La segunda, por la API con la que cuenta AirSim para comunicarse con un script de Python, ya que es en este lenguaje en el que se implementa el algoritmo de aprendizaje por refuerzo. Esta API permite una comunicación bidireccional entre ambos programas, pudiendo enviar al dron comandos de movimiento, y recibir información de la simulación, como las lecturas de los sensores.

TensorFlow y Keras-RL

TensorFlow es una librería de código abierto para Python desarrollada por Google que contiene las herramientas necesarias para diseñar e implementar redes neuronales profundas en aplicaciones de aprendizaje supervisado de forma directa.

Keras-RL es una capa de software superior a TensorFlow que ayuda a trabajar al programador a un más alto nivel de una manera más amigable y transparente que usando las instrucciones de bajo nivel que ofrece Tensorflow. Estas herramientas permiten implantar aquellos algoritmos necesarios para realizar aplicaciones de aprendizaje por refuerzo profundo, como el "DQN" o "DDQN".

OpenAI Gym

El proyecto publicado por OpenAI permite la formalización de entornos virtuales ajustándose a una plantilla de funciones que describen el comportamiento general de un entorno de aprendizaje por refuerzo. Se describen a continuación las funciones necesarias para ello:

- `init(self)`: inicializa todos los parámetros del entorno. Se ejecuta una sola vez al generar el entorno como una variable del script. Devuelve el parámetro de una observación inicial del entorno.
- `step(self, action)`: describe el comportamiento de cada paso en el entorno (no incluye el comportamiento del agente) en función de cada acción disponible. Devuelve como parámetros una nueva observación del entorno tras realizar la acción especificada, la recompensa que ha supuesto realizar dicha acción, y una variable booleana que especifica si ha finalizado el episodio.
- `reset(self)`: describe las instrucciones que se llevan a cabo cuando finaliza un episodio para devolver el entorno a su estado inicial.

Estas tres funciones conforman la estructura principal de un entorno descrito en formato Gym. Sin embargo, es común añadir las siguientes funciones que serán llamadas desde la función "step", con el objetivo de aportar limpieza al código:

- `get_observation(self)`: trata la información del entorno y la sintetiza en la observación que devuelve.
- `get_reward(self)`: analiza el estado actual del entorno y devuelve la recompensa (escalar) que le corresponda.
- `get_done(self)`: estudia si el episodio ha finalizado, ya sea porque se ha cumplido el objetivo, el agente ha cometido un error que resulta terminal, o se ha alcanzado un límite de tiempo del episodio, y devuelve una variable booleana con el resultado.

Este esquema general es aplicable a la mayoría de los problemas de aprendizaje por refuerzo.

Más adelante, en el apartado de **"Programación de los entornos"**, se expone como se ha aplicado este esquema general al problema concreto del dron autónomo en particular, describiendo en detalle cada una de las funciones del entorno.

Hardware

En la Tabla 4 se presentan las prestaciones del ordenador que se ha utilizado para la implementación y el entrenamiento de los agentes.

CPU	Intel Core i7 (8 núcleos)
GPU (Tarjeta gráfica)	Nvidia GTX 970
RAM	16 GB

Tabla 4: Especificaciones del hardware utilizado

2.2 Descripción del sistema

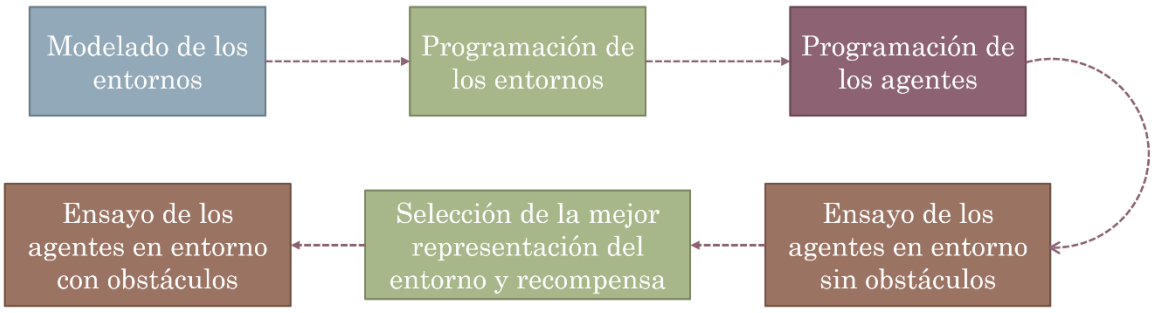


Figura 6: Flujo de trabajo empleado

En respuesta al esquema general del aprendizaje por refuerzo, se ha creado cada uno de los elementos que lo conforman, ajustándose al problema presentado, con las herramientas que se han descrito previamente. Se describe a continuación el flujo de trabajo de la Figura 6.

1. Se han descrito distintos entornos simplificados (sin obstáculos), variando la estructura de sus observaciones y sus funciones de recompensa, con el objetivo de encontrar aquella observación y función de recompensa que permitan un mejor aprendizaje del agente.
2. Se han programado los distintos agentes que mejor se ajustan a las estructuras de observaciones presentadas por los entornos, y se han evaluado los resultados, manteniendo al agente y al entorno con mejores resultados de aprendizaje.
3. Se ha expuesto a dicho agente a tres entornos complejos (con presencia de obstáculos), y se ha evaluado su aprendizaje en cada uno de ellos.

2.2.1 Modelado de los entornos

El modelado de los entornos se ha llevado a cabo íntegramente en Unreal Engine. Se ha diseñado una estructura de aspa con tres secciones que convergen en el centro, como se presenta en la Figura 7. Cada sección está diseñada con obstáculos que representan tres situaciones de interés para la aplicación de un dron autónomo: un bosque, una ciudad, y una región montañosa. Se consigue dicho efecto variando el tamaño de los obstáculos presentes en cada sección, sin embargo, dicha dimensión es constante a lo largo de toda la altura de la estructura, ya que el

dron será capaz de moverse en un plano paralelo al plano horizontal, manteniendo su altura constante.

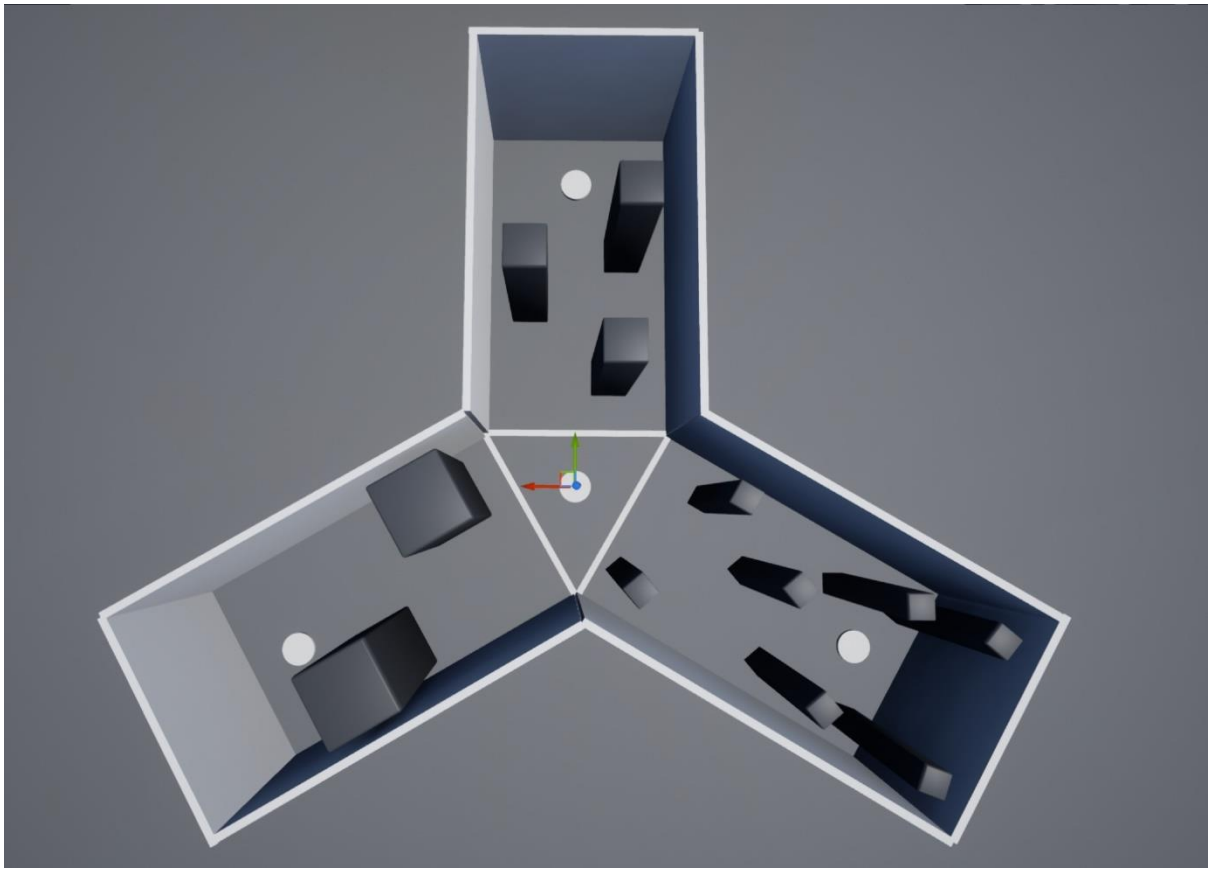


Figura 7: Planta del entorno completo con obstáculos

2.2.1.1 Entorno “Forest”

Cuenta con obstáculos de bajo grosor que tratan de simular la presencia de árboles.

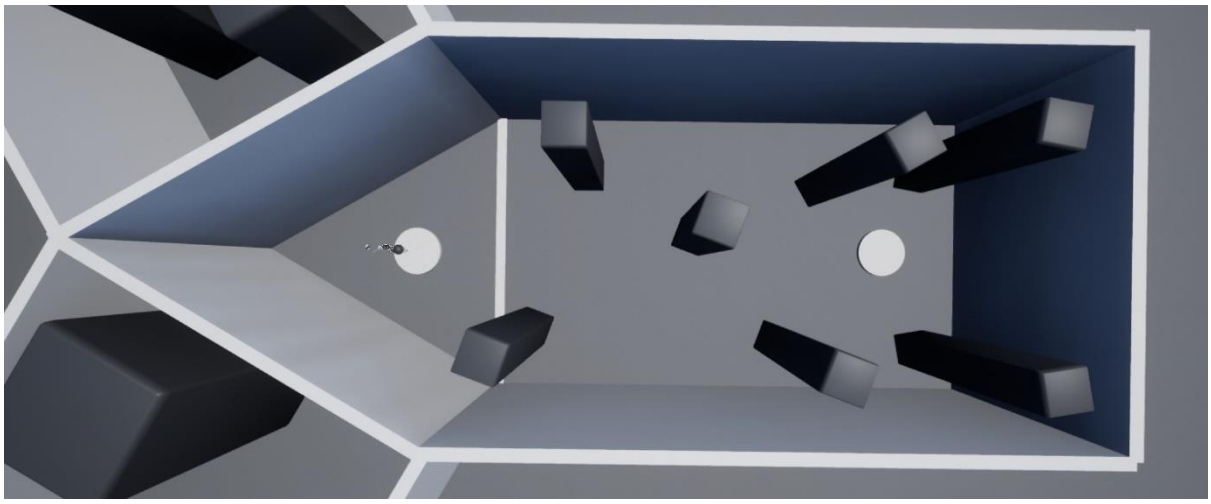


Figura 8: Planta del entorno "Forest"

2.2.1.2 Entorno “City”

Presenta obstáculos de tamaño mediano simulando la presencia de edificios.

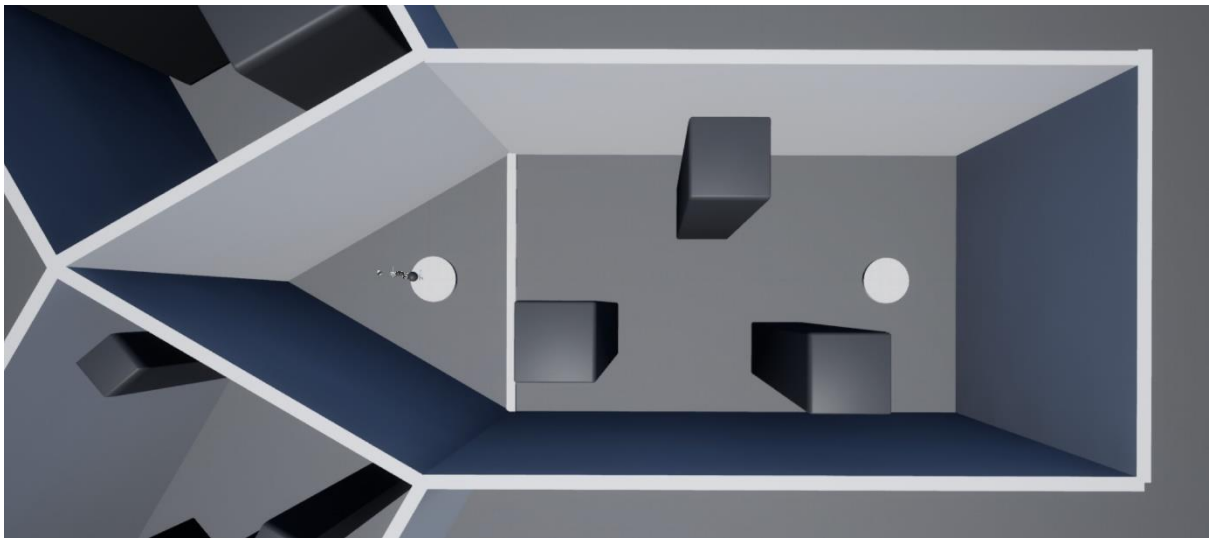


Figura 9: Planta del entorno "City"

2.2.1.3 Entorno “Mountains”

Recoge los obstáculos de mayor tamaño recordando a generaciones naturales grandes como montañas

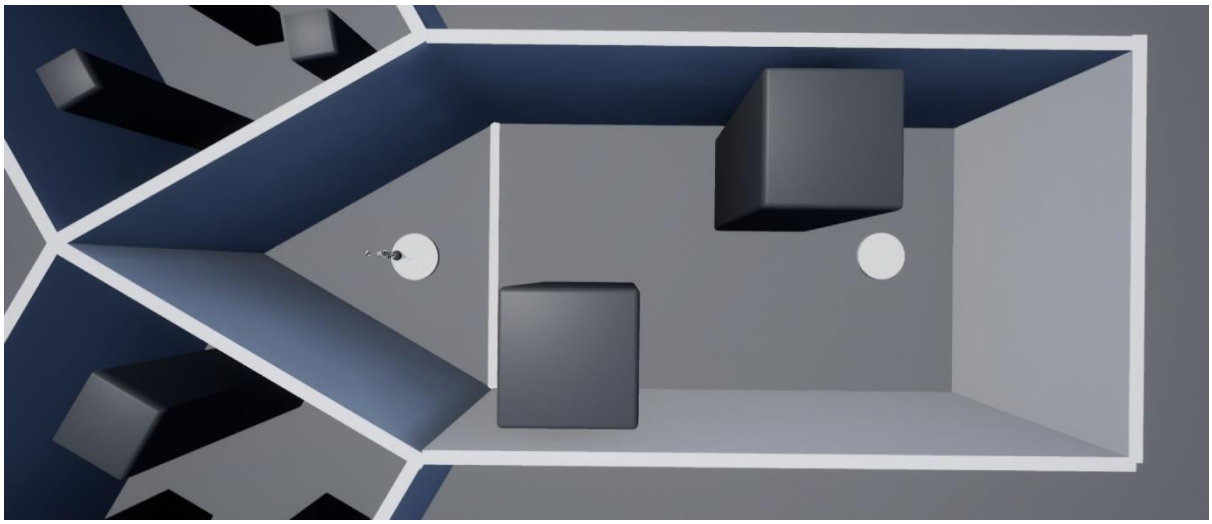


Figura 10: Planta del entorno "Mountains"

Con el objetivo de buscar aquella estructura de observación y función de recompensa más adecuadas sin alargar los tiempos de simulación, el entorno se ha limpiado de obstáculos y se han posicionado cuatro puntos de aparición y objetivo en la estructura, como se muestra en la Figura 11. Dichos puntos serán aleatorizados al principio de cada episodio, generando veinte rutas distintas por las que se resolvería el problema. Una vez encontradas la observación y recompensa adecuadas, se presenta el agente al aprendizaje en los entornos de "Forest", "City", y "Mountains".

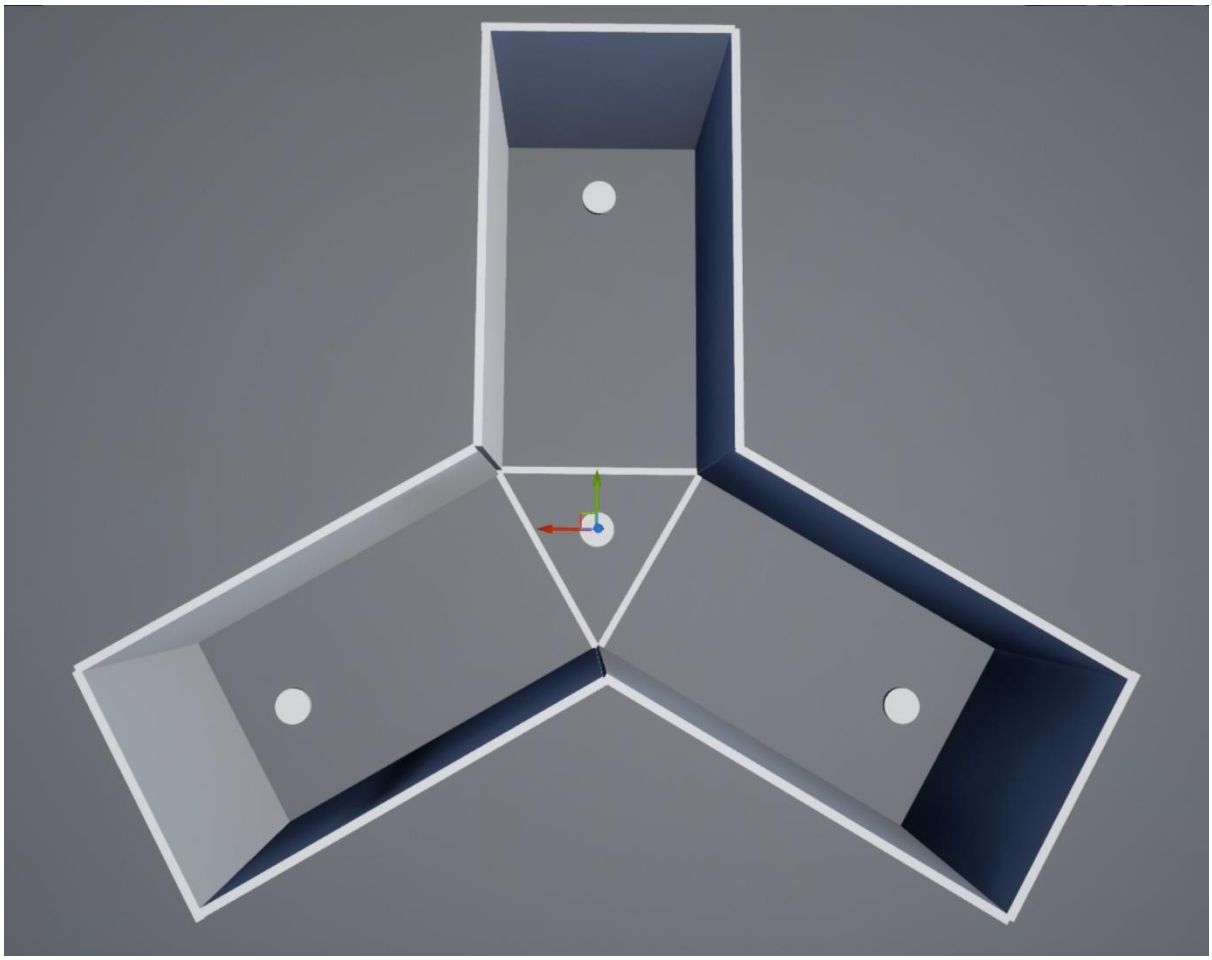


Figura 11: Planta del entorno completo sin obstáculos

2.2.2 Programación de los entornos

Como se menciona en el apartado de **"Descripción de las tecnologías"**, para el objetivo de la formalización del entorno de forma programática, de manera que un agente sea capaz de interactuar con él, se ha utilizado la herramienta Gym. El producto de la empresa OpenAI presenta una librería con las funciones que resemblan un entorno general aplicado al campo del aprendizaje por refuerzo. De esta manera, únicamente es necesario modificar aquellas funciones que definen al entorno personalizado, y que describen por lo tanto las acciones llevadas a cabo al inicializar el entorno, al llevar a cabo un paso en el entorno en función de una acción realizada, y al reiniciar el entorno cuando finaliza un episodio.

Los distintos entornos programados quedan entonces representados por las funciones de “init”, “step” y “reset”. Dado que la función de “step” debe devolver la nueva observación, la recompensa, e indicar si el episodio ha finalizado, se añaden tres funciones adicionales que serán llamadas desde “step” para obtener estos valores: “get_observation”, “get_reward” y “get_done”. Será entonces en las funciones de “get_observation” y “get_reward” que quedarán recogidas las principales diferencias entre los distintos entornos, por lo tanto, se exponen a continuación las explicaciones de los dos entornos utilizados, seguidos de los ficheros necesarios para definirlos.

Entornos definidos según sus observaciones y sus acciones

- **“AirGym – Sensors":**

  Se definen sus observaciones como un vector de 32 sensores de distancia con un alcance entre 0m y 40m, que varían su ángulo de guiñada con respecto al sistema de referencia del dron, y distribuidos uniformemente alrededor del mismo. Queda por lo tanto descrita la observación con 11,25º de resolución entre cada sensor, como se indica en la Figura 12.

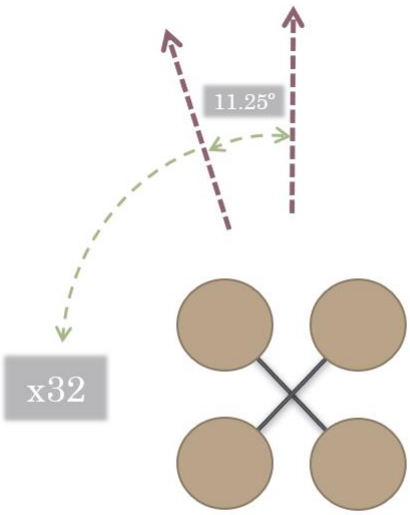


Figura 12: Disposición de los sensores de distancia en el entorno "AiGym - Sensors"

A este vector de distancias se le añade al final el ángulo entre el eje x del dron y el objetivo como indica la Figura 13.

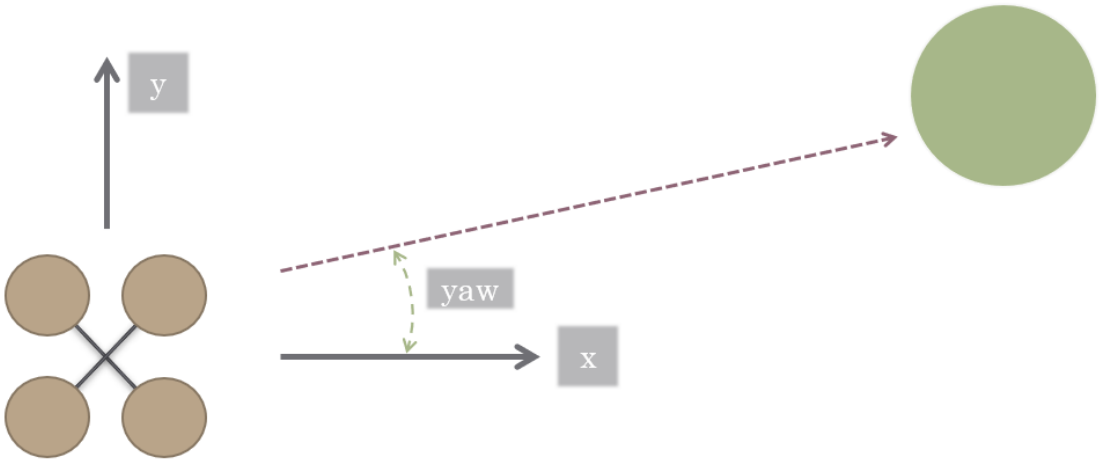


Figura 13: Sistema de referencia angular en el entorno "AirGym - Sensors"

Las acciones disponibles en este entorno serán las de moverse en los cuatro sentidos solidarios a los ejes del dron (siempre en un movimiento paralelo al plano horizontal), a saber: delante, atrás, izquierda y derecha.

- **AirGym – Camera:**

  Recoge las observaciones mediante una cámara de profundidad. Esta cámara devuelve una imagen cuyos píxeles varían su valor entre 0 y 1, según lo lejos o lo cerca que esté el objeto que resemblan. Cuentan con un alcance máximo de 120 m, y un alcance mínimo de 0 m. Se describe gráficamente el comportamiento de la cámara en la Figura 14.

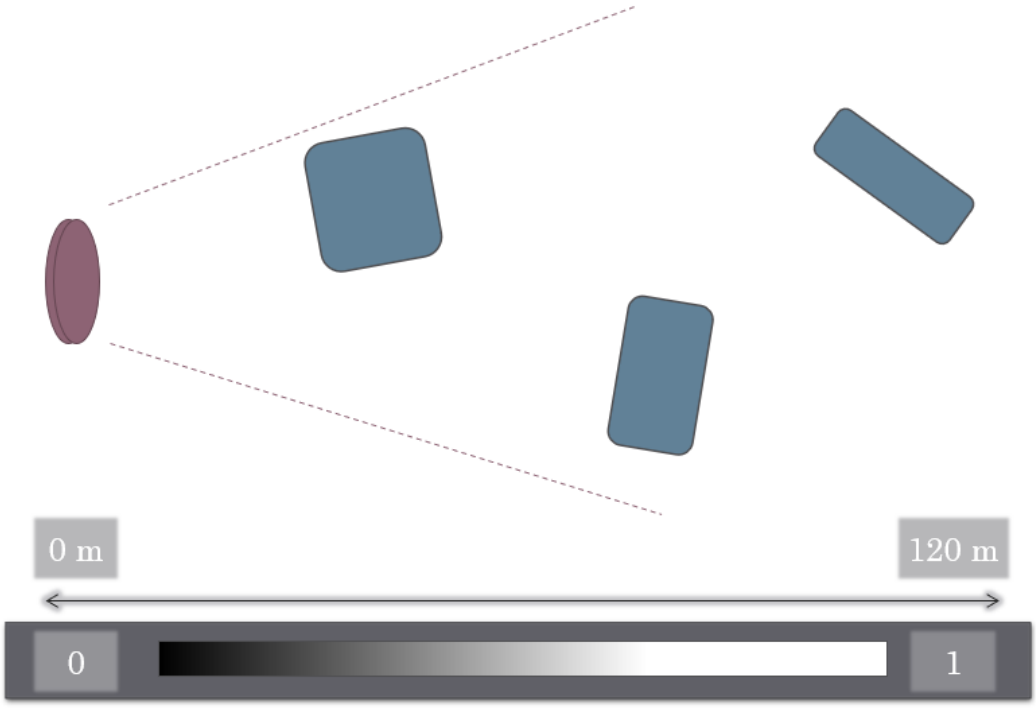


Figura 14: Descripción del funcionamiento de la cámara de profundidad en el entorno "AirGym - Camera"

Dado que la observación es una imagen, la información del ángulo al que se encuentra el objetivo se ha introducido en una sección de la imagen de profundidad que funciona a modo de brújula. Esta sección es completamente blanca, a excepción de una franja que indica la dirección del objetivo, como se indica en la Figura 15.

Figura 15: Tratamiento del ángulo con el objetivo en el entorno "AirGym - Camera"

De esta forma, una posible observación completa de este entorno puede apreciarse en la Figura 16.

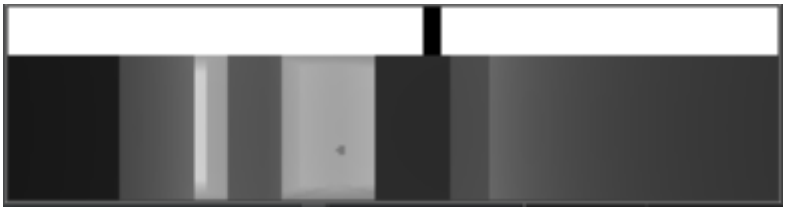


Figura 16: Observación completa del entorno "AirGym - Camera"

Al estar la cámara de profundidad del dron apuntando en la dirección positiva de su eje x, y contar con un campo de visión de 90º, contar con las mismas acciones del entorno basado en sensores no resultaría prudente, ya que podría moverse hacia atrás o hacia los lados sin ver realmente los obstáculos que pueda encontrarse en esa dirección. Por ello, las acciones disponibles en este entorno son: avanzar hacia delante, rotar hacia la derecha y rotar hacia la izquierda. De esta forma, el dron podrá avanzar únicamente en la dirección hacia la que apunta su cámara de profundidad.

Para definir cada uno de los entornos son necesarios dos ficheros: el primero, un fichero "settings.json" que describe los sensores con los que cuenta el dron; el segundo, el fichero "airgym.py" que describe el comportamiento del entorno en cada caso. Los ficheros correspondientes a ambos entornos se encuentran en el **APENDICE I**, mientras que los ficheros descriptivos de los sensores lo hacen en el **APÉNDICE IV**.

Entornos definidos según sus funciones de recompensa

Hasta ahora se han expuesto los distintos entornos en función de sus observaciones y de sus distintas acciones. A continuación, se exponen las dos funciones de recompensa que se han valorado. Ambas han sido probadas en el entorno con observaciones basadas en sensores de distancia "AirGym – Sensors", ya que la eficacia de la recompensa debería resultar independiente a la observación, siempre que las observaciones contengan la misma información (distancias y ángulo con el objetivo).

- **Recompensa – "Delta":**

```
def get_reward(self):

    …

    # Collision
    if self.has_collided:
        return -1000

    # Distance reduction
    else:
        return 10 * (self.previous_distance - self.current_distance)
```

Si el dron colisiona con un obstáculo, el agente pierde 1000 puntos.

El agente recibe un valor proporcional al incremento de distancia con el objetivo en cada paso. En caso de alejarse recibe una recompensa negativa, mientras que, en el caso de acercarse, la recompensa será positiva.

- **Recompensa – "Dist":**

```
def get_reward(self):

    …

    # Collision
    if self.has_collided:
        return -1000

    # Distance
    else:
        return reward = -0.01 * self.current_distance
```

Si el dron colisiona con un obstáculo, el agente pierde 1000 puntos.

El agente recibe un valor proporcional a la distancia actual con el objetivo (cambiada de signo) en cada paso. El objetivo será que trate de minimizar dicha distancia para minimizar el castigo. Dados los parámetros utilizados en las cinemáticas del dron (movimientos de 10m/s durante 0.1s), la recompensa estará comprendida en el intervalo [-10, 10].

Una vez establecidas las cuatro posibles variaciones, se procedió a ensayar distintos agentes en los distintos entornos, en el escenario compuesto por el aspa completa, pero sin obstáculos, descrito anteriormente. A continuación, se exponen aquellos agentes que han obtenido los mejores resultados.

2.2.3 Programación de los agentes

Dado que los dos entornos sujetos a estudio que se presentan contienen observaciones que recogen la información del entorno en estructuras distintas, será necesario crear dos agentes que se adapten a la estructura de datos que se presente, a saber: información de los sensores de forma vectorial, e imágenes de profundidad con estructura matricial.

Para cada agente se han de fijar una serie de hiperparámetros que son los siguientes:

- Descripción general y justificación de la estructura.
- **Épsilon** (variable)**:** factor de compromiso entre exploración y explotación. Este parámetro se hace variar linealmente a lo largo del entrenamiento, de tal forma que al principio predomina la exploración del entorno, mientras que al final lo hace la explotación del mismo. Queda definido, por lo tanto, por sus valores inicial y final.
- **Gamma:** factor de descuento para recompensas futuras.
- **Ratio de aprendizaje** (Learning rate)

- **Número de pasos** del aprendizaje

Para más detalle de la estructura de cada uno de los agentes, se referencia al **APÉNDICE II**, donde se encuentran los ficheros de Python que recogen la programación de los éstos.

- **Agente basado en estructura de red neuronal densa**

  Como se explicó en la introducción, para el tratamiento de información en forma vectorial, la estructura más simple de red neuronal que se presenta es una estructura de capas densas completamente conectadas. Este agente será utilizado en el entorno "AirGym – Sensors". En la Figura 17 se presenta un esquema que especifica el número de capas y sus propiedades más generales, mientras que en la Tabla 5 se encuentran los principales parámetros anteriormente mencionados.

Parámetro	Valor
Épsilon inicial	1.0
Épsilon final	0.1
Gamma	0.99
Ratio de aprendizaje	2.5e-4
Número de pasos	500.000

Tabla 5: Parámetros de aprendizaje del agente basado en estructura de red neuronal densa

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 33)                0
_________________________________________________________________
dense (Dense)                (None, 256)               8704
_________________________________________________________________
dense_1 (Dense)              (None, 256)               65792
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 1028
=================================================================
Total params: 75,524
Trainable params: 75,524
Non-trainable params: 0
_________________________________________________________________
```

Figura 17: Descripción del modelo de red neuronal densa

- **Agente basado en estructura de red neuronal convolucional**

En vista a las propiedades del segundo entorno de "AirGym - Camera", que presenta las observaciones en estructura matricial, se hace uso de la estructura de capas más apropiadas para el análisis de imagen, las capas convolucionales. En la Figura 18 se especifican los rasgos generales de la red escogida, basándose en trabajos anteriores de análisis de imagen

para la resolución de juegos de la consola Atari, desarrollada por OpenAI, con ligeras adaptaciones para el problema propuesto. En la Tabla 6 se encuentran los principales parámetros del modelo utilizado.

Parámetro	Valor
Épsilon inicial	1.0
Épsilon final	0.1
Gamma	0.99
Ratio de aprendizaje	2.5e-4
Número de pasos	1.000.000

Tabla 6: Parámetros de aprendizaje del agente basado en estructura de red neuronal convolucional

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 30, 126)       832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 32, 15, 63)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 7, 31)         18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 64, 3, 15)         0
_________________________________________________________________
flatten (Flatten)            (None, 2880)              0
_________________________________________________________________
dense (Dense)                (None, 256)               737536
_________________________________________________________________
dense_1 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 645
=================================================================
Total params: 790,405
Trainable params: 790,405
Non-trainable params: 0
_________________________________________________________________
```

Figura 18: Descripción del modelo de red neuronal convolucional

Una vez entrenados los agentes correspondientes a cada una de las observaciones propuestas (“Sensors” y “Camera”) con la recompensa “delta”, y una vez entrenado el agente “Sensors” con ambas recompensas (“delta” y dist”), se escoge aquella observación y función de recompensa que respondan a un mejor proceso de aprendizaje. Dicha observación y función de recompensa serán las utilizadas en los ensayos con obstáculos en los entornos de “Forest”, “City” y “Mountains”. Se exponen los resultados de ambas variables en los apartados 3.1 y 3.2 del **“CAPÍTULO 3”**.

Para el entrenamiento en cada uno de los escenarios propuestos con obstáculos, se ha dividido el proceso en dos partes, la primera con el objetivo de ahondar en la exploración del entorno, mientras que la segunda con el de explorarlo. Posteriormente, en los resultados, se ha apreciado que dividir el entrenamiento de esta forma puede desestabilizar el aprendizaje obteniendo en ocasiones peores recompensas al finalizar la segunda parte que al haber finalizado la primera, por lo que dependerá de este factor que el modelo final escogido para cada entorno sea el posterior a la parte I o la parte II. Quedan definidas cada una de las partes del entrenamiento mediante la épsilon inicial y final, y el número de pasos en la Tabla 7.

Parámetro	Parte I	Parte II
Épsilon inicial	1.0	0.1
Épsilon final	0.1	0.01
Número de pasos	1.000.000	200.000

Tabla 7: Descripción del entrenamiento seccionado

2.2.4 Vinculación del entorno y el agente

Los archivos que definen el entorno y el agente son independientes y por si mismos no cuentan con una comunicación entre ellos. Por ello, se han desarrollado tres ficheros con la función de vincularlos en las distintas situaciones que puedan ser necesarios:

- **Fichero “train.py”:**

  Recoge los parámetros necesarios para el entrenamiento e inicializa un nuevo agente de forma aleatoria que entrena durante el número de pasos indicado. Guarda el agente en un fichero donde quedan registrados su estructura y la última actualización de sus pesos, y genera un breve informe del entrenamiento donde quedan recogidas las recompensas.

- **Fichero “load.py”:**

  Permite cargar un agente ya entrenado con los parámetros de entrenamiento que se especifiquen, durante el número de pasos indicado. Guarda el nuevo agente en un fichero donde quedan registrados su estructura y la última actualización de sus pesos, y genera un breve informe del entrenamiento donde quedan recogidas las recompensas.

- **Fichero “test.py”:**

  Permite cargar un agente ya entrenado y probar la política que éste haya inferido del entorno. En este caso no se realiza ningún aprendizaje, es decir, los pesos de la red neuronal en la que se basa el agente permanecen congelados. Además, al no realizar el aprendizaje, el factor épsilon (que indica la presencia de acciones aleatorias con el objetivo de la exploración del entorno) es nulo, por lo que las acciones que realiza el agente responden en todo momento a la política estimada en el aprendizaje.

Pueden encontrarse estos tres ficheros en el **APÉNDICE III**.

CAPÍTULO 3: ANÁLISIS DE RESULTADOS

En este capítulo se exponen los resultados obtenidos para los distintos entrenamientos llevados a cabo. En primer lugar, se discuten las distintas funciones de recompensa sujetas a estudio, seguidas de las distintas estructuras de observaciones escogidas. Para aquella recompensa y observación que hayan concluido en unos mejores resultados, se exponen las gráficas descritas por las siguientes variables en los escenarios de "Forest", "City" y "Mountains":

- **Épsilon:** variación del factor de exploración – explotación a lo largo del entrenamiento.
- **Recompensa media** (por paso)**:** evolución de la recompensa media recibida por el agente en cada paso. El aprendizaje queda reflejado por una tendencia ascendente en la gráfica, con un valor máximo igual a la máxima recompensa obtenible en el entorno.
- **"Loss":** Representa la diferencia entre la recompensa que el agente espera recibir en cada paso (estimación), y lo que realmente recibe (realidad). El aprendizaje queda reflejado por una tendencia descendente en la gráfica.
- **Precisión:** El porcentaje de episodios en los que el agente logra llegar al objetivo sin colisionar, sobre el total de episodios. El aprendizaje queda reflejado por una tendencia ascendente en la gráfica, con un valor máximo del 100%. Junto con esta gráfica, se dibuja (en línea discontinua) el número de episodios totales y el número de episodios en los que el agente cumple el objetivo.

Cada una de las gráficas toma valores para cada intervalo de 10.000 pasos, es decir, cada vez que el modelo de red neuronal sobre el que se toma las decisiones de las acciones realizadas se iguala al modelo de red neuronal sobre el cual se aplica el aprendizaje, según describe el algoritmo de "Double Deep Q-Learning".

1.1 Evaluación de las recompensas

Se presenta, para cada una de las funciones de recompensa propuestas ("Delta" y "Dist"), la gráfica descrita por la precisión del agente a lo largo del entrenamiento. Se ha escogido esta variable como comparativa de ambas recompensas ya que, junto con la recompensa media obtenida, permite distinguir el progreso a lo largo del aprendizaje. Sin embargo, no resulta de interés comparar las recompensas medias por cada paso recibidas por el agente, ya que en este caso la recompensa es totalmente distinta.

1.1.1 Recompensa – “Delta”

Si el dron colisiona con un obstáculo, el agente pierde 1000 puntos.

El agente recibe un valor proporcional al incremento de distancia con el objetivo en cada paso. En caso de alejarse recibe una recompensa negativa, mientras que, en el caso de acercarse, la recompensa será positiva.

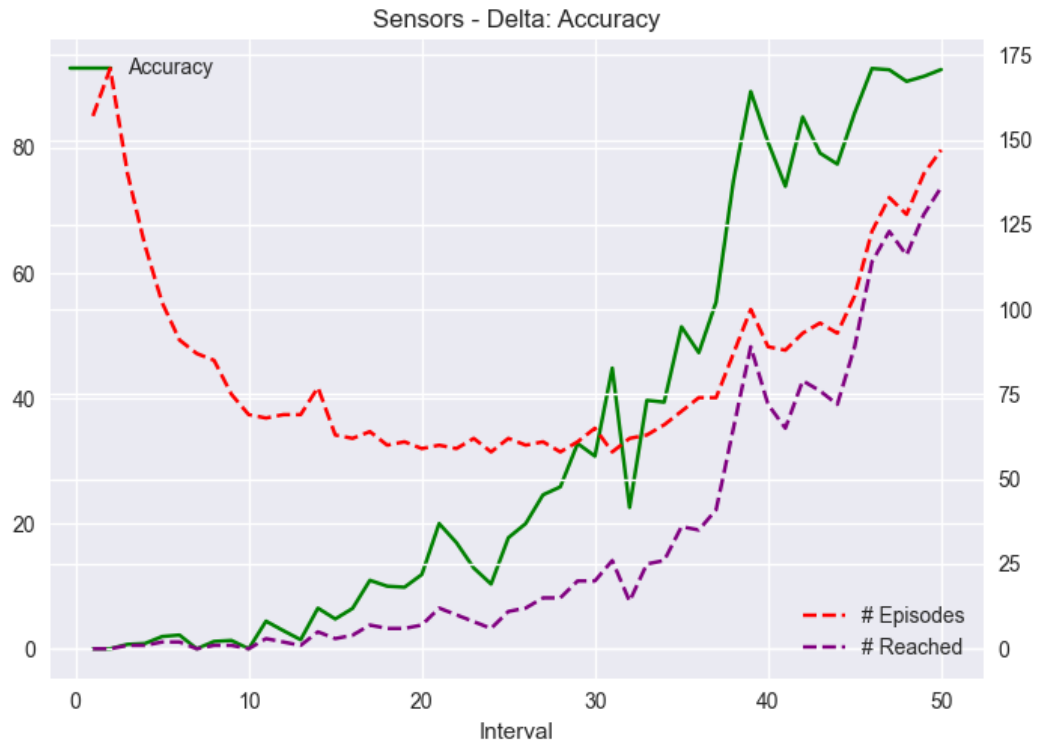


Figura 19: Precisión de la función de recompensa "Delta"

Se aprecia un aprendizaje favorable en el desarrollo ascendente de la precisión, llegando a un valor final de 92.5%.

En cuanto al comportamiento del número total de episodios en cada intervalo, cabe mencionar que al comienzo del entrenamiento resulta en un valor alto, debido tanto a la aleatoriedad de las acciones realizadas, como al desconocimiento del entorno por parte del agente, que ocasionan colisiones frecuentes. A la mitad del aprendizaje, el agente comienza a evitar colisionar, lo que reduce el número de episodios, que finalmente vuelve a aumentar cuando el agente optimiza la calidad de sus acciones en base a la recompensa recibida.

Se puede inferir, por lo tanto, que la función de recompensa “Delta” representa correctamente el problema propuesto.

1.1.2 Recompensa – “Dist”

Si el dron colisiona con un obstáculo, el agente pierde 1000 puntos.

El agente recibe un valor proporcional a la distancia actual con el objetivo (cambiada de signo) en cada paso. El objetivo será que trate de minimizar dicha distancia para minimizar el castigo.

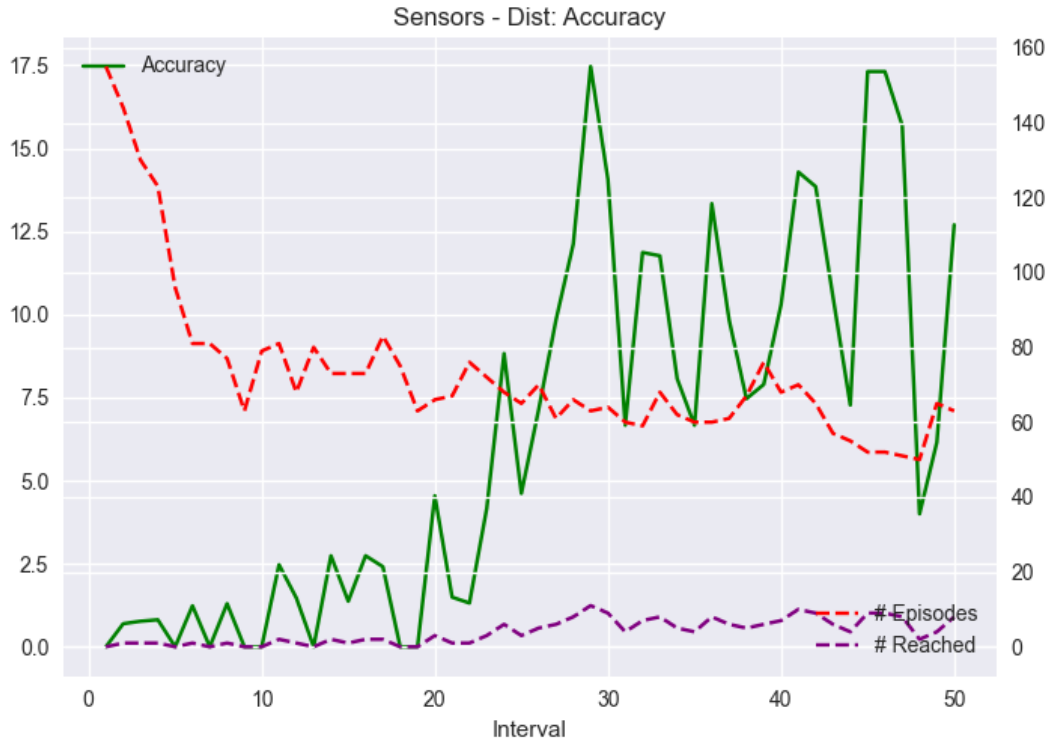


Figura 20: Precisión de la función de recompensa "Dist"

Se aprecia un aprendizaje inestable en la función de precisión a lo largo del entrenamiento, a pesar de una ligera tendencia ascendente a lo largo del mismo, alcanzando un valor máximo de 17.5%, y un valor final alrededor de 12.5%.

Se concluye que la función de recompensa “Dist” no representa correctamente el problema propuesto, por lo que no contribuye a un aprendizaje favorable por parte del agente.

En vista del resultado aportado por la función de recompensa “Delta” frente al aportado por “Dist”, se realizan el resto de los ensayos manteniendo la primera y descartando la segunda como función de recompensa representativa del problema.

1.2 Evaluación de observaciones

Para cada uno de los entornos propuestos con distintas estructuras de observaciones, y distintos espacios de acciones (y ensayados con distintas estructuras de agentes), se presentan las gráficas descritas por las variables introducidas al inicio del capítulo.

1.2.1 Entorno "AirGym – Sensors"

- **Observaciones:** vector definido por las lecturas de 32 sensores de distancias junto con el ángulo de guiñada entre el eje x del sistema de referencia del dron con la recta que lo une con el objetivo.
- **Acciones:** moverse hacia delante, atrás, izquierda y derecha.

- **Épsilon:**

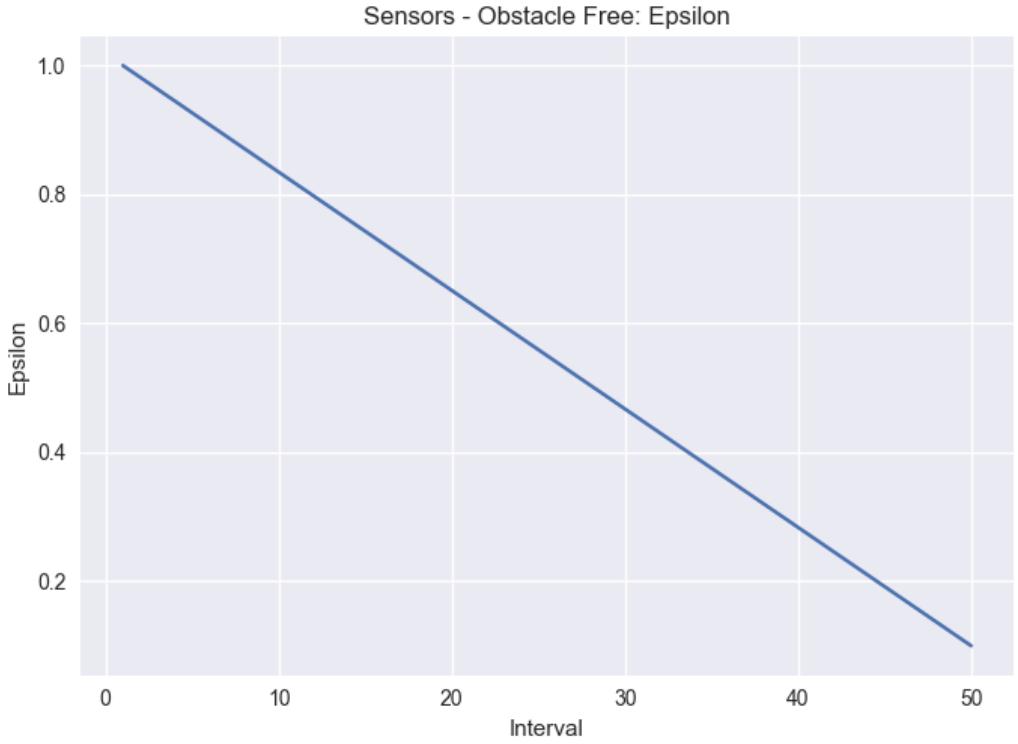


Figura 21: "AirGym - Sensors": Épsilon

Se expone el desarrollo del factor épsilon con el objetivo de tener en cuenta el porcentaje aleatorio de acciones que realiza el agente en cada intervalo del entrenamiento.

- **Recompensa media** (por paso)**:**

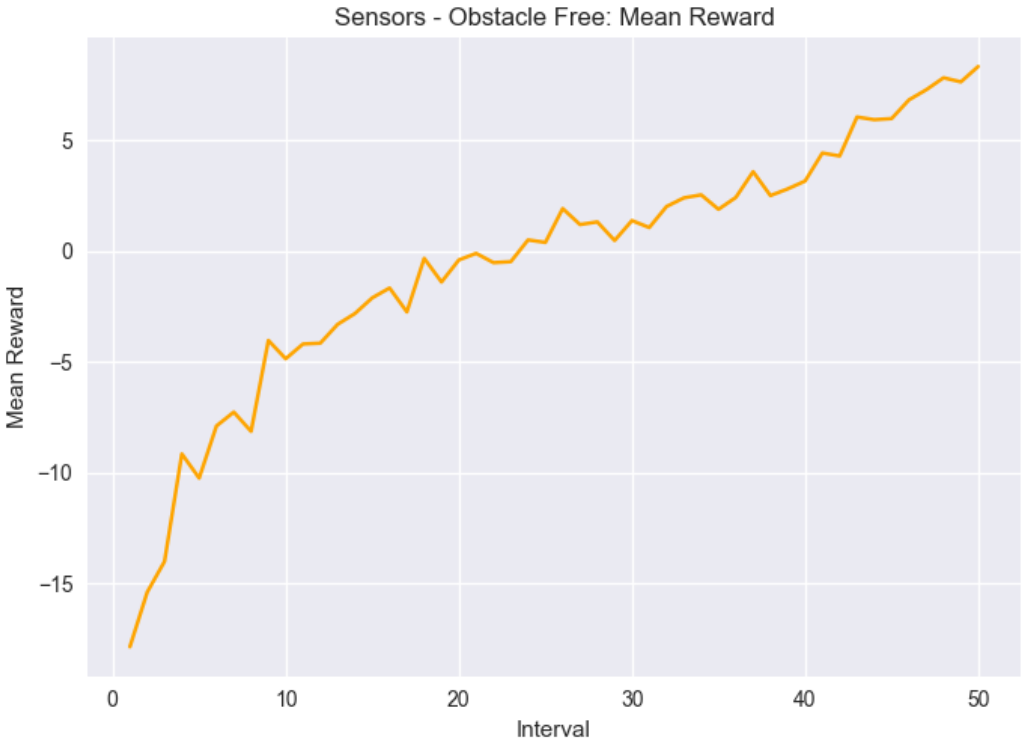


Figura 22: "AirGym - Sensors": Recompensa media

Se reconoce un correcto aprendizaje en la tendencia ascendente de la recompensa media obtenida por paso por parte del agente, que llega a tener un valor final y máximo de 8,321.

La derivada de la gráfica representada no llega a ser nula, lo que indica que podría seguir creciendo, y que los valores de los últimos episodios no se han estabilizado. Sin embargo, éste se trata de un entrenamiento orientativo con el objetivo de encontrar la mejor estructura de observación, por lo que una visión global del aprendizaje resulta suficiente.

- **“Loss”:**

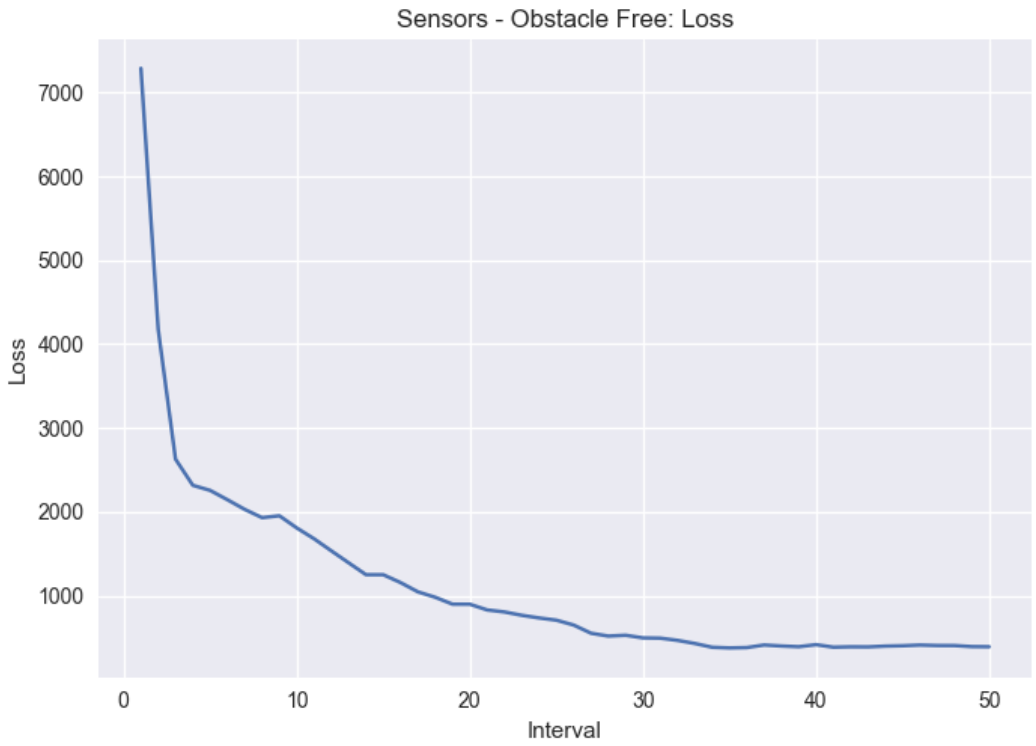


Figura 23: "AirGym - Sensors": "Loss"

Junto con la recompensa media obtenida, la “loss” refleja un correcto aprendizaje, con una tendencia descendente y disminuyendo su valor rápidamente al comienzo del episodio.

- **Precisión**

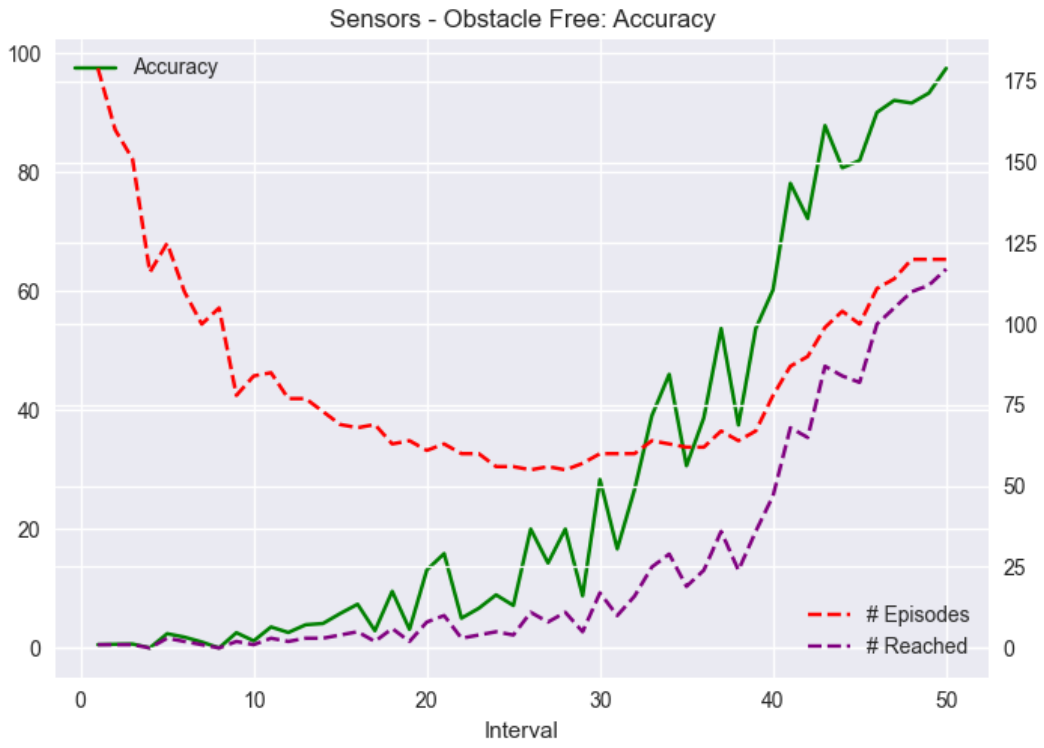


Figura 24: "AirGym - Sensors": Precisión

Finalmente, la precisión también refleja un aprendizaje favorable, con la tendencia ascendente esperada, y alcanzando un valor máximo de 97.5%.

Se vuelve a observar el mismo comportamiento seguido por el número total de episodios por intervalo que en la recompensa "Delta", que refleja un gran número de colisiones al principio, frente a un gran número de alcances del objetivo al final.

- **Conclusión:**

De las variables analizadas se infiere que, en un entorno sin abundancia de obstáculos, la representación del estado en base a los sensores de distancia permite un correcto aprendizaje del problema a resolver.

1.2.2 Entorno “AirGym – Camera”

- **Observaciones:** imagen de profundidad con una franja que representa el ángulo de guiñada entre el eje x del sistema de referencia del dron con la recta que lo une con el objetivo.
- **Acciones:** moverse hacia delante, variar el ángulo de guiñada hacia la derecha, y variar el ángulo de guiñada hacia la izquierda.

- **Épsilon:**

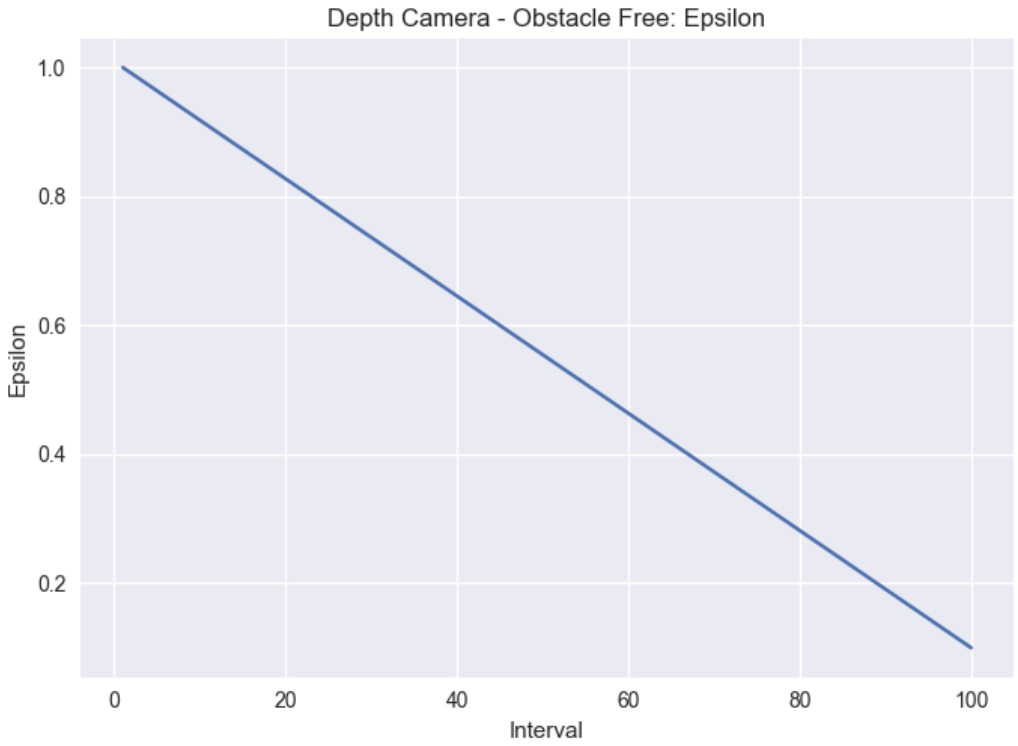


Figura 25: "AirGym - Camera": Épsilon

Se expone el desarrollo del factor épsilon con el objetivo de tener en cuenta el porcentaje aleatorio de acciones que realiza el agente en cada intervalo del entrenamiento.

- **Recompensa media** (por paso)**:**

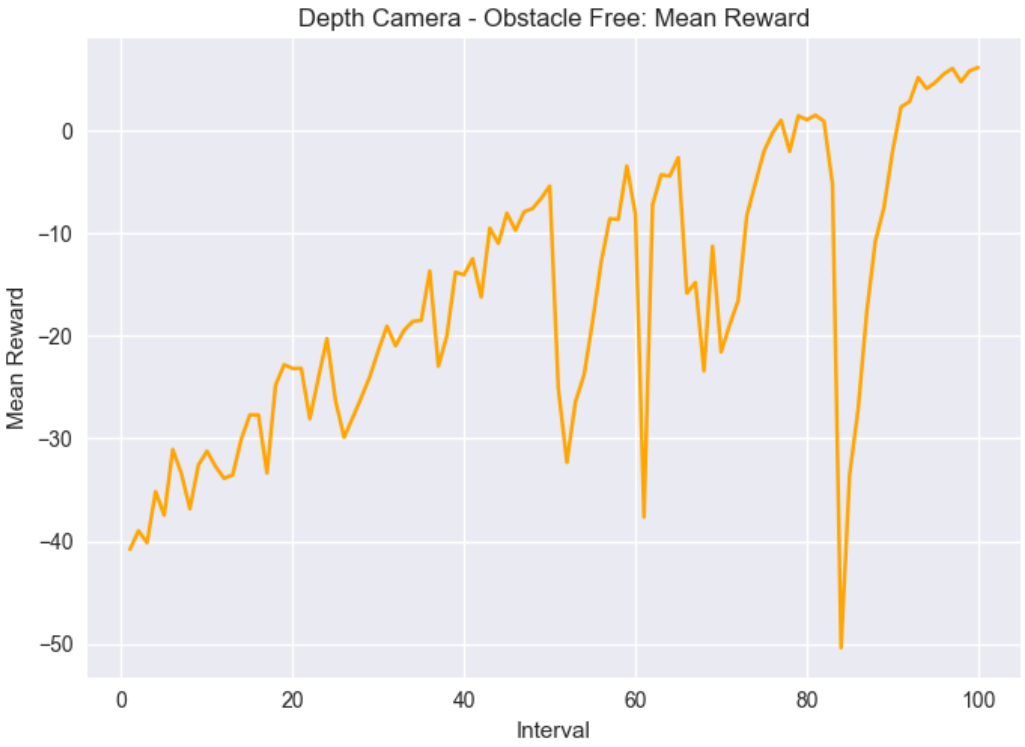


Figura 26: "AirGym - Camera": Recompensa media

Se aprecia un aprendizaje inestable en el desarrollo de la recompensa media obtenida. A pesar de su tendencia ascendente, se localizan ciertos intervalos en los que hay picos de recompensas extremadamente bajas. No se ve reflejado un aprendizaje favorable.

2. **“Loss”:**

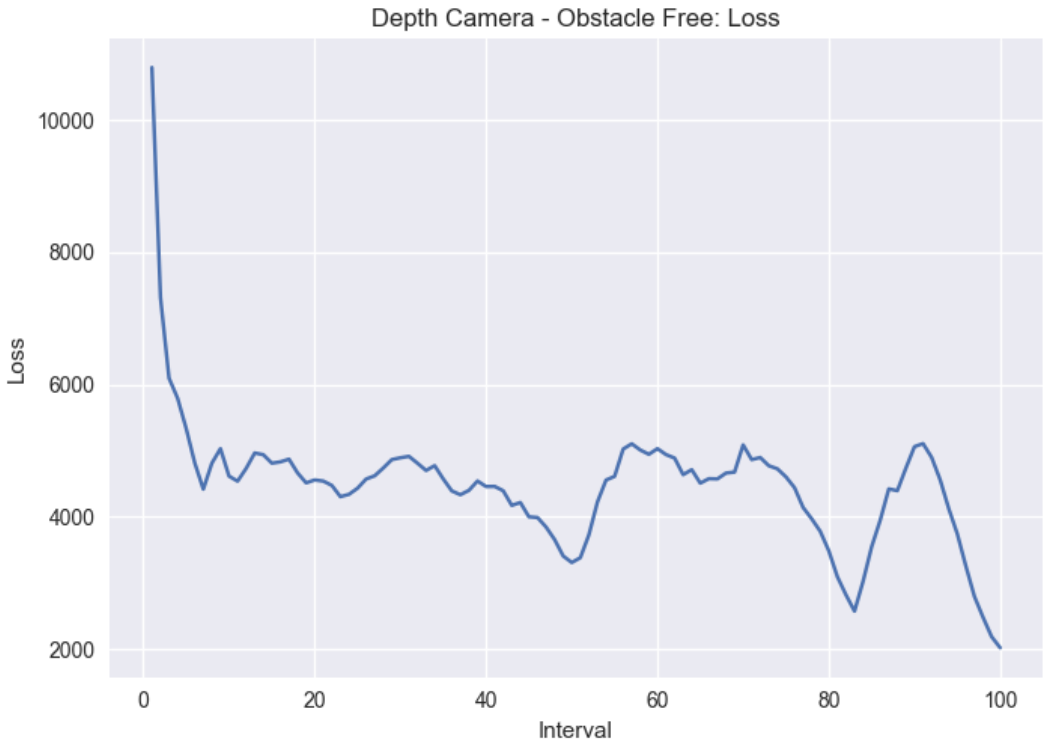


Figura 27: "AirGym - Camera": "Loss"

De la misma forma que en la recompensa media obtenida, la “loss” cuenta con demasiadas oscilaciones, lo cual no refleja un aprendizaje estable, a pesar de su tendencia descendente.

- **Precisión:**

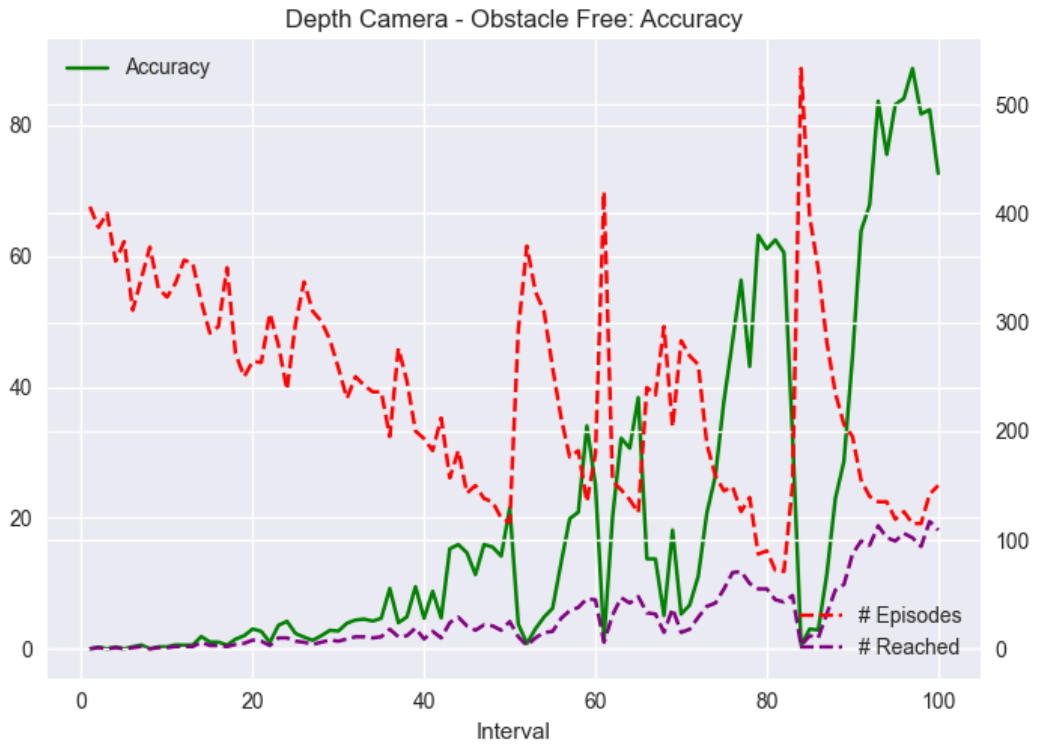


Figura 28: "AirGym - Camera": Precisión

Finalmente, la precisión tampoco avala un aprendizaje correcto por parte del agente, contando con gran cantidad de ruido, a pesar de una tendencia ascendente, y un valor final de 72,66%.

- **Conclusión:**

En base a las variables analizadas, se puede concluir que la observación escogida en el entorno de "AirGym – Camera" no representa correctamente el estado, y por lo tanto no contribuye a un correcto aprendizaje del entorno. Es probable que esto se deba a que la representación del ángulo de guiñada con el objetivo, que se trata de un escalar, se ha representado de forma gráfica, y el agente tiene que realizar un sobreesfuerzo en interpretar su significado. Una posible solución sería contar con una red neuronal mixta que tratase la imagen de profundidad a la entrada, y el escalar del ángulo de guiñada se añadiese una vez tratada la imagen.

En vista de los resultados favorables obtenidos en el entorno "AirGym – Sensors" frente a las inestabilidades ocasionadas en el entorno "AirGym – Camera", se procede a realizar el resto de los entrenamientos en presencia de obstáculos con la primera representación del estado, basada en sensores de distancia, y a entrenar al agente basado en la red neuronal densa completamente conectada.

3.3 Resultados en el entorno “Forest”

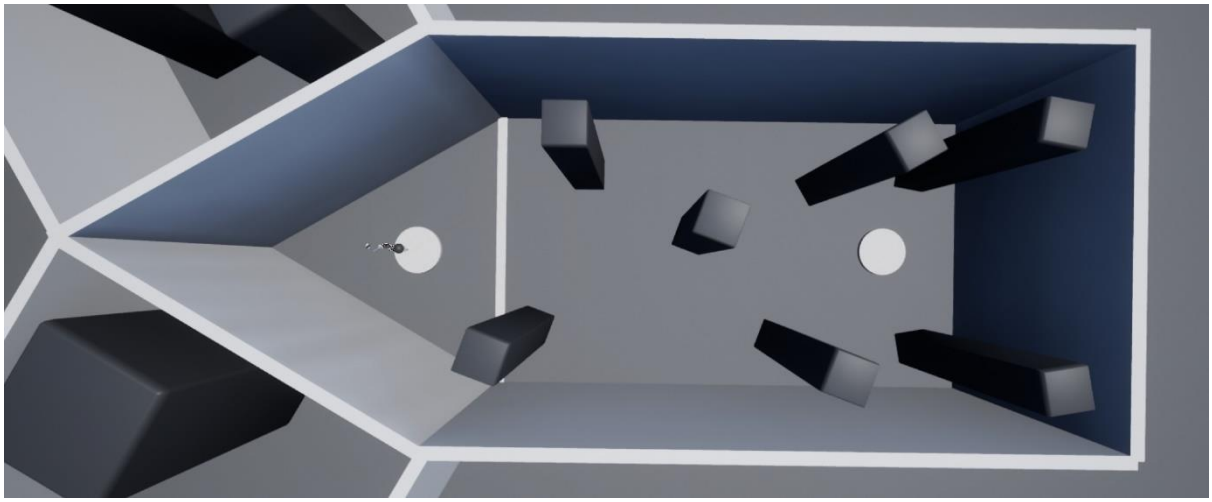


Figura 29: Planta del entorno "Forest"

El agente ha respondido satisfactoriamente al entrenamiento propuesto en ambas partes, por lo que se exponen los resultados del entrenamiento completo.

- **Épsilon:**

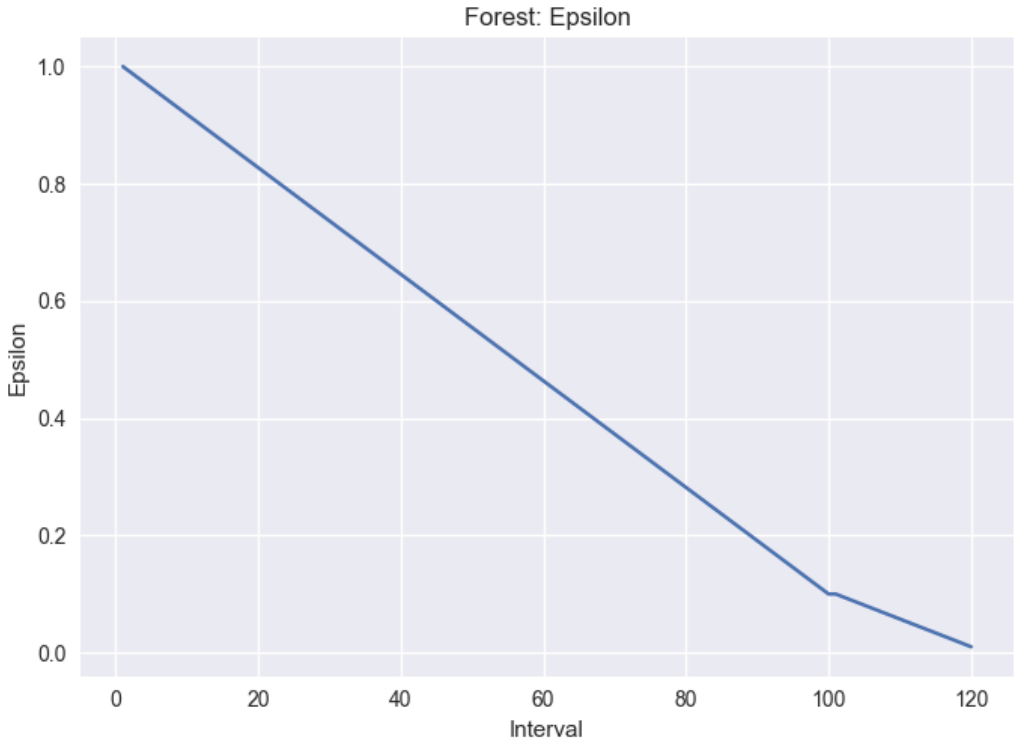


Figura 30: Entrenamiento en "Forest": Épsilon

El cambio de pendiente en el factor de exploración describe la frontera entre la primera parte del entrenamiento en la que predomina la exploración mediante acciones aleatorias, y la segunda, en la que predomina la explotación de la política aprendida.

- **Recompensa media** (por paso)**:**

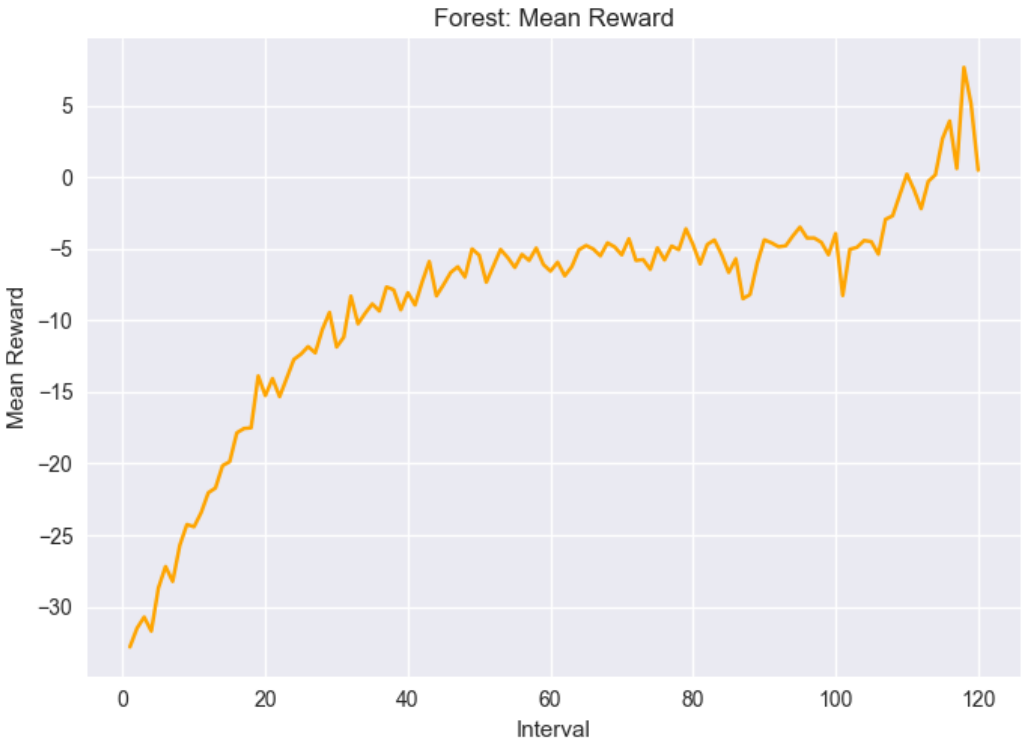


Figura 31: Entrenamiento en "Forest": Recompensa media

Se aprecia un aprendizaje en la tendencia al alza de la recompensa media por paso, la cual llega a alcanzar valores positivos. Se considera favorable el efecto de la segunda parte del entrenamiento dado que la recompensa sigue creciendo a partir del intervalo 100.

- **“Loss”:**

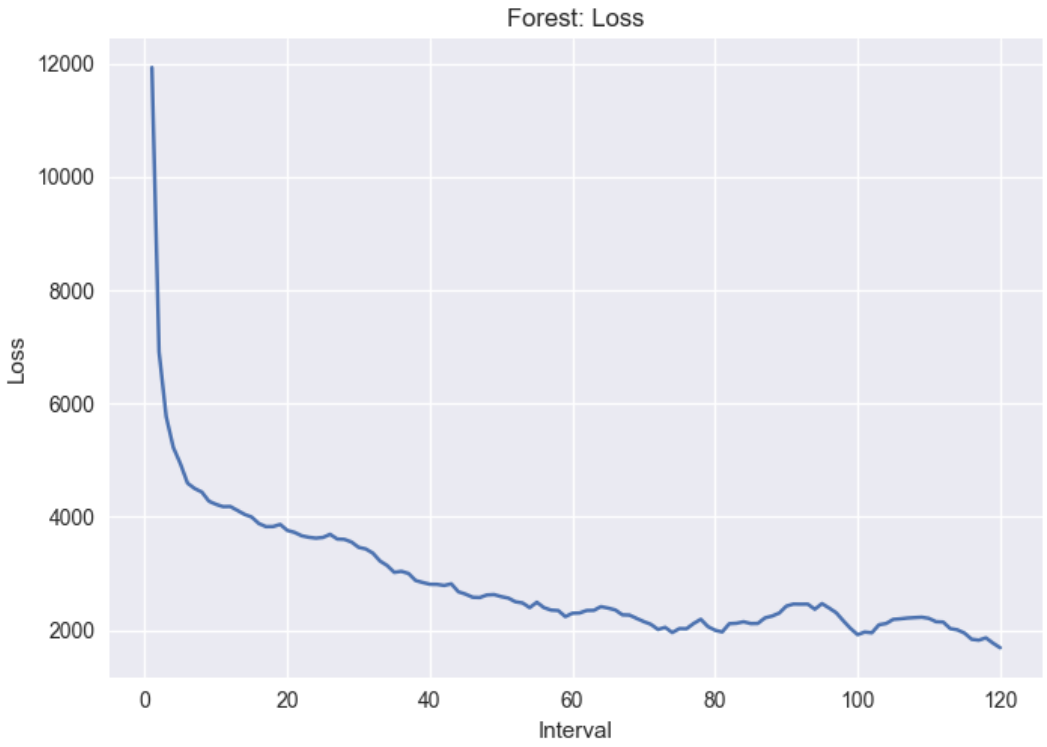


Figura 32: Entrenamiento en "Forest": "Loss"

La “loss” también sugiere un aprendizaje estable con ligeras oscilaciones en el cambio de entrenamiento, que finalmente se estabilizan.

- **Precisión:**

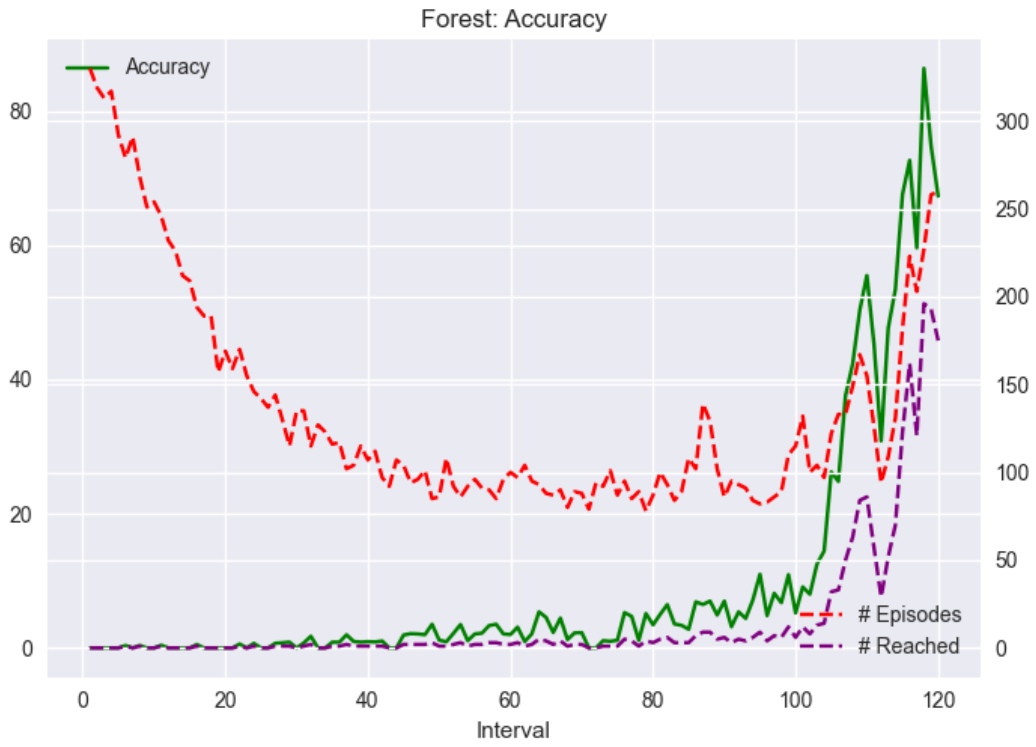


Figura 33: Entrenamiento en "Forest": Precisión

Finalmente, se puede confirmar que el agente ha contado con un correcto aprendizaje en vista de los valores finales alcanzados por la precisión. Se finaliza el entrenamiento con un valor de precisión de 67,31%, aunque este valor no indica el verdadero comportamiento final del agente, el cual está sujeto a realizar acciones aleatorias al no ser nulo el factor de exploración. Posteriormente, en el apartado de "Comparativa de los agentes" se estudia el comportamiento de estos sin presencia de acciones aleatorias.

- **Conclusión:**

Las tres variables sujetas a estudio a lo largo del ensayo confirman que el agente entrenado en el entorno "Forest" ha aprendido satisfactoriamente a esquivar obstáculos y a alcanzar el objetivo indicado.

3.4 Resultados en el entorno “City”

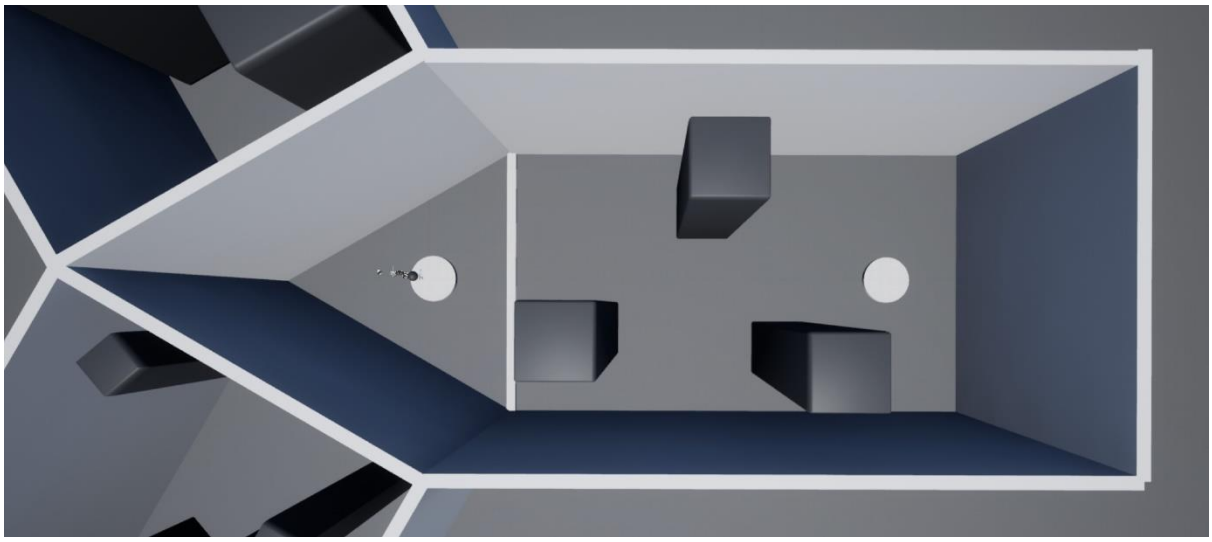


Figura 34: Planta del entorno "City"

El agente no ha respondido satisfactoriamente al entrenamiento propuesto en ambas partes, por lo que se exponen los resultados obtenidos tras la Parte I.

- **Épsilon:**

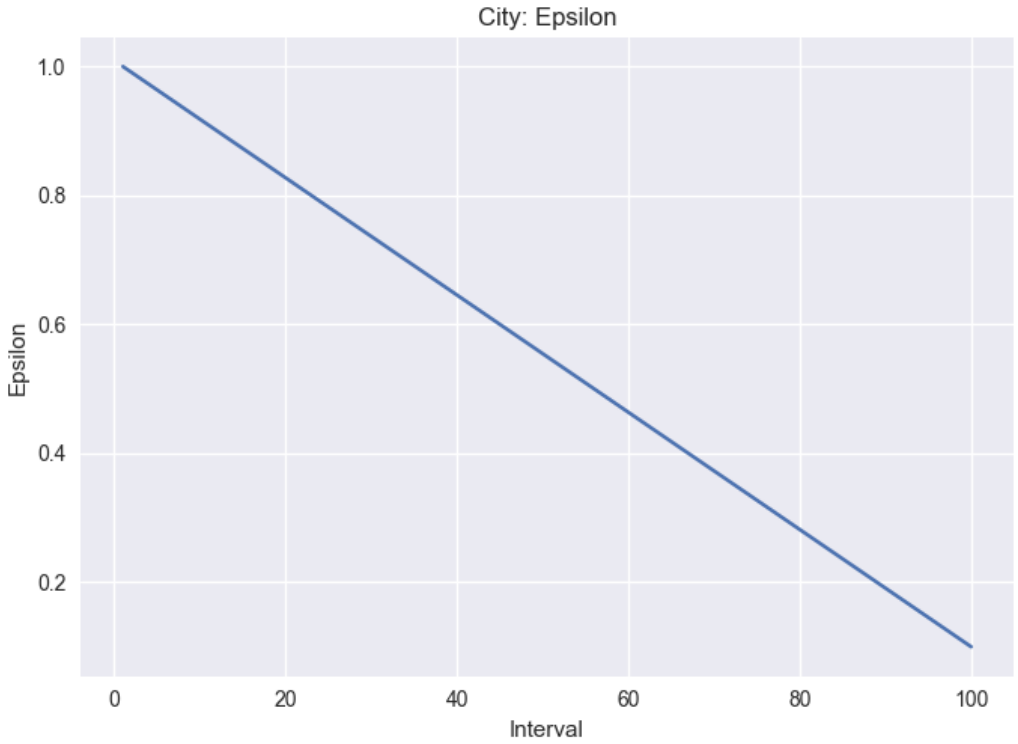


Figura 35: Entrenamiento en "City": Épsilon

Se aprecia en el desarrollo de la épsilon a lo largo de la Parte I del entrenamiento.

- **Recompensa media** (por paso)**:**

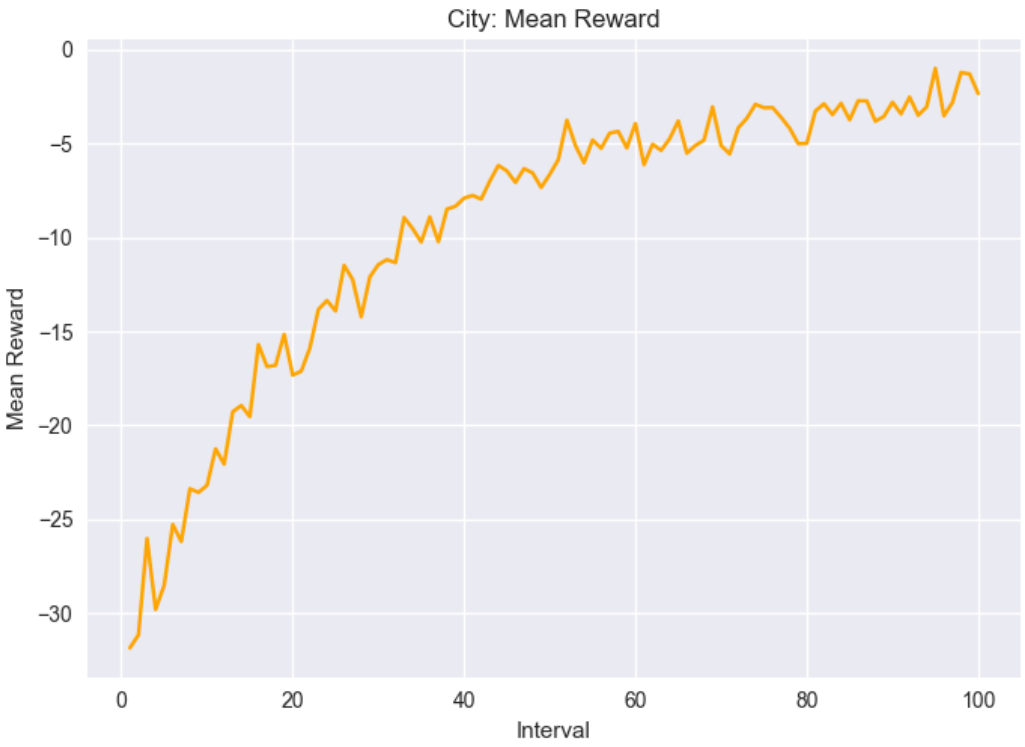


Figura 36: Entrenamiento en "City": Recompensa media

Se aprecia un aprendizaje en la tendencia al alza de la recompensa media por paso, sin embargo, esta no llega a alcanzar valores positivos, lo que indica que no está compensando todo lo que podría el efecto negativo de las colisiones con un avance predominante hacia delante.

- **“Loss”:**

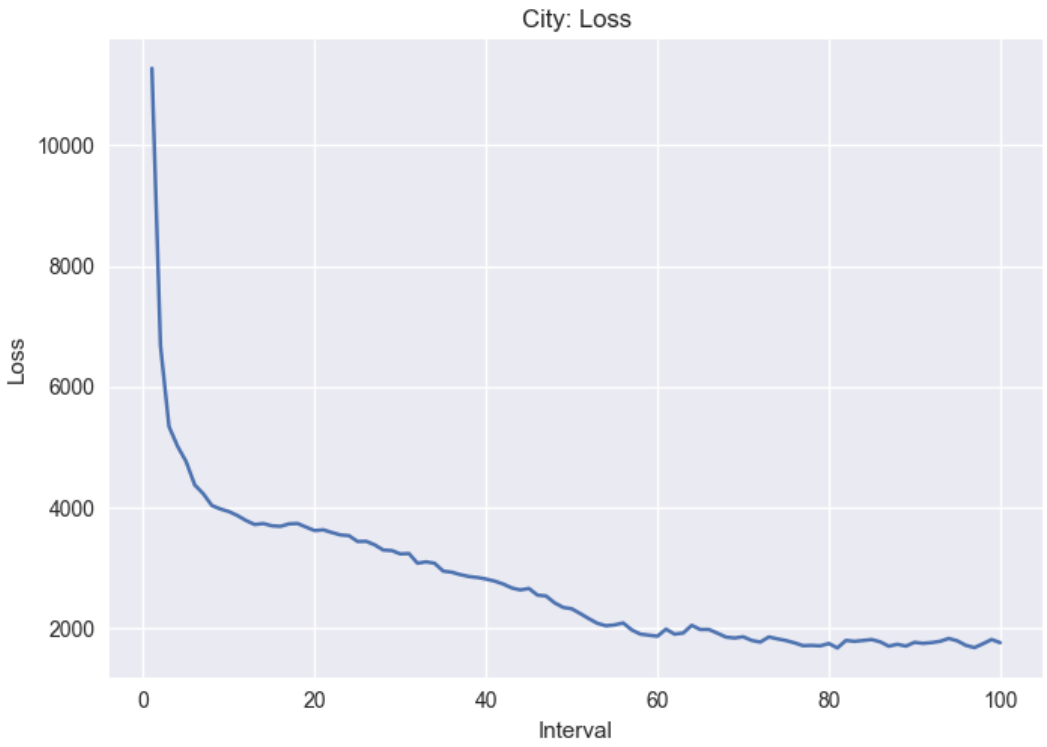


Figura 37: Entrenamiento en "City": "Loss"

La “loss” sugiere un aprendizaje estable, disminuyendo su valor drásticamente al comienzo del ensayo, y manteniendo su valor bajo.

- **Precisión:**

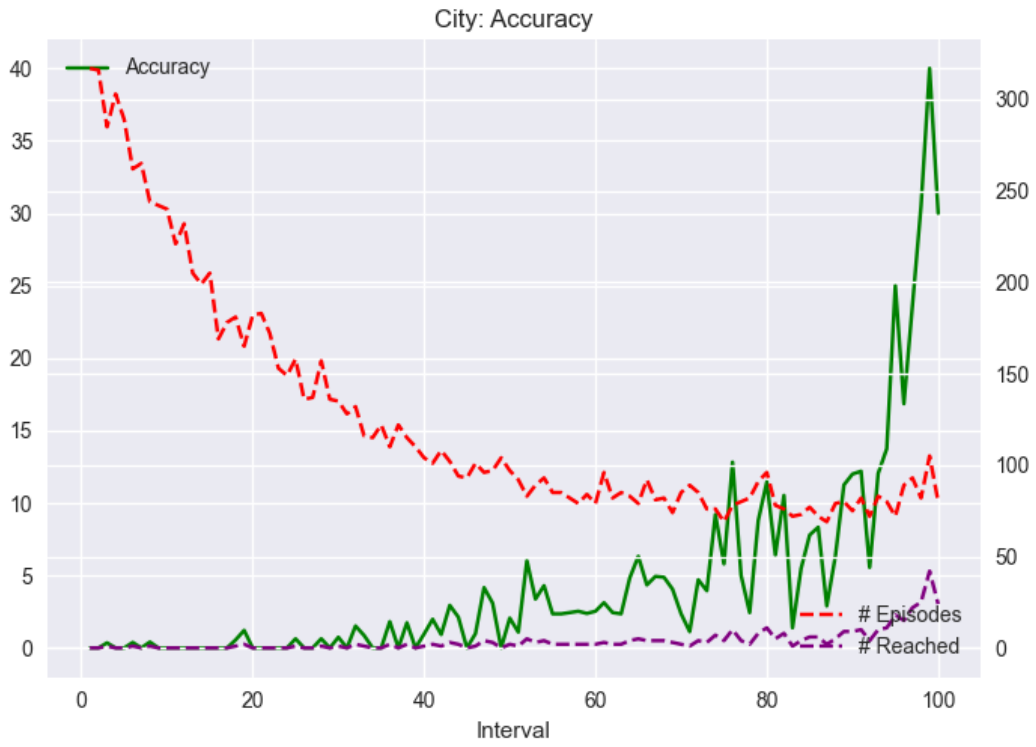


Figura 38: Entrenamiento en "City": Precisión

La precisión, a pesar de incrementar su valor hacia el final del entrenamiento, no alcanza valores tan elevados como los obtenidos en el entorno "Forest". A pesar de realizarse acciones aleatorias al final del entrenamiento, no se considera que la ausencia de estas vaya a aumentar considerablemente la precisión final del agente, que actualmente toma un valor del 30%.

- **Conclusión:**

A pesar del descenso de la "loss" y el aumento de la recompensa media característicos de un correcto aprendizaje, la precisión plantea un dilema a la hora de considerar aptos o no los resultados del agente entrenado en el entorno "City", ya que el dron solo alcanza el objetivo un 30% de las veces.

3.5 Resultados en el entorno “Mountains”

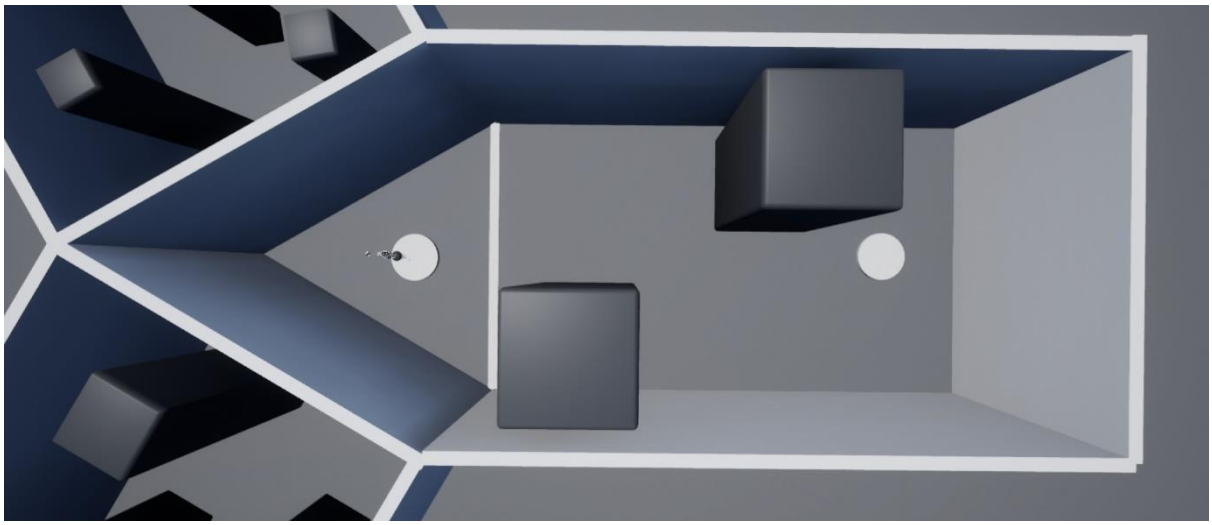


Figura 39: Planta del entorno "Mountains"

El agente no ha respondido satisfactoriamente al entrenamiento propuesto en ambas partes, por lo que se exponen los resultados obtenidos tras la Parte I.

- **Épsilon:**

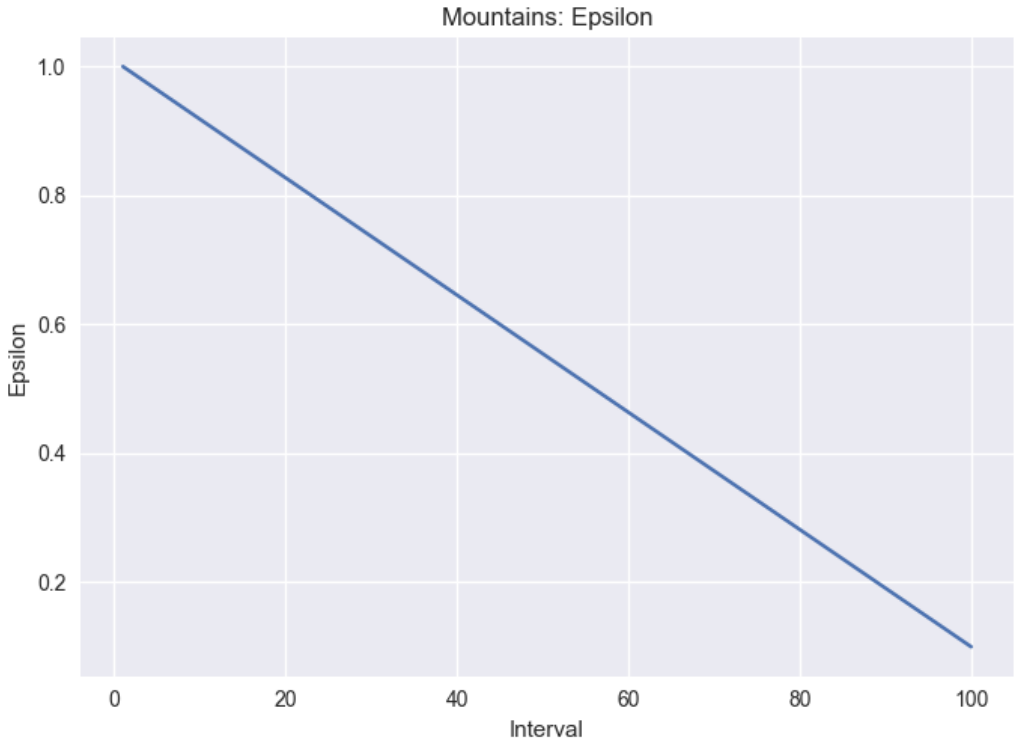


Figura 40: Entrenamiento en "Mountains": Épsilon

Se aprecia en el desarrollo de la épsilon a lo largo de la Parte I del entrenamiento.

- **Recompensa media** (por paso)**:**

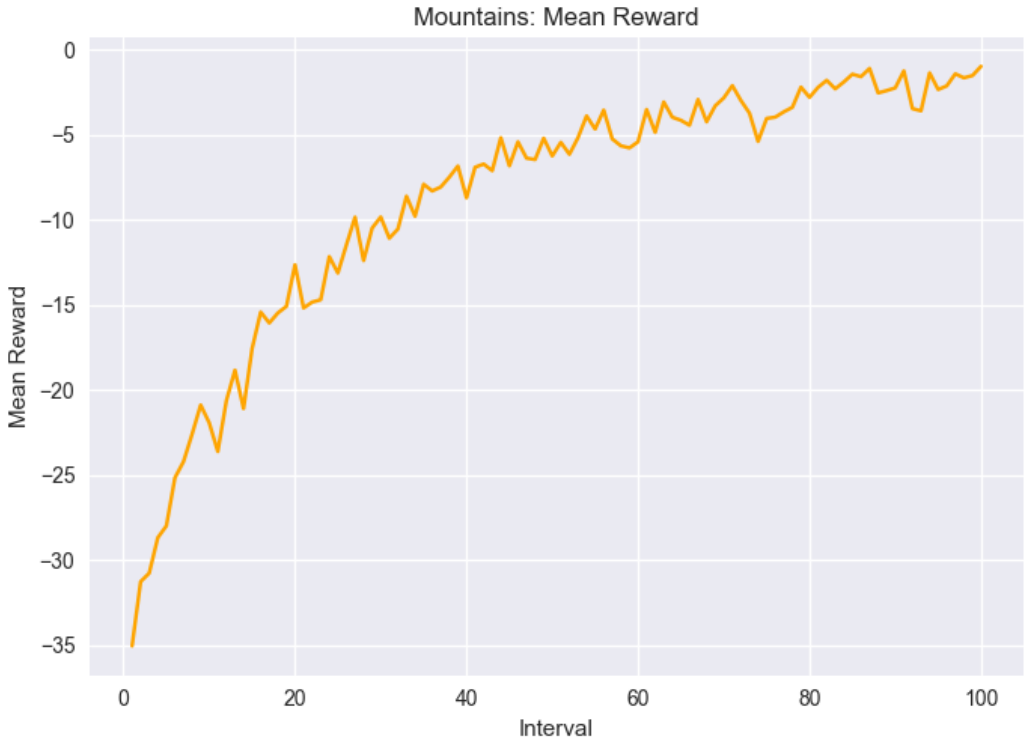


Figura 41: Entrenamiento en "Mountains": Recompensa media

Se aprecia un aprendizaje en la tendencia al alza de la recompensa media por paso, sin embargo, esta no llega a alcanzar valores positivos, lo que indica que no está compensando todo lo que podría el efecto negativo de las colisiones con un avance predominante hacia delante.

- **“Loss”:**

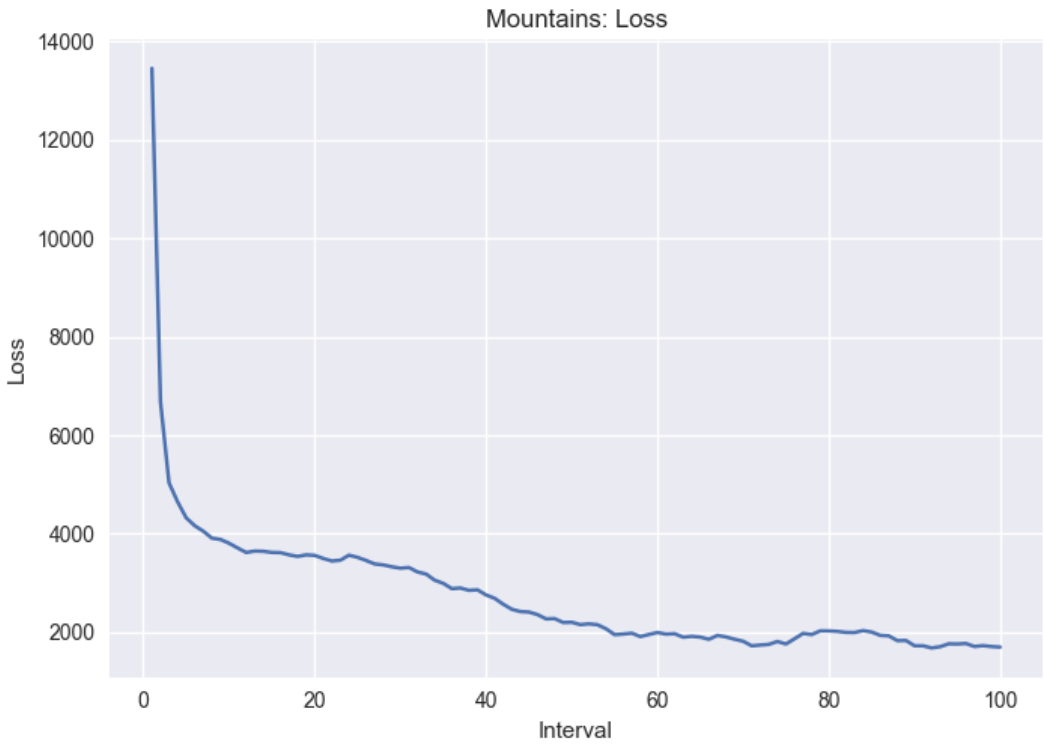


Figura 42: Entrenamiento en "Mountains": "Loss"

La “loss” sugiere un aprendizaje estable, disminuyendo su valor drásticamente al comienzo del ensayo, y manteniendo su valor bajo.

- **Precisión:**

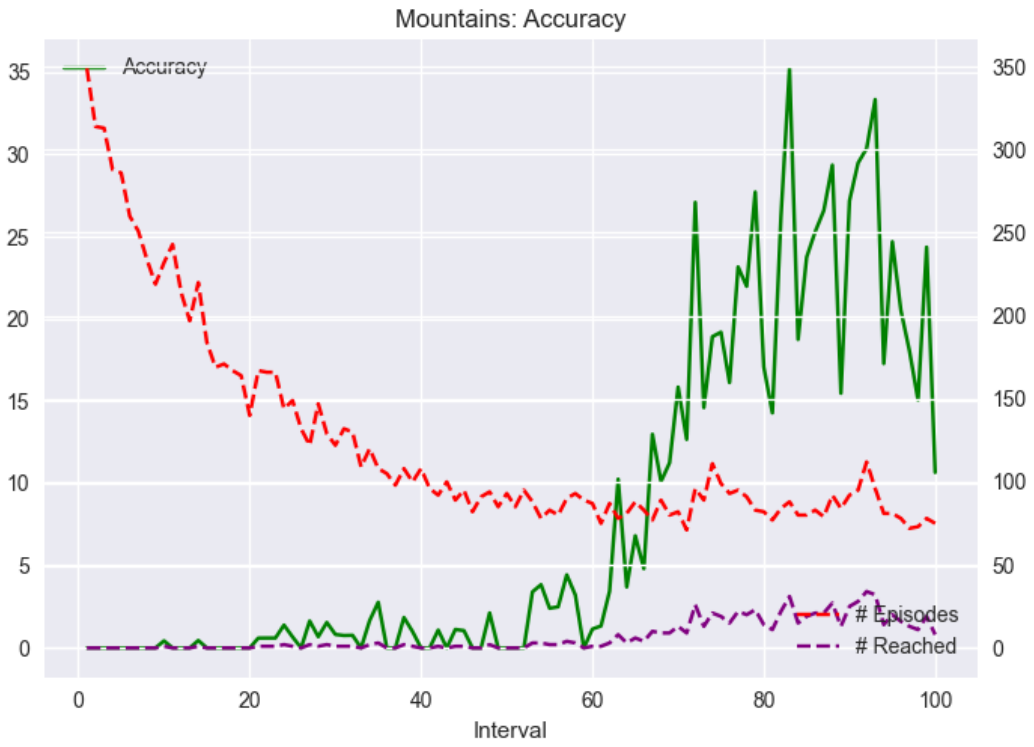


Figura 43: Entrenamiento en "Mountains": Precisión

Se aprecian elevadas oscilaciones en el desarrollo de la precisión del agente, que, a pesar de alcanzar un valor máximo de 35%, disminuye posteriormente hasta un valor final de 10,66%.

- **Conclusión:**

El crecimiento final descrito por la precisión no alcanza un valor suficientemente elevado y estable como para indicar que el agente haya aprendido a llegar al objetivo y esquivar obstáculos. Se considera por lo tanto que el agente ensayado en el entorno “Mountains” no es apto para la resolución del problema propuesto.

3.6 Comparativa de los agentes

Con el objetivo de concluir la aptitud de los agentes se les ha ensayado haciendo uso del fichero "test.py", que evalúa la política inferida por el agente en ausencia de acciones exploratorias (aleatorias), y sin llevar a cabo ningún proceso de aprendizaje, es decir, los pesos de las redes neuronales se mantienen constantes.

Para evaluar la aptitud de los agentes a la hora de resolver el problema propuesto en su entorno de entrenamiento, se ha establecido un umbral de éxito a partir del 50% de precisión.

Se exponen los resultados de los ensayos en la Tabla 8.

	Modelo entrenado en el entorno		
	Forest	**City**	**Mountains**
Recompensa media	0.4920	-2.3366	-0.9589
"Loss"	1690	1763	1701
Precisión (test) [%]	98	30	23
Aptitud	Apto	No apto	No apto

Tabla 8: Comparativa de los agentes entrenados

A pesar de no haber logrado el objetivo propuesto en los agentes ensayados en los entornos "City" y "Mountains", se obtiene un resultado extraordinario en el agente ensayado en el entorno "Forest", con un 98% de precisión. Cabe destacar que el 2% de error se debe al alcance del tiempo límite por episodio y no a la colisión del dron con un obstáculo.

CAPÍTULO 4: CONCLUSIONES Y TRABAJOS FUTUROS

Se considera que el objetivo del proyecto ha sido cumplido al obtener un modelo de agente capaz de trazar una ruta en un entorno con obstáculos sin colisionar, con una precisión del 98%. Sin embargo, ha de cuestionarse la gran diferencia entre la precisión obtenida en dicho agente y los otros dos agentes ensayados, que obtienen precisiones de 30% y 23%. Con el objetivo de mejorar la precisión de estos dos últimos, y continuar la línea de investigación de este proyecto, se proponen distintas modificaciones a los ensayos realizados.

1. Ampliación de las observaciones en el espacio temporal:

Las observaciones propuestas a lo largo de los ensayos aportan información únicamente del instante en el que se toman, sin recoger información del historial temporal de observaciones pasadas. De esta forma, se pierde la información de la velocidad y aceleración con la que cuenta el dron en cada momento.

Se proponen dos soluciones que implican la ampliación del espacio de las observaciones:

- Aportar al agente las tres últimas observaciones, con el objetivo de que sea capaz de extrapolar de tres puntos temporales, valores de velocidad y aceleración.
- Añadir al espacio de las observaciones los valores de velocidades y aceleraciones lineales en el plano.

2. Aplicación de estructura mixta de red neuronal para el análisis de imagen:

Una posible razón por la cual se ha producido un aprendizaje inestable en el ensayo del entorno con observaciones recibidas de cámaras de profundidad es el aporte de la información del ángulo de guiñada con el objetivo en formato de imagen en forma de "brújula", ya que el agente no recibe la información como el escalar que es, y debe hacer un trabajo adicional para extrapolarla.

Dotar al agente de una estructura mixta, que tratase primero la imagen, y una vez descompuesta analizase el ángulo escalar obviaría ese trabajo adicional, dando una mejor oportunidad al entorno con observaciones en estructura de imagen.

3. Ampliación tridimensional del entorno:

Los ensayos realizados a lo largo del proyecto se han llevado a cabo en un espacio bidimensional, con el objetivo de mantener el espacio de estados en un tamaño asequible para las especificaciones del hardware utilizado. En caso de contar con un hardware capaz de soportar unos tiempos de entrenamiento elevados, resultaría interesante dotar al agente de la capacidad de moverse verticalmente.

4. Implementación del modelo en un dron físico:

En caso de contar con un gemelo digital de un dron físico, se propone repetir los ensayos realizados en el proyecto, con las posibles modificaciones anteriormente descritas, con el objetivo de implementar el modelo de agente resultante en un dron físico. De esta forma, el salto del mundo virtual al físico necesitaría de la menor cantidad de ajustes posible.

APÉNDICES

APÉNDICE I: CÓDIGO DEL ENTORNO

airgym_sensors.py

```
import airsim
import math
import numpy as np
import time

import gym
from gym.spaces import Discrete
from gym.spaces import Box


class AirGym(gym.Env):

    def __init__(self):

        # Connecting to AirSim
        self.client = airsim.MultirotorClient()
        self.client.confirmConnection()
        self.client.enableApiControl(True)
        self.client.armDisarm(True)

        # Movement parameters
        self.linear_velocity = 8         # [m/s]
        self.angular_velocity = 1        # [rad/s]
        self.movement_duration = 0.1     # [seconds]
        self.height = -6                 # [meters]

        # Environment Shapes
        self.observation_space = Box(low=0, high=1, shape=(1, 33))
        self.action_space = Discrete(n=4)        # Change to 6 to enable yaw

        # Episode duration
        self.start_time = time.time()
        self.episode_duration = 40       # [seconds]

        # Goal and initial distance
        self.goal_threshold = 1.5          # [meters]

        '''
        self.points = [[00.00, 00.00],
                       [00.00, 50.00],
                       [43.03, -25.52],
                       [-43.53, -24.57]]
        '''

        self.goal_reached = 0
        self.start = [00.00, 00.00]
        self.goal = [-43.53, -24.57]         # Forest
        #self.goal = [00.00, 50.00]           # City
        #self.goal = [43.03, -25.52]          # Mountains


        self.current_distance = 2 * self.goal_threshold
```

```
        # Initial state
        self.state = self.get_state()

    def get_state(self):

        # Compass image section
        state = self.client.getMultirotorState()
        position = [state.kinematics_estimated.position.x_val,
state.kinematics_estimated.position.y_val]
        self.drone_yaw =
airsim.to_eularian_angles(state.kinematics_estimated.orientation)[2]
        self.distance = np.matmul(np.array([[np.cos(self.drone_yaw),
np.sin(self.drone_yaw)],
                                            [-np.sin(self.drone_yaw),
np.cos(self.drone_yaw)]]),
                                  np.array([self.goal[0] - position[0],
self.goal[1] - position[1]]))

        goal_yaw = (math.atan2(self.distance[1], self.distance[0]) / (2 *
np.pi)) + 0.5

        distance_sensor_data =
[self.client.getDistanceSensorData(str(i)).distance / 41 for i in
range(32)]

        observation = distance_sensor_data + [goal_yaw]

        return observation

    # Calculates reward, returned to step
    def get_reward(self):

        # Distance to goal scalar
        self.previous_distance = self.current_distance
        self.current_distance = np.sqrt(self.distance[0] ** 2 +
self.distance[1] ** 2)

        # Collision detection
        self.has_collided = self.client.simGetCollisionInfo().has_collided

        # Collision
        if self.has_collided:
            reward = -1000

        # Distance reduction
        else:

            reward = 10 * (self.previous_distance - self.current_distance)

            # reward = -0.01 * self.current_distance
# dist
            # reward = 10 * (self.previous_distance -
self.current_distance)      # delta

        return reward

    def get_done(self):

        if self.current_distance < self.goal_threshold:
            self.goal_reached += 1
            print(" - Goal Reached: " + str(self.goal_reached))
```

```
        return self.has_collided or time.time() - self.start_time >
self.episode_duration \
               or self.current_distance < self.goal_threshold

    # Takes action, called by step
    def take_action(self, action):

        # Move forward
        if action == 0:
            self.client.moveByVelocityZAsync(np.cos(self.drone_yaw) *
self.linear_velocity,
                                             np.sin(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Move backwards
        elif action == 1:
            self.client.moveByVelocityZAsync(-np.cos(self.drone_yaw) *
self.linear_velocity,
                                             -np.sin(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Move right
        elif action == 2:
            self.client.moveByVelocityZAsync(-np.sin(self.drone_yaw) *
self.linear_velocity,
                                             np.cos(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Move left
        elif action == 3:
            self.client.moveByVelocityZAsync(np.sin(self.drone_yaw) *
self.linear_velocity,
                                             -np.cos(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Yaw right
        elif action == 4:
            self.client.moveByAngleRatesZAsync(0, 0, -
self.angular_velocity,
                                               self.height,
self.movement_duration).join()

        # Yaw left
        elif action == 5:
            self.client.moveByAngleRatesZAsync(0, 0, self.angular_velocity,
                                               self.height,
self.movement_duration).join()

    def step(self, action):

        # Action
        self.take_action(action)
```

```
        # Observation
        self.state = self.get_state()

        # Reward
        reward = self.get_reward()

        # Done
        done = self.get_done()

        return self.state, reward, done, {}

    def reset(self):

        # Drone reset
        self.client.reset()
        time.sleep(0.1)
        self.client.enableApiControl(True)
        self.client.armDisarm(True)

        pose = self.client.simGetVehiclePose()
        pose.position.x_val = self.start[0]
        pose.position.y_val = self.start[1]
        pose.position.z_val = self.height
        self.client.simSetVehiclePose(pose, True)

        goal_pose = self.client.simGetObjectPose("Sphere_5")
        goal_pose.position.x_val = self.goal[0]
        goal_pose.position.y_val = self.goal[1]
        self.client.simSetObjectPose("Sphere_5", goal_pose)

        # Parameters reset
        self.has_collided = False
        self.start_time = time.time()

        # State reset
        self.state = self.get_state()
        self.current_distance = np.sqrt(self.distance[0] ** 2 +
self.distance[1] ** 2)
        self.previous_distance = self.current_distance

        return self.state
```

airgym_one_camera.py

```
import airsim
import cv2
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import time

import gym
from gym.spaces import Discrete
from gym.spaces import Box


class AirGym(gym.Env):

    def __init__(self):

        # Connecting to AirSim
        self.client = airsim.MultirotorClient()
        self.client.confirmConnection()
        self.client.enableApiControl(True)
        self.client.armDisarm(True)

        # Movement parameters
        self.linear_velocity = 8        # [m/s]
        self.angular_velocity = 1        # [rad/s]
        self.movement_duration = 0.1     # [seconds]
        self.height = -6                 # [meters]

        # Environment Shapes
        self.observation_space = Box(low=0, high=1, shape=(64, 256))
        self.action_space = Discrete(n=3)       # Change to 6 to enable yaw

        # Episode duration
        self.start_time = time.time()
        self.episode_duration = 90      # [seconds]

        # Goal and initial distance
        self.goal_threshold = 1.5         # [meters]

        self.points = [[00.00, 00.00],
                       [00.00, 50.00],
                       [43.03, -25.52],
                       [-43.53, -24.57]]

        self.current_distance = 2 * self.goal_threshold
        self.goal = [999, 999]

        # Initial state
        self.state = self.get_state()

    def get_state(self):

        # Compass image section
        state = self.client.getMultirotorState()
        position = [state.kinematics_estimated.position.x_val,
state.kinematics_estimated.position.y_val]
```

```
        self.drone_yaw =
airsim.to_eularian_angles(state.kinematics_estimated.orientation)[2]
        self.distance = np.matmul(np.array([[np.cos(self.drone_yaw),
np.sin(self.drone_yaw)],
                                            [-np.sin(self.drone_yaw),
np.cos(self.drone_yaw)]]),
                                  np.array([self.goal[0] - position[0],
self.goal[1] - position[1]]))

        goal_yaw = math.atan2(self.distance[1], self.distance[0])

        compass_img = np.ones((16, 256))
        for i, column in enumerate(compass_img[0]):
            if i == int((goal_yaw + np.pi) * 128 / np.pi):
                compass_img[:, i - 3:i + 3] = 0
                break

        # Depth view image
        response = self.client.simGetImages([airsim.ImageRequest("0",
airsim.ImageType.DepthVis, False, False)])[0]

        depth_img = np.frombuffer(response.image_data_uint8,
dtype=np.uint8) / 255
        depth_img = depth_img.reshape((response.height, response.width,
3))[:, :, 0]
        depth_img = depth_img[48:96, :]

        # Full observation
        try:
            observation = cv2.vconcat([compass_img, depth_img])
        except:
            observation = self.state

        '''
        # Debug render
        cv2.imshow("DepthView", observation)
        if cv2.waitKey(1) == 0xFF & ord('q'):
            pass
        '''

        return observation

    # Calculates reward, returned to step
    def get_reward(self):

        # Distance to goal scalar
        self.previous_distance = self.current_distance
        self.current_distance = np.sqrt(self.distance[0] ** 2 +
self.distance[1] ** 2)

        # Collision detection
        self.has_collided = self.client.simGetCollisionInfo().has_collided

        # Collision
        if self.has_collided:
            return -1000

        # Distance reduction
        else:
            return 10 * (self.previous_distance - self.current_distance)
```

```
    def get_done(self):

        if self.current_distance < self.goal_threshold:
            print(" - Goal Reached")

        return self.has_collided or time.time() - self.start_time >
self.episode_duration \
               or self.current_distance < self.goal_threshold

    # Takes action, called by step
    def take_action(self, action):

        # Move forward
        if action == 0:
            self.client.moveByVelocityZAsync(np.cos(self.drone_yaw) *
self.linear_velocity,
                                             np.sin(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Yaw right
        elif action == 1:
            self.client.moveByAngleRatesZAsync(0, 0, -
self.angular_velocity,
                                               self.height,
self.movement_duration).join()

        # Yaw left
        elif action == 2:
            self.client.moveByAngleRatesZAsync(0, 0, self.angular_velocity,
                                               self.height,
self.movement_duration).join()


        # Move right
        elif action == 3:
            self.client.moveByVelocityZAsync(-np.sin(self.drone_yaw) *
self.linear_velocity,
                                             np.cos(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Move left
        elif action == 4:
            self.client.moveByVelocityZAsync(np.sin(self.drone_yaw) *
self.linear_velocity,
                                             -np.cos(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()

        # Move backwards
        elif action == 3:
            self.client.moveByVelocityZAsync(-np.cos(self.drone_yaw) *
self.linear_velocity,
                                             -np.sin(self.drone_yaw) *
self.linear_velocity,
                                             self.height,
self.movement_duration).join()
```

```
    def step(self, action):

        # Action
        self.take_action(action)

        # Observation
        self.state = self.get_state()

        # Reward
        reward = self.get_reward()

        # Done
        done = self.get_done()

        return self.state, reward, done, {}

    def reset(self):

        # Drone reset
        self.client.reset()
        time.sleep(0.1)
        self.client.enableApiControl(True)
        self.client.armDisarm(True)

        # Start and Goal
        start, self.goal = random.sample(self.points, k=2)

        pose = self.client.simGetVehiclePose()
        pose.position.x_val = start[0]
        pose.position.y_val = start[1]
        pose.position.z_val = self.height
        self.client.simSetVehiclePose(pose, True)

        goal_pose = self.client.simGetObjectPose("Sphere_5")
        goal_pose.position.x_val = self.goal[0]
        goal_pose.position.y_val = self.goal[1]
        self.client.simSetObjectPose("Sphere_5", goal_pose)

        # Parameters reset
        self.has_collided = False
        self.start_time = time.time()

        # State reset
        self.state = self.get_state()
        self.current_distance = np.sqrt(self.distance[0] ** 2 +
self.distance[1] ** 2)
        self.previous_distance = self.current_distance

        return self.state
```

APÉNDICE II: CÓDIGO DEL AGENTE

agent.py

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Input, Dense, Flatten, Conv2D,
MaxPooling2D
from tensorflow.keras.optimizers import Adam

from rl.agents import DQNAgent
from rl.policy import EpsGreedyQPolicy, LinearAnnealedPolicy
from rl.memory import SequentialMemory

import matplotlib.pyplot as plt
import numpy as np
import os
import time


class DroneAgent:

    def __init__(self, env, initial_epsilon, final_epsilon, gamma,
target_model_update,
                 learning_rate, steps):

        # Environment specs
        self.env = env
        self.observation_shape = env.observation_space.shape
        self.n_actions = env.action_space.n

        # Hyperparameters
        self.initial_epsilon = initial_epsilon
        self.final_epsilon = final_epsilon
        self.gamma = gamma
        self.target_model_update = target_model_update
        self.learning_rate = learning_rate
        self.steps = steps

        # Model and agent
        self.model = self.build_model()
        self.agent = self.build_agent()

        self.train_history = None
        self.model_path = None

    def build_model(self):

        # Sensor data
        model = Sequential()
        model.add(Flatten(input_shape=(self.observation_shape)))
        model.add(Dense(256, activation='relu'))
        model.add(Dense(256, activation='relu'))
        model.add(Dense(self.n_actions, activation='linear'))

        '''
        # One depth camera
```

```
        model = Sequential()
        model.add(Conv2D(32, (5, 5), strides=2, activation='relu',
input_shape=(1,) + self.observation_shape, data_format='channels_first'))
        model.add(MaxPooling2D(data_format='channels_first'))
        model.add(Conv2D(64, (3, 3), strides=2, activation='relu',
data_format='channels_first'))
        model.add(MaxPooling2D(data_format='channels_first'))
        model.add(Flatten())
        model.add(Dense(256, activation='relu'))
        model.add(Dense(128, activation='relu'))
        model.add(Dense(self.n_actions, activation='linear'))
        '''

        return model

    def build_agent(self):

        policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps',
value_max=self.initial_epsilon,
                                      value_min=self.final_epsilon,
value_test=0, nb_steps=self.steps)
        memory = SequentialMemory(limit=100000, window_length=1)
        dqn = DQNAgent(model=self.model, policy=policy, gamma=self.gamma,
memory=memory, nb_actions=self.n_actions,
                       target_model_update=self.target_model_update,
enable_double_dqn=True)

        return dqn

    def fit(self):

        self.agent.compile(Adam(lr=self.learning_rate), metrics=['mae'])
        self.train_history = self.agent.fit(self.env, nb_steps=self.steps,
visualize=False, verbose=1,
                                             action_repetition=3).history

    def save(self):
        self.model_path = 'models/ddqn_' + str(time.time())
        os.mkdir(self.model_path)
        self.model.save_weights(self.model_path + '/weights.h5',
overwrite=True)

    def load(self, path):

        self.model.load_weights(path)
        self.build_agent()
        self.agent.compile(Adam(lr=self.learning_rate), metrics=['mae'])

    def test(self, episodes):
        self.agent.test(self.env, nb_episodes=episodes, visualize=False)

    def plot(self):

        # Plot rewards

        rewards = self.train_history['episode_reward']
        kernel_size = int(0.001 * self.steps)

        mean_rewards = np.zeros(len(rewards) - kernel_size)
        for episode_number in range(mean_rewards.size):
            for i in range(episode_number, episode_number + kernel_size):
```

```
                mean_rewards[episode_number] += rewards[i]
            mean_rewards[episode_number] /= kernel_size

        plt.style.use('seaborn')
        fig1, ax1 = plt.subplots()
        fig2, ax2 = plt.subplots()

        ax1.plot(rewards, linestyle='-', label='Rewards')
        ax1.plot(mean_rewards, color='orange', linestyle='--', label='Mean
Rewards')
        ax2.plot(mean_rewards, color='orange', linestyle='-', label='Mean
Rewards')

        ax1.set_title('City - DDQN: D:256 - D:256 - D:4')
        ax1.set_xlabel("Episode Number")
        ax1.set_ylabel("Reward")
        ax1.legend(loc=1)

        ax2.set_title('City - DDQN: D:256 - D:256 - D:4')
        ax2.set_xlabel("Episode Number")
        ax2.set_ylabel("Reward")
        ax2.legend(loc=1)

        fig1.savefig(self.model_path + '/rewards.png')
        fig2.savefig(self.model_path + '/mean_rewards.png')

        # Plot model
        tf.keras.utils.plot_model(self.agent.model, to_file=self.model_path
+ '/.model.png', show_shapes=True,
                                  show_dtype=False, show_layer_names=False,
rankdir='TB', expand_nested=False, dpi=96)
```

APÉNDICE III: CÓDIGOS DE VINCULACIÓN ENTORNO – AGENTE

train.py

```
from airgym_sensors import AirGym
from agent import DroneAgent

import tensorflow as tf
config =
tf.compat.v1.ConfigProto(gpu_options=tf.compat.v1.GPUOptions(per_process_gp
u_memory_fraction=0.8))
config.gpu_options.allow_growth = True
session = tf.compat.v1.Session(config=config)
tf.compat.v1.keras.backend.set_session(session)

# Environment
env = AirGym()
env.reset()

# Hyperparameters
initial_epsilon = 1.0
final_epsilon = 0.1             # Exploration factor
gamma = 0.99                    # Future action weight
target_model_update = 10000     # original = 1e-2
learning_rate = 2.5e-4          # 2.5e-4 prev
steps = 1000000

# Agent definition
agent = DroneAgent(env,
                   initial_epsilon=initial_epsilon,
final_epsilon=final_epsilon,
                   gamma=gamma,
                   target_model_update=target_model_update,
                   learning_rate=learning_rate,
                   steps=steps)

# Model summary
agent.model.summary()

# Agent training
agent.fit()

# Model saving
agent.save()
agent.plot()

# Test
agent.test(episodes=10)
```

load.py

```
from airgym_sensors import AirGym
from agent import DroneAgent

import tensorflow as tf
config =
tf.compat.v1.ConfigProto(gpu_options=tf.compat.v1.GPUOptions(per_process_gp
u_memory_fraction=0.8))
config.gpu_options.allow_growth = True
session = tf.compat.v1.Session(config=config)
tf.compat.v1.keras.backend.set_session(session)

# Environment
env = AirGym()
env.reset()

# Hyperparameters
initial_epsilon = 0.1
final_epsilon = 0.01
gamma = 0.99                    # Future action weight
target_model_update = 10000     # original = 1e-2
learning_rate = 2.5e-4
steps = 200000

# Agent definition
agent = DroneAgent(env,
                   initial_epsilon=initial_epsilon,
final_epsilon=final_epsilon,
                   gamma=gamma,
                   target_model_update=target_model_update,
                   learning_rate=learning_rate,
                   steps=steps)

# Model summary
agent.model.summary()

# Model loading
agent.load('models\ddqn_forest_ini\weights.h5')

# Agent training
agent.fit()

# Model saving
agent.save()
agent.plot()

# Test
agent.test(episodes=10)
```

test.py

```
from airgym_sensors import AirGym
from agent import DroneAgent

import tensorflow as tf
config =
tf.compat.v1.ConfigProto(gpu_options=tf.compat.v1.GPUOptions(per_process_gp
u_memory_fraction=0.8))
config.gpu_options.allow_growth = True
session = tf.compat.v1.Session(config=config)
tf.compat.v1.keras.backend.set_session(session)

# Environment
env = AirGym()
env.reset()

# Hyperparameters
initial_epsilon = 1.0
final_epsilon = 0.05
gamma = 0.99                    # Future action weight
target_model_update = 10000     # original = 1e-2
learning_rate = 2.5e-4
steps = 10000

# Agent definition
agent = DroneAgent(env,
                   initial_epsilon=initial_epsilon,
final_epsilon=final_epsilon,
                   gamma=gamma,
                   target_model_update=target_model_update,
                   learning_rate=learning_rate,
                   steps=steps)

# Model summary
agent.model.summary()

agent.load('models\ddqn_forest_large\weights.h5')
agent.test(episodes=10)
```

APÉNDICE IV: CONFIGURACIÓN DE AIRSIM

settings.json – "AirGym Sensors"

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "ClockSpeed": 2,
  "Vehicles": {

    "Drone": {

      "VehicleType": "SimpleFlight",
      "Sensors": {

        "0": {
          "SensorType": 5,
          "Enabled" : true,
          "Yaw": 0, "Pitch": 0, "Roll": 0,
          "DrawDebugPoints": true
        },

        "1": {
          "SensorType": 5,
          "Enabled" : true,
          "Yaw": 11.25, "Pitch": 0, "Roll": 0,
          "DrawDebugPoints": true
        },

        "2": {
          "SensorType": 5,
          "Enabled" : true,
          "Yaw": 22.5, "Pitch": 0, "Roll": 0,
          "DrawDebugPoints": true
        },

        "3": {
          "SensorType": 5,
          "Enabled" : true,
          "Yaw": 33.75, "Pitch": 0, "Roll": 0,
          "DrawDebugPoints": true
       …
        }
      }
    }
  }
}
```

settings.json – "AirGym Camera"

```
{
  "SettingsVersion": 1.2,
  "SimMode": "Multirotor",
  "ClockSpeed": 2,

  "CameraDefaults": {
    "Gimbal": {
      "Stabilization": 1,
      "Pitch": 0, "Roll": 0
    }
  },

  "Vehicles": {

    "Drone": {

      "VehicleType": "SimpleFlight",
      "Cameras": {

        "0": {
          "CaptureSettings": [{
              "ImageType": 3
          }],
          "X": 0, "Y": 0, "Z": -1,
          "Pitch": 0, "Roll": 0, "Yaw": 0,
          "Gimbal": {
            "Stabilization": 1,
            "Pitch": 0, "Roll": 0
          }
        }
      }
    }
  }
}
```

APÉNDICE V: OBJETIVOS DEL DESARROLLO SOSTENIBLE

En 2015, los países miembros de las Naciones Unidas se reunieron para aprobar 17 Objetivos como parte de la Agenda 2030 para el Desarrollo Sostenible. Dichos objetivos tienen como finalidad poner fin a la pobreza, proteger el planeta y mejorar las vidas y las perspectivas de las personas en todo el mundo.

La búsqueda de la autonomía en los drones, y en la robótica en general, es una herramienta de la que los ODSs pueden verse realmente beneficiados, ya que la aplicación de la autonomía es algo que podría verse implementado en prácticamente cualquier ámbito actual de la robótica.

A continuación, se exponen aquellos Objetivos del Desarrollo Sostenible que más pueden verse beneficiados por el área de investigación relacionada con los ensayos llevados a cabo a lo largo de este proyecto:

1. **ODS 2:** Poner fin al hambre, lograr la seguridad alimentaria y la mejora de la nutrición y promover la agricultura sostenible.

   - **Objetivo 2.1:** Poner fin al hambre y asegurar el acceso de todas las personas, en particular los pobres y las personas en situaciones vulnerables, incluidos los lactantes, a una alimentación sana, nutritiva y suficiente durante todo el año.

     Con la presencia de drones autónomos en países menos desarrollados, el reparto de alimentos frescos y agua podría suponer el abastecimiento de aquellas zonas menos comunicadas y con dificultad de acceso para vehículos terrestres.

   - **Objetivo 2.3:** Duplicar la productividad agrícola y los ingresos de los productores de alimentos en pequeña escala, en particular las mujeres, los pueblos indígenas, los agricultores familiares, los pastores y los pescadores, entre otras cosas mediante un acceso seguro y equitativo a las tierras, a otros recursos de producción e insumos, conocimientos, servicios financieros, mercados y oportunidades para la generación de valor añadido y empleos no agrícolas.

     Actividades de regadío eficiente podrían ser llevadas a cabo por drones autónomos, aumentando las áreas de cultivo sin suponer un coste adicional significativo en cuanto a infraestructuras de mantenimiento, como los aspersores o tuberías de plástico actuales.

2. **ODS 8:** Promover el crecimiento económico inclusivo y sostenible, el empleo y el trabajo decente para todos.
   - **Objetivo 8.2:** Lograr niveles más elevados de productividad económica mediante la diversificación, la modernización tecnológica y la innovación, entre otras cosas centrándose en los sectores con gran valor añadido y un uso intensivo de la mano de obra.

El mercado del dron autónomo, actualmente en expansión, aporta un valor añadido a aquellas industrias con la posibilidad de implementar estas nuevas tecnologías. Además, el crecimiento de esta área de investigación implica la generación de nuevos empleos destinados a ingenieros, técnicos, diseñadores, y todo aquel personal implicado tanto en el proceso de creación del producto, como en la implementación de este en la industria.

3. **ODS 11:** Lograr que las ciudades sean más inclusivas, seguras, resilientes y sostenibles

- **Objetivo 11.6:** De aquí a 2030, reducir el impacto ambiental negativo per cápita de las ciudades, incluso prestando especial atención a la calidad del aire y la gestión de los desechos municipales y de otro tipo.

  La incorporación de vehículos aéreos autónomos en el reparto de paquetería supondría la supresión de emisiones generadas por todos aquellos vehículos que actualmente se utilizan para esta tarea, mejorando así la calidad del aire que respiramos.

BIBLIOGRAFÍA

[1] Richard S. Sutton and Andrew G. Barto. "*Reinforcement Learning: An Introduction"*, (2014, 2015), Bradford Books.

[2] David Silver. *"Reinforcement Learning Course, DeepMind"*, https://www.youtube.com/playlist?list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ, (2015), (visitado 10/2020).

[3] Andrew Ng. *"Deep Learning Specialization, Course 1: Neural Networks and Deep Learning"*, (visitado 12/2020).

[4] Sandro Skansi. *"Introduction to Deep Learning"*, (2018), SPRINGER.

[5] Habibi Aghdam, Hamed, Jahani Heravi, Elnaz. *"Guide to Convolutional Neural Networks"*, (2017), SPRINGER.

[6] Harrison Kinsley and Daniel Kukiela. *"Neural Networks From Scratch in Python"*, (2020), Sentdex.

[7] Microsoft. *"AirSim Documentation"*, microsoft.github.io, (visitado 06/2021).

[8] OpenAI. *"OpenAI Gym Documentation"*, https://gym.openai.com/docs/, (visitado 02/2021).

[9] Google. *"TensorFlow Documentation"*, API Documentation, TensorFlow Core v2.5.0, (visitado 03/2021)

[10] Kjell-K. *"AirSim as an OpenAI Gym Environment"*, https://github.com/Kjell-K/AirGym, (visitado 01/2021).

[11] Rodrigo de Lazcano Pérez Vicente. *"Plataforma para Simulacion del Comportamiento de un Dron Mediante Algoritmos de Aprendizaje por Refuerzo"*, (2020).

[12] Greg Surma. *"Atari – Solving Games with AI"*, https://gsurma.medium.com/atari-reinforcement-learning-in-depth-part-1-ddqn-ceaa762a546f, (visitado 02/2021).

[13] Naciones Unidas. *"Objetivos de desarrollo sostenible"*, https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/, (visitado 07/2021).