AGENTES INTELIGENTES PARA IMITAR EL COMPORTAMIENTO DE LOS JUGADORES DE PÁDEL

ESPECIALIDAD DE COMPUTACIÓN

TRABAJO DE FIN DE GRADO

Autor: Ivan López Rodríguez
Director: Carlos Andujar Gran
Codirector: Mohammadreza Javadiha
Tutor GEP: David Pardos

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Índice de contenidos

1. Resumen 6

2. Contextualización 6
- 2.1 Introducción y contexto 6
- 2.2 Descripción del problema 12
- 2.3 Definición de conceptos 13
- 2.4 Agentes implicados 14

3. Justificación 14
- 3.1 Soluciones existentes 14
- 3.2 Solución tomada 15
- 3.3 Estudio de Unity en el aprendizaje por refuerzo 15
  - 3.3.1 Métodos de entrenamiento 16
  - 3.3.2 MultiAgent Posthumous Credit Assignment 18
  - 3.3.3 Resumen de los algoritmos 22

4. Alcance 22
- 4.1 Objetivos y subobjetivos 22
- 4.2 Requerimientos 23
  - 4.2.1 Requerimientos funcionales 23
  - 4.2.2 Requerimientos no funcionales 23
- 4.3 Obstáculos y riesgos 24

5. Metodología 24

6. Herramientas 25

7. Planificación temporal 26
- 7.1 Descripción inicial de las tareas 26
  - 7.1.1 Gestión del proyecto 26
  - 7.1.2 Trabajo previo 28
  - 7.1.3 Desarrollo del entorno videojuego 28
  - 7.1.4 Desarrollo del agente 29
- 7.2 Recursos 30
  - 7.2.1 Recursos humanos 30
  - 7.2.2 Recursos materiales 30
- 7.3 Gestión del riesgo 30
- 7.4 Cambios en la planificación inicial 32

8. Gestión económica 34
- 8.1 Presupuesto 34
  - 8.1.1 Costes de personal por actividad 34
  - 8.1.2 Costes genéricos 35
  - 8.1.3 Contingencia 36
  - 8.1.4 Costes de los Imprevistos 37
  - 8.1.5 Coste total 37
- 8.2 Control de gestión 38

9. Sostenibilidad **38**
9.1 Autoevaluación 39
9.2 Dimensión económica 39
9.3 Dimensión ambiental 40
9.4 Dimensión social 41

10. Desarrollo del proyecto **42**
10.1 Explicación del software utilizado 42
10.2 Generación del entorno básico 47
10.2.1 Implementación del jugador 49
10.2.2 Implementación de un bot 51
10.2.3 Implementación de la pelota 52
10.2.4 Implementación del primer agente 52
10.2.5 Modificación en el código de la pelota 53
10.2.6 Modificación en el código del jugador 53
10.2.7 Implementación de un controlador del entorno 54
10.3 Entrenamiento del agente básico 55
10.4 Implementación del entorno avanzado 65
10.4.1 Implementación del apuntado dinámico 65
10.4.2 Entrenamiento del agente con apuntado dinámico 66
10.4.3 Implementación de los distintos tipos de golpe 72
10.4.4 Entrenamiento del agente de los distintos tipos de golpe 72
10.5 Implementación del entorno 2 vs 2 78
10.6 Entrenamiento del agente de 2 vs 2 79
10.7 Resumen de los parámetros en los distintos agentes 85
10.8 Configuración del entrenamiento 86

11. Conclusiones **88**
11.1 Futuras mejoras 89

Referencias **92**

Anexo **94**
Diagrama de Gantt inicial 94
Diagrama de Gantt final 95
Vídeos de los resultados 96

Índice de figuras

1. Ejemplo redes neuronales densas 6
2. Ejemplo redes neuronales convolucionales 7
3. Ejemplo de un perceptrón 7
4. Ejemplo del descenso del gradiente 9
5. Elementos necesarios para el aprendizaje por refuerzo 12
6. Ejemplo Curriculum Learning 17
7. Ejemplo Environment Parameter Randomization 18
8. Actor descentralizado multi-agente 19

9. Cálculo baseline para un agente 20
10. Resultados estudio MA-POCA 21
11. Modelo de una metodología ágil 25
12. Tabla descriptiva de la planificación 32
13. Tabla descriptiva de la planificación actualizada 33
14. Tabla del coste de personal 35
15. Jerarquía de objetos Unity 43
16. Escena, cámara y animator de Unity 44
17. Ejemplo transición de animaciones 45
18. Ejemplo del inspector de Unity 45
19. Librerías del entorno virtual de Anaconda 47
20. Entorno del uno contra uno de pádel 49
21. Collider del área de golpeo del jugador 51
22. Resultados de la prueba 55
23. Ejemplo de entrenamiento con un solo entorno 56
24. Ejemplo de entrenamiento con ocho entornos 56
25. Comparación al entrenar más de un entorno a la vez 57
26. Resultados del primer agente del entorno simple 57
27. Muestra del rayo para detectar la red 59
28. Resultados del agente final del entorno simple 59
29. Movimiento de los jugadores en 50 puntos 61
30. Demostración de un punto en el entorno simple 63
31. Demostración de un segundo punto en el entorno simple 65
32. Demostración del eje utilizado para el punto utilizado para apuntar 66
33. Resultados del agente con apuntado 67
34. Movimiento de los jugadores con apuntado en 50 puntos 68
35. Demostración del apuntado 69
36. Segunda demostración del apuntado 71
37. Resultados del agente con nuevos tipos de golpe 73
38. Movimiento de los jugadores con nuevos golpes en 50 puntos 73
39. Demostración al aplicar una víbora 74
40. Velocidad de la pelota al aplicarle una víbora 74
41. Valores de la velocidad en entornos con un solo golpe 75
42. Demostración al aplicar un globo 76
43. Velocidad de la pelota al aplicarle un globo 77
44. Comparación de los valores de posición de la pelota 77
45. Porcentajes de uso de cada tipo de golpe 78
46. Entorno del dos contra dos de pádel 78
47. Resultados del entorno de dos contra dos 80
48. Movimiento de los jugadores en el entorno de dos contra dos 81
49. Leyenda del siguiente conjunto de gráficas 81
50. Conjunto de gráficas del entorno de dos contra dos 84
51. Fichero de configuración del entrenamiento 88
52. Diagrama de Gantt 94
53. Diagrama de Gantt actualizado 95

Índice de tablas

1. Comparativa motores gráficos 15
2. Comparativa de los algoritmos de ML-agents 22
3. Costes por hora del personal 34
4. Plan de contingencia del 10% 36
5. Coste de los imprevistos 37
6. Coste total del proyecto 38
7. Tabla resumen vectores de observación 85

Índice de ecuaciones

1. Cálculo realizado por una neurona 8
2. Cálculo de los nuevos pesos a través del descenso del gradiente 9
3. Cálculo de la función de estado-acción 19
4. Cálculo de la velocidad para definir la trayectoria de la pelota 50

1. Resumen

El objetivo de este proyecto es el desarrollo de una aplicación que permita jugar un partido de pádel, donde los jugadores van a ser controlados por unos agentes entrenados mediante aprendizaje por refuerzo. También podemos hacer que alguno de estos jugadores sea controlado por una persona real, para entonces enfrentarnos a los modelos entrenados, o incluso por si queremos enseñar a jugar a una persona novel.

2. Contextualización

2.1 Introducción y contexto

Desde hace unos años, se ha empezado a utilizar las redes neuronales para la implementación de muchos algoritmos nuevos, como pueden ser los algoritmos recomendadores de *Youtube*, *Spotify* o *Netflix*.

Dependiendo del problema que queramos resolver, se utilizará un tipo de arquitectura u otro. Por ejemplo, para implementar un clasificador, podemos trabajar con las típicas redes neuronales densas (figura 1), pero si nuestra entrada son imágenes, probablemente sería preferible trabajar con redes convolucionales (figura 2). También existen otro tipos de redes, como las recurrentes o las generativas antagónicas (GANs), las cuales no vamos a tratar en detalle, ya que no son arquitecturas que vayamos a utilizar.

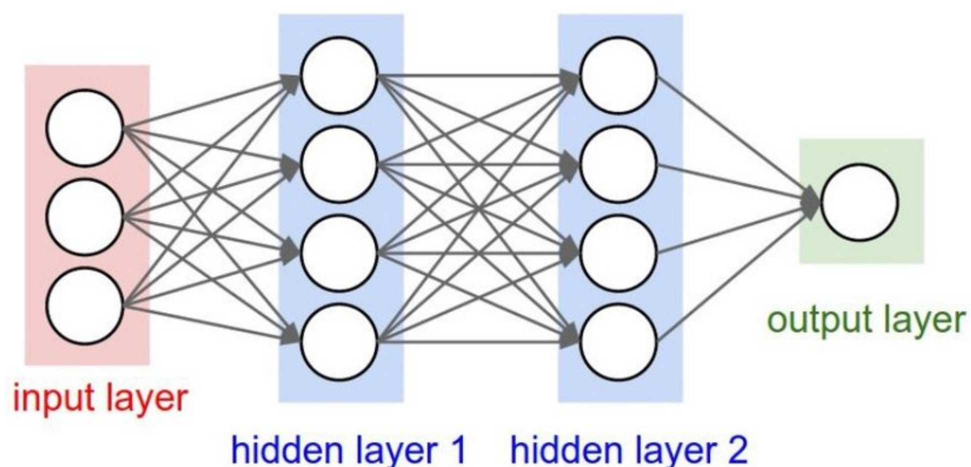


Figura 1: Ejemplo de arquitectura de una red neuronal densa. Fuente: [1]

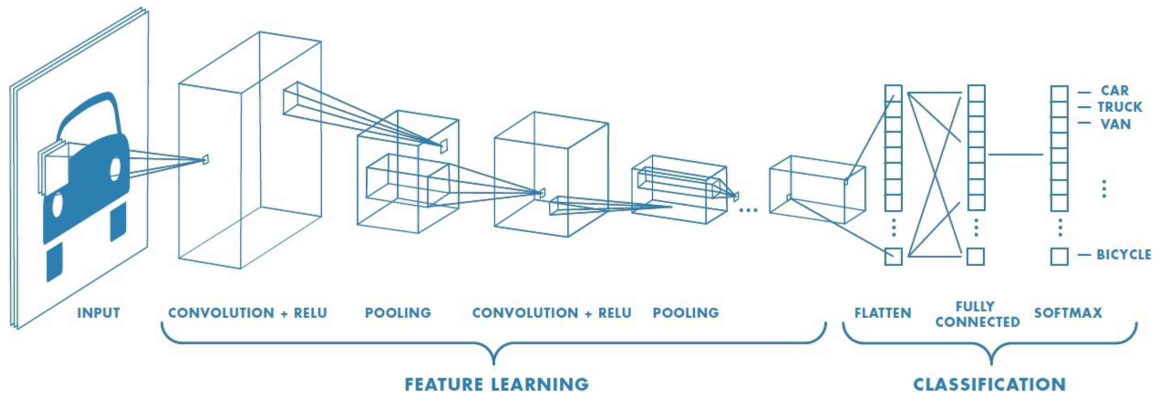


Figura 2: Ejemplo de arquitectura para una red neuronal convolucional. Fuente: [2]

Para poder entender todas estas arquitecturas, primero tenemos que tener claro cómo funciona una neurona. Para ello vamos a hablar sobre el perceptrón, una unidad de red neuronal.

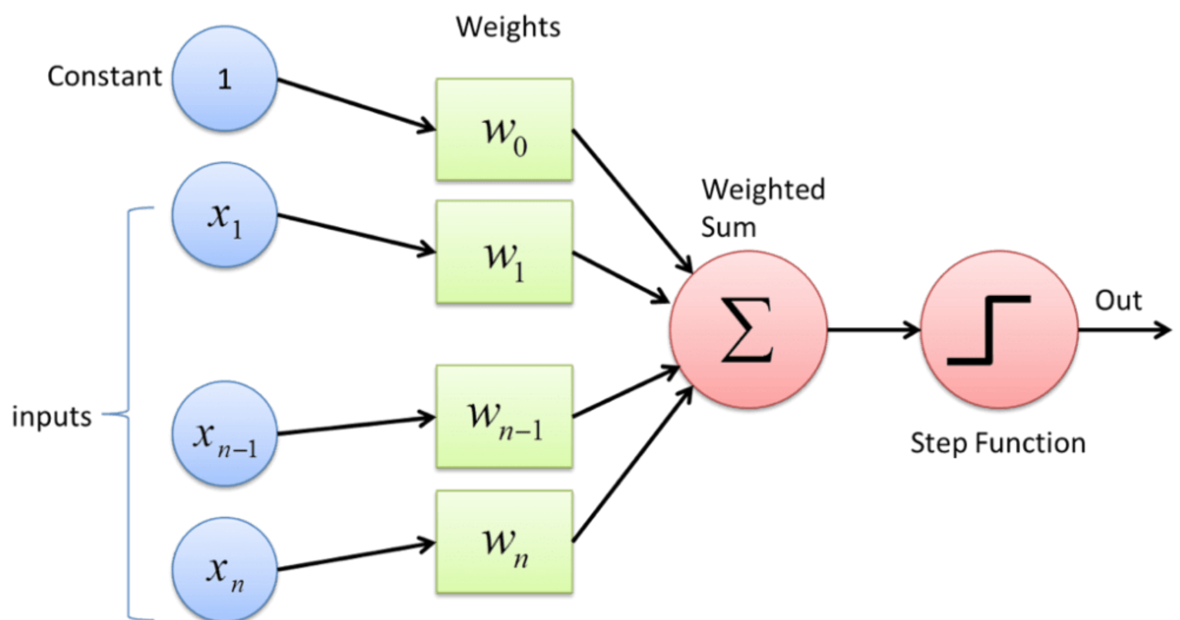


Figura 3: Ejemplo de un perceptrón. Fuente: [16]

Si comparamos la figura 1 con la 3, podríamos decir que la figura 3 es como la 1, pero solo con una capa oculta y de una sola neurona. Aunque hay que destacar que la figura 3 nos está explicando cual es el cálculo que se hace en cada neurona. Concretamente, el cálculo que se hace en cada neurona es:

$$Y = f(\sum_{i=1}^{n} w_i x_i)$$

$$Y = f(X_n * W_n + ... + X_1 * W_1 + W_0)$$

Ecuación 1: Cálculo realizado por una neurona. Fuente: Elaboración propia

Por tanto, tenemos *n* valores como entrada en la neurona, y para cada uno de ellos tenemos un valor por el cual los multiplicamos, estos valores se conocen como pesos (*weights*). Luego, tenemos el valor w_0, el cual se deja igual ya que es como si lo multiplicamos por uno, éste se conoce como *bias*. El siguiente paso es, sumar todos los resultados de las multiplicaciones, y entonces, al valor resultante se le aplica una función *f*, también conocida como función de activación. Una de los motivos para utilizar las funciones de activación, es el hecho de ajustar el rango de los resultados de la red neuronal, por ejemplo, si queremos definir un rango entre 0 y 1, utilizaremos una sigmoide.

Ahora ya sabemos cual es el cálculo que lleva a cabo cada neurona. Sin embargo, no sabemos qué valores tienen los pesos. Normalmente los pesos se inicializan de forma aleatoria, y luego irán variando durante el entrenamiento. Para ajustar los pesos se utiliza una función de coste (*loss function*), que sirve para calcular el error, junto con un algoritmo de optimización como el descenso del gradiente. En el descenso del gradiente simplificado, nos tenemos que imaginar una figura tridimensional, donde vamos a aparecer en un punto de forma aleatoria (por la forma en la que se inicializan los pesos). Ahora, podemos calcular la pendiente de este punto gracias a las derivadas (en este caso es una derivada parcial de la función de coste respecto a un valor de entrada), pero la idea de calcular el gradiente, es para utilizar su valor negado, ya que nosotros queremos minimizar el error. Por tanto, el cálculo del nuevo valor de un peso es el que se puede observar en la ecuación 2. Tenemos otra variable que es el ratio de aprendizaje. Aunque no vamos a entrar mucho en detalle, este ratio sirve para definir el tamaño de los pasos que va a dar la red neuronal. Encontrar un buen valor para el ratio de aprendizaje no es algo trivial, ya que si le damos un valor muy alto, puede ser que nunca encuentre un mínimo, o si por el contrario le asignamos un valor muy bajo, va a estar haciendo muchas iteraciones innecesarias.

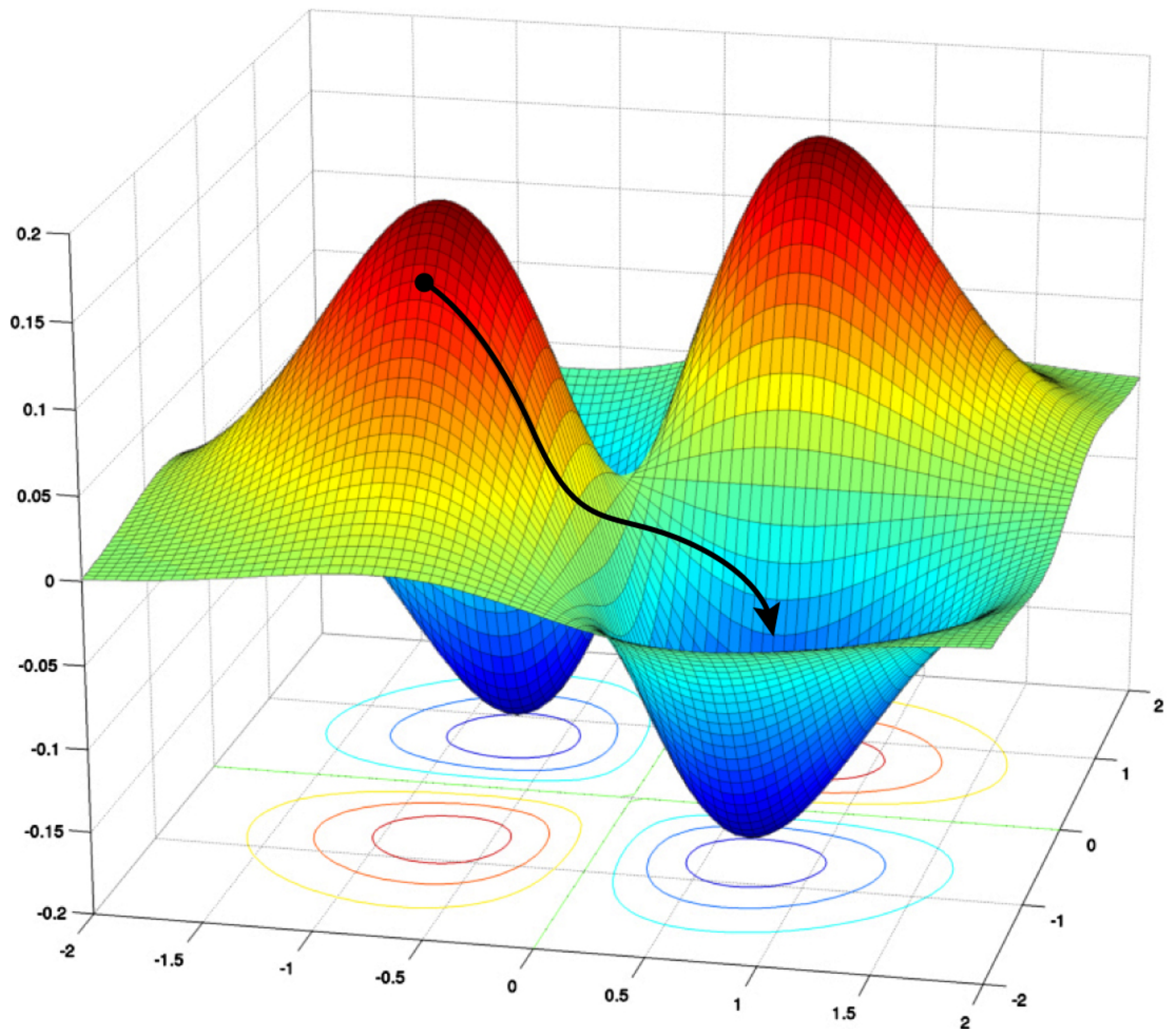


Figura 4: Ejemplo del descenso del gradiente. Fuente: [17]

$$w' = w - (\alpha \ * \ dL/dw)$$

Ecuación 2: Cálculo de los nuevos pesos a través del descenso del gradiente. Fuente: Elaboración propia

En la ecuación 2 tenemos la w', que hace referencia al nuevo valor que se le va a asignar al peso, mientras que w es el valor actual del peso. Luego, tenemos α, que es el ratio de aprendizaje, y por último, tenemos dL/dw, esto es la derivada parcial de la función de coste respecto a cada X, en otras palabras, nos dice cuánto varía la función de coste respecto a los cambios del peso.

El funcionamiento de una red neuronal se puede separar en dos pasos:

- Paso hacia delante (forward propagation): Este paso recibe los valores de las neuronas de la capa de entrada, para entonces ir moviéndose hacia las capas de la derecha, aplicando los cálculos necesarios en cada neurona, hasta llegar a la capa de salida.
- Paso hacia atrás (backward propagation): Ahora que tenemos los valores en la capa de salida, podemos calcular el error con la función de coste, y entonces, calcular los nuevos pesos mediante el descenso del gradiente.

Habiendo explicado de forma simplificada el funcionamiento del perceptrón, podemos construir arquitecturas más grandes como el perceptrón multicapa, que sería algo como lo que se puede observar en la figura 1. Donde podemos tener más de una capa intermedia, y más de una neurona en cada una de las capas.

Otra arquitectura bastante conocida es la de red convolucional, las cuales se utilizan cuando nuestros datos de entrada son matrices, normalmente pensado para imágenes. Añaden dos estrategias nuevas, la convolución y el *pooling*. La convolución se podría definir como un filtro que se le aplica a la imagen utilizando una máscara, en este caso los valores a entrenar son los del filtro. Por otra parte, el pooling trata de reducir el tamaño de la matriz manteniendo la información más relevante.

Aparte de conocer distintas arquitecturas, tenemos que hablar sobre los distintos tipos de aprendizaje que hay, ya que hasta ahora solo hemos hecho referencia a uno de estos.

En todo momento nos hemos centrado en el *aprendizaje supervisado*, es decir disponemos de unos datos de los cuales sabemos el resultado para unos valores de entrada, y por eso mismo somos capaces de calcular el error. Como comentario, normalmente estos datos se separan en dos, en un conjunto de entrenamiento y otro de test. De manera que, en la fase de entrenamiento solo se utilizan los datos de entrenamiento, y una vez la fase de entrenamiento ha terminado, comprobamos la precisión del modelo mediante el conjunto de test. Esto sirve para averiguar si la red realmente ha sido capaz de generalizar la resolución del problema, o si simplemente ha memorizado los datos de entrenamiento (conocido como *overfitting*).

El segundo tipo de entrenamiento que existe es el contrario del supervisado, el *aprendizaje no supervisado*. Es decir, el conjunto de datos que vamos a utilizar, no están etiquetados, por tanto, no conocemos los resultados que la red nos debería devolver. Este tipo de algoritmo normalmente se suele utilizar

para tareas de agrupación, encontrando grupos similares en los valores de entrada.

Para finalizar, tenemos el *aprendizaje por refuerzo*, que es el que nosotros vamos a utilizar. Este tipo se puede decir que es algo intermedio respecto a los dos anteriores, ya que el algoritmo va a recibir algo de feedback a partir de un sistema de puntuación, es decir, cuando sepamos que ha hecho algo bien le daremos una recompensa positiva, y en el caso contrario, una negativa.

En este caso, se trabajará mediante el aprendizaje por refuerzo. Esta metodología se define por estos elementos importantes, que son:

- **Agente**, programa que entrenaremos para “aprender” a hacer una función.
- **Entorno**, mundo donde el agente trabajará.
- **Acciones**, conjunto de movimientos y otras acciones que puede realizar el agente.
- **Recompensas**, sistema de puntuación que evaluará la decisión de elegir una acción u otra por parte del agente (*feedback* que recibirá la red neuronal).
- **Estado,** forma de describir la situación del entorno. En nuestro caso podría ser la posición de los jugadores, posición y velocidad de la pelota y a lo mejor algo más. O incluso podríamos dar como estado toda la imagen del juego, en ese caso deberíamos utilizar una red neuronal convolucional.
- **Política**, secuencia de acciones que ha tomado el agente. En muchos algoritmos de aprendizaje por refuerzo se habla de *Q-values*, los cuales se pueden definir como la probabilidad de escoger una acción para un estado en concreto.

Como bien se puede observar en la figura 5, la iteración principal del aprendizaje por refuerzo consiste en que el agente elige una acción, se efectúa dicha acción en el mundo donde se encuentra el agente, y mediante el resultado de aplicar esa acción en el mundo, se calculará la recompensa para el agente, la cual puede ser positiva o negativa o nula, ya que el agente no va a estar recibiendo recompensas en todas las iteraciones, sino sólo en aquellas que sepamos con certeza que ha hecho algo bien o mal.

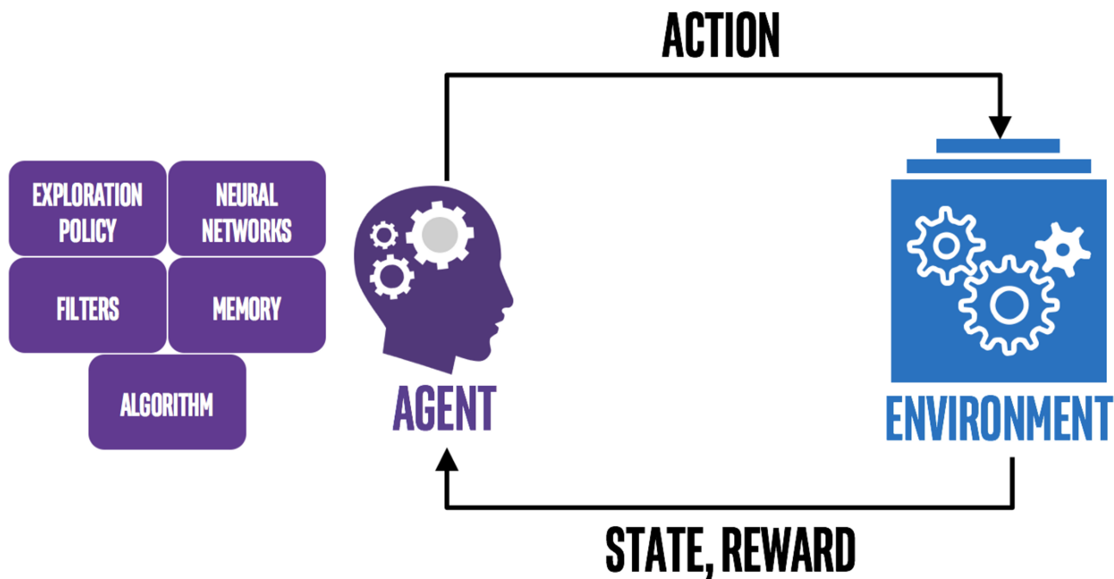


Figura 5: Conjunto de pasos a seguir en el aprendizaje por refuerzo. Fuente: [3]

En cuanto al contexto de este trabajo en el marco académico, en el grado de ingeniería informática de la Universidad Politécnica de Catalunya, impartido por la Facultad de Informática de Barcelona, existen distintas especialidades a elegir por el alumno a partir del tercer año de curso. La elección que tomé, y por tanto, la especialidad de este trabajo, es la de computación. Concretamente, este trabajo abarca tanto al ámbito de gráficos por ordenador, como el de inteligencia artificial, ya que entrenaré un agente mediante aprendizaje por refuerzo y se creará un entorno de videojuego para visualizar los resultados de su entrenamiento.

El proyecto se ha realizado en la modalidad A, es decir en el propio centro. Con Carlos Andujar (responsable de la asignatura de Gráficos) como director, y Mohammadreza Javadiha como codirector.

2.2 Descripción del problema

El objetivo general es el desarrollo de una aplicación donde un usuario podrá jugar a un videojuego de pádel contra un agente entrenado, o enfrentar a dos agentes entrenados, o incluso hacer partidos dos contra dos, donde todos son agentes, o un jugador sea controlado por una persona real. Cabe distinguir las dos partes principales del trabajo, que son:

- Implementación propia del videojuego, sin tener en cuenta que dicha aplicación la utilizará un agente.

- Creación, incorporación y entrenamiento del agente, dentro del entorno creado previamente.

Por tanto, el primer paso será aprender cómo funciona el motor gráfico (Unity3D) y empezar a crear un entorno funcional. Para ello deberemos tener dos modelos, uno para la pista de pádel y otro para los jugadores, las animaciones de los jugadores y la implementación de los jugadores y de la pelota.

La idea es hacerlo sin tener en cuenta que trabajaremos con un agente. Es decir, hacer un videojuego que vaya a ser jugado por una persona con un teclado. Así, cuando se haya finalizado esa primera iteración, crearemos el agente, y las acciones entre las cuales deberá elegir, serán exactamente las que habremos implementado para el jugador que supuestamente iba a controlar una persona.

Cuando ya tengamos vinculado al agente con nuestro entorno correctamente, querremos entrenarlo y comprobar si lo hace correctamente, ya que si no es así, deberemos modificar algunos parámetros.

2.3 Definición de conceptos

Vamos a trabajar con redes neuronales para aplicar la metodología de aprendizaje por refuerzo, por tanto debemos conocer cómo funcionan este tipo de aprendizaje, incluyendo la creación de un entorno correctamente, las acciones entre las que elegirá el agente y el sistema de recompensas que obtendrá el agente por cada acción tomada. En la especialidad de computación, tenemos una asignatura complementaria de especialidad, *Aprendizaje Automático*, la cual explica cómo funcionan las redes neuronales simples. Además hice un curso de JEDI, para profundizar en el mundo *Deep Learning,* y vimos algunos ejemplos de aprendizaje por refuerzo.

Una de las herramientas que se utilizarán es *Unity.* Se trata de un motor de videojuegos *open-source,* con el cual se generará el entorno y la programación de las acciones del agente, por tanto, se deberá estudiar el funcionamiento del mismo.

De alguna forma debemos comunicar el entorno que vamos a generar mediante *Unity* con la respuesta que nos devolverá la red neuronal. Para ello utilizaremos la librería *ML-agents* que se ha incorporado recientemente a *Unity*. Comentar que en Unity, el lenguaje de programación que normalmente se utiliza es **C#,** mientras que la librería, está implementada con **Python**. Es algo de lo más normal, ya que **Python** tiene librerías/entornos muy conocidos

y utilizados para las redes neuronales, como por ejemplo lo son **PyTorch** o **TensorFlow**.

2.4 Agentes implicados

Normalmente cuando se habla de implementar un videojuego, se trabaja con un grupo de distintos profesionales, donde se reparte el trabajo de modelado, animación, programación y otras tareas. Al ser un trabajo individual, no se debe esperar un videojuego pulido a la perfección, ya que gran parte del trabajo viene al estudio e implementación de un agente en el entorno creado. Sin embargo, este producto se puede decir que va dirigido a toda la industria del videojuego, queriendo difundir el cambio de los típicos bots por este tipo de agentes.

Así, este proyecto puede llegar a ser uno, de los muchos ejemplos que hay, que muestre cual es el potencial que tienen los agentes entrenados mediante el aprendizaje por refuerzo.

Este producto podrá ser utilizado por una persona, tanto de espectador, si quiere enfrentar a unos agentes con otros, o incluso como participante, si quiere jugar contra uno de ellos.

El propio motor gráfico, puede llegar a conseguir más visibilidad, gracias a proyectos como este mediante la librería *ML-agents*, o incluso algunas desarrolladoras de videojuegos, que decidan implementar dicha tecnología.

3. Justificación

3.1 Soluciones existentes

Las redes neuronales cada vez se utilizan más, y por eso mismo quería aprender más sobre estas. Me pareció interesante aplicarlas en el área de los videojuegos, que siempre me han llamado la atención, pudiendo utilizar la nueva librería de *Unity*.
Como bien se puede ver en la propia página oficial [4,5], hay distintos ejemplos de videojuegos utilizando dicha tecnología. Pero hasta ahora, no hay ninguno sobre pádel, un deporte de raqueta en auge. Además, en general hay pocos juegos de dicho deporte. Otra aplicación de este TFG, además del entretenimiento para videojuegos, puede ser el hecho de utilizar esta aplicación para mostrar de otra forma las reglas y golpes básicos de pádel, es decir con un enfoque didáctico.

Muchos videojuegos que se encuentran en el mercado, utilizan la inteligencia artificial para definir el comportamiento de distintos personajes, aunque sólo algunos se basan en aprendizaje por refuerzo.

3.2 Solución tomada

Las herramientas que he decidido utilizar son *Unity* como motor gráfico, y la librería ML-agents, que permite entrenar un agente y conectarlo con el entorno del videojuego de forma sencilla. Otros motores gráficos, como *Unreal Engine*, hasta ahora no disponen de una librería que permita entrenar un agente y añadirlo al entorno fácilmente. Lo que supondría tener que generar una interfaz para comunicar el entorno con el agente.

Un ejemplo de estos intermediarios, puede ser los que se encuentran en *OpenAI* [6], donde tenemos la implementación de algunos videojuegos con la propia conexión que deberemos hacer con la red neuronal que vamos a crear nosotros. Pero esto es totalmente distinto a lo que yo he querido hacer, ya que quería tener la libertad de poder implementar la aplicación que yo quisiera.

Si/No	**Crear entorno 3D**	**Conexión agente con el entorno**
GameMaker	Si	No
Unreal Engine	Si	No
Unity	Si	Si

Tabla 1: Comparativa de motores gráficos.

3.3 Estudio de Unity en el aprendizaje por refuerzo

Unity incorpora una librería llamada ***ml-agents***, que permite generar y entrenar agentes mediante el entorno que hemos generado nosotros mismos con el motor gráfico. Esto nos facilita mucho la conexión del agente con la red neuronal y con el entorno, ya que está todo agrupado en la misma herramienta.

La librería *ml-agents* tiene muchas herramientas y muchas formas distintas de poder entrenar nuestros agentes. Si nos fijamos en la fuente [19], podremos observar todas las formas distintas en las que podemos entrenar nuestros agentes.

Primero hay una distinción entre los tipos de modelos por lo que van a utilizar para entrenar, es decir, de qué forma se va a definir los estados del entorno.

Podemos trabajar con un vector de observaciones, que no deja de ser un vector con unos valores con los que podríamos definir el entorno. En este caso se utilizará una red neuronal densa, como la que habíamos visto en la figura 1. Por otro lado, podemos recoger la imagen completa del mundo a partir de los objetos cámara que utiliza Unity. Al estar tratando imágenes, se utilizará una red neuronal convolucional. Incluso podríamos trabajar con agentes con memoria, en este caso se utilizan redes neuronales recurrentes, este tipo de arquitectura no se ha explicado ya que no se va a utilizar, además es algo más allá del objetivo del proyecto.

3.3.1 Métodos de entrenamiento

A continuación, encontramos dos metodologías de entrenamiento, una que depende del entorno y otra que no. En el caso que no depende del entorno, tenemos dos técnicas de aprendizaje, *Deep Reinforcement Learning,* donde podemos utilizar dos algoritmos distintos, *Proximal Policy Optimization (PPO)* o *Soft Actor-Critic (SAC)*, donde la principal diferencia es que SAC es *off-policy*, es decir que aprende de todas las experiencias que ha vivido el agente, mientras que PPO escogerá entre las políticas que mejor han funcionado. Dentro de los entrenamientos agnósticos al entorno, también tenemos el aprendizaje por imitación, cuyo funcionamiento consiste en grabar un video para demostrar cual es el comportamiento que queremos que tenga el agente, y entonces utilizarlo para la fase de entrenamiento. En este caso también disponemos de dos algoritmos distintos. *Generative Adversarial Imitation Learning (GAIL)* utiliza dos redes neuronales, una para decidir qué acción va a tomar el agente, y otra que trata de ver la diferencia entre el video y los estados/acciones del agente. Esta red dará una recompensa mayor cuando el agente se acerque más al comportamiento de la demo. En cada iteración el agente trata de aprender a maximizar su recompensa, pero la otra red va a aprender a diferenciar mejor el video con lo que hace el agente, así que cada vez será más estricta. Luego, tenemos el *Behavioral Cloning (BC)*, el cual trata de copiar las acciones mostradas en los videos. Se puede utilizar conjuntamente con PPO o SAC. Es muy importante que para este algoritmo se trate de mostrar todos los estados posibles, ya que la única fuente del entrenamiento son las grabaciones.

Por otro lado, tenemos los entrenamientos que dependen de los entornos. En nuestro caso podemos decir que el entrenamiento dependerá del entorno, ya que cada vez que la pelota bote dos veces o toque algún cristal se terminará un punto. Para este tipo de entrenamientos ML-agents proporciona cuatro algoritmos distintos. El primero de todos se utiliza para un entorno competitivo, es decir donde vamos a enfrentar un agente contra otro, y no importa si es un juego simétrico o asimétrico. Con simétrico nos referimos a

que los agentes que se enfrentan están con las mismas condiciones y tienen el mismo objetivo, como por ejemplo el fútbol. En cambio, en un juego asimétrico los agentes tendrán objetivos, y por tanto, comportamientos distintos, como sucede por ejemplo en el escondite, donde un agente tendrá el objetivo de esconderse y otro el de encontrar al agente que se ha escondido. El algoritmo en cuestión se conoce como *Self-Play*, la idea es que el agente aprende al competir con la mejor versión del agente hasta el momento. Se puede complementar con los algoritmos de PPO y SAC.

El siguiente tipo de entrenamiento se llama *Curriculum Learning*, y se suele utilizar cuando queremos entrenar un agente el cual se va a enfrentar a nuevas dificultades a lo largo del tiempo. Se asemeja a cómo aprendemos los humanos, ya que nos enfrentamos a nuevos retos a lo largo de nuestra vida. En la figura 6 nos encontramos con un entorno que se encuentra entre los distintos ejemplos proporcionados por la propia librería (Wall Jump). En este caso, se quería entrenar un agente que se tiene que mover hasta un punto en concreto (cuadrado verde), pero el agente se va a encontrar un obstáculo el cual tendría que pasar mediante dos saltos, unos para subirse a la plataforma blanca, y otro para pasar a la otra parte de la zona. Entonces, para entrenar al agente a conseguir dicho comportamiento, se ha utilizado la técnica de *Curriculum Learning*, mediante la construcción de distintos entornos con paredes de distintas alturas, comenzando desde el caso más simple donde no hay pared, hasta llegar al entorno más difícil de todos, donde para pasar a la zona del cuadrado verde tendrá que saltar a la plataforma blanca, para luego saltar a la otra parte del campo.

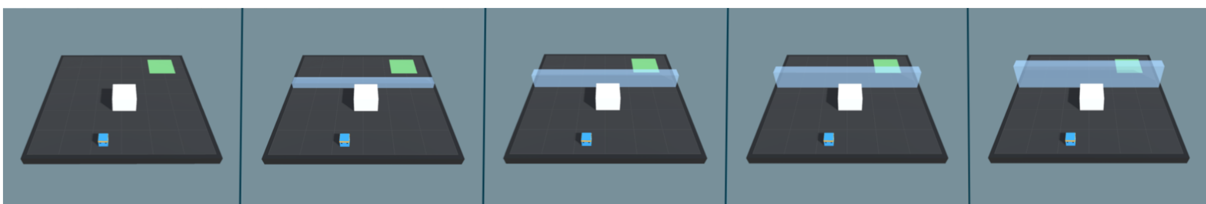


Figura 6: Entorno aplicado al Curriculum Learning. Fuente: [19]

El siguiente tipo de entrenamiento es algo similar al anterior, ya que de alguna forma entrenamos al agente para entornos algo distintos. Nos referimos al *Environment Parameter Randomization*. Éste es el nombre que le ha dado Unity, pero está basado en el concepto de *Domain Randomization* [20]. La idea es la de variar los valores de algunos parámetros del entorno durante el entrenamiento, haciendo así que el agente resultante sea más robusto. Los cambios que hagamos deben hacer que el agente se enfrente a distintas situaciones, las cuales tienen que ser similares, ya que si las modificaciones difieren mucho del entorno anterior, puede que también varíe

el comportamiento que debe aprender el agente, haciendo que el entrenamiento no sea efectivo.

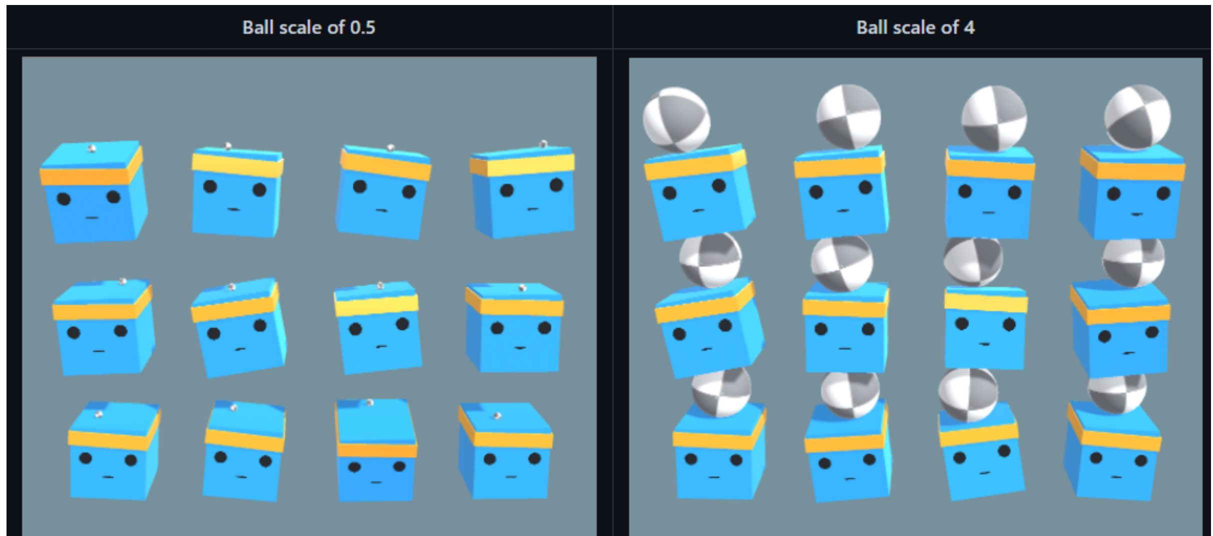


Figura 7: Entorno aplicado al Environment Parameter Randomization. Fuente: [19]

3.3.2 MultiAgent Posthumous Credit Assignment

Por último, tenemos un algoritmo para entrenar agentes en un entorno de cooperación (como en nuestro caso), el *MultiAgent Posthumous Credit Assignment* (MA-POCA). Para poder hablar sobre el algoritmo, primero debemos conocer a qué problema hace referencia. Nos referimos al *Credit Assignment Problem (CAP)* [21], es el problema de determinar qué acciones nos van a llevar a maximizar la puntuación del agente. En el caso de Unity, han ido un poco más allá, y han formulado el problema de *Posthumous Credit Assignment Problem*, el cual hace referencia a entornos donde un agente que ha sido eliminado no va a ser consciente de las siguientes recompensas que pueden conseguir los demás agentes, y a lo mejor el agente que se ha eliminado, se ha tenido que sacrificar para que sus compañeros lleguen al estado donde van a conseguir una mayor recompensa. En nuestro entorno esto no va a suceder, ya que si un agente va a ser eliminado del entorno, quiere decir que se ha finalizado un punto (del partido del pádel), así que en realidad se van a eliminar todos los agentes para reiniciar el entorno al completo. Ésto es así porque en este trabajo consideramos cada punto del partido como la principal unidad de recompensa.

Comentar que muchos de los puntos que se van a tratar del algoritmo han sido extraídos del artículo publicado por Unity [22], que a su vez hace referencias a otros dos artículos [23,24]. La importancia de los artículos en los que se basa Unity reside en el hecho de trabajar con arquitecturas con más de un agente a la vez, utilizando el paradigma de una ejecución descentralizada pero con un entrenamiento centralizado. Así, los agentes

utilizan observaciones locales, pero la información global está disponible durante el entrenamiento.

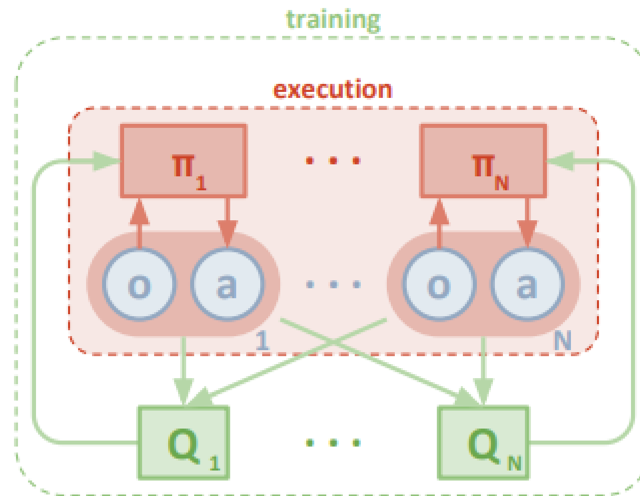


Figura 8: Actor descentralizado multi-agente. Fuente: [23]

La figura 8 ilustra la idea de conocer toda la información de los distintos agentes a la vez (entrenamiento centralizado). EL símbolo **π** hace referencia al conjunto de políticas de los **N** agentes del entorno; **o** es la observación (descripción del entorno) del agente; **a** define la acción que tomará el agente; por último tenemos **Q**, que representa una función centralizada que recibe las acciones de todos los agentes, y algo de información del estado como las observaciones de todos los agentes. Dicha función nos devolverá el *Q-value* para un agente, el cual modificará la política del agente. En el caso de MA-POCA, el valor de esta función se calcula:

$$Q^{\boldsymbol{\pi}}(s_t, \boldsymbol{a_t}) = r(s_t, \boldsymbol{a_t}) + \mathbb{E}_{\boldsymbol{\pi}}\left[\sum_{l=1}^{\infty} \gamma^l r(s_{t+l}, \boldsymbol{a_{t+l}})\right]$$

Ecuación 3: Cálculo de la función de estado-acción. Fuente: [22]

La **r** hace referencia a la función que dado un estado y una acción devuelve la recompensa asociada, mientras que la **γ**, es un factor de reducción, su rango de valores va desde el cero (incluido) hasta el 1 (excluido), $\gamma \in [0, 1)$.

Hay un punto que se comenta en los artículos citados, que es el de absorber estados. Este es un estado en el que se suele poner los agentes inactivos, es decir cuando un agente se ha eliminado del entorno, pero el mundo sigue en ejecución con otros agentes. Esto está conectado con el problema *CAP*, el hecho de poner un agente en este estado, es para generar un camino en el

espacio de estados para poder propagar las recompensas al agente eliminado. Sin embargo, en el caso del *MA-POCA*, ya que este número de estados puede crecer bastante cuantos más agentes tengamos en nuestro entorno, Unity ha preferido utilizar una capa de neuronas de **atención** [25]. No vamos a entrar en mucho detalle en este tipo de neuronas, pero digamos que se utilizan para poder memorizar algo de información. Con esto vamos a poder atribuir la recompensa de un futuro a agentes que terminaron su ejecución.

Otro aspecto frecuente en la literatura es el hecho de distinguir qué agente es el que más ha contribuido a conseguir la recompensa grupal. Hasta ahora hemos hablado de recompensa como algo que va a recibir un agente individual, pero en un entorno de cooperación, vamos a tener un conjunto de agentes que van a pertenecer a un equipo. Por tanto, estos agentes pueden recibir recompensas de forma individual, pero también pueden recibir recompensas grupales. Poniendo el caso de nuestro entorno, cuando un agente golpee la pelota, éste va a recibir una recompensa positiva de forma individual, y cuando un equipo gane un punto, la recompensa va a ser a nivel de equipo. Un aspecto relevante pues es determinar cuál ha sido la contribución de cada agente a cada recompensa. En los artículos se refieren a este punto como *Counterfactual Baselines*, para tratar de reflejar la contribución individual de los agentes en la recompensa total. No hemos entrado en mucho detalle en los cálculos del algoritmo, ya que se considera que están fuera del objetivo del proyecto; sin embargo, vamos a explicar la parametrización del *baseline* para poder comentar unos puntos que se utilizan en la arquitectura.

$$Q_\psi(RSA(g_j(o_t^j), f_i(o_t^i, a_t^i)_{\substack{1 \leq i \leq k_t \\ i \neq j}}))$$

Figura 9: Cálculo del baseline para el agente j. Fuente: [22]

Como bien podemos observar en el cálculo de la figura 9, se utiliza un bloque RSA (algoritmo de cifrado), y no solo se utiliza aquí, sino que en muchos otros cálculos de la arquitectura utiliza este sistema. Luego tenemos dos funciones, la *g* que recibe las observaciones de un agente, y la *f* obtiene tantos las observaciones como la acción. Estas funciones realmente son unas redes para codificar la entrada que reciben.

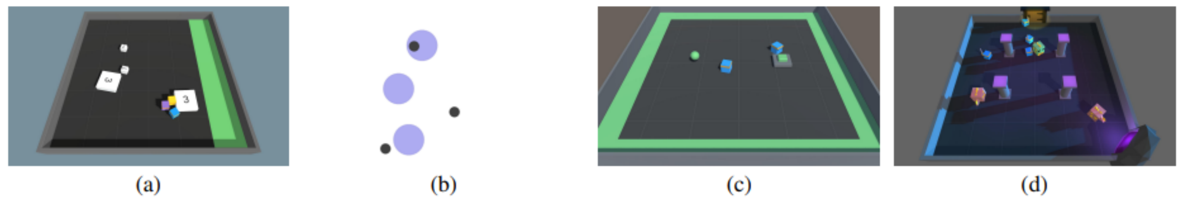


Figure 2: **(a) Collaborative Push Block**. Agents (blue, yellow, purple) must push white blocks to green area; larger blocks require more agents to push. **(b) Simple Spread**. Agents (large circles) must move to cover targets (small circles) without colliding with one another. **(c) Baton Pass**. Blue agents must grab green food and hit green button to spawn another agent, who can grab the next food, and so on. **(d) Dungeon Escape**. Blue agents must kill green dragon by sacrificing one of them to reveal a key. Teammates must pick up key and reach the door, while avoiding pink dragons.

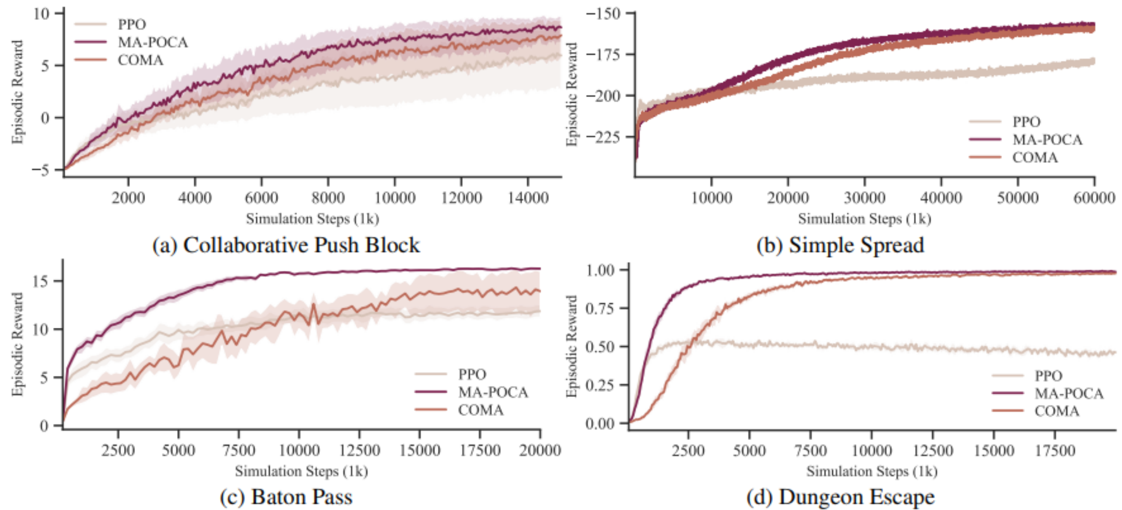


Figura 10: Resultados del estudio del algoritmo MA-POCA. Fuente: [22]

La descripción de la propia figura 10, nos describe brevemente cual es el objetivo de los distintos entornos. En el caso del (a), los agentes deben empujar los bloques blancos a las áreas verdes, y cuanto más grandes sean estos bloques más agentes se necesitaran para poder empujarlos. Los agentes (círculos grandes) del entorno (b) deben cubrir los objetivos (círculos pequeños) sin colisionar entre ellos. En el ejemplo (c), los agentes deben coger la comida verde y pulsar el botón para generar otro agente, el cual podrá coger la siguiente comida. Por último, en el entorno (d), los agentes de color azul deben matar los dragones verdes haciendo que uno de los agentes se sacrifique para mostrar una llave, los demás compañeros deberán llevar la llave hasta la puerta evitando a los dragones rosas.

En la anterior figura nos encontramos con los resultados del estudio del algoritmo MA-POCA. Se compara tanto con PPO, que es uno de los algoritmos que aunque sea sencillo puede conseguir buenos resultados, y con COMA, que es el algoritmo del que parte Unity para generar MA-POCA (definido en [24]). Estos algoritmos se han probado en cuatro entornos diferentes, y un factor importante es que tenemos entornos donde el número de los agentes es constante, y otros donde puede aumentar o disminuir. Esto es importante porque al estudiar este algoritmo se ha hecho mucho énfasis

en trabajar con entornos donde el número de agentes puede variar durante la ejecución, pero no es el caso del entorno que nosotros vamos a construir. En todas estas situaciones el algoritmo de *MA-POCA* es el que ha obtenido mejores resultados, *COMA* no se queda tan atrás, pero *PPO* en los entornos que no son tan triviales como el de la *dungeon escape*, es la mitad de eficiente que los demás.

3.3.3 Resumen de los algoritmos

Para finalizar vamos a generar una tabla con los algoritmos que hemos explicado, y que así podamos clasificarlos y compararlos rápidamente.

Agnósticos del entorno	No agnósticos del entorno
Proximal policy optimization (PPO)	Self-Play (entorno multi-agente competitivo)
Soft Actor-Critic (SAC)	MA-POCA (entorno multi-agente cooperativo)
Generative Adversarial Imitation Learning (GAIL)	Curriculum Learning
Behavioral Cloning (BC)	Environment Parameter Randomization

Tabla 2: Comparativa de los algoritmos de ML-agents.

4. Alcance

A continuación definiremos el alcance y objetivos del proyecto, el cual tiene una componente de investigación relevante. Dentro de este tipo de proyectos, se pueden encontrar muchas funcionalidades opcionales, por lo que trataré de definir cuales son las que puntualizamos como principales y que se deberán implementar obligatoriamente.

4.1 Objetivos y subobjetivos

El principal objetivo es programar un videojuego de padel con un agente unido e integrado al entorno virtual. El proyecto se divide en primero implementar el videojuego y comprobar que funciona correctamente, para luego implementar y añadir el agente al entorno previamente generado. Sin embargo, hay que tener en cuenta que en la primera iteración se desarrollará un juego más simple, por ejemplo, sólo se dispondrá de un tipo de golpe, para así poder comprobar que el agente entrena correctamente. Entonces más adelante se implementarán otros tipos de golpe (globo, remate u otros).

Cabe destacar que el proyecto se centra en el aprendizaje y comportamiento de los agentes en cuanto a su posición y movimiento en la pista. En ningún momento se está buscando una representación totalmente realista. Por ello, los modelos que se utilizarán para la pista y para los jugadores serán sencillos, se utilizará una versión simplificada del reglamento de pádel (por ejemplo, impidiendo la devolución de la pelota fuera de pista) y se utilizará una simulación básica de la trayectoria de la pelota, ignorando por ejemplo las fuerzas de fricción.

También se harán distintas iteraciones dentro del apartado de los agentes. Primero trabajaremos en partidos uno contra uno, para luego estudiar partidos dos contra dos, donde se deberá informar al agente que dispone de un compañero de equipo. Lo cual nos lleva a decidir sobre qué información debemos mostrar al agente. Tanto esta parte como el sistema de recompensas contienen una componente de investigación, la cual se resolverá a partir de prueba y error. Por ejemplo, la información que considero básica para este tipo de agentes es: posición y velocidad del agente, posición y velocidad de la pelota. Un sistema de recompensas inicial podría ser, otorgar 0.1 puntos cada vez que se golpee a la pelota, incrementar 1 cada vez que gane un punto o restarlo si se ha perdido.

4.2 Requerimientos

4.2.1 Requerimientos funcionales

Algunos de estos requisitos se han definido en el apartado anterior, pero también debemos recordar que la implementación de la librería *ML-agents* está diseñada para **Python**, por lo que deberemos tenerlo instalado.
Contemplar además, esos distintos modos que se han comentado brevemente, donde todos los jugadores son manipulados por agentes, o la posibilidad de que un jugador sea controlado por una persona real.

4.2.2 Requerimientos no funcionales

Un requisito no funcional a destacar es el tiempo de ejecución a la hora de entrenar estos agentes, ya que no se puede llegar a estimar cuánto necesitaremos para obtener buenos resultados (el problema de generar la arquitectura de las redes neuronales es NP-hard).
En cuanto a reusabilidad, la idea es que cuando tengamos algunos modelos de agentes funcionales, almacenarlos para así poder utilizarlos cuando queramos, pero también tendremos la posibilidad de entrenar a agentes nuevos, existiendo la posibilidad que comiencen con el conocimiento de un agente entrenado o desde cero.

4.3 Obstáculos y riesgos

Algunos de los riesgos del proyecto ya se han ido identificando en el documento. El hecho de que las redes neuronales sea un problema NP-hard, no podemos encontrar una arquitectura perfecta ni unos parámetros perfectamente adecuados para un problema.

Además la fase de entrenamiento supondrá una gran cantidad de tiempo del proyecto, donde no sabemos con certeza si el sistema de recompensas creado y la información que recibe el agente será suficiente para obtener el resultado que esperamos. Al no existir un ejemplo de pádel, existe una componente importante de exploración para dichos parámetros.

Una solución para el entrenamiento, es el paso de dividir el proyecto en primero una iteración más sencilla, con solo un tipo de golpe, antes de abordar el partido en equipo. Otra solución que también se puede aplicar al entrenamiento es el uso de servicios externos, como por ejemplo ***Microsoft Azure*** o ***Amazon Web Services***.

5. Metodología

El desarrollo de esta aplicación se divide principalmente en tres pasos: aprender el funcionamiento de las herramientas a utilizar, la implementación del videojuego, y la creación y unión del agente con el entorno del mundo virtual. En este último punto entra en juego la fase de investigación, para que mediante el entrenamiento del agente obtengamos un comportamiento correcto.

Aunque hayamos dividido el proyecto en esos puntos, estos se van a subdividir con tareas más específicas, (se visualizará mejor en el diagrama de Gantt), todas comparten la idea de la metodología ágil de desarrollo (figura 11). La idea siempre será: diseñar, implementar y probar cada funcionalidad nueva. Por tanto lo que puede ser una tarea como implementar un jugador, se fragmentará en los tres puntos que sigue la metodología ágil. Para luego presentar los resultados de forma semanal al director, para entonces seguir con lo que se había planeado primeramente o modificarlo.

Figura 11: Pasos a seguir en una metodología ágil. Fuente: [7]

Todo el proyecto se desarrollará mediante el equipo personal del que dispongo, el cual debería ser suficiente para entrenar a los agentes.

A la hora de comprobar que las funcionalidades se comportan correctamente, se les aplicará un conjunto de pruebas diseñadas por mí, pero también se le preguntará al director del proyecto para algunas comprobaciones adicionales, tratando así de reducir al máximo el posible número de errores que pueda contener la aplicación.

6. Herramientas

Las herramientas que se utilizaran para desarrollar la aplicación, ya se han ido comentando y será Unity y su librería ML-agents. La documentación necesaria para implementar el proyecto es la que se puede encontrar en [5] para la librería, y [8] para el entorno del videojuego.

Para las reuniones semanales con el director, se utilizará Google Calendar para definir qué días se harán, y Google Meet para llevarlas a cabo.

En lo que se refiere a la parte de código, se ha utilizado *Visual studio code* [27] como editor ya que es el que Unity utiliza por defecto.

Las gráficas que se han generado para mostrar los resultados de los entrenamientos, se han creado de dos formas distintas. La primera ha sido con *tensorboard* [28], la cual genera de forma automática un conjunto de gráficas a partir de los datos del entrenamiento. Con estas podremos conocer cómo ha ido evolucionando la recompensa a lo largo del tiempo, y cómo ha variado la duración de las iteraciones. La segunda ha sido mediante *Google Colab* [29], se recogen los datos almacenados en el vector de observaciones, para así poder mostrar los partidos jugados por los agentes. En este caso se ha utilizado *seaborn* [30] para generar las gráficas.

Además, para la construcción de la planificación, se diseñará un diagrama de *gantt,* mediante Gantter [9].

7. Planificación temporal

Para llevar un buen control del proyecto y saber cómo estamos avanzando, es muy importante llevar una buena planificación. Vamos a explicar la planificación de este trabajo, explicando cada uno de sus apartados.

El trabajo comenzó a finales de enero de 2022, y la entrega de la memoria está definida para el 22 de junio, mientras que la presentación se hará el 29. Contando que se termina el proyecto el día 27 de junio, contamos con un total de 150 días aproximadamente y una duración estimada de 482 horas.

Se espera una dedicación diaria de 3 horas aproximadamente, aunque es probable que los fines de semana sea cuando se pueda dedicar más tiempo al proyecto.

7.1 Descripción inicial de las tareas

En este apartado se documentan las distintas tareas en las que se ha dividido la planificación del proyecto. Dentro de estas tareas, encontramos subtareas en las que se ha dividido la más general. En la figura 12 se pueden encontrar todas las tareas y subtareas con toda la información, como puede ser la duración y dependencias. Además, en la figura 49 (anexo) mostramos toda esta planificación con un diagrama de Gantt.

7.1.1 Gestión del proyecto

En este apartado se contempla todo lo relacionado con planificar y documentar el proyecto, y reuniones para comprobar cómo avanzamos. Aproximadamente se invertirán unas 155 horas.

Alcance (GP1)

Antes de nada debemos saber que vamos a hacer, el objetivo, que se queda dentro o fuera del proyecto, y las herramientas que se utilizarán (previamente se habrá estudiado entre las distintas herramientas disponibles, comparándolas y explicando porqué hemos elegido una u otra). La duración aproximada es de 25 horas.

Planificación (GP2)

Para todas las tareas definidas en el alcance del proyecto, hacemos una planificación con sus respectivos materiales y requerimientos. También explicamos los riesgos y algunas soluciones que podemos tomar si surgen. Se utilizarán unas 10 horas.

Presupuesto (GP3)

Para cuantificar el proyecto, se va a realizar un estudio de su coste. Tendremos en cuenta los costes de personal, aplicado a los distintos tipos de profesionales envueltos, y el hardware necesario. Se utilizarán unas 5 horas.

Informe de sostenibilidad (GP4)

Se estudiará el impacto medioambiental, económico y social. El tiempo dedicado será de 10 horas.

Reuniones (GP5)

La idea es tener una reunión con el director y el codirector semanalmente o de forma quincenal. El objetivo de estas reuniones es presentar el trabajo realizado en ese tiempo, y analizar si estamos consiguiendo las tareas planificadas en el tiempo que le asignamos. Aproximadamente se invertirán unas 30 horas.

Documentación (GP6)

En la entrega del proyecto, se debe adjuntar una memoria, la cual se irá documentando a lo largo de la implementación, es decir se irá trabajando paralelamente en la implementación y en la documentación. Se espera un total de 60 horas para ello.

Presentación (GP7)

La parte final de este proyecto es la presentación hacia el tribunal que lo evaluará. Por tanto, debemos preparar correctamente dicha presentación, con el material que utilizaremos y algún ensayo. Para llevar a cabo una correcta presentación, se esperan unas 15 horas invertidas.

7.1.2 Trabajo previo

Dentro de este apartado se comenta que es lo que se ha hecho antes de comenzar a la implementación del proyecto. Por tanto, engloba tanto el estudio de las distintas herramientas que se pueden utilizar, la instalación de las herramientas escogidas, y el estudio de la documentación y ejemplos de dichas herramientas. Aproximadamente se invertirán unas 57 horas.

Estudio de las posibles herramientas (TP1)

Tenemos que investigar qué motores gráficos hay en el mercado (open-source), y cómo implementar una red neuronal para que podamos conectar el entorno del juego con la red. Por eso mismo se eligió Unity, ya que con la librería ML-agents se nos facilita esa conexión. Se estima unas 5 horas.

Instalación de las herramientas (TP2)

Después de escoger qué herramientas se utilizaran en la implementación del proyecto, debemos instalarlas. Para ello nos tendremos que instalar Unity, la librería ML-agents, Python, Pytorch, y otras dependencias para poder entrenar la red mediante GPU (como CUDA). Se espera invertir unas 2 horas.

Estudio de la documentación y ejemplos (TP3)

Para poder trabajar correctamente, se prefiere hacer un estudio previo de las herramientas, para luego estar más familiarizado. Es muy importante además de leer la documentación oficial, ver distintos ejemplos donde también se utiliza la librería ML-agents, para tener una ligera idea de cómo funciona. Cabe destacar que este estudio no es simplemente previo, es decir, mientras estemos implementado, también deberemos buscar información en la documentación, o visualizar distintos ejemplos de donde podemos captar ideas para nuestro trabajo. Se invertirán unas 50 horas.

7.1.3 Desarrollo del entorno videojuego

El proyecto se ha dividido en dos tareas principales, una es la creación del entorno del videojuego, con distintas subtareas, y luego la implementación del agente y el entrenamiento. En esta primera parte se espera una duración de 125 horas.

Implementación básica del entorno (DEV1)

La primera iteración es el desarrollo básico del videojuego, es decir se contempla tanto la creación de la pista como la de los jugadores, y la implementación de las distintas acciones básicas del jugador (movimiento y un tipo de golpe). También es posible que se implemente un bot tanto para probar el juego como para la fase de entrenamiento de los agentes. Como

seguimos una metodología ágil, este apartado se separa en tres, diseño (20 horas), desarrollo (30 horas) y testeo (20 horas).

Implementación extra del entorno (DEV2)

Después del desarrollo básico del entorno, y de haber creado y entrenado un agente con dichas características, se expandirán los tipos de golpe que podrá usar el agente. Se estima unas 10 horas de diseño, 15 horas de desarrollo y 10 horas para el testeo.

Implementación de un posible menú (DEV3)

Al no ser de una gran importancia, simplemente se cuenta como una posible implementación extra, pero si se puede, es deseable generar un menú con el que interactuar para cambiar entre los distintos modos de juego. Se espera unas 5 horas para el diseño, 10 para el desarrollo y otras 5 para el testeo.

7.1.4 Desarrollo del agente

En esta segunda tarea se va a implementar y entrenar los distintos agentes, primero partiremos del agente con un solo tipo de golpe, y cuando comprobemos que entrena correctamente y obtenemos un buen resultado, entonces implementaremos los nuevos golpes para luego añadirlos al nuevo agente. Se espera unas 145 horas.

Implementación del agente básico (DA1)

Después de tener el entorno, se generará todo lo necesario para implementar el agente y que este pueda entrenar. De igual forma se separará en tres, con unas 10 horas para el diseño, 15 para la implementación y 10 para el testeo.

Entrenamiento del agente básico (DA2)

Lo siguiente a implementar y probar el agente, es entrenarlo y, parte muy importante, comprobar que el entrenamiento está siendo efectivo, ya que si no es así, habrá que modificar algo, como podría ser el sistema de recomendación (dicha comprobación se incluye en el apartado anterior). Se decide separar esta fase del apartado DA1, porque supone una gran cantidad de horas. Se invertirán unas 30 horas.

Implementación del agente extra (DA3)

Al comprobar que el agente simple funciona correctamente, se implementarán los nuevos golpes para luego crear un nuevo agente y añadirlos. Se esperan unas 5 horas de diseño, 10 de desarrollo y 15 de testeo.

Entrenamiento del agente extra (DA4)

Ahora deberemos entrenar a este nuevo agente, el cual tendrá nuevas acciones entre las cuales podrá elegir. Se estiman unas 50 horas.

7.2 Recursos

7.2.1 Recursos humanos

Asignamos las distintas tareas a cuatro roles, los cuales son los profesionales necesarios para desarrollar dicho proyecto. Destacar que al hacer un juego simple, no contemplamos el hecho de contratar a un profesional para la generación de animaciones o modelos.

1. Jefe de proyecto: Planifica, lidera y documenta el proyecto.
2. Investigador: Estudia sobre qué herramientas se puede llevar a cabo el trabajo y diseña distintas técnicas.
3. Programador: Encargado de implementar el entorno y el agente.
4. Tester: Responsable de comprobar que toda implementación funciona correctamente.

7.2.2 Recursos materiales

Para llevar a cabo todo el desarrollo del trabajo, necesitamos un equipo con algunas especificaciones mínimas, además de las distintas herramientas que ya hemos ido comentando.

1. Ordenador: Equipo necesario para llevar a cabo la implementación del proyecto.
2. Unity: Motor gráfico con el cual se desarrollará el entorno. [8]
3. ML-agents: librería de Unity con la cual implementaremos el agente. [4][5]
4. Gantter: Herramienta para crear diagramas de Gantt. [9]

7.3 Gestión del riesgo

Al implementar dichos proyectos es probable que surjan algunos problemas, y por eso mismo debemos pensar cuales son los riesgos, y cómo podemos solucionarlos para el caso en que aparezcan.

1. Dificultades imprevistas: Ya que se trabaja con herramientas con las que no somos familiarizados, es posible que no podamos implementar todo lo que queramos en el tiempo esperado, por eso mismo se ha decidido diseñar un modelo más sencillo, para luego si funciona implementar uno con funcionalidades extra.

2. Recursos limitados y/o rendimiento: Cuando trabajamos con redes neuronales, se debe tener en cuenta, que una gran parte del trabajo es el entrenamiento de estas. Si se detecta que el equipo personal no es suficiente para entrenarlas, se podrían entrenar mediante **Amazon Web Services** o **Microsoft Azure.**

3. Estimación del tiempo: Es posible que la estimación que hemos hecho para las distintas tareas, sea de más, o por el contrario, falten horas. Por eso mismo la idea es dejar unos días entre medias del trabajo para mediar con la posible falta de tiempo para el desarrollo de algunas fases. Y entonces volver a hacer de nuevo las estimaciones de las siguientes tareas, para ir ajustando la planificación.

	Nombre	Recursos	Tiempo	Dependencias
1	**⊟Gestión del proyecto - GP**	-	**155**	-
2	Alcance - GP1	Ordenador	25	TP1
3	Planificación - GP2	Ordenador, Gantter	10	GP1
4	Presupuesto - GP3	Ordenador	5	GP1
5	Informe sostenibilidad - GP4	Ordenador	10	GP1
6	Reuniones - GP5	Ordenador	30	-
7	Documentación - GP6	Ordenador	60	-
8	Presentación - GP7	Ordenador	15	-
9	**⊟Trabajo previo - TP**	-	**57**	-
10	Estudio herramientas - TP1	Ordenador	5	-
11	Instalación herramientas - TP2	Ordenador	2	TP1
12	Estudio documentación - TP3	Ordenador	50	TP1
13	**⊟Desarrollo entorno - DEV**	-	**125**	**TP**
14	**⊟Implementación básica entorno - DEV1**	**Ordenador, Unity**	**70**	**TP**
15	Diseño DEV1 - DEV1.1	Ordenador, Unity	20	-
16	Desarrollo DEV1 - DEV1.2	Ordenador, Unity	30	DEV1.1
17	Testeo DEV1 - DEV1.3	Ordenador, Unity	20	DEV1.2
18	**⊟Implementación extra entorno - DEV2**	**Ordenador, Unity**	**35**	**DEV1**
19	Diseño DEV2 - DEV2.1	Ordenador, Unity	10	-
20	Desarrollo DEV2 - DEV2.2	Ordenador, Unity	15	DEV2.1
21	Testeo DEV2 - DEV2.3	Ordenador, Unity	10	DEV2.2
22	**⊟Implementación menú - DEV3**	**Ordenador, Unity**	**20**	**TP**
23	Diseño DEV3 - DEV3.1	Ordenador, Unity	5	-
24	Desarrollo DEV3 - DEV3.2	Ordenador, Unity	10	DEV3.1
25	Testeo DEV3 - DEV3.3	Ordenador, Unity	5	DEV3.2
26	**⊟Desarrollo agente - DA**	**Ordenador, Unity, ML-agents**	**145**	**TP**
27	**⊟Implementación agente básico - DA1**	**Ordenador, Unity, ML-agents**	**35**	**DEV1**
28	Diseño DA1 - DA1.1	Ordenador, Unity, ML-agents	10	-
29	Desarrollo DA1 - DA1.2	Ordenador, Unity, ML-agents	15	DA1.1
30	Testeo DA1 - DA1.3	Ordenador, Unity, ML-agents	10	DA1.2
31	Entreno agente básico - DA2	Ordenador, Unity, ML-agents	30	DA1
32	**⊟Implementación agente extra - DA3**	**Ordenador, Unity, ML-agents**	**30**	**DEV2**
33	Diseño DA3 - DA3.1	Ordenador, Unity, ML-agents	5	-
34	Desarrollo DA3 - DA3.2	Ordenador, Unity, ML-agents	10	DA3.1
35	Testeo DA3 - DA3.3	Ordenador, Unity, ML-agents	15	DA3.2
36	Entreno agente extra - DA4	Ordenador, Unity, ML-agents	50	DA3

Figura 12: Tabla de tareas con los recursos, duración y dependencias. Generada mediante Gantter. [9]

7.4 Cambios en la planificación inicial

Realmente no ha habido grandes cambios respecto a la planificación inicial. Lo primero es que se ha eliminado la parte de implementación de un menú,

ya que esta funcionalidad no es de vital importancia y se decidió invertir el tiempo en puntos más importantes.
La otra diferencia es que algunas tareas han llevado más tiempo del que se estimó, sobre todo las que tienen que ver con el entorno extra, tanto implementación del entorno como implementación y entrenamiento del agente. En cuanto a horas totales, en total se han invertido unas 657 a diferencia de las 482 que se habían estimado; el añadido de horas viene sobre todo por los distintos agentes que se han ido entrenando, ya que se tuvo que invertir bastante más tiempo del que se previó inicialmente.

El desglose de horas actualizado se puede comprobar en la siguiente tabla.

	Nombre	Recursos	Tiempo	Dependencias
1	⊟**Gestión del proyecto - GP**	-	**155**	-
2	Alcance - GP1	Ordenador	25	TP1
3	Planificación - GP2	Ordenador, Gantter	10	GP1
4	Presupuesto - GP3	Ordenador	5	GP1
5	Informe sostenibilidad - GP4	Ordenador	10	GP1
6	Reuniones - GP5	Ordenador	30	-
7	Documentación - GP6	Ordenador	60	-
8	Presentación - GP7	Ordenador	15	-
9	⊟**Trabajo previo - TP**	-	**57**	-
10	Estudio herramientas - TP1	Ordenador	5	-
11	Instalación herramientas - TP2	Ordenador	2	TP1
12	Estudio documentación - TP3	Ordenador	50	TP1
13	⊟**Desarrollo entorno - DEV**	-	**120**	**TP**
14	⊟**Implementación básica entorno - DEV1**	**Ordenador, Unity**	**85**	**TP**
15	Diseño DEV1 - DEV1.1	Ordenador, Unity	25	-
16	Desarrollo DEV1 - DEV1.2	Ordenador, Unity	35	DEV1.1
17	Testeo DEV1 - DEV1.3	Ordenador, Unity	25	DEV1.2
18	⊟**Implementación extra entorno - DEV2**	**Ordenador, Unity**	**35**	**DEV1**
19	Diseño DEV2 - DEV2.1	Ordenador, Unity	10	-
20	Desarrollo DEV2 - DEV2.2	Ordenador, Unity	15	DEV2.1
21	Testeo DEV2 - DEV2.3	Ordenador, Unity	10	DEV2.2
22	⊟**Desarrollo agente - DA**	**Ordenador, Unity, ML-agents**	**325**	**TP**
23	⊟**Implementación agente básico - DA1**	**Ordenador, Unity, ML-agents**	**45**	**DEV1**
24	Diseño DA1 - DA1.1	Ordenador, Unity, ML-agents	15	-
25	Desarrollo DA1 - DA1.2	Ordenador, Unity, ML-agents	20	DA1.1
26	Testeo DA1 - DA1.3	Ordenador, Unity, ML-agents	10	DA1.2
27	Entreno agente básico - DA2	Ordenador, Unity, ML-agents	100	DA1
28	⊟**Implementación agente extra - DA3**	**Ordenador, Unity, ML-agents**	**30**	**DA2**
29	Diseño DA3 - DA3.1	Ordenador, Unity, ML-agents	5	-
30	Desarrollo DA3 - DA3.2	Ordenador, Unity, ML-agents	10	DA3.1
31	Testeo DA3 - DA3.3	Ordenador, Unity, ML-agents	15	DA3.2
32	Entreno agente extra - DA4	Ordenador, Unity, ML-agents	150	DA3
33	TOTAL	-	657	-

Figura 13 : Tabla de tareas actualizada con los recursos, duración y dependencias. Generada mediante Gantter. [9]

Si luego nos fijamos en el diagrama de Gantt actualizado (figura 50), podemos observar que las fechas de las tareas no han variado, ya que a nivel de tareas sí que nos hemos ceñido al plan inicial. Por tanto, a nivel de

implementación se finalizó a mediados del mes de Mayo para poder enfocarnos de pleno en la memoria. Obviamente, la documentación se comenzó antes de esta fecha, aprovechando muchas veces el hecho de dejar entrenando un agente para documentar el trabajo.

8. Gestión económica

En esta sección se van a definir los costes estimados del proyecto. Por una parte nos encontramos con el coste de personal, es decir para los distintos profesionales que definimos en la parte de planificación, coste de herramientas, coste de equipo (hardware) y del espacio donde se trabajará. Además, para los posibles problemas que definimos en el apartado de riesgos, se realizará un plan de contingencias, una partida de imprevistos y puntos para controlar el presupuesto.

Este apartado se ha actualizado solo con el presupuesto final, donde la diferencia con el inicial viene dado por la falta de horas estimadas. En la planificación inicial nos había salido un presupuesto de 19.775,86€.

8.1 Presupuesto

8.1.1 Costes de personal por actividad

Calculamos el coste por hora de los distintos roles identificados. Destacar que son sueldos brutos, incluyendo el IRPF y la seguridad social, para ello cogemos el coste bruto y lo multiplicamos por 1.3. Comentar que todos los cálculos se harán con el sueldo teniendo en cuenta la seguridad social.

Rol	Coste/hora	Coste SS/hora
Jefe de proyecto	30€/h	39€/h
Investigador	20€/h	26€/h
Programador	16€/h	20.8€/h
Tester	16€/h	20.8€/h

Tabla 3: Costes por hora del personal. Información obtenida de un ejemplo de años anteriores.

En la figura 14 se calcula cual es el coste total de personal, para ello aprovechamos la tabla que mostramos en la planificación, añadiendo la columna de los costes (teniendo en cuenta el coste de la seguridad social).

Para ello debemos saber que rol será el que trabajará para cada tarea definida.
En el caso de la gestión del proyecto, todo se realizará por el jefe de proyecto, a excepción de las reuniones que estarán todos, y la documentación que estará tanto el jefe de proyecto como el investigador.
El estudio de las herramientas será realizado por el investigador, mientras que la instalación y el estudio de la documentación se encargará el programador.
Todas las fases de diseño son para el investigador, el desarrollo para el programador y el testeo para el tester. Los apartados de entrenar al agente también están destinados para el programador.

	Nombre	Recursos	Tiempo	Dependencias	Coste SS	Roles
1	⊟**Gestión del proyecto - GP**	-	**155**	-	**9633**	
2	Alcance - GP1	Ordenador	25	TP1	975	JP
3	Planificación - GP2	Ordenador, Gantter	10	GP1	390	JP
4	Presupuesto - GP3	Ordenador	5	GP1	195	JP
5	Informe sostenibilidad - GP4	Ordenador	10	GP1	390	JP
6	Reuniones - GP5	Ordenador	30	-	3198	JP, I, P, T
7	Documentación - GP6	Ordenador	60	-	3900	JP, I
8	Presentación - GP7	Ordenador	15	-	585	JP
9	⊟**Trabajo previo - TP**	-	**57**	-	**1211.6**	
10	Estudio herramientas - TP1	Ordenador	5	-	130	I
11	Instalación herramientas - TP2	Ordenador	2	TP1	41.6	P
12	Estudio documentación - TP3	Ordenador	50	TP1	1040	P
13	⊟**Desarrollo entorno - DEV**	-	**120**	**TP**	**2678**	
14	⊟**Implementación básica entorno - DEV1**	**Ordenador, Unity**	**85**	**TP**	**1898**	
15	Diseño DEV1 - DEV1.1	Ordenador, Unity	25	-	650	I
16	Desarrollo DEV1 - DEV1.2	Ordenador, Unity	35	DEV1.1	728	P
17	Testeo DEV1 - DEV1.3	Ordenador, Unity	25	DEV1.2	520	T
18	⊟**Implementación extra entorno - DEV2**	**Ordenador, Unity**	**35**	**DEV1**	**780**	
19	Diseño DEV2 - DEV2.1	Ordenador, Unity	10	-	260	I
20	Desarrollo DEV2 - DEV2.2	Ordenador, Unity	15	DEV2.1	312	P
21	Testeo DEV2 - DEV2.3	Ordenador, Unity	10	DEV2.2	208	T
22	⊟**Desarrollo agente - DA**	**Ordenador, Unity, ML-agents**	**325**	**TP**	**6864**	
23	⊟**Implementación agente básico - DA1**	**Ordenador, Unity, ML-agents**	**45**	**DEV1**	**1014**	
24	Diseño DA1 - DA1.1	Ordenador, Unity, ML-agents	15	-	390	I
25	Desarrollo DA1 - DA1.2	Ordenador, Unity, ML-agents	20	DA1.1	416	P
26	Testeo DA1 - DA1.3	Ordenador, Unity, ML-agents	10	DA1.2	208	T
27	Entreno agente básico - DA2	Ordenador, Unity, ML-agents	100	DA1	2080	P
28	⊟**Implementación agente extra - DA3**	**Ordenador, Unity, ML-agents**	**30**	**DA2**	**650**	
29	Diseño DA3 - DA3.1	Ordenador, Unity, ML-agents	5	-	130	I
30	Desarrollo DA3 - DA3.2	Ordenador, Unity, ML-agents	10	DA3.1	208	P
31	Testeo DA3 - DA3.3	Ordenador, Unity, ML-agents	15	DA3.2	312	T
32	Entreno agente extra - DA4	Ordenador, Unity, ML-agents	150	DA3	3120	P
33	TOTAL	-	657	-	20386.6	

Figura 14: Contiene la tabla con el coste de personal teniendo en cuenta la seguridad social. Roles: JP - Jefe de proyecto; I - Investigador; P - programador; T - tester.

8.1.2 Costes genéricos

En el caso de los costes a partir de las herramientas utilizadas son nulos, ya que todas son gratuitas o open-source.

A nivel de hardware, hablamos que se va a utilizar un equipo propio, para el cual se va a tener en cuenta los componentes más importantes como son el procesador, tarjeta gráfica y memoria RAM.

- Procesador Ryzen 5 1600x. (250€) [10]
- Gráfica Asus Dual GeForce RTX 2060 Super. (600€) [11]
- Memoria RAM 2x8 GB DDR4. (70€) [12]

Si queremos tener la idea de un ordenador completo, nos podemos fijar en el equipo [13], que contiene componentes parecidos, es decir unos 1200€. Para calcular la amortización del mismo, tenemos en cuenta que un año tiene 220 días hábiles y 8 horas laborales al día, y una vida útil del equipo de 4 años. Por tanto si aplicamos la fórmula del coste por hora: *coste equipo / (vida útil * 220 * 8).* Obtenemos un coste por hora de 0.17€. Contando que el ordenador se utiliza para todas las fases del proyecto, se utiliza en un total de 482 horas, así que la amortización es de 82€.

También debemos tener en cuenta el espacio en el que se trabajará, en este caso yo simplemente voy a hacerlo desde mi casa. Ya que vivo en Gerona, para encontrar algo similar, nos podemos fijar en el precio de un espacio coworking, para que pudiéramos trabajar distintas personas.
Como bien podemos observar en el siguiente link, el precio de dicho lugar es de 150€ al mes. Por tanto, si contamos que aproximadamente el proyecto dura 5 meses, tenemos un coste de 750€. [14]

8.1.3 Contingencia

Al hablar de un proyecto es probable que algunas cosas no salgan tal y como se han planificado, por eso mismo es importante añadir un sobrecoste por si acaso. Fijamos un 10% de sobrecoste.

Tipo	Coste	Contingencia
Espacio coworking	750€	75€
Software	0€	0€
Hardware	82€	8.2€
Personal	20,386.6€	2,038.66€
Total	**21,218.6€**	**2,121.86€**

Tabla 4: Tabla que contiene el plan de contingencia del 10%. Elaboración propia.

8.1.4 Costes de los Imprevistos

Ahora calcularemos los costes de algunos imprevistos que pueden surgir a lo largo del trabajo. Definiremos cual es su coste añadido, y con qué porcentaje puede llegar a ocurrir dicho imprevisto.

1. Dificultades imprevistas: Necesitaríamos más horas de desarrollo y de comprobación. Añadiremos 30 horas de desarrollo y 20 de testeo, por lo tanto, 30 horas del programador y 20 del tester, un total de 1040€. La probabilidad de que surja es del 20%.
2. Recursos limitados y/o rendimiento: Para resolverlo utilizaremos **Amazon Web Services** o **Microsoft Azure**, en ambos casos nos referimos a la prueba gratuita que ofrecen. Por tanto el coste añadido sería de 0€. La probabilidad de necesitar estos servicios es del 10%
3. Estimación del tiempo: Este caso es similar al primero, pero ahora ha sido por una mala estimación de tiempo. Para resolverlo deberemos contratar durante más horas probablemente al programador y al tester. Digamos que de nuevo necesitaríamos 30 horas para el programador y 20 más para el tester. El coste total será de 1040€, con una probabilidad del 25%.

Imprevisto	Coste	Probabilidad	Coste total
Dificultades	1,040€	20%	208€
Recursos limitados	0€	10%	0€
Estimación del tiempo	1,040€	25%	260€
Total	**2,080€**	**-**	**468€**

Tabla 5: Tabla con el coste de los imprevistos. Elaboración propia.

8.1.5 Coste total

Después de definir todos los costes del trabajo, los vamos a agrupar para presentar el total. Como bien se puede observar en la tabla 5, el coste total del proyecto es de 23808.46€. La diferencia del presupuesto inicial con el final es de 4032.6€.

Tipo	Coste
Personal	20,386.6€

Software	0€
Hardware	82€
Espacio coworking	750€
Contingencia	2,121.86€
Imprevistos	468€
Total	**23,808.46€**

Tabla 6: Tabla con el coste total del proyecto. Elaboración propia.

8.2 Control de gestión

Aunque ahora ya tengamos definido cual es el presupuesto inicial del proyecto, esto puede llegar a variar, ya sea porque hemos sobreestimado o subestimado alguno de los costes. Por tanto, cada vez que se termine alguna tarea, se deberá imputar cuales han sido sus horas, y por tanto, su coste real. Para llevar el control de horas, se irán imputando en un excel, para luego calcular la desviación con la planificación y alomejor modificar alguna de las siguientes tareas para re-modificar la planificación inicial.
A continuación definimos los descriptores para el control, los cuales se aplicarán cada vez que se termine alguna tarea de las planificadas:

1. Desviación del coste de personal por tarea: (coste estimado - coste real) * horas reales.
2. Desviación de las horas de las tareas: (horas estimadas - horas reales) * coste real.
3. Desviación de las horas totales de las tareas: coste estimado total - coste real total.
4. Desviación total de los costes genéricos: coste estimado total - coste real total.
5. Desviación total de los imprevistos: coste estimado imprevistos - coste real imprevistos.
6. Desviación total de horas: horas estimadas - horas reales.

9. Sostenibilidad

En todos los proyectos se debe analizar la sostenibilidad del mismo, enfocándose en el ámbito económico, ambiental y social. En esta sección, encontramos una autoevaluación sobre la competencia de la sostenibilidad, para luego analizar los tres campos de la sostenibilidad a partir de unas preguntas.

9.1 Autoevaluación

Normalmente todo el mundo (donde me incluyo), solo pensamos en el ámbito económico al hablar sobre un proyecto. Ya que al comenzar a planificar un proyecto, siempre se suele hacer un estudio sobre el presupuesto inicial, para luego decidir si es algo rentable. Sin embargo, pocas veces se tiene en cuenta los otros dos ámbitos, el social y el ambiental.

Hay casos donde ni siquiera se hace un estudio sobre el impacto ambiental que puede suponer un proyecto, sin importar la cantidad de recursos que necesitan. Por eso mismo hay que ser más consciente del uso de algunos recursos, donde hay veces que los consumimos sin realmente necesitarlos. Esto además se verá reflejado en una reducción de los costes.

Todos los proyectos nacen con un objetivo, pero además hay veces que sin ser un objetivo principal, estos trabajos pueden ayudar a la sociedad. En este caso, se puede tomar como un ejemplo a la familiarización de distintas técnicas para el aprendizaje por refuerzo.

9.2 Dimensión económica

Considero que el presupuesto presentado es correcto, ya que se está trabajando con técnicas relativamente nuevas, y que además se debe hacer un pequeño estudio para su correcto funcionamiento. Es conocido que cada vez se utilizan más redes neuronales para nuevos proyectos, gracias al gran potencial que tienen. En este caso se aplican al aprendizaje por refuerzo. La resolución de este proyecto se podría enfocar de distintas maneras, a lo mejor haciendo un bot. La principal diferencia con una inteligencia artificial es que el comportamiento se consigue definiendo explícitamente en el código que queremos que haga, por tanto si está bien implementado funcionaria desde el segundo en que ejecutemos la aplicación, sin embargo, el resultado será menos real que el que se puede conseguir con una inteligencia artificial.

Se ha tratado de reducir el coste desde el inicio del trabajo. Por ejemplo, para trabajar se ha alquilado un espacio coworking en vez de una nave entera, el equipo con el que se ha desarrollado el proyecto no tiene un hardware de última generación, pero ya es suficiente para poder llevarlo a cabo, en la asignación de tareas se ha tratado de ser lo más realista posible, optimizando el hecho de asignar más o menos personas en distintas tareas. Hay otro coste, el cual no se ha cuantificado, que es el de la electricidad. Aunque no se ha tenido en cuenta para el cálculo del presupuesto, se ha tratado de minimizar mediante el trabajo en paralelo de distintas tareas a la vez.

La diferencia entre el presupuesto que se hizo inicialmente con el final es de 4032.6€. Esta diferencia aparece por las horas que se han añadido en la planificación, ya que se estimaron bastante menos en los entrenamientos de los distintos agentes.

En cuanto a la vida útil hay que diferenciar entre dos escenarios distintos. Si dejamos el desarrollo del proyecto parado en el punto en el que se va a presentar, no tendríamos porqué contar un coste añadido, ya que los modelos entrenados ya son funcionales. Sin embargo, si se quiere seguir trabajando con el proyecto, el coste extra vendría por las horas que se invertirán.

Hay algunos puntos que pueden hacer que tengamos mejores o peores resultados, pero no tanto como para perjudicar en la viabilidad del proyecto. Es decir, por el hecho de que no exista una única forma de desarrollar la aplicación, como por ejemplo la definición de las observaciones o el sistema de puntuación del agente. A través de prueba y error, se llegará hasta un punto donde el comportamiento del agente se asemeje al que se espera obtener.

9.3 Dimensión ambiental

En este caso, el proyecto no tiene un gran impacto medioambiental, ya que los recursos que se van a utilizar son, un solo ordenador, tanto para la parte de documentación como la de desarrollo, y la luz dentro de las horas invertidas para el trabajo. En otras ocasiones, se utiliza más de un equipo para distintas tareas, o más de una tarjeta gráfica. Además para reducir el impacto, se ha trabajo en más de una tarea a la vez, por ejemplo, a la vez que entrenamos un agente, se aprovecha para implementar la siguiente mejora de la aplicación o para documentar la memoria.

No creo que pudiera hacer este proyecto con menos recursos, ya que considero que los utilizados son los mínimos. Aunque es cierto que ahora con el conocimiento obtenido, podría hacer el trabajo de nuevo con un menor número de horas, ya que hay muchas de estas horas las cuáles se han utilizado en las pruebas por la parte de investigación que presenta el proyecto.

A nivel ambiental si comparamos esta solución con otras, podemos decir que no influirá tanto en la huella ambiental, ya que trabajamos con el mínimo equipo posible y tratando de minimizar el consumo de electricidad mediante el trabajo en paralelo de distintas tareas.

Otro punto positivo del proyecto es que si hablamos de su vida útil, solo tendríamos que invertir más recursos si queremos seguir con su desarrollo, pero gracias al conocimiento que tenemos ahora de las herramientas utilizadas, la cantidad de recursos a añadir sería menor. Además, gracias a la variedad de distintos agentes que tenemos entrenados, podemos entrenar alguno nuevo haciendo que parta con el conocimiento de estos (esta funcionalidad se ha utilizado para entrenar nuevas versiones de distintos agentes, haciendo que no tengamos que entrenar el agente desde cero, y ahorrando así distintos recursos).

Existe la posibilidad de que no obtengamos modelos con los comportamientos esperados, y esto se verá reflejado en una modificación en los parámetros de entrenamiento para luego entrenar un agente de nuevo. Haciendo que necesitemos un mayor consumo de electricidad y que entonces, aumente la huella ecológica del proyecto. O incluso, que algunos de los componentes del ordenador utilizado se estropeen, y se tengan que cambiar por nuevos.

9.4 Dimensión social

Desde antes de entrar a la universidad me gustaba y me interesaba por el mundo de los videojuegos, por eso mismo me parece muy interesante trabajar con **Unity**, siendo este uno de los motores gráficos más conocidos. Para entonces trabajar en el ámbito de la inteligencia artificial con su librería **ML-agents**, mundo el cual me llamó la atención desde que cursé la asignatura de Inteligencia Artificial.

Este proyecto no nace realmente de una necesidad explícita, pero, sin embargo, de forma implícita, va a ayudar a la familiarización de estas técnicas de una forma más amena, a través del mundo de los videojuegos, ya que podemos observar los resultados de las redes a nivel visual.

A través de la realización del proyecto se ha reflexionado a nivel personal si me gustaría dedicarme en un futuro por completo al campo de la investigación , o que en realidad prefiero el desarrollo de aplicaciones con nuevas herramientas/tecnologías.

Mediante este proyecto tenemos otro nuevo ejemplo donde se pueden aplicar las redes neuronales, por tanto los principales beneficiarios son, los desarrolladores que apuestan por estas nuevas tecnologías, el propio motor Unity y la librería ML-agents, y la industria de los videojuegos. Por otra parte, no considero que pueda haber algún colectivo que se vea perjudicado por este trabajo.

Si comparamos los resultados obtenidos con el problema que se planteó al inicio, hemos conseguido desarrollar agentes que han aprendido a jugar a pádel, pasando primero por algunas versiones más simples y en las cuáles los resultados han sido más satisfactorios. La parte que podemos decir ha funcionado peor, ha sido en el dos contra dos, ya que hay algunas ocasiones donde al agente no le parece importar el hecho de estar jugando con un compañero.

Existe la posibilidad que mediante el desarrollo de la aplicación se encuentre algún fallo de la librería, y por tanto que se vean afectados los creadores de esta. Incluso a lo mejor este fallo no nos permite continuar con el desarrollo de la aplicación. Aunque hay que decir que hasta las herramientas utilizadas han funcionado perfectamente.

10. Desarrollo del proyecto

10.1 Explicación del software utilizado

En este apartado se va a explicar más en detalle las herramientas que se van a utilizar para implementar el entorno y el agente.

Como motor gráfico vamos a utilizar Unity, ya que es *open-source* y muy reconocido en el ambiente de videojuegos. Otro de los principales motivos que se ha elegido Unity es por la librería *ML-agents*, la cúal permite conectar el entorno con el agente y con la red neuronal automáticamente.

A continuación se van a mostrar distintas imágenes de la interfaz de Unity, para explicar brevemente las distintas ventanas que disponemos y que función tiene cada una.

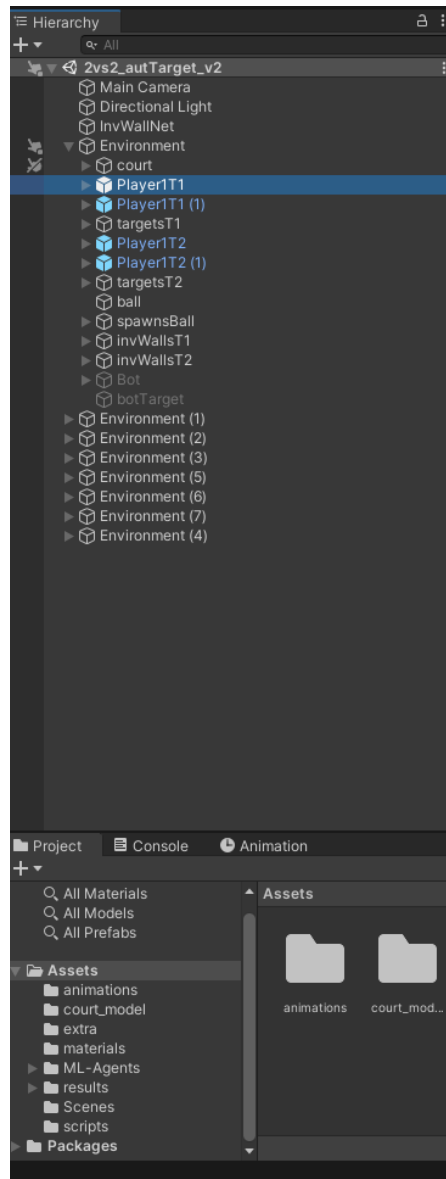


*Figura 15: Jerarquía de objetos y apartado de proyecto para los distintos archivos utilizados, **Console** para leer los fallos o mensajes y **Animation** para generar animaciones.*

En la figura 15 tenemos la jerarquía de objetos que tenemos en nuestra escena. Se puede observar que tenemos distintos objetos llamados *Environment*, donde cada uno de estos es un campo de pádel con sus jugadores.

Además, podemos listar objetos dentro de otros, formando la típica forma de árbol, en nuestro caso lo hemos utilizado para poder generar más de un entorno de forma rápida, y entonces utilizar los distintos agentes para el entrenamiento.

En la parte de debajo tenemos tres ventanas:

- Project: Lista todos los archivos que tenemos en el directorio del proyecto, facilitando la navegación entre los distintos elementos por si queremos abrir alguno de estos.
- Console: Típica consola donde podremos mostrar prints si los implementamos, y donde se nos notificará de los errores de compilación.
- Animation: Este apartado es el que se utiliza para generar las animaciones a partir del entorno de Unity.

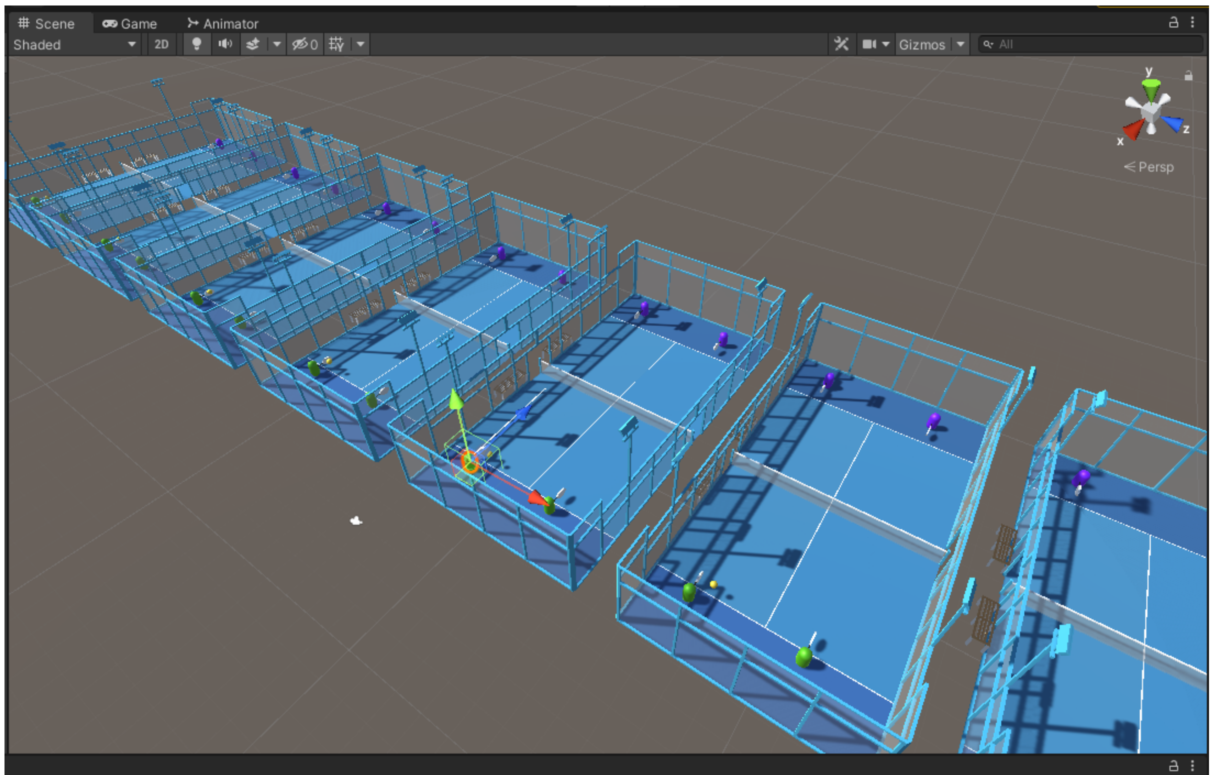


Figura 16: Imagen que muestra la ventana para visualizar la escena, la vista de la cámara y el animator para definir los estados y transiciones entre animaciones.

En la parte central de la interfaz tenemos otras tres ventanas:

- Scene: En esta ventana nos podemos mover por toda la escena que hemos montado o que vamos a montar, por tanto, se utiliza para ir generando los distintos elementos del entorno y posicionarlos a nuestro gusto.
- Game: Aquí dentro solo se muestra la imagen de la cámara. Como en la mayoría de los videojuegos, el jugador no dispone de la visión del entorno al completo, simplemente ve una parte de este. En este caso nuestra cámara está apuntando a un solo campo de pádel.
- Animator: Este apartado se utiliza para gestionar las distintas animaciones del entorno. Permitiendo hacer distintas transiciones entre varias animaciones siguiendo el formato de un grafo.

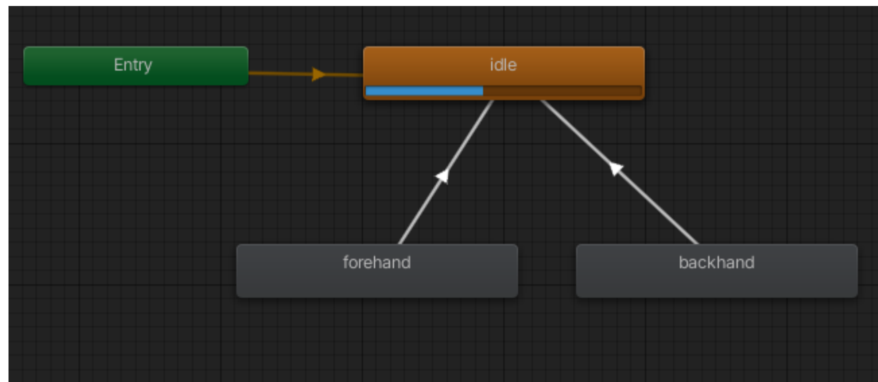


Figura 17: Ejemplo de transición de animaciones.

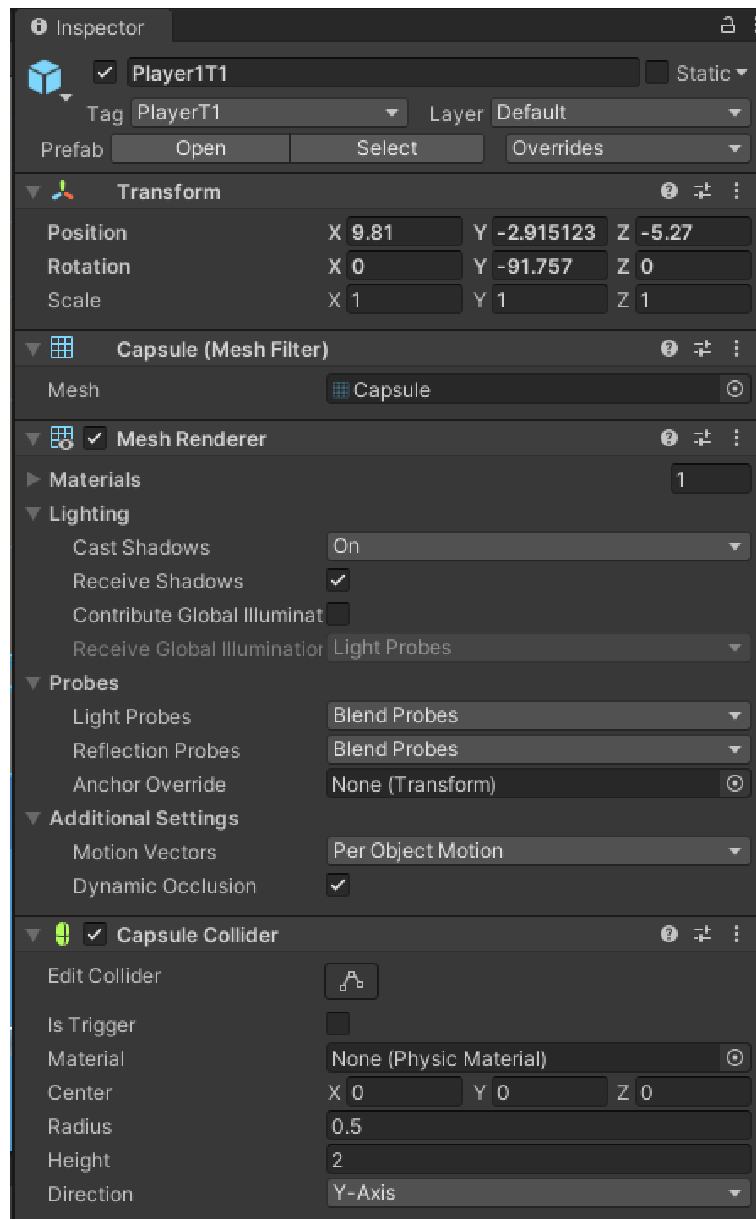


Figura 18: Ejemplo del inspector, que permite visualizar la información de un objeto de Unity.

Por último, en la parte de la derecha nos encontramos con el *Inspector*. En esta ventana se nos va a mostrar toda la información del elemento escogido. Otro punto muy importante es el de añadir componentes a dicho objeto, como por ejemplo: asignar un script, un collider (componente utilizado para detectar las colisiones con otros objetos del entorno), o un rigidbody (componente que permite definir un objeto con física real, como la fuerza de la gravedad).

Hasta aquí esta breve explicación sobre la interfaz de Unity. Ahora vamos a pasar a explicar qué necesitamos para el entrenamiento del agente y qué hemos hecho para ello.

Para llevar a cabo el entrenamiento del agente, a parte de tener instalado Unity con ML-agents, también necesitaremos Python y algunas librerías como TensorFlow o PyTorch. Para ello vamos a montar un entorno desde Anaconda [15]. Anaconda navigator es una interfaz que viene con algunas librerías y softwares instalados por defecto, pero que además podemos crear distintos entornos con distintas instalaciones por separado.

Por tanto nos vamos a crear un entorno nuevo, el cúal utilizaremos solo para entrenar el agente. Las únicas librerías que tendremos que instalar nosotros manualmente son: *mlagents* y *mlagents-envs*. Ya que las demás vienen por defecto al generar un nuevo entorno de anaconda.

```
# Name                    Version               Build  Channel
absl-py                   1.0.0                pypi_0    pypi
attrs                     21.4.0               pypi_0    pypi
ca-certificates           2022.2.1         haa95532_0
cachetools                5.0.0                pypi_0    pypi
cattrs                    1.5.0                pypi_0    pypi
certifi                   2021.10.8        py39haa95532_2
charset-normalizer        2.0.12               pypi_0    pypi
cloudpickle               2.0.0                pypi_0    pypi
google-auth               2.6.0                pypi_0    pypi
google-auth-oauthlib      0.4.6                pypi_0    pypi
grpcio                    1.44.0               pypi_0    pypi
h5py                      3.6.0                pypi_0    pypi
idna                      3.3                  pypi_0    pypi
importlib-metadata        4.11.2               pypi_0    pypi
markdown                  3.3.6                pypi_0    pypi
mlagents                  0.28.0                dev_0    <develop>
mlagents-envs             0.28.0                dev_0    <develop>
numpy                     1.22.2               pypi_0    pypi
oauthlib                  3.2.0                pypi_0    pypi
openssl                   1.1.1m           h2bbff1b_0
pillow                    9.0.1                pypi_0    pypi
pip                       21.2.4           py39haa95532_0
protobuf                  3.19.4               pypi_0    pypi
pyasn1                    0.4.8                pypi_0    pypi
pyasn1-modules            0.2.8                pypi_0    pypi
pypiwin32                 223                  pypi_0    pypi
python                    3.9.7            h6244533_1
pywin32                   303                  pypi_0    pypi
pyyaml                    6.0                  pypi_0    pypi
requests                  2.27.1               pypi_0    pypi
requests-oauthlib         1.3.1                pypi_0    pypi
rsa                       4.8                  pypi_0    pypi
setuptools                58.0.4           py39haa95532_0
six                       1.16.0               pypi_0    pypi
sqlite                    3.37.2           h2bbff1b_0
tensorboard               2.8.0                pypi_0    pypi
tensorboard-data-server   0.6.1                pypi_0    pypi
tensorboard-plugin-wit    1.8.1                pypi_0    pypi
torch                     1.7.1+cu110          pypi_0    pypi
typing-extensions         4.1.1                pypi_0    pypi
tzdata                    2021e            hda174b7_0
urllib3                   1.26.8               pypi_0    pypi
vc                        14.2             h21ff451_1
vs2015_runtime            14.27.29016      h5e58377_2
werkzeug                  2.0.3                pypi_0    pypi
wheel                     0.37.1           pyhd3eb1b0_0
wincertstore              0.2              py39haa95532_2
zipp                      3.7.0                pypi_0    pypi
```

Figura 19: Librerías instaladas en el entorno virtual de Anaconda.

Por tanto, cada vez que vayamos a entrenar un agente, lo haremos desde la consola de anaconda y desde este entorno.

10.2 Generación del entorno básico

La principal idea de la implementación es llevarla a cabo de forma iterativa, es decir tal y como se ha mostrado en el diagrama de Gantt, siguiendo la metodología ágil. Por tanto el primer paso es generar un entorno de padel sencillo, además se va a generar obviando el hecho de que será utilizado por una red neuronal, como si fuera un juego que va a ser jugado por una persona.

Primero voy a explicar a qué me refiero cuando digo que va a ser un entorno básico. Esta primera implementación sólo contempla el escenario de 1 contra 1, por tanto tendremos una pista, la cual se ha obtenido a partir de una página web donde se cuelgan y se pueden descargar modelos de forma gratuita [26]. Luego, obviamente tendremos a los distintos jugadores, donde uno será controlado por nosotros y el otro, de momento será un bot, mediante el cual seremos capaces de poder probar las diferentes implementaciones que iremos haciendo, como por ejemplo el golpe de la pelota o el sistema de puntuación. Las únicas acciones que seremos capaces de hacer serán, mover el jugador y golpear a la pelota con un solo tipo de golpe. El conjunto de reglas que se va a implementar se basará en las del juego original, pero se simplificará, ya que de esta forma nos facilita el entrenamiento de la red neuronal. Por ejemplo, consideramos que un punto se ha acabado si: la pelota ha botado dos veces en un mismo campo o si un jugador golpea la pelota y toca un cristal. El saque será servido por el mismo jugador y no necesariamente tendrá que ser en diagonal.

Al cargar el modelo, tendremos que escalarlo para trabajar con un tamaño acorde a los demás objetos del entorno. Además otro de los problemas que tuve al cargar el modelo de la pista, fue el hecho que no contuviera *colliders*, esto es un componente que podemos añadir a los objetos de Unity para definir su forma y que así sean capaces de colisionar físicamente contra otros objetos. No me percaté de este problema hasta que adelanté un poco más en el desarrollo, fue en el punto de mover el jugador y darme cuenta que era capaz de atravesar los cristales de la pista.

Una vez configurada la pista de pádel, el siguiente paso era crear los jugadores. Ya que el aspecto visual realmente no es un punto principal para este proyecto, decidí representar el cuerpo con un objeto cápsula de Unity, y la pala se construyó a partir de dos cilindros. A cada jugador se le ha añadido un componente *rigidbody*, el cual define la masa del objeto, la posibilidad que tenga gravedad o no, parámetros de velocidad y más.

Para terminar el entorno de forma visual, ya que aún queda realizar las distintas implementaciones, vamos a generar una pelota. Para ello vamos a partir del objeto esfera que nos proporciona Unity. Como componentes principales contendrá un *collider*, al cual le pondremos un material físico, permitiéndonos definir el rebote que se espera de una pelota de pádel, y un *rigidbody*.

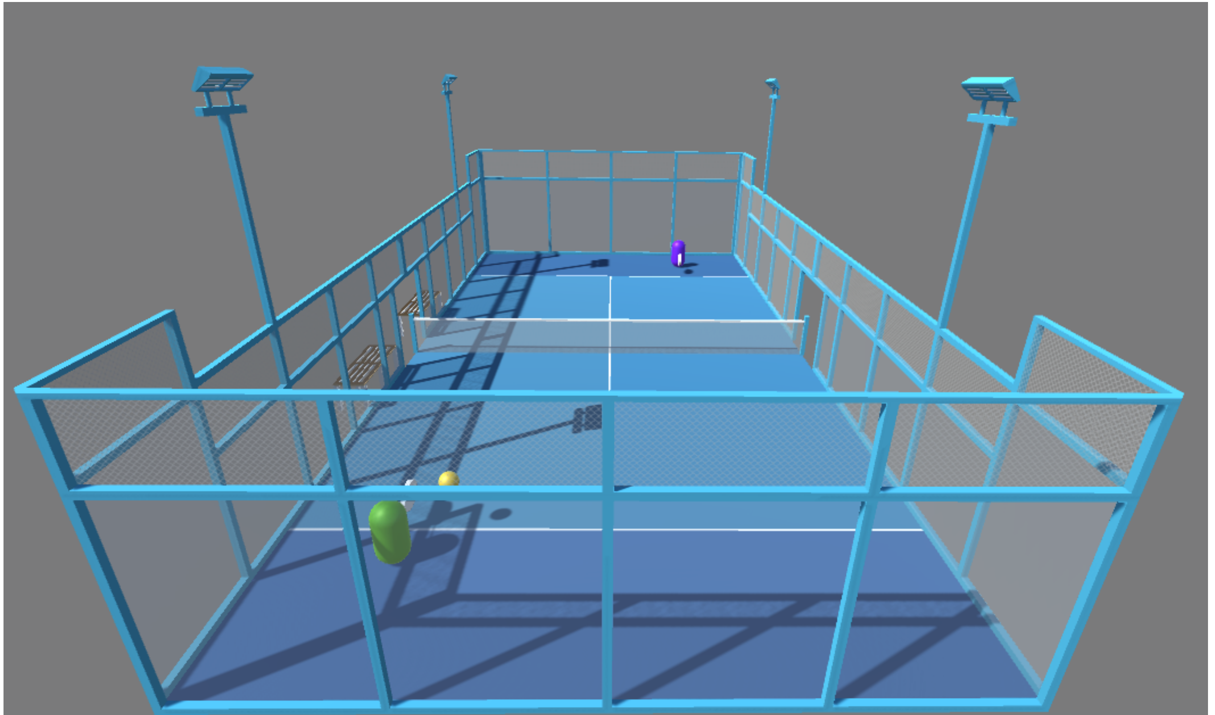


Figura 20: Entorno de pádel, uno contra uno.

10.2.1 Implementación del jugador

Lo primero que vamos a programar del jugador va a ser el movimiento, para ello vamos a utilizar la función **Input.GetAxisRaw()** de Unity, la cual recibe un string como parámetro que tiene que ser “Vertical” o “Horizontal”. Esta función lee como entrada ya sean teclas (en caso de usar un teclado) o un joystick (en caso de mando), cual es el movimiento que desea aplicar la persona que está controlando el jugador. Por ejemplo, en caso del eje horizontal, la función nos devolverá un 0 si no ha tocado ninguna tecla del movimiento horizontal (A o D), 1 si desea moverse hacia la derecha o -1 si es hacia la izquierda. Entonces, con estos resultados podemos aplicar un *Translate* al jugador para realizar los movimientos pertinentes al personaje.

El siguiente punto importante a implementar del jugador es el golpeo de la pelota. Para ello vamos a utilizar las distintas funcionalidades que nos otorga el hecho de usar colliders. Nuestro jugador hasta ahora contiene un collider en forma de cápsula, ya que es la forma propia del personaje. Pero ahora queremos añadir uno nuevo, el cual nos servirá para definir el área donde es capaz de golpear la pelota, y así en el momento que la pelota toque el área de dicho collider se le aplicará una fuerza representando el golpe del jugador. Para poder llevar a cabo dicha implementación, tendremos que poner a True el campo *isTrigger* del nuevo collider. Para entender cuál es el funcionamiento de este campo, primero debemos explicar qué funciones se activan con los colliders.

- OnTriggerEnter() - Esta función se ejecuta cada vez que un objeto entra en el área definida del collider, solo si uno de los objetos tiene el campo *isTrigger* a True.
- OnCollisionEnter() - En este caso se ejecuta en todas las colisiones.

Por tanto, como nosotros queremos solo tener en cuenta este collider para cuando la pelota toca ese área, utilizaremos la función **OnTriggerEnter()**. Recibe como parámetro el collider del objeto con el que ha colisionado, con el cual podemos comprobar que el objeto es la pelota, y entonces, modificar su velocidad para simular así un golpe. Para ello primero debemos decidir hacia dónde queremos apuntar. Para el apuntado de momento vamos a generar tres puntos en la pista (centro, lado izquierdo, lado derecho), entre los cuales se va a elegir de forma aleatoria hacia donde se va a enviar la pelota. Aunque se consideran sólo tres puntos como objetivo del golpe, a nivel táctico en pádel con frecuencia se consideran fundamentalmente tres tipos de golpes en cuanto a dirección: cruzados, en paralelo, y al medio de la pista.

Haciendo los cálculos necesarios podemos variar la velocidad de la pelota para enviarla hacia el punto que se ha escogido aleatoriamente. En nuestro caso dicho cálculo se divide en tres partes:

1. Cálculo del vector dirección (punto a enviar la pelota respecto la posición actual de la misma) normalizado.
2. Fuerza con la que se golpea a la pelota.
3. Vector **up**, el cual determinará la altura de la pelota.

$$v = normalizar(pos_{objetivo} - pos_{pelota}) \ * \ fuerza \ + \ up)$$

Ecuación 4: Cálculo de la velocidad para definir la trayectoria de la pelota. Fuente: Elaboración propia

A partir de estos tres parámetros podemos generar distintos tipos de golpes, aunque en el entorno básico vamos a trabajar solo con uno. El golpe utilizado es simple, ya que no se le aplica mucha fuerza, y con una altura suficiente para asegurarnos que prácticamente siempre pase la pelota por encima de la red.

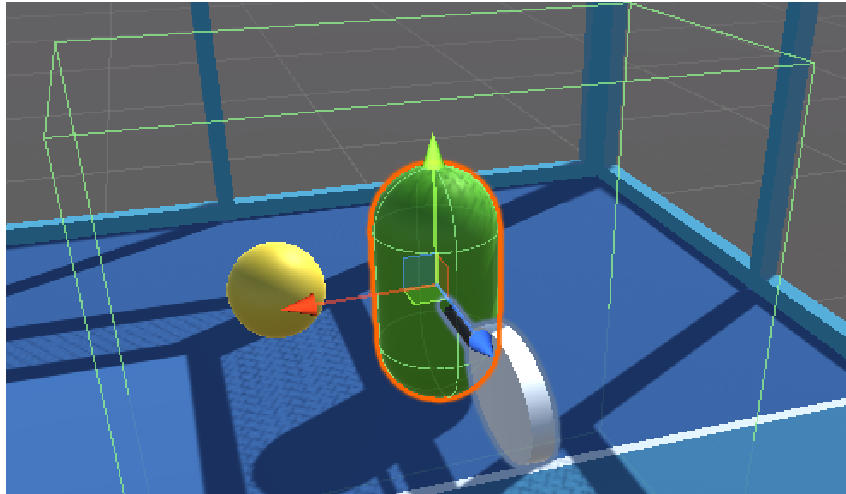


Figura 21: Muestra del collider que define el área donde detecta la pelota para golpearla.

Finalmente se han añadido dos animaciones distintas, simulando un golpe de derecha o de revés. Para ello se aprovecha la función que detecta la colisión para ejecutar una u otra dependiendo de la posición de la pelota respecto a la del jugador.

10.2.2 Implementación de un bot

A continuación se va a implementar un bot, con el objetivo de poder probar las distintas funcionalidades que hemos añadido al jugador. De hecho para generar dicho bot, vamos a partir del código del jugador.

La principal diferencia va a ser el movimiento, ya que el jugador se movía a partir de los inputs que nosotros envíamos mediante el teclado, pero en el caso del bot, queremos que se mueva por sí solo. Para ello vamos a hacerlo lo más simple posible, que va a ser que siga la coordenada X de la pelota en todo momento, asegurándonos así que siempre llegue a la pelota para devolverla.

El golpe del autómata va a ser exactamente el mismo que utilizamos para nuestro jugador, al igual que el sistema de apuntado. En el punto que nos encontramos ahora mismo, podemos probar que el golpe de la pelota funciona correctamente.

Lo próximo que debemos implementar para ya tener nuestro entorno funcional, es el sistema de puntuación, para así poder detectar cuando se ha terminado un punto y entonces reiniciar el entorno.

10.2.3 Implementación de la pelota

Para poder implementar el sistema de detección de un punto, debemos tener información constante de la pelota, para saber cuándo la pelota ha botado dos veces en un lado de la pista u otro, cuándo un jugador manda la pelota directamente hacia un cristal, y qué jugador fue el último que golpeó la pelota.

Para ello, lo primero que vamos a hacer es crear 8 paredes invisibles, siendo tres de estas las cristaleras y una más para el suelo del lado del campo de un jugador, y otras cuatro para el otro lado del campo. Por tanto mediante los colliders, vamos a poder detectar cuando la pelota ha tocado alguna de estas paredes invisibles. Además también queremos saber qué jugador ha sido el último jugador que golpeó la pelota.

Un punto entonces acabará cuando la pelota haya botado dos veces en un campo, o cuando un agente mande la pelota directamente al cristal. Cuando una de estas dos opciones ocurra, lo siguiente a hacer será reiniciar los objetos de la escena, lo cual implica:

- Mover los jugadores a la posición inicial,
- Mover la pelota a la posición inicial,
- Poner a cero la velocidad de la pelota,
- Reiniciar las variables utilizadas para detectar el punto, como quién ha sido el último jugador que golpeó la pelota y el número de botes de la pelota.

Ahora podemos probar con el jugador que movemos nosotros mismos contra el bot, para poder comprobar las distintas implementaciones que hemos hecho hasta ahora. Al hacerlo se constató que con el golpe predefinido, al ser sus parámetros constantes, si los jugadores están bastante alejados de la red, el golpe no consigue una altura suficiente para hacer pasar la pelota al otro lado de la pista. Para este primer entorno, queremos que este golpe sea “perfecto”, que nunca se quede corto, así que incrementamos la altura del mismo para asegurarnos de que la pelota siempre pasará la red.

10.2.4 Implementación del primer agente

Para poder definir el primer agente que vamos a implementar tendremos que modificar el código que hemos hecho. Hasta ahora nuestro programa se dividía en tres scripts distintos: jugador, bot y pelota. De momento el bot se dejará aparte ya que su principal función era la de probar las primeras implementaciones, la pelota seguirá teniendo la función de notificar cuando se ha terminado un punto, y el jugador será modificado para que esta vez sea jugado por un agente. Además generamos un script nuevo que controlará todo el entorno, siendo responsable de dar las recompensas al agente y de reiniciar el entorno cuando se haya terminado un punto.

10.2.5 Modificación en el código de la pelota

Comenzamos con la pelota ya que es la que menos modificaciones tiene. Su principal función (la cual mantiene) es detectar cuando se ha finalizado un punto y qué equipo ha sido el que lo ha ganado y el que lo ha perdido. Hasta ahora también se encargaba de reiniciar el entorno, sin embargo ahora un código específico para el entorno, por tanto será este quien se encargue de esto. Así que cuando la pelota detecte que se ha finalizado un punto, se lo notificará al controlador y este dará unas recompensas a cada agente de cada equipo y reiniciará el entorno.

10.2.6 Modificación en el código del jugador

Lo primero que tenemos que cambiar para que nuestro jugador pueda ser controlado por el agente, es que, la clase que teníamos heredando de **MonoBehaviour** (clase predeterminada de Unity) lo haga de **Agent** (clase predeterminada de ML-agents).

Antes de ponernos a implementar los cambios necesarios para el agente, debemos pensar los parámetros necesarios para que el agente entrene:

- Observaciones del agente: Conjunto de parámetros que se va a utilizar como input para la red neuronal. Deben de ser lo suficiente para que el entrenamiento llegue a un punto donde podamos detectar que el comportamiento del agente está siendo el de jugar a pádel. Para esta primera versión vamos a enviarle la posición del jugador (solo coordenadas x,z), posición de la pelota (x,y,z) y el vector velocidad de la pelota. Lo que hace un total de 8 valores como input.
- Acciones del agente: Conjunto de movimientos que puede ejecutar el agente. Las únicas acciones que podemos decidir es el movimiento del jugador, haciendo distinción entre si el movimiento es en el eje horizontal o vertical. Esto se ve reflejado en la red neuronal como dos outputs con valores discretos, teniendo tres posibilidades: no mover, mover hacia un lado o hacia el otro, teniendo en cuenta que la velocidad del jugador es constante.
- Sistema de recompensas: Puntuación positiva o negativa que recibirá el agente al ocurrir distintos sucesos en el entorno. Cada vez que el agente golpee a la pelota recibirá 0.1 puntos, y cuando un punto termine, el equipo ganador obtendrá 1 punto mientras que al perdedor se le quitará uno.

Al tener estos parámetros estudiados, añadiremos dos componentes a los jugadores, *Decision Requester* y *Behaviour Parameters.* El primero es el encargado de solicitar qué decisión tomar a la red neuronal, mientras que en

el otro es donde definimos el tamaño del vector de observación y de acciones.

Ahora sí que podemos entrar en la implementación necesaria para que el agente funcione correctamente. Ya que el agente no tiene noción sobre el golpe a la pelota, esta parte será la misma, a excepción de un pequeño añadido. Cuando se detecte que el jugador va a golpear la pelota se le dará una recompensa de 0.1, mostrando así al agente que debe moverse en dirección hacia la pelota para golpearla.

Lo siguiente es re-implementar (*override)* distintas funciones de la clase: ***Initialize***, inicializamos los variables necesarias del código; ***CollectObservations****,* rellenamos en un vector las observaciones del agente; ***Heuristic****,* esta simplemente se utiliza en el modo heurístico del agente, el cual nos permite poder controlar el agente por nuestra cuenta, la función devuelve el conjunto de decisiones tal y como haría la red neuronal; ***OnActionReceived***, dado un conjunto de decisiones (ya sea de la red neuronal o del heurístico) reflejarlo en el entorno. En nuestro caso, estas decisiones se resumen en mover o no al jugador mediante dos ejes.

Como comentario extra, se ha añadido una enumeración para saber de qué equipo es cada jugador, para así cuando se haga un punto, saber a quién debemos incrementar o disminuir su puntuación.

10.2.7 Implementación de un controlador del entorno

El siguiente script va a ser el encargado de configurar de nuevo el entorno cada vez que sea necesario, es decir, cada vez que se haga un punto deberá devolver a los jugadores y a la pelota a sus posiciones iniciales.

Para poder llevar a cabo dichas funciones, cuando la pelota detecte que se ha terminado un punto, se lo comunicará al controlador llamando a una función, la cual recibe como parámetro qué equipo ha sido el ganador. Comentar que para preparar el código para el entorno de dos contra dos, se va a utilizar la clase *SimpleMultiAgentGroup*, la cual sirve para almacenar distintos agentes dentro de un mismo grupo (para nosotros un grupo se define como un equipo u otro del partido). Así, cada vez que se haga un punto, esta recompensa no se va a añadir a nivel de jugador, sino a nivel de equipo. La otra recompensa que vamos a implementar será para cuando el jugador golpee la pelota, de momento esta la vamos a hacer a nivel de jugador.

Con esto terminamos las implementaciones principales para comenzar a generar y entrenar un agente dentro de este primer entorno básico.

10.3 Entrenamiento del agente básico

Si nos fijamos en los distintos ejemplos que hay en el github oficial de ML-agents, veremos que se hacen copias de los entornos para que los entrenamientos sean más rápidos, ya que al controlar distintas instancias al mismo tiempo, estudia cada una por separado, pero el aprendizaje que obtiene es a nivel general.

Antes de hacer esas copias de entornos, vamos a ejecutar nuestro único entorno durante unos minutos, para comprobar que toda la conexión agente-entorno se ha definido correctamente. El objetivo de esto es darnos cuenta que el agente se puede mover entre los dos ejes, es decir, que el agente recibe las decisiones de la red neuronal y las aplica, percatarse si cuando se termina un punto, los jugadores y la pelota se reinician correctamente, y al mismo tiempo, que el sistema de recompensas se está aplicando correctamente.

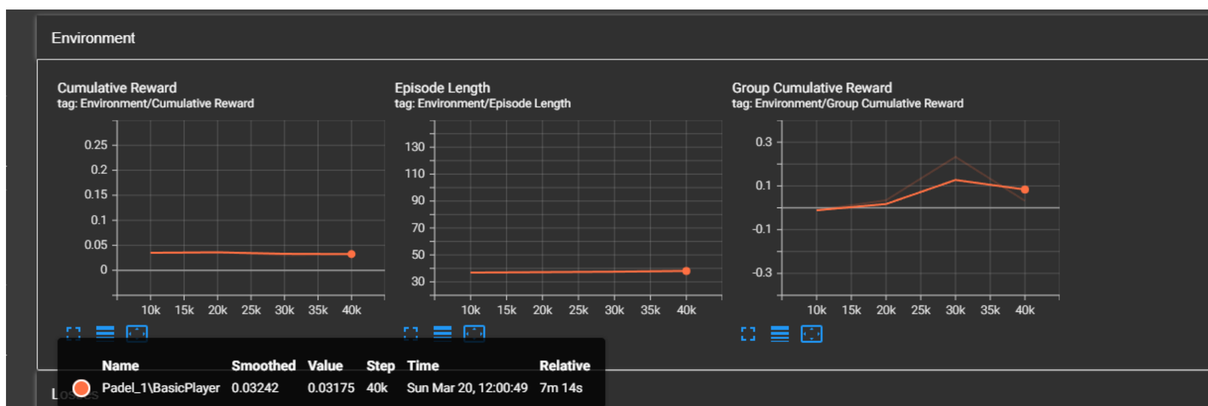


Figura 22: Resultados del entrenamiento para comprobar que todo funciona correctamente.

En la imagen anterior podemos observar los resultados del pequeño entrenamiento que hemos hecho para comprobar el correcto funcionamiento del entorno. Realmente no tiene mucha importancia ya que en un rango de tiempo tan reducido y con solo un entorno, no vamos a detectar señales importantes de aprendizaje. Pero nos va a servir para cuando montemos distintas copias de entornos, demostrar que en el mismo tiempo utilizado vamos a multiplicar el número de pasos realizados.
Por tanto el siguiente paso es construir una nueva escena en Unity, aplicando distintas copias del entorno, y volveremos a ejecutar el comando para el entrenamiento. Como ya sabemos que nuestro entorno se comporta correctamente, vamos a dejarlo un tiempo entrenando. Entonces comprobaremos si el agente ha aprendido a moverse hacia la pelota para devolverla y así, tratar de ganar el punto.

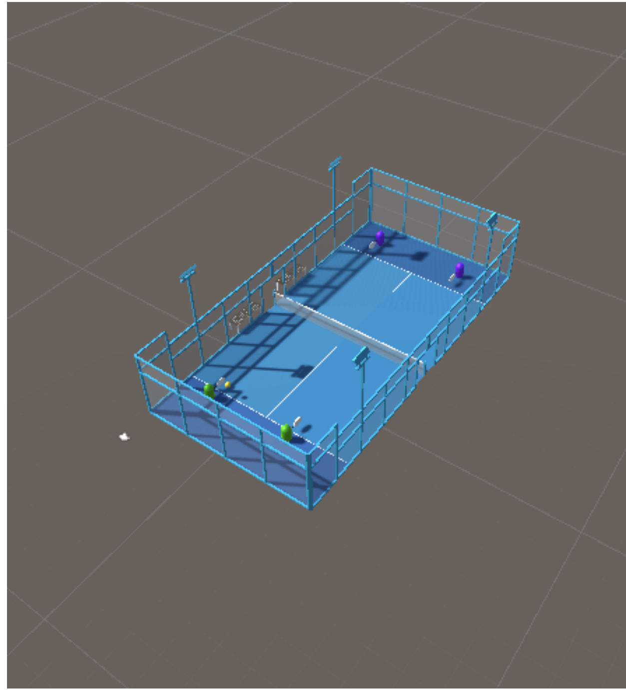


Figura 23: Ejemplo de entrenamiento con un solo entorno.

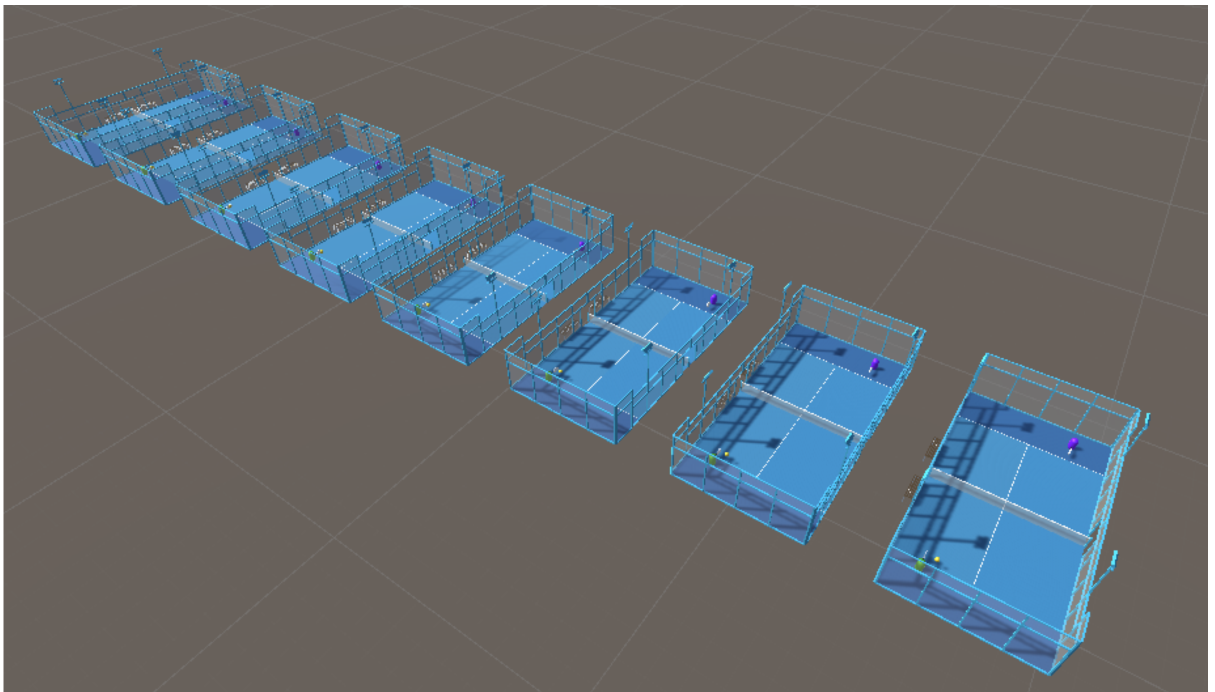


Figura 24: Ejemplo de entrenamiento con ocho entornos.

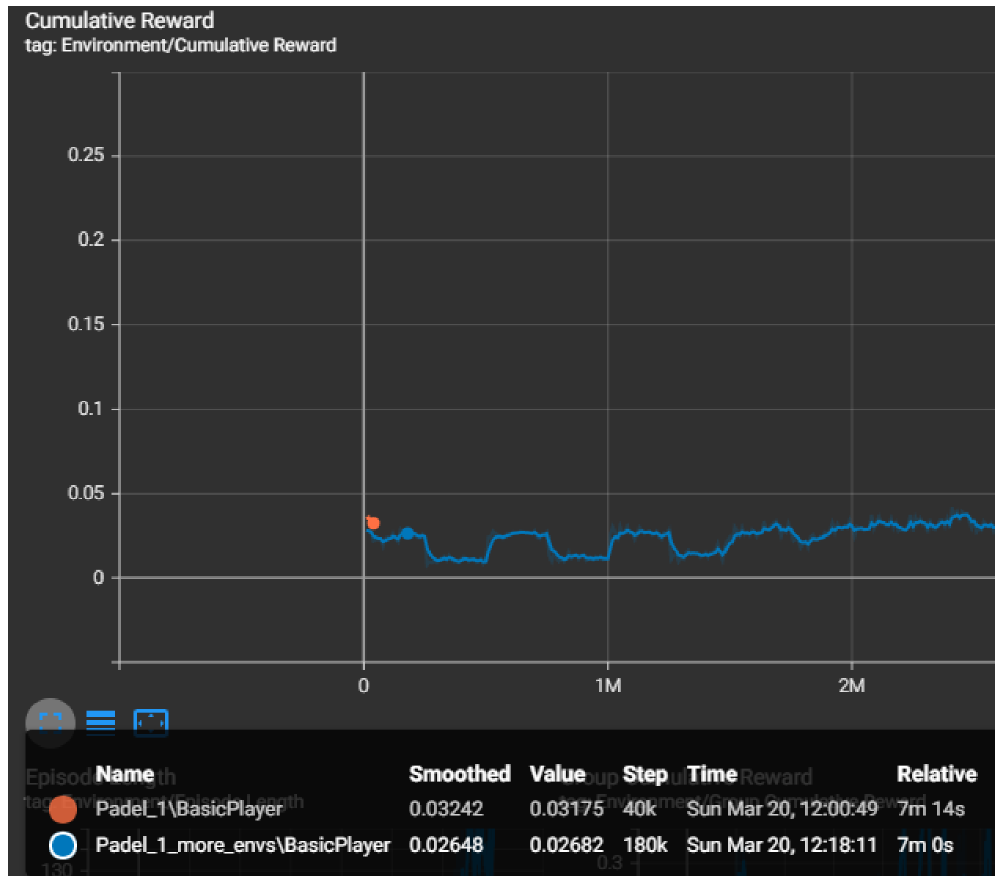


Figura 25: Comparación de pasos al utilizar más de un entorno durante el entrenamiento.

En la imagen anterior podemos observar como el primer agente que habíamos entrenado durante unos siete minutos había hecho unos 40.000 pasos, mientras que el nuevo, que está trabajando con 8 entornos al mismo tiempo, ha hecho un total de 180.000 pasos en el mismo tiempo. Estamos consiguiendo 4.5 pasos más en el mismo tiempo transcurrido, por tanto realmente es una mejora considerable.

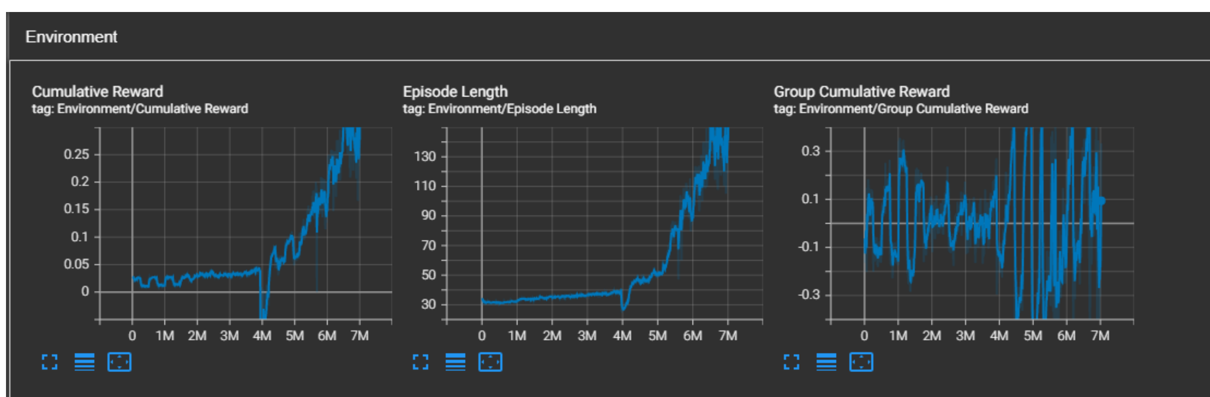


Figura 26: Resultados del primer agente del entorno simple.

Hemos entrenado a este agente durante unos siete millones de pasos, y hemos obtenido el comportamiento esperado, ya que el agente se está moviendo hacia la pelota. Pero se ha tenido que mediar con distintos problemas los cuales se han ido resolviendo durante el entrenamiento.

Al entrenar el agente, la red neuronal escoge de forma aleatoria sus pesos, y esto se ve reflejado en un comportamiento aleatorio al inicio. En este caso los agentes se movían hacia atrás, evitando así, golpear a la pelota, así que la pelota botaba dos veces y se terminaba el punto, haciendo que el equipo contrario tuviera una puntuación alta sin esfuerzo. Para resolver dicho problema se ha hecho que el saque varíe de equipo en cada punto, y la pelota se ha acercado más al jugador, haciendo así que cuando empiece un nuevo punto sea prácticamente instantáneo el golpeo. Con esta nueva implementación, tenemos que el agente entienda que al golpear la pelota obtiene una puntuación positiva, y se reparte en ambos equipos al hacer que los saques varíen entre equipos.

Aunque el entrenamiento del agente se hubiera pausado para arreglar estos problemas, se ha obtenido un agente funcional, pero quería construir un agente el cual no tenga que cambiarse en mitad del entrenamiento.

En consecuencia, hice un nuevo entrenamiento con un nuevo agente partiendo de cero con las últimas implementaciones. Gracias de nuevo a la aleatoriedad de los pesos de la red neuronal, el agente empezó a moverse hacia delante sin parar, llegando hasta la red del campo. Ahí fue donde entonces me di cuenta que no había implementado la regla de no tocar la red. Para hacer consciente al agente que no debía tocarla, hago que si colisiona con esta, se le da una puntuación negativa de 1 (como cuando pierde un punto), y se reinicia la escena.

De nuevo había cambiado algo en medio del entrenamiento del agente, pero lo seguí entrenando por si detectaba algún problema nuevo. El resultado final fue de nuevo un agente correcto, aunque en este caso no se movía prácticamente en el eje vertical. Esto viene dado a que el agente se dio cuenta que si se movía mucho hacia delante iba a tocar la red, obteniendo una puntuación negativa, así que simplemente se movía en el eje horizontal para llegar hacia la pelota.

Como aún no había conseguido ese agente funcional sin tener que cambiar nada durante el entrenamiento, creé uno nuevo para ello.

Primero hice dos cambios que no considero que sean muy significativos:

- La puntuación de golpear la pelota pasó de ser individual a grupal, no supone un gran cambio ya que el grupo ahora mismo solo está compuesto por un jugador, pero quería tener toda la puntuación en un único parámetro.
- Se añade la distancia de la pelota con el jugador como observación, hasta ahora el agente recibía la posición de la pelota como su propia posición, así que podía inferirla.

Por último quería que el agente también se moviera algo más en el eje vertical, para ello le añadí un nuevo componente al jugador. Un rayo que detecte la distancia hasta la red.

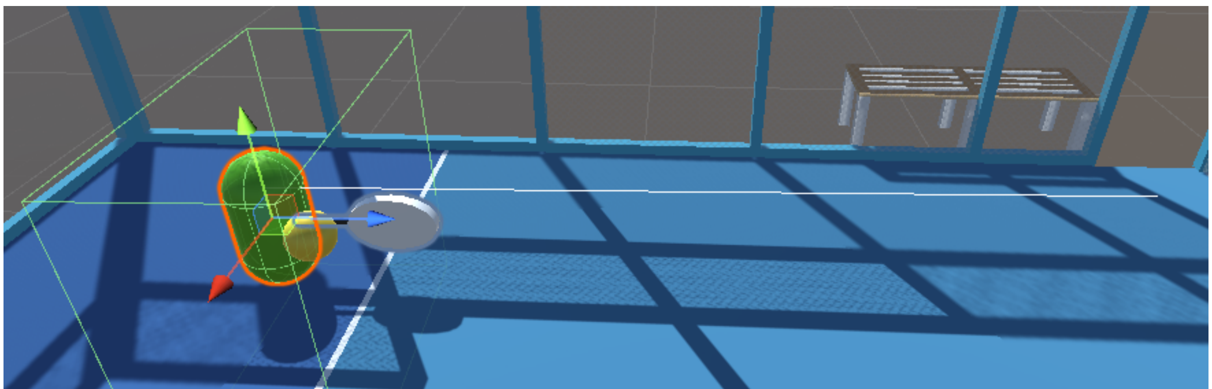


Figura 27: Muestra del rayo para detectar la red.

Con todos estos cambios entrené un nuevo agente, el cual resultó en un agente funcional sin tener que haber cambiado nada a mitad del entrenamiento.

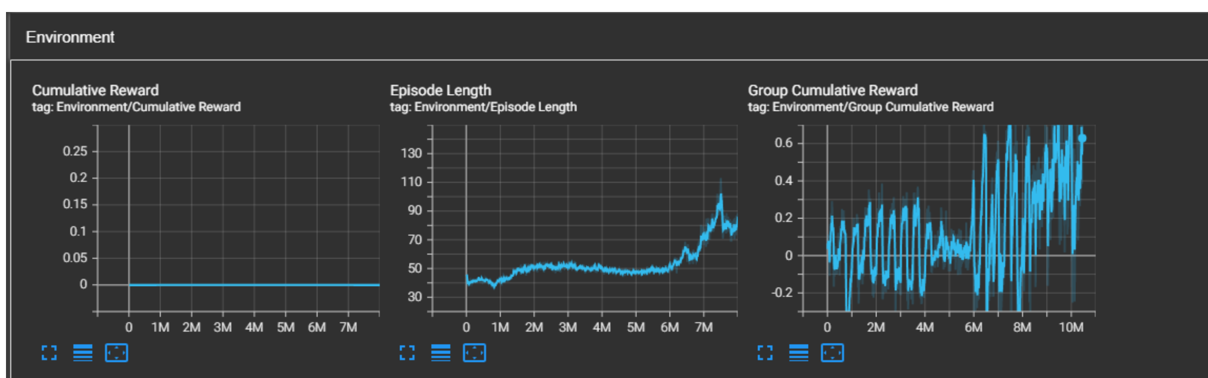


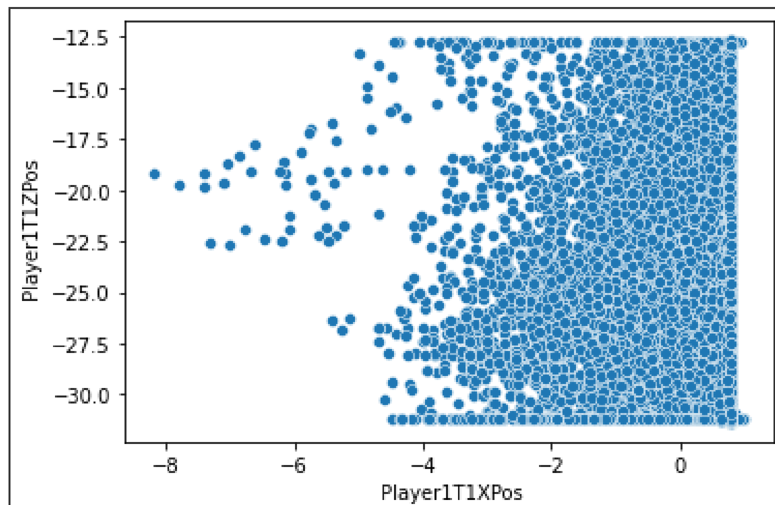
Figura 28: Resultados del agente final del entorno simple.

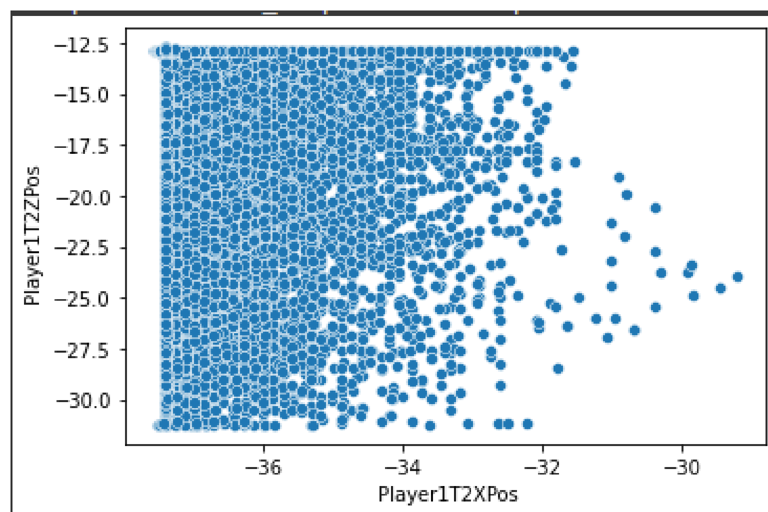
En los resultados del entrenamiento se puede ver como la primera gráfica es lineal, esto viene dado a que hemos pasado toda la puntuación a nivel de grupo. La segunda mide la duración de los episodios, se puede observar como cada vez duran más debido a que el agente va aprendiendo a jugar. Si nos fijamos en la tercera, vemos que de forma general la puntuación ha ido

aumentando a lo largo del entrenamiento, por tanto podemos comprobar que el agente ha aprendido durante el tiempo.

Aparte de los gráficos anteriores, los cuales se generan automáticamente mediante tensorboard con la información del entrenamiento, vamos a generar algunos nosotros mismos. Para ello, una vez tengamos el agente ya entrenado, vamos a generar un fichero **.csv** para almacenar información como la posición de los jugadores, posición de la pelota, velocidad de la pelota u otros factores importantes de las siguientes implementaciones. Estos atributos se van a guardar cada 30 fotogramas en un total de 50 puntos.

Una vez hayamos generado este fichero, mediante un script de Python podremos generar nuevos plots con la librería *seaborn*. Por ejemplo podemos mostrar la posición de los jugadores, para así ver cuál ha sido el movimiento de estos.





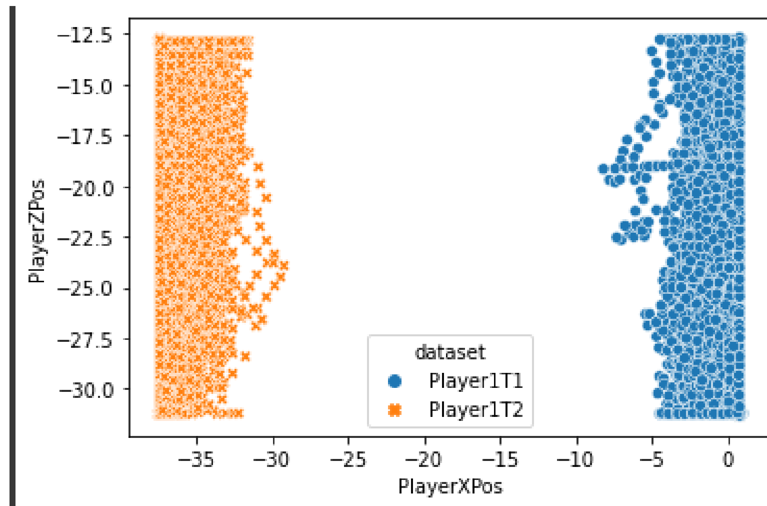
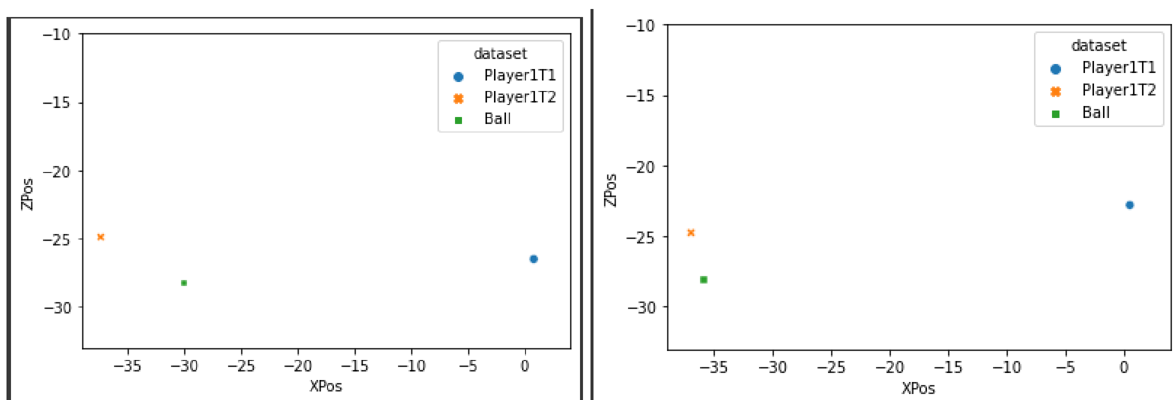
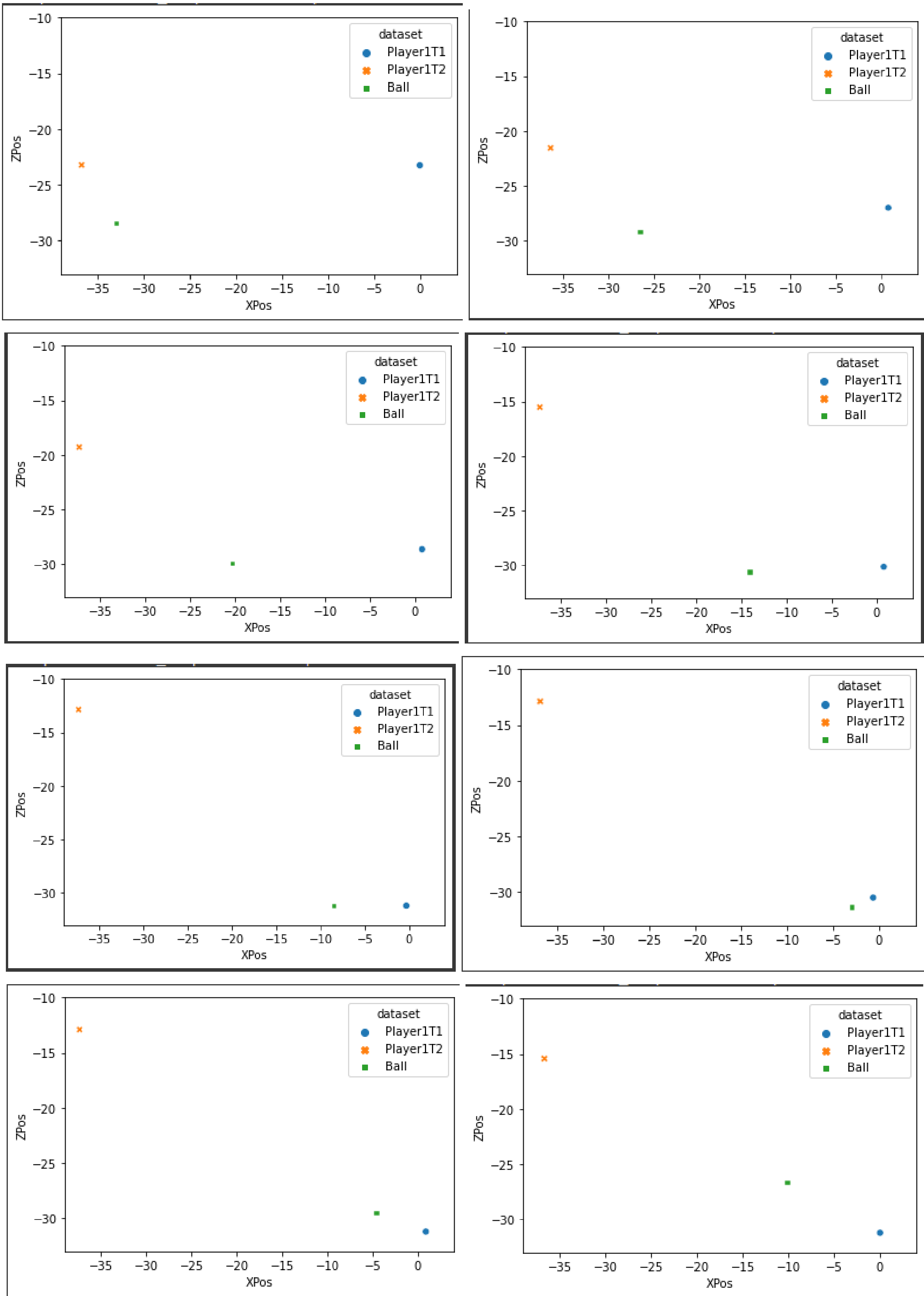


Figura 29: Movimiento de los jugadores en 50 puntos.

El tercer gráfico de los mostrados no es más que una concatenación de los datos de los dos primeros. En todos lo que se está mostrando es la posición de los jugadores en todas las tomas de las variables que se han hecho. Se puede observar como los jugadores no se acercan mucho hacia la red, prácticamente todo el rato se quedan en la parte de detrás de la pista.

Ahora se va a mostrar un conjunto de gráficos para demostrar que el agente ha aprendido a moverse hacia la pelota. Para ello se ha escogido uno de los 50 puntos sobre los que se ha recogido la información del entorno, y se genera el conjunto de gráficos. Se está mostrando de forma resumida las 50 últimas tomas de información antes de finalizar el punto y solo generando el gráfico de las tomas múltiples de 5, es decir si un punto ha finalizado en la toma 100, los gráficos mostrados son los de las tomas 50, 55, 60.




dataset
Player1T1
Player1T2
Ball
ZPos
XPos

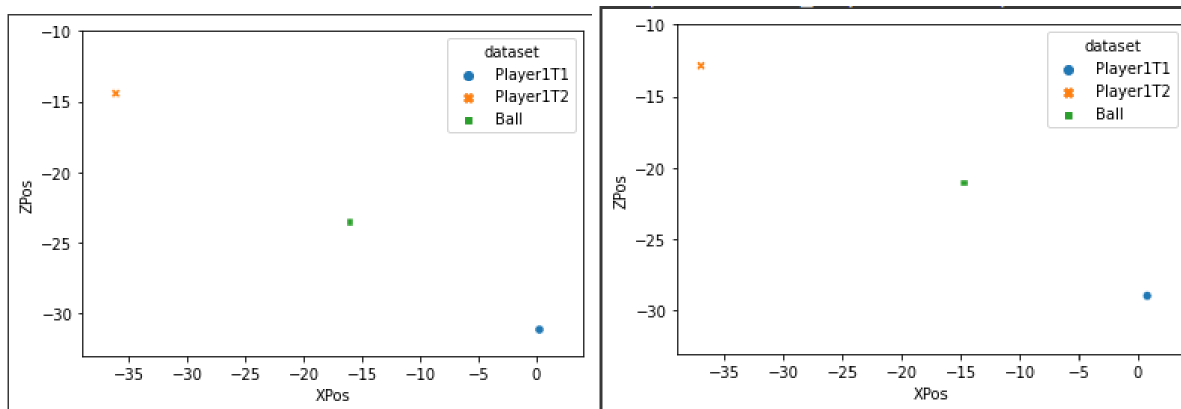
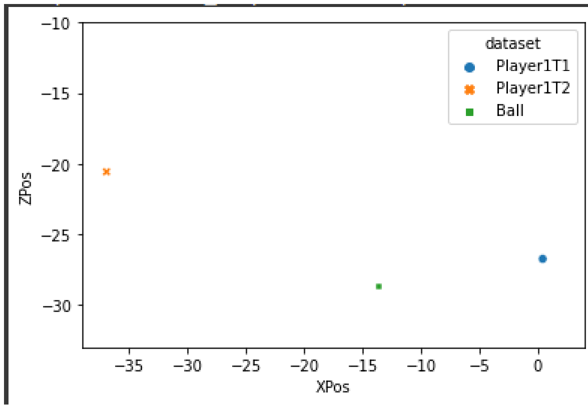
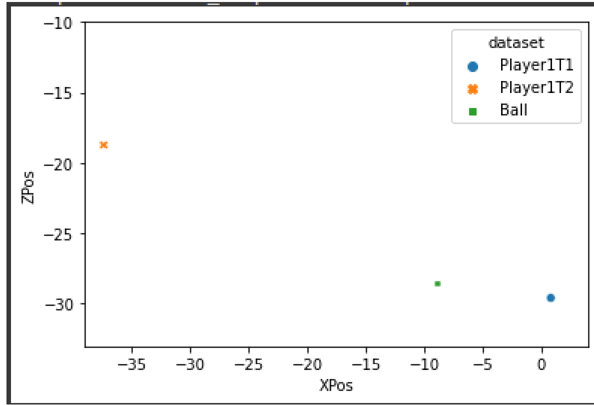
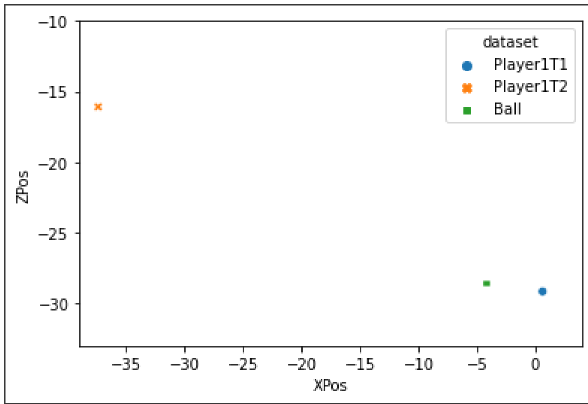
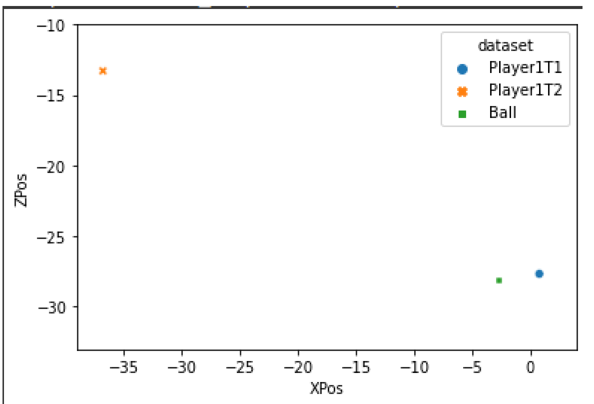
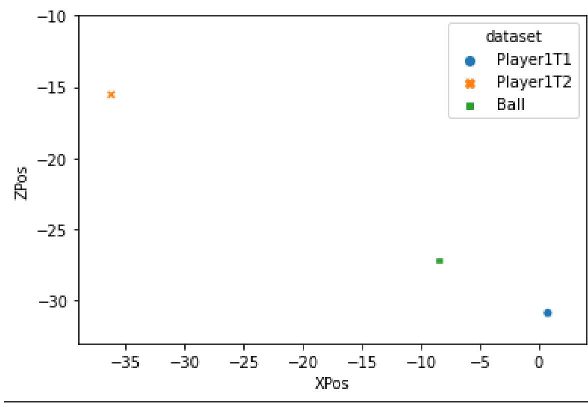
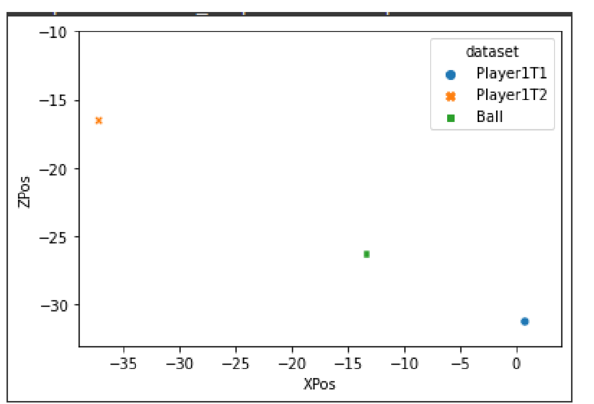
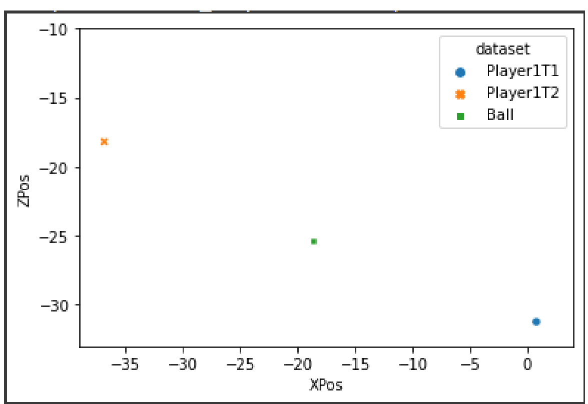
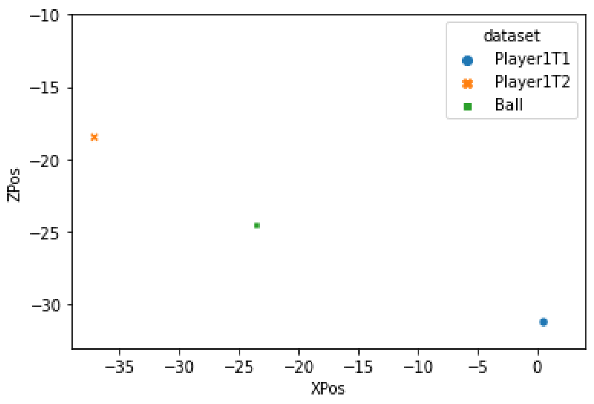


Figura 30: Demostración de un punto a partir de un conjunto de gráficas.

El orden correcto para seguir el partido es de arriba hacia abajo y de izquierda a derecha. Aclarar por como se ve en la leyenda, el punto azul es un jugador, el naranja es el otro jugador y el verde es la pelota.

Al principio de la escena se ve como la pelota se está acercando hacia el jugador de color naranja, el cual se ha ido moviendo para cuando esté lo suficientemente cerca golpearla. Entonces la pelota se mueve hasta llegar al campo del jugador azul, aunque este gráfico sea en dos dimensiones, sabemos que la pelota ha pasado la red porque no ha vuelto hacia el campo del jugador naranja. A continuación el jugador azul se está moviendo hacia la posición de la pelota, para entonces devolverla al campo contrario. Cuando la pelota se comienza a alejar del jugador azul, significa que este la ha golpeado. Si comparamos el último gráfico con el penúltimo, podemos ver cómo la pelota se ha acercado hacia el campo de jugador azul, cosa que quiere decir que la pelota ha tocado la red y ha vuelto al campo del jugador que la había golpeado, por tanto, este es el motivo por el que finaliza este punto.

En el siguiente conjunto de gráficos se observa cómo la pelota va hacia el campo del jugador azul y éste se mueve para golpear la pelota. Entonces la pelota va hacia el campo del jugador naranja; éste, sin embargo, no consigue posicionarse correctamente para devolverla. En este caso el punto finaliza porque la pelota bota dos veces en el campo del jugador naranja, ya que el jugador no se ha movido lo suficiente como para golpearla antes de que haga el segundo bote.


dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

dataset
Player1T1
Player1T2
Ball
ZPos
XPos

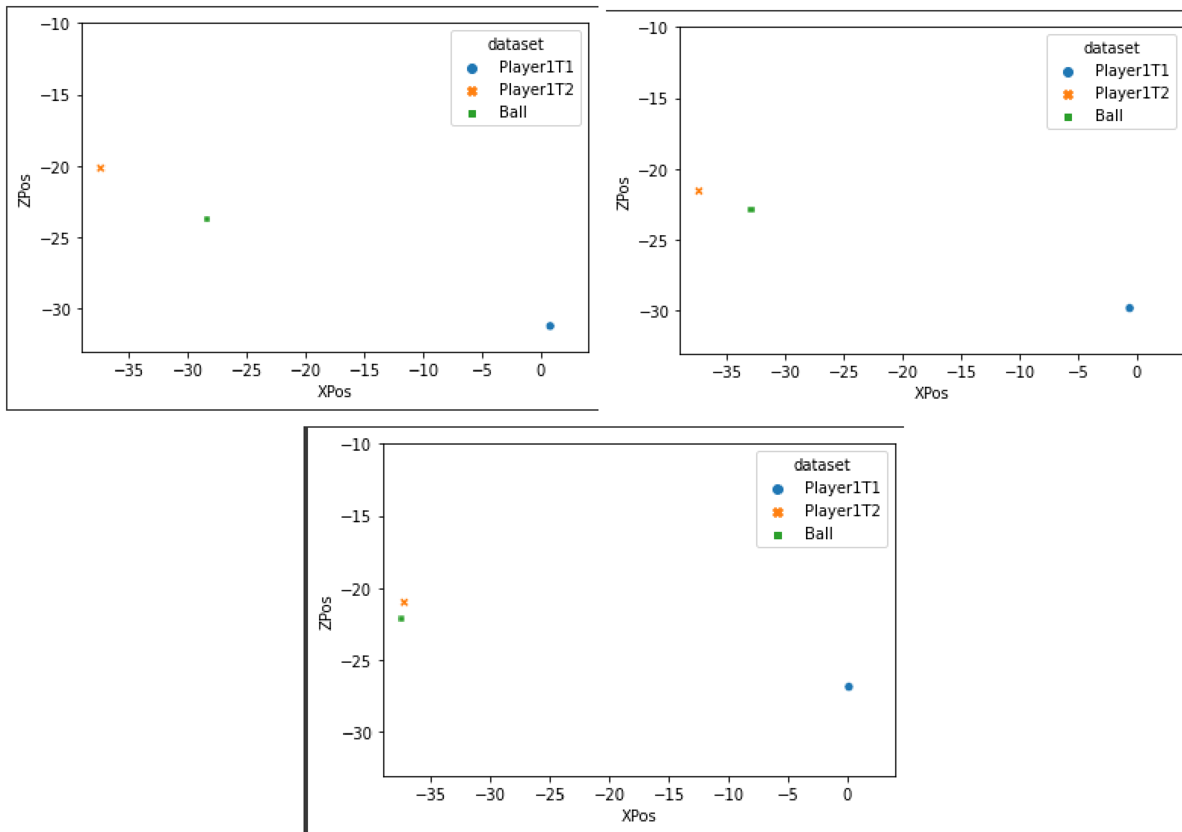


Figura 31: Demostración de un segundo punto a partir de un conjunto de gráficas.

Por tanto, ahora tenemos un agente para el caso 1vs1 que ya funciona. Estudiando el modelo obtenido creo que el golpe utilizado es algo lento, pero esto se solucionará al añadir nuevos golpes en el siguiente entorno. Sin embargo considero que sería más interesante cambiar el apuntado del golpe, dejar de elegir de forma aleatoria entre tres puntos constantes. Trabajaremos con un punto el cual se podrá mover horizontalmente para elegir dónde enviar la pelota, entonces haremos consciente a la red neuronal de dónde quiere enviar la pelota.

10.4 Implementación del entorno avanzado

10.4.1 Implementación del apuntado dinámico

La idea de implementar este apuntado es dejar que el agente decida hacia dónde quiere enviar la pelota, eliminando así que se envíe hacia un punto u otro de forma aleatoria. De igual forma que teníamos tres puntos entre los cuales elegimos a dónde enviar la pelota, ahora tendremos un punto en el centro de la pista y el agente podrá ir moviéndolo en el eje horizontal para definir la trayectoria de su próximo golpe.

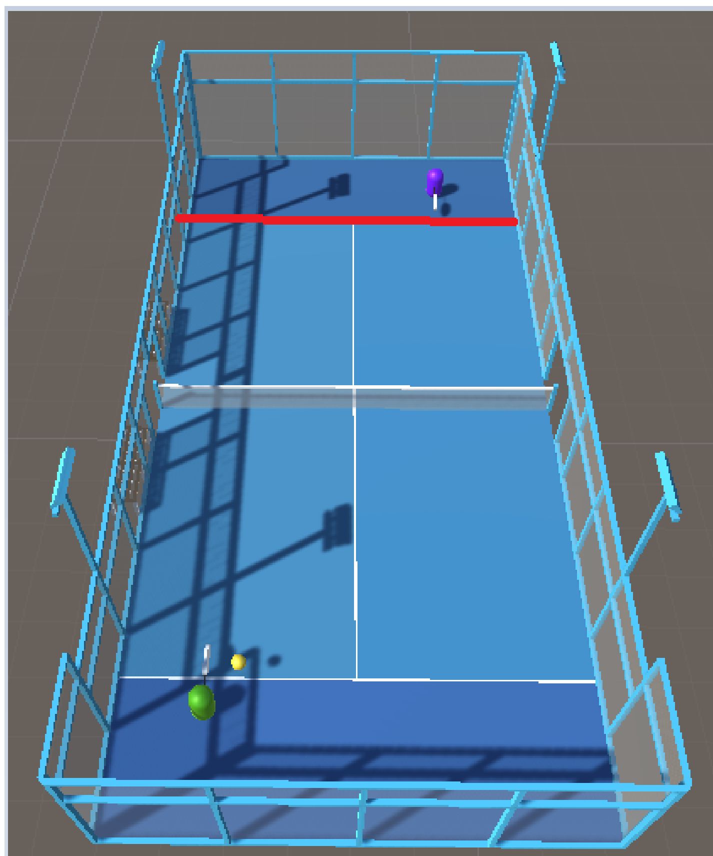


Figura 32: Demostración del eje por el que se moverá el punto utilizado como objetivo.

A nivel de entorno, lo único que varía con el anterior es que ahora en vez de tener tres puntos invisibles tendremos uno. En el código del controlador del entorno, añadimos que cada vez que se haga un reinicio, también se reinicie la posición del punto utilizado como apuntado.

En el agente tendremos que añadir una rama nueva de acciones discretas, la cual tendrá tres valores posibles:

- 0 - No mueve el punto
- 1 - Mueve el punto hacia la derecha
- 2 - Mueve el punto hacia la izquierda

A nivel de código, la función OnActionReceived, la cual recibe como parámetro las decisiones tomadas por la red, añadiremos el movimiento del punto para el apuntado tal y como hacemos para mover el jugador. Finalmente, cuando salte el trigger porque el jugador tiene la pelota a su alcance, ésta recibirá un golpe y ahora la trayectoria vendrá definida por este punto.

10.4.2 Entrenamiento del agente con apuntado dinámico

Antes de pasar al entrenamiento del agente con este nuevo añadido, lo primero que se ha hecho ha sido probarlo mediante la opción del heurístico,

la cual nos permite controlar el agente por nuestra cuenta, para comprobar como el punto que define la trayectoria se mueve correctamente y la pelota sigue esa dirección.

El siguiente paso es entrenar a un nuevo agente. Pero para ello no vamos a entrenar un agente desde cero, ya que existe una opción con la que podemos entrenar un nuevo agente partiendo de otro. Por tanto este nuevo agente va a partir del modelo anterior, que ya ha aprendido a moverse hacia la pelota.

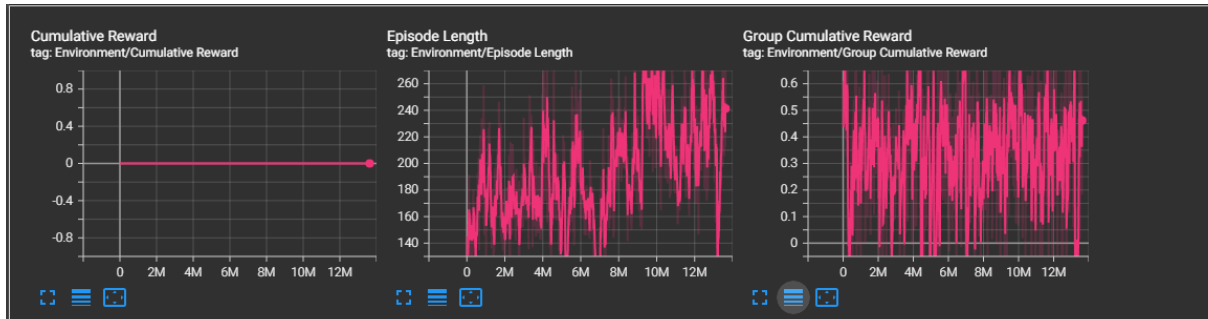


Figura 33: Resultados del agente con la posibilidad de elegir hacia dónde apuntar.

Primero dejar claro que hay un pico negativo muy pronunciado (se puede observar en la segunda gráfica hacia el final), ya que un día al reanudar el entrenamiento del agente, hubo un error al cargar el entorno de Unity, ya que mientras se entrenaba este agente se estaba implementado los cambios para la siguiente mejora.
En cuanto a los resultados de las gráficas, en la segunda vemos que la duración de los episodios ha crecido, lo cual es buena señal. En este caso la recompensa varía bastante, pero se puede observar como cada vez que se adelanta el entrenamiento, los valores aumentan. Además, el modelo del nuevo agente funciona correctamente, ya que se sigue moviendo hacia la pelota, y se puede observar cómo varía el apuntado de sus golpes a medida que se desarrolla el partido.

Para poder mostrar que el apuntado dinámico está funcionando correctamente, se va a recoger la información del entorno en un fichero, de igual forma que se hizo para el anterior agente. En este caso, la información que se almacenará será la misma, y además, la coordenada z del punto utilizado para definir la trayectoria de la pelota, ya que las demás coordenadas no varían.

Lo primero que vamos a mostrar va a ser el conjunto de puntos que define el movimiento de los agentes a lo largo de los 50 puntos jugados.

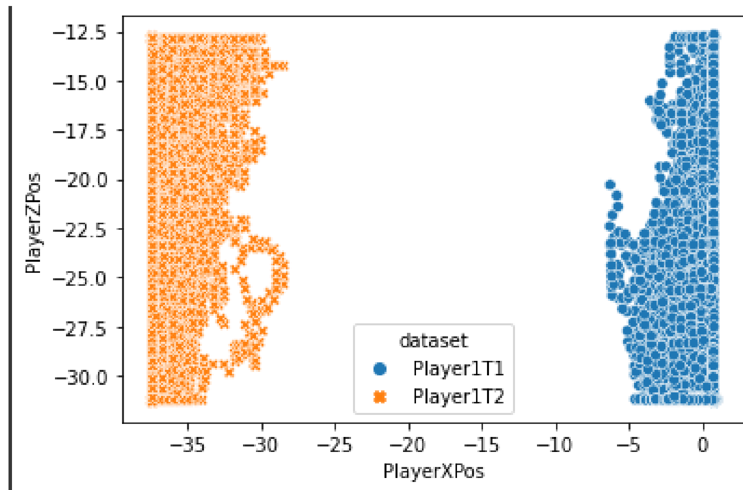
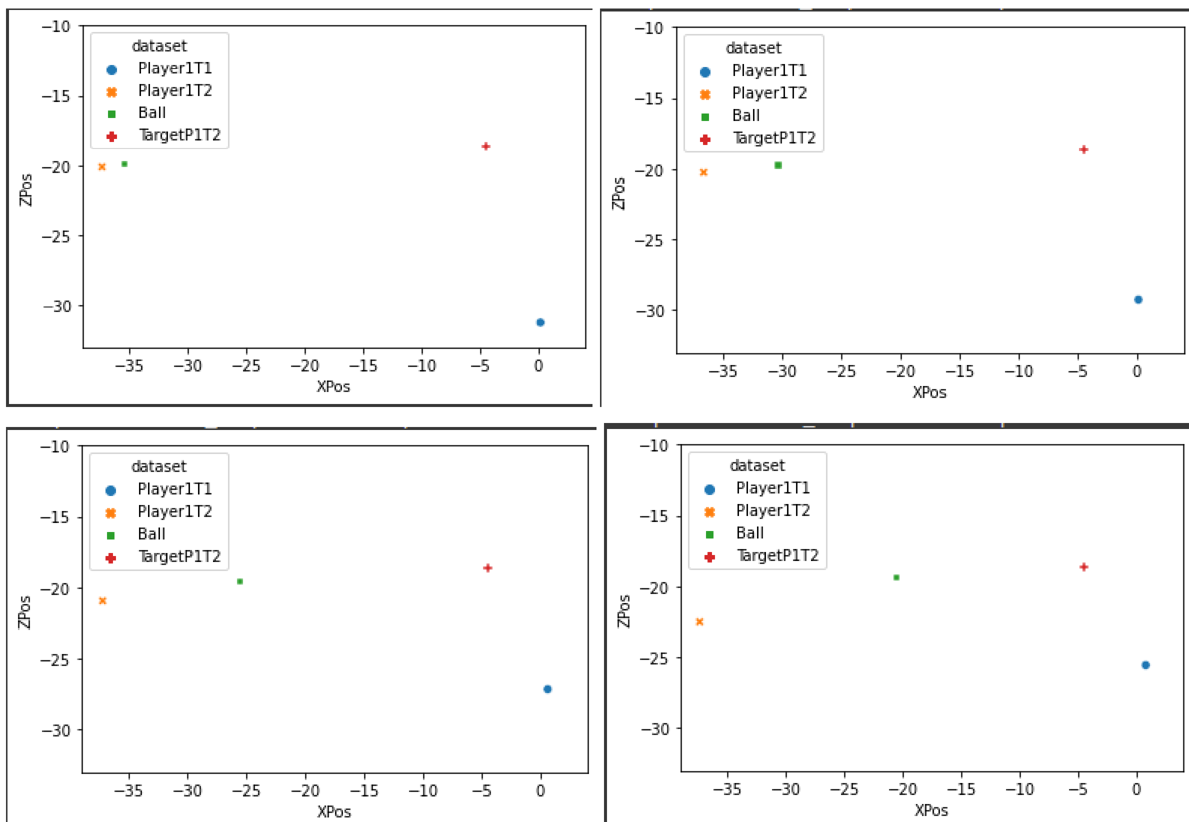


Figura 34: Movimiento de los jugadores a lo largo de 50 puntos.

En la mayoría de los puntos tomados, ambos jugadores juegan en la parte final de la pista, este comportamiento es similar al que se vió para el anterior agente.

A continuación, se van a representar distintas jugadas para demostrar el añadido del apuntado dinámico, y cómo el agente es capaz de controlarlo.



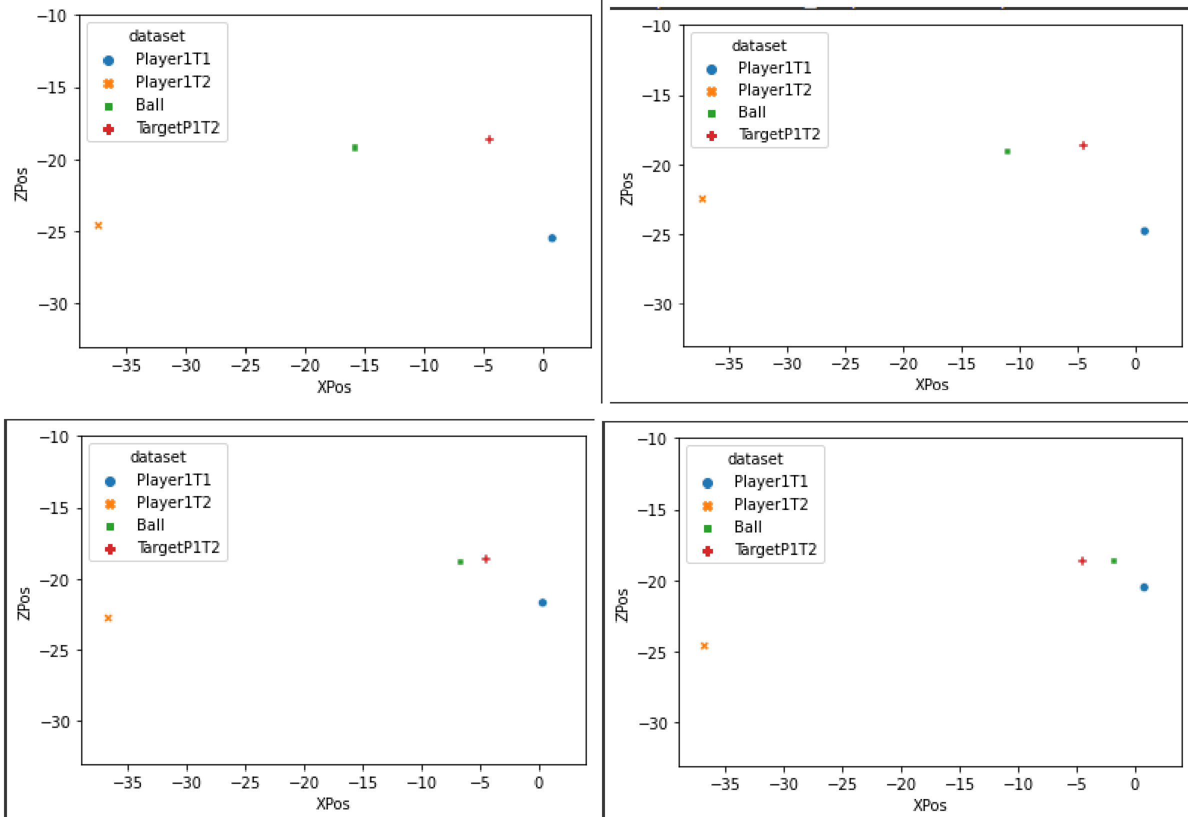
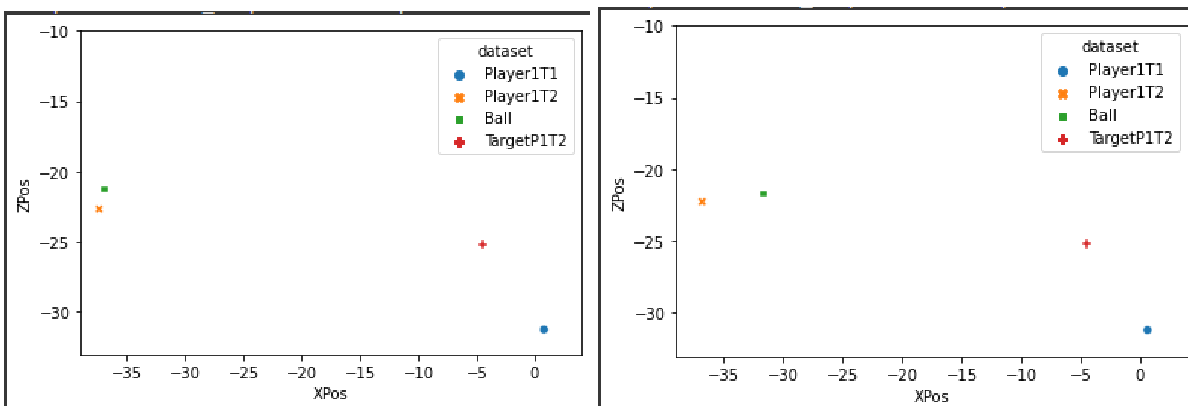
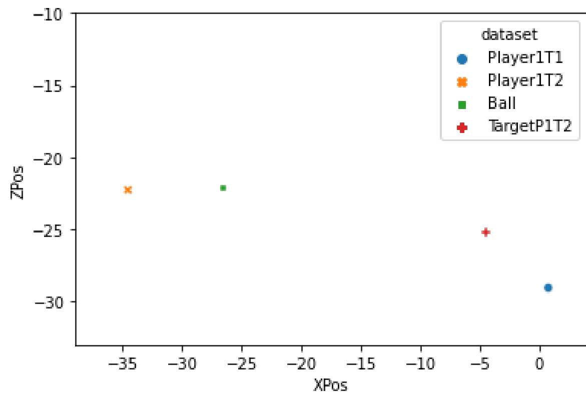
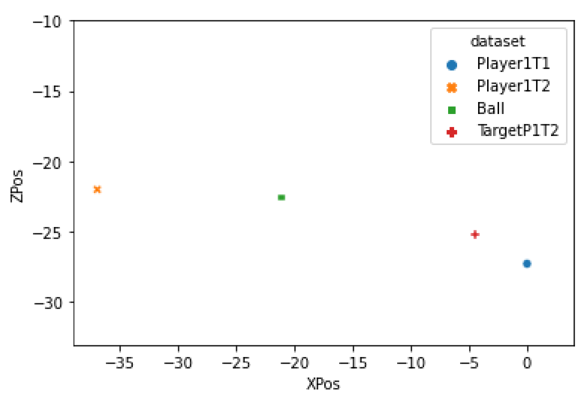


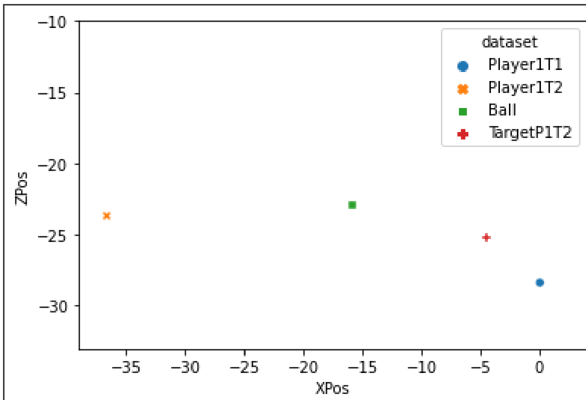
Figura 35: Demostración del nuevo apuntado a partir de un conjunto de gráficas.

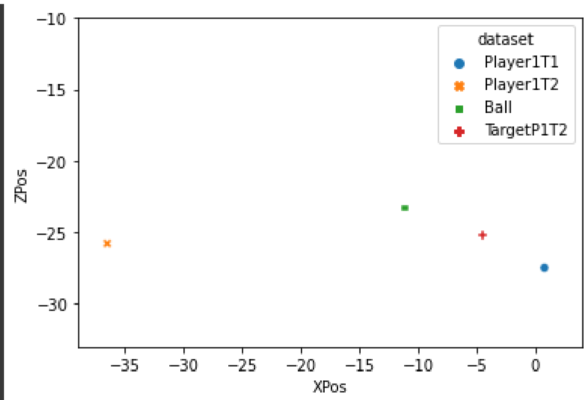
En este conjunto de imágenes se muestra la posición donde el jugador naranja quiere enviar la pelota, y si nos fijamos la pelota está yendo donde el agente ha elegido, y el jugador azul se mueve hasta dicha posición para devolverla. En este punto en concreto, los agentes estaban enviando la pelota cerca del centro de la pista. Por eso, ahora se va a mostrar otro partido donde los puntos escogidos entre los distintos agentes difieren más.

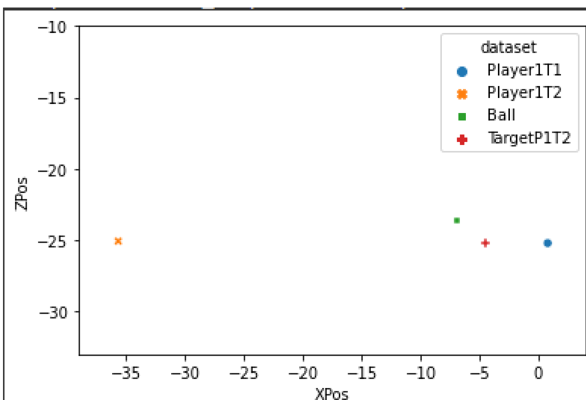


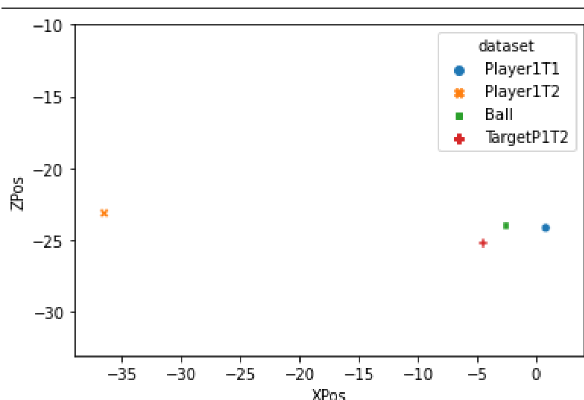

dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0

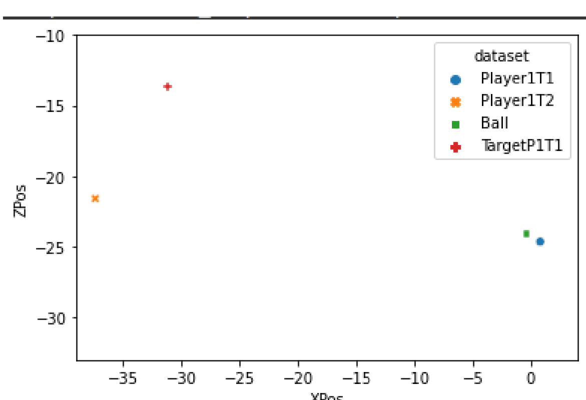

dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0

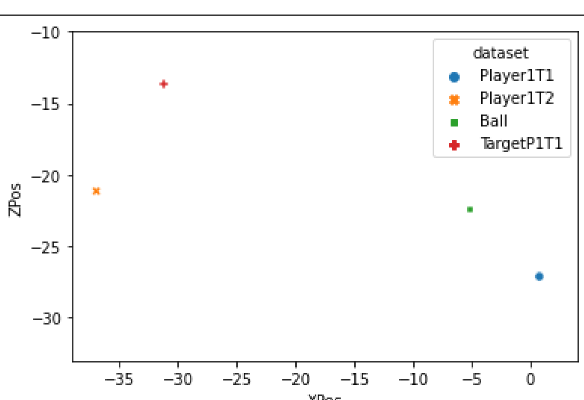

dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0


dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0


dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0


dataset
Player1T1
Player1T2
Ball
TargetP1T2
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0


dataset
Player1T1
Player1T2
Ball
TargetP1T1
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0


dataset
Player1T1
Player1T2
Ball
TargetP1T1
ZPos
XPos
−10
−15
−20
−25
−30
−35
−30
−25
−20
−15
−10
−5
0

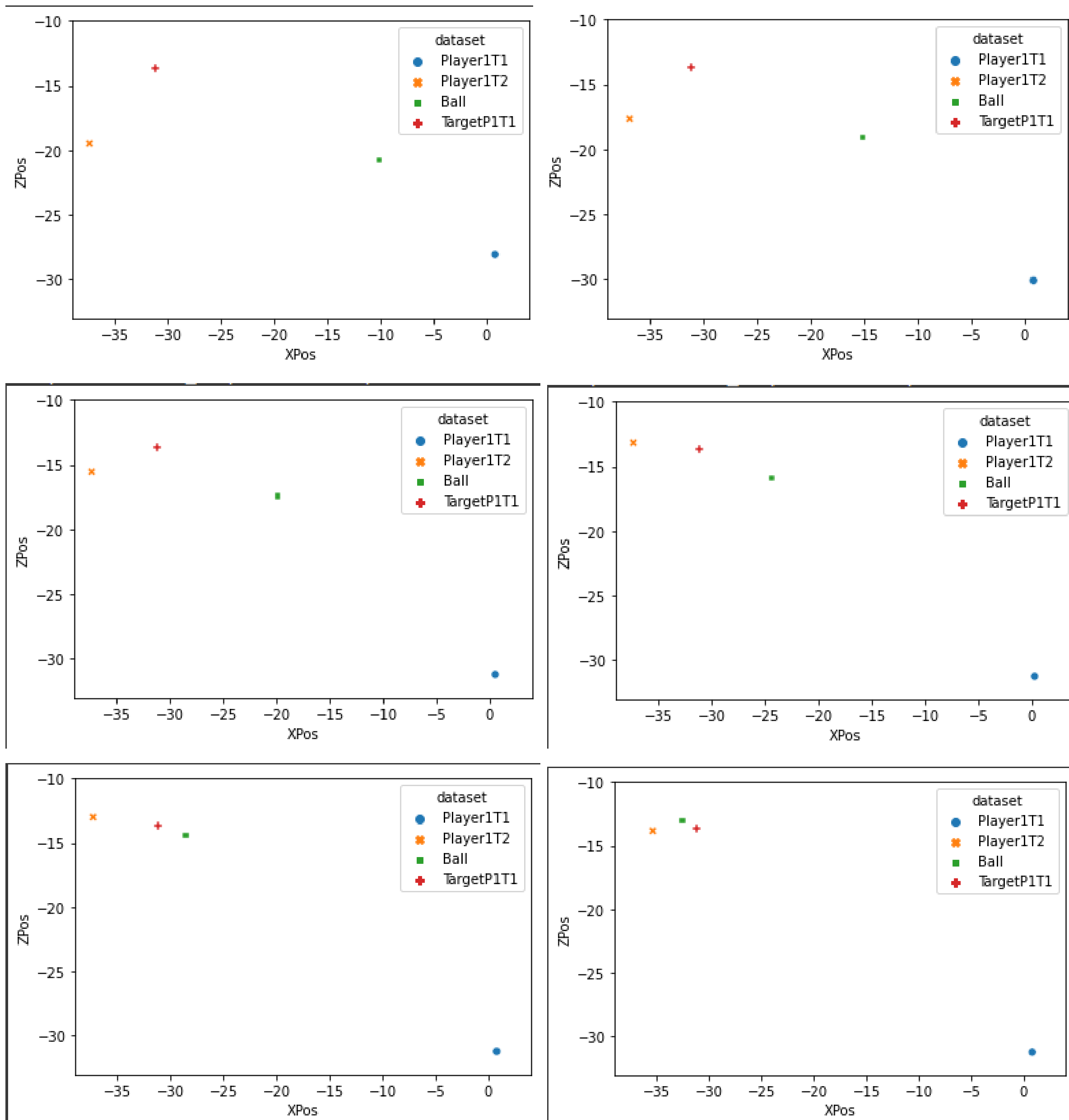


Figura 36: Segunda demostración del nuevo apuntado a partir de un conjunto de gráficas.

Ahora, el punto escogido por el jugador naranja está algo más cerca de la parte izquierda de la pista. Si nos fijamos, la pelota a lo largo del transcurso del tiempo se está moviendo hacia ese punto. El jugador azul también se mueve hacia ese destino para golpear la pelota, y el punto escogido para definir la trayectoria de la pelota es en diagonal. De igual forma, con las siguientes capturas se comprueba que la trayectoria cumple con el destino que había elegido el agente, además, el jugador naranja se mueve hacia ahí para golpear la pelota de nuevo.

Con este conjunto de gráficos, hemos demostrado de nuevo que el agente se está moviendo hacia la pelota, y que con la nueva implementación, el agente

es capaz de escoger hacia dónde quiere enviar la pelota con el siguiente golpeo.

10.4.3 Implementación de los distintos tipos de golpe

El siguiente punto que se quiere añadir es la posibilidad de utilizar distintos tipos de golpe. Hemos considerado un total de cuatro golpes distintos, donde la diferenciación entre éstos vendrá dada por los valores de los distintos parámetros que se utilizan para aplicar la trayectoria de la pelota (fuerza y vector up); en ningún caso se distingue entre golpes de derecha o de revés:

- Golpe “normal”: Este golpe es el que hemos utilizado hasta ahora, he decidido llamarlo “normal”, ya que la velocidad a la que llega la pelota no es muy alta, y la trayectoria tiene altura media.
- Globo: Este es uno de los golpes más característicos del pádel. Se trata de un golpe donde la pelota se envía hacia el final del campo contrario. Por tanto la fuerza que se le aplica a la pelota es bastante baja, sin embargo, se busca una gran altura mediante el golpe, haciendo así, que llegue hasta el final de la pista pasando por encima de los rivales.
- Víbora: Este golpe se caracteriza por la fuerza que se le aplica a la pelota. La idea es llevar el bote de la pelota cerca del cristal del final, por tanto rebotará contra el cristal con poca altura, y hará que el segundo bote sea prácticamente instantáneo.
- Dejada: Para finalizar se ha implementado una dejada. Este es un tipo de golpe donde, tanto la fuerza como la altura van a ser valores bajos, ya que se busca que la pelota pase la red, y el bote sea lo más cercano posible a la misma.

Ya que los parámetros con los que se van a definir los golpes van a ser constantes, la red neuronal lo único que hará será elegir qué tipo de disparo quiere hacer. Además, esperamos que aprenda que si está alejada de la red, no es buena idea hacer una dejada, ya que la pelota botará en su campo y perderá el punto, así que, aprenderá qué tipo de golpe es el más adecuado para un momento determinado.

10.4.4 Entrenamiento del agente de los distintos tipos de golpe

Para el entrenamiento de este nuevo agente, vamos a seguir la misma idea del último agente. Es decir este agente no va a partir de cero, sino que va a cargar el conocimiento que tiene nuestro último modelo entrenado, que ya sabe moverse hacia la pelota, y decidir la trayectoria de la pelota.

Mediante las gráficas que sacamos con tensorboard, podemos observar que la puntuación va variando de la misma forma que vimos en algunos modelos anteriores, pero la duración de los episodios ha llegado un punto donde ha

comenzado a decrementar. Esto puede venir por dos factores, sobreentrenamiento o que el agente haya aprendido en qué momento utilizar un golpe. Ya que puede darse el caso, que un agente aplique una víbora al punto contrario donde se encuentra su adversario, y que a este no le dé tiempo a devolverla.

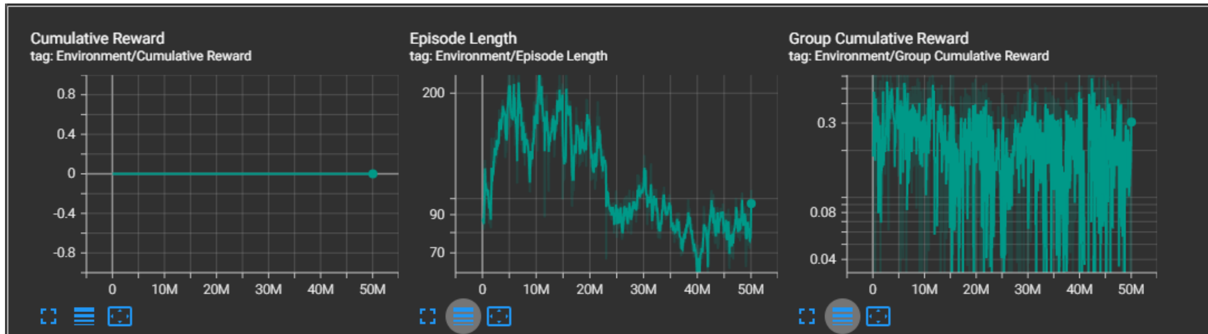


Figura 37: Resultados del agente con nuevos tipos de golpe.

Siguiendo el mismo procedimiento que se ha seguido en los anteriores agentes, se toma la información del entorno de 50 puntos. En este caso, también se informa que tipo de golpe es el que va a utilizar el agente.

Primero, se muestra el movimiento de los agentes a lo largo de los 50 puntos.

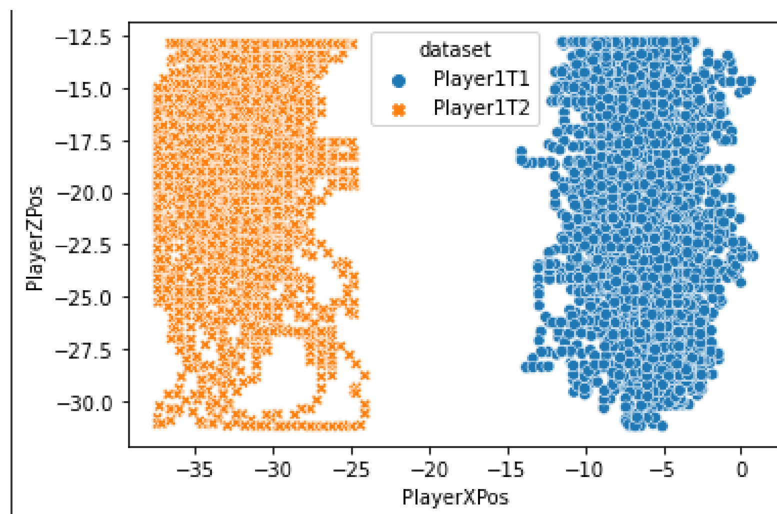


Figura 38: Movimiento de los jugadores a lo largo de 50 puntos.

En este caso, se puede observar cómo los agentes se han acercado más hacia la red, que era uno de los comportamientos que buscábamos.

Para poder mostrar el uso de los distintos tipos de golpe, se van a mostrar unos conjuntos de jugadas aplicando unos golpes u otros. Por ejemplo, si se

aplica una víbora, la pelota se va a mover a mayor velocidad que si se hubiera hecho un globo.

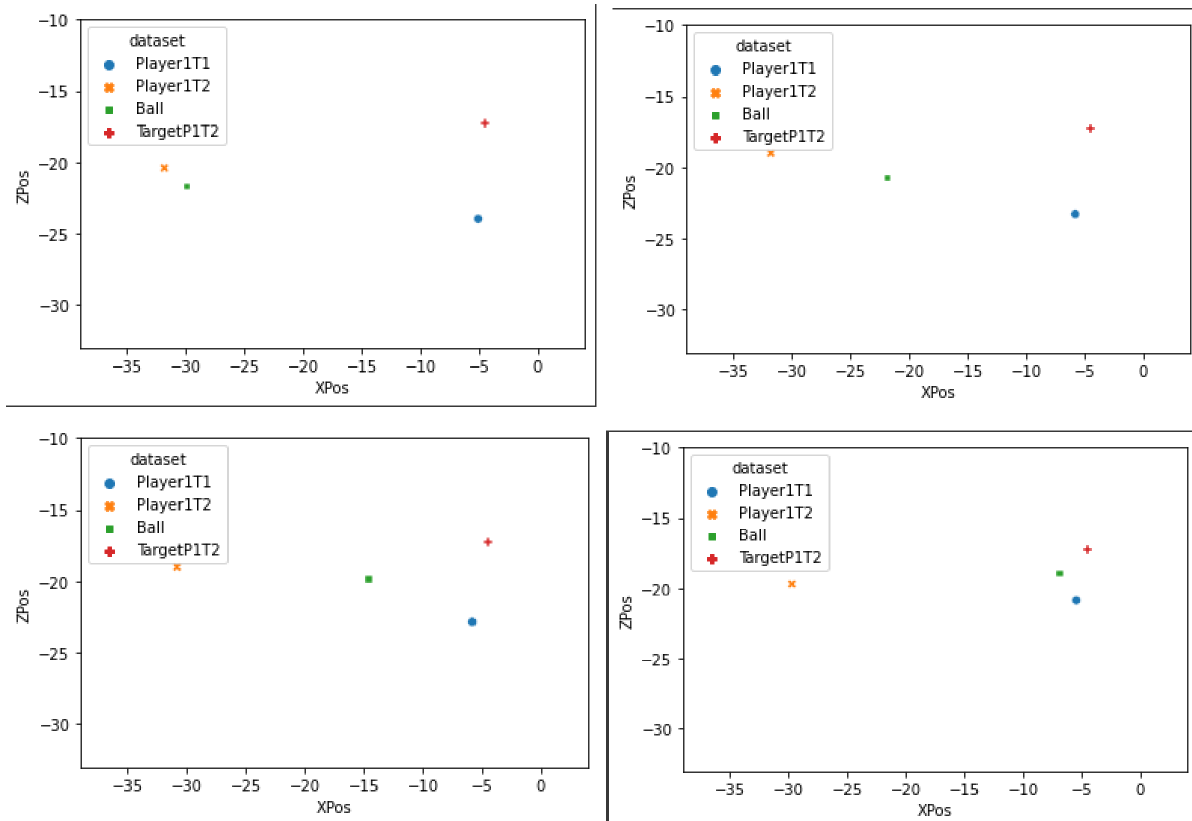


Figura 39: Demostración al aplicar una víbora.

Por parte del agente del jugador naranja, la red neuronal ha decidido utilizar una víbora, la diferencia de la distancia recorrida entre este tipo de golpe y el que se estaba utilizando hasta ahora, es bastante notoria, ya que solo con tres capturas la pelota ya ha cruzado el campo por completo, cuando antes necesitamos unas cuantas más. Otra forma de comprobarlo, es comparar los valores que alcanza la velocidad de la pelota; vamos a recoger los valores mínimos y máximos del anterior entorno, es decir los del golpe normal, y los valores justo después que el jugador naranja golpeara la pelota con una víbora.

BallVelocityX	20.25459
BallVelocityY	5.615092
BallVelocityZ	2.424042

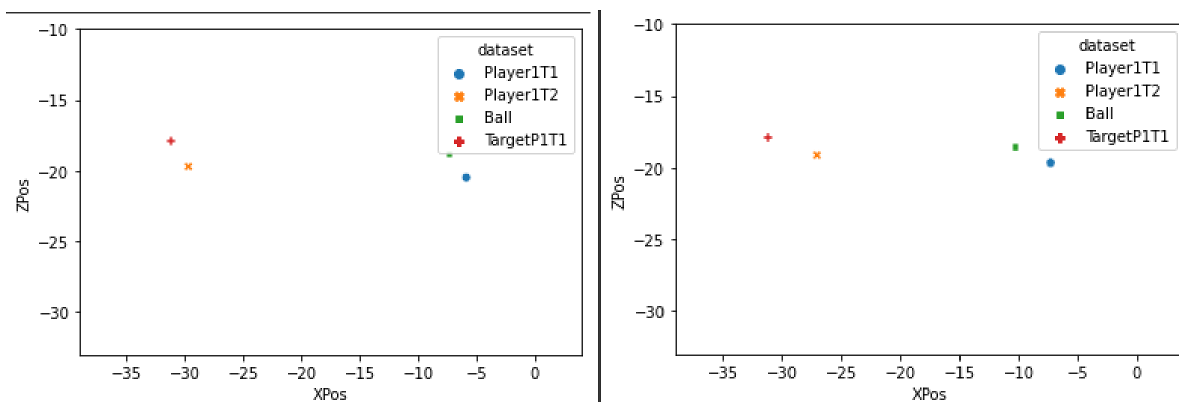
Figura 40: Velocidad de la pelota justo al aplicarle una víbora.

	BallVelocityX	BallVelocityY	BallVelocityZ
count	15723.000000	15723.000000	15723.000000
mean	0.041606	0.031396	0.249337
std	11.030391	5.282659	2.392583
min	-11.999630	-11.376530	-5.427330
25%	-11.573060	-3.702236	-1.526460
50%	1.694605	0.419447	0.231715
75%	11.553130	4.450912	1.924695
max	11.999450	9.813076	8.525377

Figura 41: Valores de la velocidad de la pelota en el entorno que solo había un tipo de golpe.

Podemos observar como el valor de la velocidad en la coordenada x de la pelota, al aplicar una víbora es de 20, el cual está fuera del rango de valores del golpe que se había utilizado siempre.

Si ahora seguimos donde habíamos dejado el punto anterior, la red neuronal ha decidido que el jugador azul utilice un globo. Para comprobar que realmente se ha utilizado este tipo de golpe, se va a enseñar la trayectoria mediante unos gráficos, la velocidad con la que sale la pelota (para compararla con el rango de velocidades del golpe normal), y se comparará los valores máximos de altura que ha alcanzado la pelota, ya que la altura es uno de los principales distintivos en el globo.



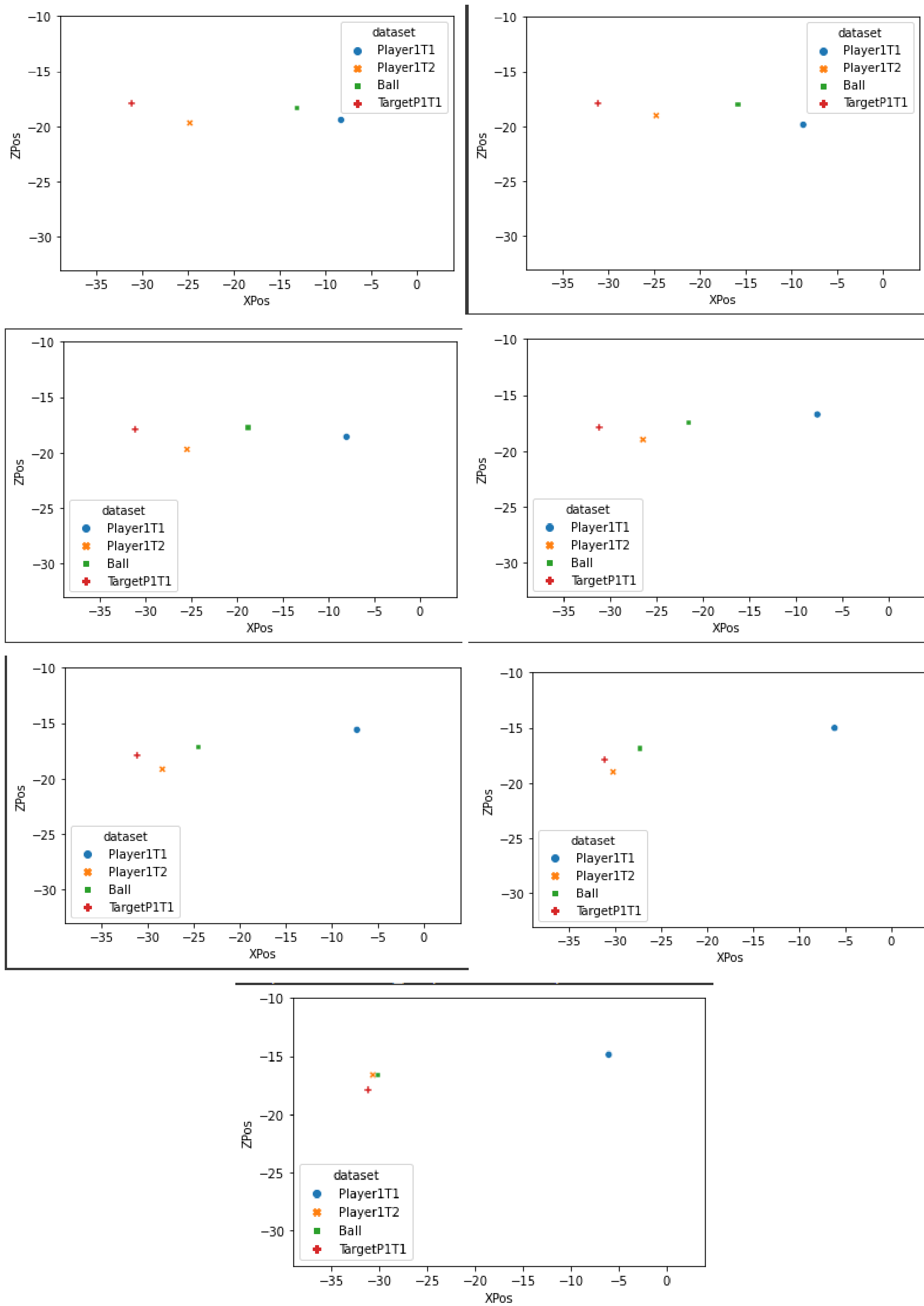


Figura 42: Demostración de aplicar un globo a la pelota.

Para el caso del globo, se observa como tarda más en completar su trayectoria, ya que se golpea a la pelota con menor fuerza.

Si comparamos el rango de velocidades en el eje Y, el valor que toma ahora la pelota al aplicarle un globo, es mayor que el máximo que se puede obtener con un golpe normal.

BallVelocityX	-7.761805
BallVelocityY	14.83594
BallVelocityZ	0.767264

Figura 43: Velocidad de la pelota al aplicarle un globo.

Con la figura 44, en la parte de la izquierda tenemos la información de la posición de la pelota en el entorno con un solo tipo de golpe, mientras que en la derecha es en este nuevo entorno. En el caso de la coordenada Y de la pelota, vemos que con esta implementación su máximo es de 140, mientras que antes casi llegaba hasta 132, por tanto ahí es cuando podemos corroborar que se ha aplicado un globo a la pelota.

	BallXPos	BallYPos	BallZPos
count	15723.000000	15723.000000	15723.000000
mean	-18.516186	128.883709	-19.752371
std	10.519679	1.792736	3.577181
min	-37.726740	124.967600	-31.383780
25%	-27.682090	127.726900	-21.966090
50%	-18.494310	128.623200	-19.776460
75%	-9.360802	130.496650	-17.311390
max	1.139331	131.904700	-12.545290

	BallXPos	BallYPos	BallZPos
count	6578.000000	6578.000000	6578.000000
mean	-18.221806	130.338808	-19.697935
std	9.298326	3.872525	3.644147
min	-37.643090	124.870200	-31.406480
25%	-25.594135	127.581025	-21.943495
50%	-18.185590	129.067700	-19.386590
75%	-11.165093	132.098700	-17.219977
max	1.230584	140.330200	-12.398440

Figura 44: Comparación de los valores de la posición de la pelota para demostrar el uso del globo.

Nos faltaría mostrar los otros dos tipos de golpe, el normal que en realidad es el que se ha estado utilizando hasta ahora, y la dejada. El golpe normal no creo que necesite mucha más explicación, sin embargo la dejada tiene un problema. Como con la dejada no se le aplica mucha fuerza a la pelota, para que pase al campo contrario el jugador debe encontrarse bastante cerca de la red. Por eso mismo, el agente ha llegado a la conclusión en la que prácticamente nunca escoge una dejada, de hecho en partidos de padel de alto nivel, la dejada es un golpe relativamente infrecuente, representando menos del 4% de los golpes. En la imagen de debajo, se muestra en porcentajes que tipo de golpe ha decidido la red neuronal para un jugador. En el caso de la dejada es del 7%, pero también tenemos que tener en cuenta que esta información se recoge cada 30 frames, así que si solo tuviéramos en cuenta la decisión del momento exacto en la que se golpea a la pelota, sería incluso menor.

smash	32.836728
normal	30.678018
globe	29.096990
drop	7.388264

Figura 45: Porcentajes de uso de cada tipo de golpe.

Hasta este punto, podemos decir que tenemos un agente que sabe jugar a pádel con las distintas implementaciones que se han ido añadiendo. Ahora el siguiente objetivo es el de generar este mismo comportamiento pero para un partido de 2 contra 2.

10.5 Implementación del entorno 2 vs 2

En cuanto a distinción del entorno respecto al de 1 contra 1, lo único en que se van a diferenciar es que en cada equipo van a haber dos jugadores. Por tanto para cada equipo se va a hacer una copia de cada jugador y de los puntos invisibles que se utilizan para definir la trayectoria del golpe de la pelota.

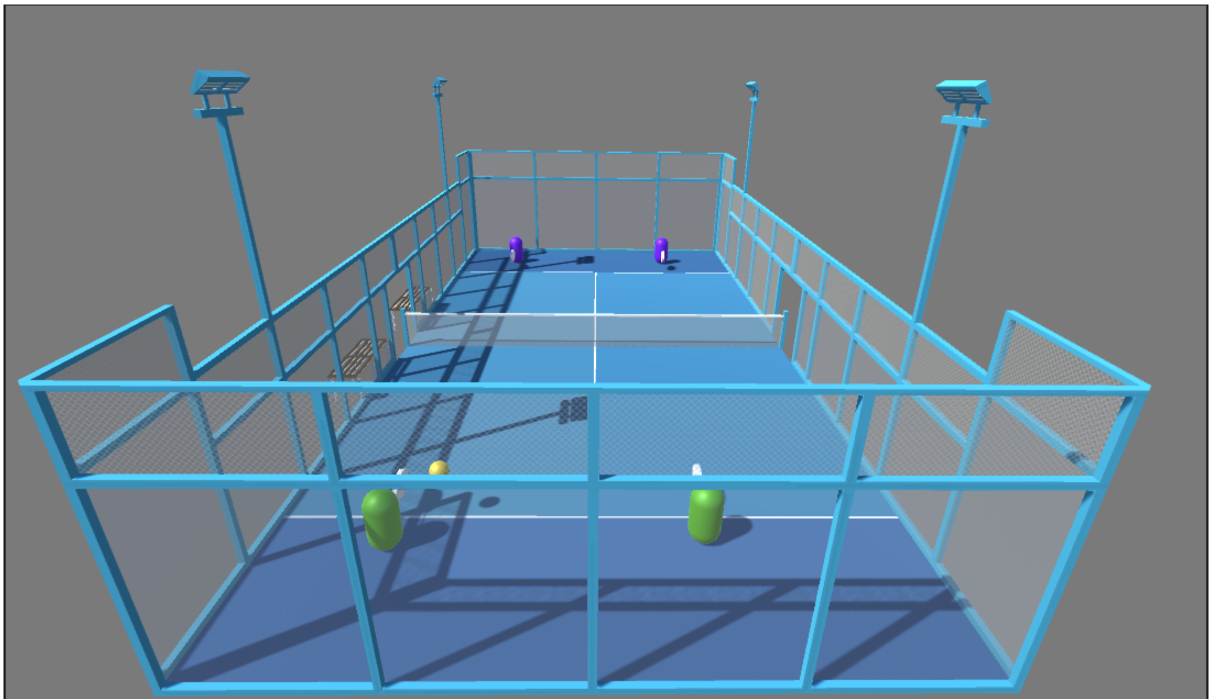


Figura 46: Entorno del dos contra dos.

A nivel de código realmente no varía nada. En el único apartado que deberíamos tener algo de cuidado sería con el controlador del entorno, pero cuando se implementó, ya se preparó pensando en este nuevo modo, ya que no se trabajaba a nivel de agente, sino a nivel de un grupo de agentes. Por

tanto lo único que haremos será añadir estos nuevos jugadores a la lista de agentes del controlador.

Ahora que ya tenemos todo preparado para este nuevo entorno, lo primero que se probó fue cargar el último modelo que habíamos entrenado para el uno contra uno, para comprobar cuál sería el comportamiento. Se esperaba que el comportamiento fuera ir hacia la pelota sin importar la presencia de su compañero. Es decir, aunque su compañero esté más cerca de la pelota, se moverá hacia esa posición. Esto es algo, lo cual se puede llegar a esperar, ya que realmente este agente no se ha entrenado con otro jugador en su equipo. Y efectivamente esto fue lo que sucedió, de hecho lo que realmente hacen es moverse a la pelota, que es lo que han aprendido en el entorno anterior. Por tanto, ahora se va a generar un nuevo agente esperando que este tenga en cuenta el hecho de jugar con un compañero.

10.6 Entrenamiento del agente de 2 vs 2

Para este nuevo entorno se han entrenado distintos agentes, ya que no se estaban consiguiendo buenos resultados. Ahora se van a resumir brevemente cuáles han sido las diferencias entre unos y otros, mostrando solo los gráficos del último de todos.

Cuando se cargó el modelo del uno contra uno, se comprobó que no funcionaba correctamente, ya que el comportamiento de los agentes de un mismo equipo era prácticamente exacto. así que sin cambiar nada respecto a las observaciones ni el sistema de puntuación, se entrenó un nuevo agente. Este resultó en un agente al cual realmente no le daba la importancia necesaria al hecho de estar jugando con un compañero.
El siguiente paso fue añadir al vector de observaciones, la distancia del compañero respecto a la pelota y las posiciones de los demás jugadores, tanto el de su equipo como los del contrario. Al finalizar el entrenamiento, que en este caso todos estos agentes se han entrenado hasta unos cincuenta millones de iteraciones, se podía observar que en algunas ocasiones, si la pelota se acercaba hacia su compañero, el otro jugador no iba hacia la pelota, sin embargo el resultado no era tan bueno como había sido para el caso del uno contra uno.

Otro de los problemas que tenían estos agentes, era como si estuvieran obviando la importancia del apuntado de la pelota. Es decir, había veces donde el jugador, incluso justo al comenzar un punto, enviaba la pelota directamente hacia el cristal, y aunque se alargará el entrenamiento, esto no se corregía. Para tratar de solucionarlo, se adjuntó la posición del punto invisible que se utiliza para definir la trayectoria de la pelota al golpearla en el

vector de observaciones. Sin embargo, esto no fue suficiente para solucionarlo.

Al final se llegó a la conclusión de modificar el sistema de puntuación. Hasta ahora no se distinguía cómo se había conseguido un punto, ya sea porque algún agente había enviado fuera la pelota, o porque la pelota había botado dos veces en el campo. Por tanto se decide que si un punto ha sido perdido porque alguien ha enviado la pelota fuera del campo, será más penalizado que no por si bota dos veces. Es decir, se penaliza explícitamente lo que en pádel se conoce como errores no forzados.

El sistema de puntuación otorgaba un punto al equipo ganador, y quitaba un punto al equipo perdedor. Ahora vamos a distinguir la posibilidad de anotar un punto al enviar la pelota fuera del campo, respecto a si la pelota ha botado dos veces en un campo. En el caso que la pelota bote dos veces en un campo, el equipo ganador obtendrá 0.75 puntos y al perdedor se le restará 0.75 puntos. Sin embargo, un punto ha finalizado porque se ha enviado la pelota fuera del campo, el equipo ganador obtendrá 0.6 puntos, ya que ha ganado el punto por fallo del otro equipo, y al perdedor se le restará 1 punto, ya que se considera más grave perder un punto de esta forma.

Con este agente sí que se ha observado la mejora esperada, ya que no está enviado la pelota fuera del campo en el primer golpe después del saque.

Mediante los gráficos generados a partir de Tensorboard, se puede observar cómo las tres variables han ido mejorando a lo largo del tiempo, lo cual es una buena señal. Además, a nivel visual se puede comprobar que el agente no envía tantas pelotas fuera del campo como antes hacía.

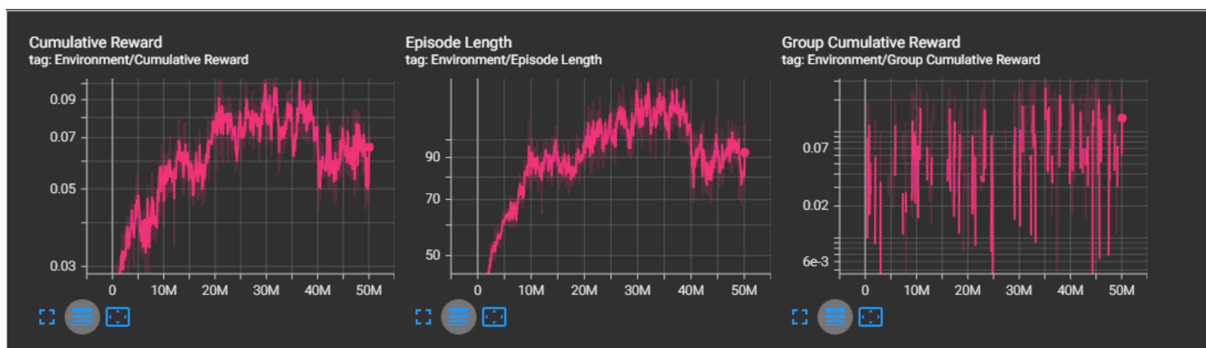


Figura 47: Resultados del entorno de dos contra dos.

Ahora que ya tenemos el agente entrenado, vamos a recoger la información de 50 puntos, como ya hemos hecho con los demás modelos. En este caso vamos a mostrar el movimiento de los cuatro jugadores, y mostraremos un

punto de los 50 jugados, ya que la única diferencia con el anterior resultado es que ahora tenemos dos jugadores por equipo.

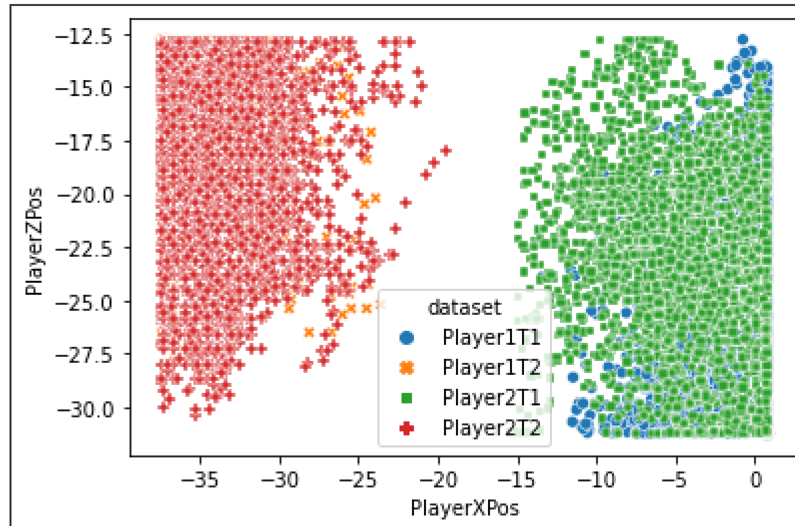


Figura 48: Movimiento de los jugadores a lo largo de 50 puntos en el entorno de dos contra dos.

Si nos fijamos en el movimiento de los cuatro agentes, no le dan la importancia que se esperaba al hecho de tener un compañero, ya que muchas veces uno invade el lado de otro y viceversa.

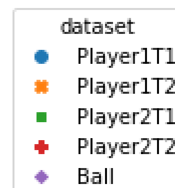
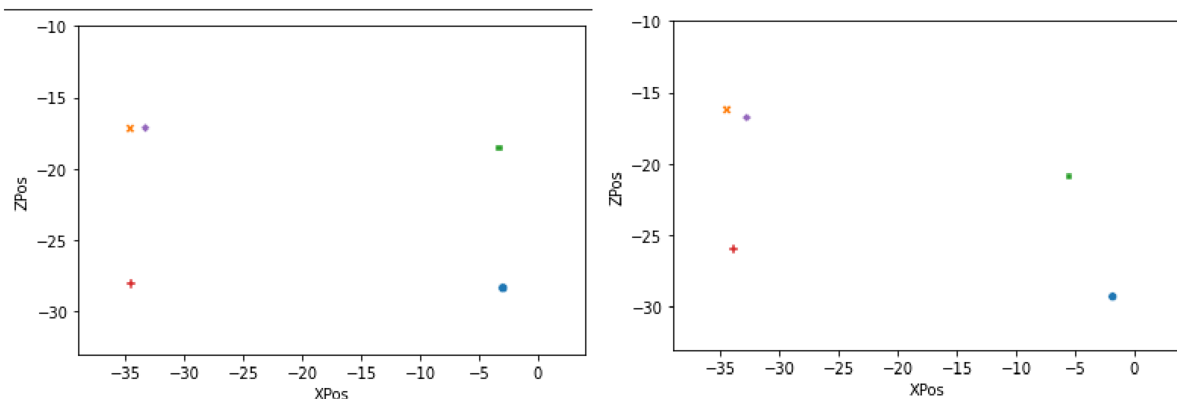
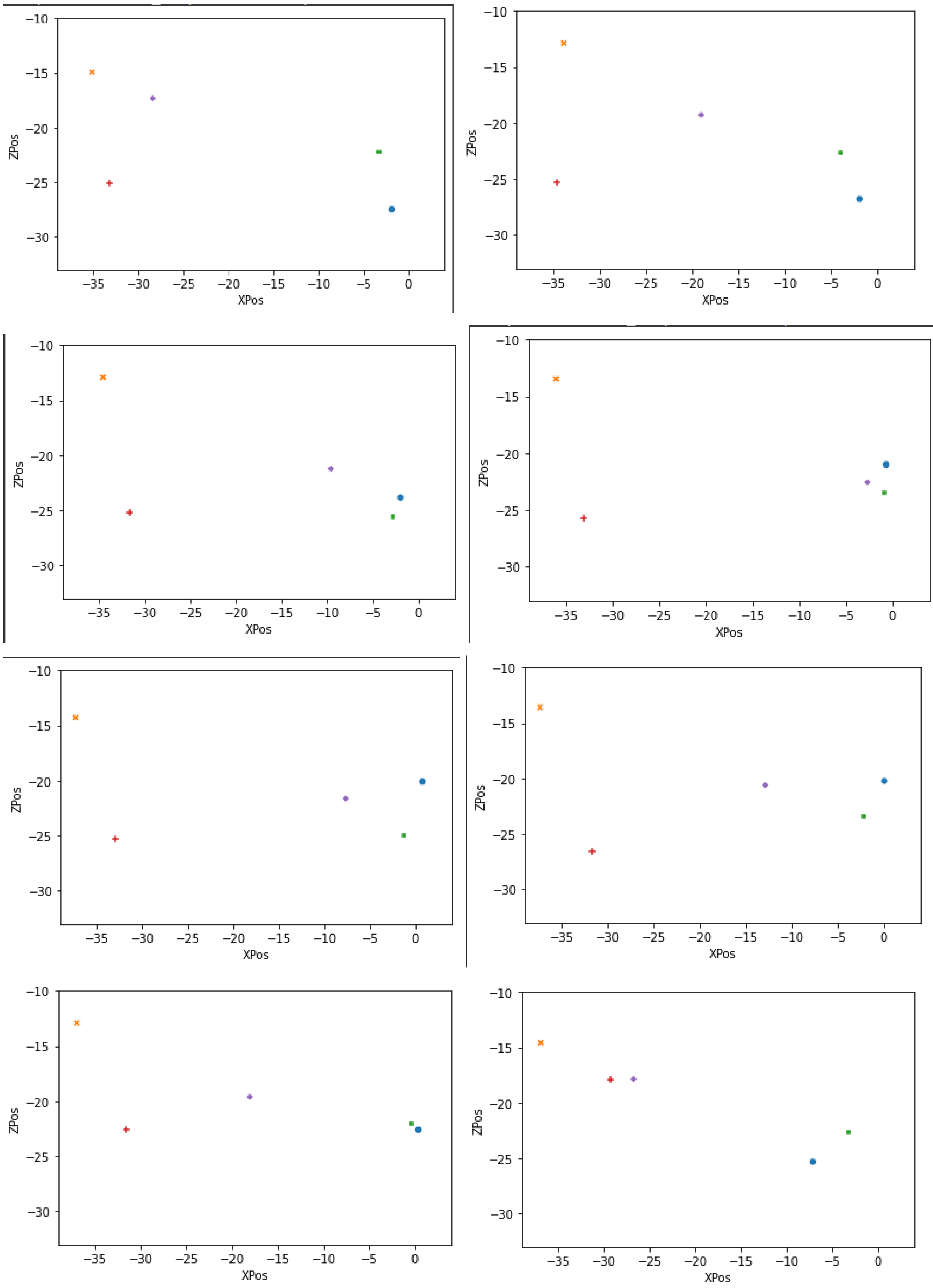
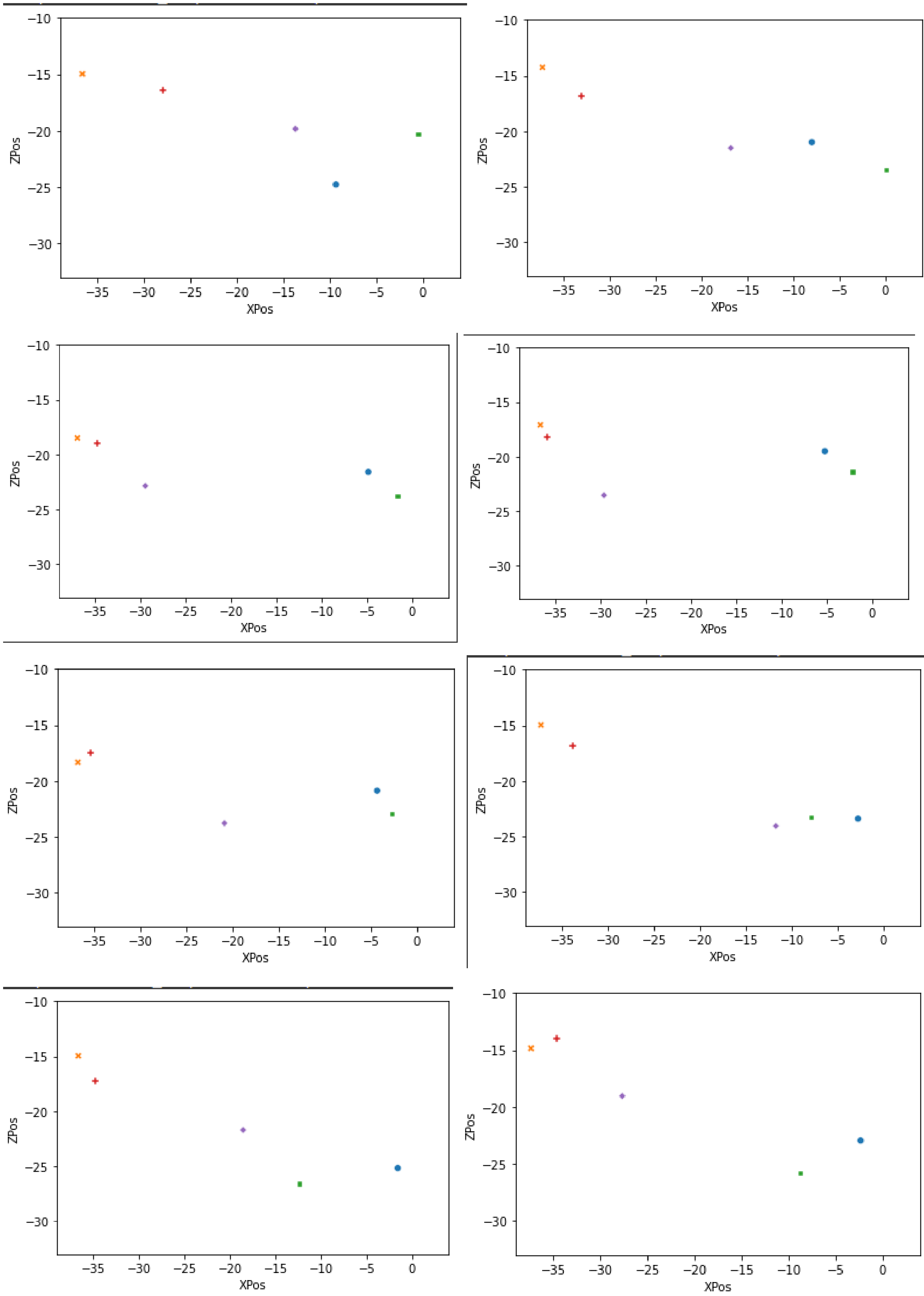


Figura 49: Leyenda del conjunto de gráficas.




−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos
−10
−15
−20
−25
−30
ZPos
−35 −30 −25 −20 −15 −10 −5 0
XPos



ZPos
XPos
−10
−15
−20
−25
−30
−35
−5
0

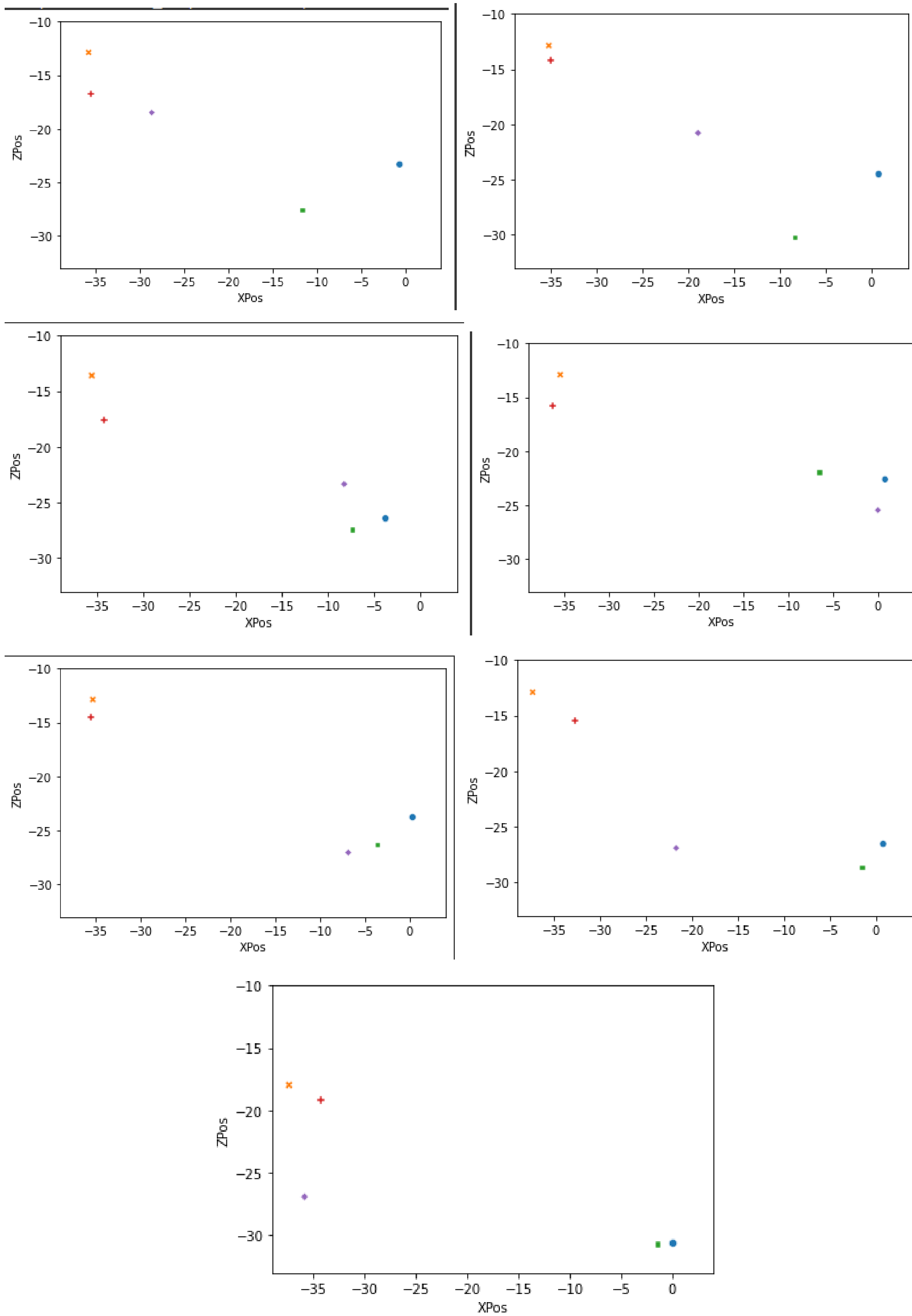


Figura 50 Conjunto de gráficas del entorno de dos contra dos.

El conjunto de gráficas anterior representa uno de los 50 puntos sobre los que se ha recogido la información del entorno. En algunas ocasiones, entre una imagen y la siguiente la pelota se ha movido bastante, y esto viene dado a que se le ha aplicado una víbora a la pelota. También podemos observar como algunas veces un agente se mueve hacia la pelota, aunque su compañero esté más cerca que él.

Cabe destacar que en el Anexo tenemos un vídeo con los resultados de cada agente.

10.7 Resumen de los parámetros en los distintos agentes

Vamos a resumir en una tabla los distintos vectores de observación que se han enviado en los distintos modelos de agentes que se han entrenado.

Parámetros	**Entorno básico**	**Apuntado dinámico**	**Diversos tipos de golpe**	**2 vs 2**
Posición jugador	X	X	X	X
Posición pelota	X	X	X	X
Velocidad pelota	X	X	X	X
Rayo detección red	X	X	X	X
Distancia propia con la pelota	X	X	X	X
Posición compañero				X
Posición oponentes				X
Distancia compañero con la pelota				X
Posición del punto para el apuntado				X

Tabla 7: Resumen de los elementos del vector de observaciones para las diferentes configuraciones

10.8 Configuración del entrenamiento

En este apartado vamos a resumir brevemente el fichero que contiene la configuración del entrenamiento, es decir, donde definimos qué algoritmo aplicaremos, la arquitectura de la red neuronal, recompensas y más.

Lo primero que debemos decidir es qué tipo de entrenamiento aplicaremos y por tanto, que algoritmo vamos a utilizar. En el apartado 2.3 hemos comentado algunos algoritmos que Unity nos ofrece, y como en nuestro caso se trata de un entorno cooperativo, se ha decidido utilizar MA-POCA.

Luego, a través de algunos parámetros (*keep_checkpoints, checkpoints_interval*) se nos permite guardar la evolución del agente durante el entrenamiento, para así guardarnos distintos modelos y quedarnos con el que mejor comportamiento tenga.

A continuación tenemos un conjunto de hiper-parámetros a definir, los cuales vamos a resumir brevemente. El *learning_rate* es el valor que define la magnitud de cada paso del descenso del gradiente, el *batch_size* hace referencia al número de experiencias utilizadas en cada iteración del descenso del gradiente, el *buffer_size* es el número de experiencias que se deben recoger antes de actualizar la política del modelo, la *beta* define la aleatoriedad de la política, el *epsilon* dicta la rapidez de la evolución de la política durante el entrenamiento, la *lambda* es la confianza que tiene el agente hacia el valor actual de la estimación al calcular un nuevo valor de estimación, el *num_epoch* es el número de pasos a realizar mediante las experiencias al hacer el descenso del gradiente, y finalmente el *learning_rate_schedule* determina cómo cambia el *learning_rate* a lo largo del entrenamiento; podemos dejarlo constante o que sea lineal, llegando hasta 0 cuando lleguemos a las iteraciones máximas definidas en el parámetro *max_steps*.

Por otra parte tenemos la configuración de la red neuronal. Podemos elegir mediante el *normalize* si queremos normalizar los valores de entrada, las *hidden_units* son el número de neuronas que habrán en cada capa oculta, el *num_layers* es el número de capas intermedias (u ocultas) que tendrá la red, por último tenemos el *vis_encode_type,* para definir qué tipo de *encoder* queremos utilizar para las observaciones visuales. En nuestro caso hemos dejado el valor por defecto, que es un encoder simple con dos capas de convolución, aunque en nuestro caso no se va a utilizar ya que sólo hemos trabajado con vectores de observación.

También existe un apartado para definir las recompensas con las que se va a trabajar durante el entrenamiento. Existen tanto las extrínsecas como las

intrínsecas, donde las extrínsecas son las que definimos nosotros para cuando ocurra algo en el entorno (termina un punto), mientras que las intrínsecas se obtienen de forma auxiliar con métodos como la *curiosity* o *GAIL* para ayudar a entornos que no reciben recompensas tan seguidamente. En la extrínseca tenemos dos campos a definir, la *strength* que será el factor por el cual se va a multiplicar las recompensas recibidas por el entorno (normalmente se deja a 1), y el *gamma*, es un factor para reducir las futuras recompensas del entorno.

Por último, hemos entrenado utilizando el *self-play* que hace que los agentes compitan con la mejor versión encontrada hasta el momento. Dentro de este apartado podemos definir el *save_steps*, que es el número de pasos durante el cual se va a guardar una copia de la política actual, *swap_steps,* número de pasos "fantasma" para modificar la política de un oponente con otra versión, esto sobretodo es importante en entornos donde el número de agentes entre equipos difiere. Los pasos fantasma hacen referencia a los pasos de un agente que está siguiendo una política fija, sin estar aprendiendo. Luego, con *window* definimos el número de versiones del agente que vamos a almacenar, una vez se llega al máximo se elimina la más antigua, el *play_against_lastest_model_ratio* define la probabilidad de que un agente juegue contra la última política almacenada.

```yaml
behaviors:
  BasicPlayer:
    trainer_type: poca
    hyperparameters:
      batch_size: 2048
      buffer_size: 20480
      learning_rate: 0.0003
      beta: 0.005
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: constant
    network_settings:
      normalize: false
      hidden_units: 512
      num_layers: 2
      vis_encode_type: simple
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    keep_checkpoints: 25
    checkpoint_interval: 2000000
    max_steps: 500000000
    time_horizon: 1000
    summary_freq: 10000
    self_play:
      save_steps: 50000
      swap_steps: 2000
      window: 10
      play_against_latest_model_ratio: 0.5
      initial_elo: 1200.0
torch_settings:
  device: null
```

Figura 51: Fichero de configuración del entrenamiento.

11. Conclusiones

En este proyecto se han presentado diversas implementaciones de un agente mediante aprendizaje por refuerzo para que éste aprendiera a jugar a padel. Para ello hemos partido de cero, es decir, se ha generado el entorno y las funcionalidades necesarias para conseguir un videojuego funcional, y entonces se han hecho distintas pruebas hasta conseguir un agente con un comportamiento similar al esperado. Aunque como ya se ha comentado en apartados anteriores, hemos partido de un entorno sencillo para adecuarnos a las herramientas y cuando se han conseguido buenos resultados con los

agentes, se ha mejorado el entorno con nuevas funcionalidades a nivel de jugabilidad, y por tanto se han entrenado nuevos agentes haciéndoles conscientes de las nuevas posibilidades.

A nivel general se han obtenido buenos resultados, ya que en los distintos agentes entrenados se puede observar el comportamiento de seguir la pelota para devolverla al campo contrario. Sin embargo, hay uno de los entornos en el que podríamos decir que los resultados dejan espacio a la mejora. Nos referimos al dos contra dos, el hecho que nos hace concluir que los resultados obtenidos no son los esperados, es porque no observamos que el agente aproveche el hecho de tener un compañero durante el partido. Con las gráficas del movimiento de los agentes, se puede observar como éstos invaden el campo de su compañero sin importarles su presencia. Es probable que los agentes no sean capaces de inferir esto, porque no tienen suficiente información o feedback, es decir, si por ejemplo les damos una recompensa positiva o negativa dependiendo de en qué parte del campo se encuentren, no invadirán tanto el campo de su compañero.

En cuanto a las implementaciones extras que se hicieron (apuntado dinámico y tipos de golpe), los resultados han sido los esperados. Por ejemplo, en el apuntado dinámico se prefirió no limitarlo a posiciones que siempre se encuentren dentro del campo, para así comprobar que la red neuronal ha sido capaz de aprender los límites del campo por ella misma, haciendo así también posible que un jugador mande la pelota fuera, que es algo que puede ocurrir en el mundo real. En la parte de los distintos tipos de golpe, en el momento de realizar la implementación, supe que la red podría llegar a tener problemas con la dejada, ya que es un tipo de golpe que solo pasa al campo contrario si el jugador se encuentra cerca de la red, y hasta ese punto, la mayoría de las veces, los jugadores jugaban desde la parte de atrás de la pista. A partir de este punto, los agentes se empezaron a mover más en el eje vertical, y además, la red aprendió que la dejada es un tipo de golpe que sólo se debe efectuar en momentos muy concretos, ya que para que la decisión de utilizarlo resulte en una recompensa positiva se daba en muy pocas ocasiones, y por eso es el golpe que menos utiliza (algo que también ocurre en el pádel de alto nivel).

11.1 Futuras mejoras

Durante el desarrollo del proyecto se han ido pensando en múltiples mejoras las cuales se podrían aplicar, e incluso distintas formas de implementar las distintas funcionalidades que se han hecho.

Ahora mismo la aplicación de un tipo de golpe u otro es constante, es decir, los parámetros que se utilizan para definir los golpes siempre son los mismos.

Esto se decidió hacerlo así porque se considera que el entrenamiento va a ser más sencillo al solo tener que elegir entre un golpe u otro. Además, si hacemos que la red elija sólo entre la altura y la fuerza del golpe, probablemente no distinguiremos tan fácilmente distintos tipos de golpe. A lo mejor, lo correcto sería definir un rango de fuerza y altura para cada tipo de golpe, entonces la red debería elegir entre el tipo de golpe y los valores de fuerza y de altura para el golpe escogido, haciendo así que los parámetros de los golpes no sean constantes, pero que seamos capaces de distinguir cuando se aplica un globo o una víbora.

En cuanto a nuevos añadidos, vamos a nombrar las ideas que se podrían aplicar en un próximo futuro, ya que hay otro conjunto de mejoras que no son triviales, como podría ser el hecho de hacer el juego más realista a partir de la representación del tiempo de reacción de los jugadores, el efecto que puede conseguir la pelota, o que el movimiento de los jugadores no sea a partir de una velocidad constante, sino que también tengamos en cuenta una aceleración. Destacar que muchos de los puntos que ahora se van a comentar no se han implementado por la limitación de tiempo que disponemos para llevar a cabo el proyecto.

El primero punto a mejorar sería el resultado del entorno de dos contra dos, aplicando una nueva recompensa dependiendo en qué parte del campo se encuentre el jugador, para tratar de evitar que un jugador se mueva hacia la parte del campo de su compañero. Hay que diferenciar cuando un agente invade el campo contrario sin motivo, o por si su compañero no va a llegar a devolver la pelota. Para tener en cuenta esos dos puntos, la recompensa negativa al invadir la parte del compañero irá incrementando, es decir, cuanto más tiempo pase el agente en la parte del campo de su compañero, más grande va a ser la recompensa negativa que va a recibir.

Por otra parte también podríamos probar a cambiar la arquitectura de la red neuronal y el algoritmo utilizado. Por ejemplo, para el entorno de uno contra uno, podríamos utilizar otro algoritmo de aprendizaje por refuerzo como el PPO (proximal policy optimization). En cuanto al cambio de arquitectura no solo me refiero a modificar el número de capas y de neuronas, sino también en pasar a utilizar redes convolucionales, haciendo que el input pase del vector de observaciones a la imagen que captura el propio objeto cámara de Unity.

Otro punto que considero importante tener en cuenta, es el tratar que la red neuronal conciba la importancia que tiene en padel el ganar la red. Esto se podría solucionar dando una recompensa positiva cuando el jugador se acerque a la red, pero esto puede llevarnos a un comportamiento que no sea el que nosotros realmente esperamos. Nosotros queremos que el equipo

trate de ganar la red, pero no queremos que siempre estén cerca de la red, ya que si el equipo contrario utiliza un globo, deberían retroceder para devolver la pelota. Por tanto, se pensó en utilizar otra herramienta que nos ofrece ML-agents, el aprendizaje por imitación. Este tipo de entrenamiento me llamó la atención desde que ví que se podía utilizar en Unity, pero tenemos el problema de llevarlo a cabo en nuestro caso, ya que si pensamos en el entorno de uno contra uno, tenemos dos jugadores, uno que va a ser controlado por nosotros, y será mediante el cual mostremos el comportamiento a seguir por el agente, pero entonces tenemos el otro jugador, que podríamos hacer por ejemplo que fuera controlado por un bot (como el que se utilizó para probar el entorno), sin embargo, es muy probable que el resultado no sería tan real como el que podemos obtener mediante el aprendizaje por refuerzo.

Para poder llevar a cabo el aprendizaje por imitación, deberemos grabar distintos tipos de puntos, enseñando así al agente cuando debe subir a la red o no. Por ejemplo, si un equipo hace un globo, obligando al equipo contrario que se muevan hacia atrás, estos deberían subir a la red, y si al contrario, un equipo está cerca de la red y los contrarios utilizan un globo, deberán volver a la parte de atrás de la pista. Este aprendizaje se aplicaría a un agente que no parte de cero, es decir se trataría de adicionar este comportamiento a un agente que ya sabe que tiene que moverse hacia la pelota para devolverla.

Otra idea que se ha tenido para aplicar al aprendizaje por imitación es, recoger la información de un partido profesional de pádel y grabar una demostración con esos puntos. Suponiendo que los jugadores van a tomar las opciones más óptimas para ganar un punto, deberíamos poder enseñar a nuestros agentes un nivel de pádel mayor, que no el que podamos conseguir con el aprendizaje por refuerzo.

Otro de los conocidos problemas que puede llegar a tener el aprendizaje por refuerzo, es en el que en un entorno no se reciba ningún tipo de recompensa en un largo tiempo de ejecución. En nuestro caso, durante todos los frames donde la pelota está viajando de una parte del campo a otra, ningún agente está recibiendo ningún tipo de feedback. Para solucionar este problema, ml-agents proporciona el uso de dos tipos de recompensas intrínsecas: *curiosity module* y *random network distillation (RND)*. En ambos casos se utilizan dos tipos de redes para el cálculo de la recompensa, no vamos a entrar mucho en detalle sobre su funcionamiento, pero el resumen sería que, en el caso de la *curiosity*, cuanto más se sorprenda al modelo más grande será la recompensa, y en la *RND*, cuantas más veces se visite un estado, más precisas serán las predicciones, y más bajas serán las recompensas que invitan al agente a explorar nuevos estados con errores de predicción mayores [19].

Referencias

[1] Fabian Amherd, Elias Rodriguez: "*Heatmap-based Object Detection and Tracking with a Fully Convolutional Neural Network*", visitado Febrero 26, 2022, dirección: https://www.researchgate.net/figure/Dense-Neural-Network_fig5_348402885

[2] Sumit Saha: "*A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*", visitado Febrero 26, 2022, dirección: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[3] Dan Lee: "*Reinforcement Learning, Part 1: A Brief Introduction*", visitado Febrero 26, 2022, dirección: https://medium.com/ai³-theory-practice-business/reinforcement-learning-part-1-a-brief-introduction-a53a849771cf

[4] Unity Technologies: "*Agentes de aprendizaje automático de Unity*", visitado Enero 23, 2022, dirección: https://unity.com/es/products/machine-learning-agents

[5] Maryam Honari: "*Unity-Technologies, ml-agents*", visitado Enero 23, 2022, dirección: https://github.com/Unity-Technologies/ml-agents

[6] OpenAI: "*Toolkit for developing and comparing reinforcement learning algorithms*", visitado Febrero 26, 2022, dirección: https://gym.openai.com

[7] Progressa Lean: "*¿Qué es la metodología Agile y por qué está de moda?*", visitado Febrero 26, 2022, dirección: https://www.progressalean.com/metodologia-agile/

[8] Unity Technologies: "*Unity User Manual*", visitado Febrero 20, 2022, dirección: https://docs.unity3d.com/Manual/index.html

[9] Gantter: "The project management tool that's perfect for remote collaboration", visitado Febrero 26, 2022, dirección: https://www.gantter.com

[10] Geektopia, "Ryzen 5 1600x", visitado Marzo 11, 2022, dirección: https://www.geektopia.es/es/product/amd/ryzen-5-1600x/

[11] Amazon, "ASUS Dual GeForce RTX 2060 Super", visitado Marzo 11, 2022, dirección: https://www.amazon.es/ASUS-Dual-GeForce-2060-Super/dp/B082LHQP2S

[12] PCcomponentes, "Corsair Vengeance DDR4 2x8GB", visitado Marzo 11, 2022, dirección: https://www.pccomponentes.com/corsair-vengeance-lpx-ddr4-3200-pc4-25600-16gb-2x8gb-cl16-negro

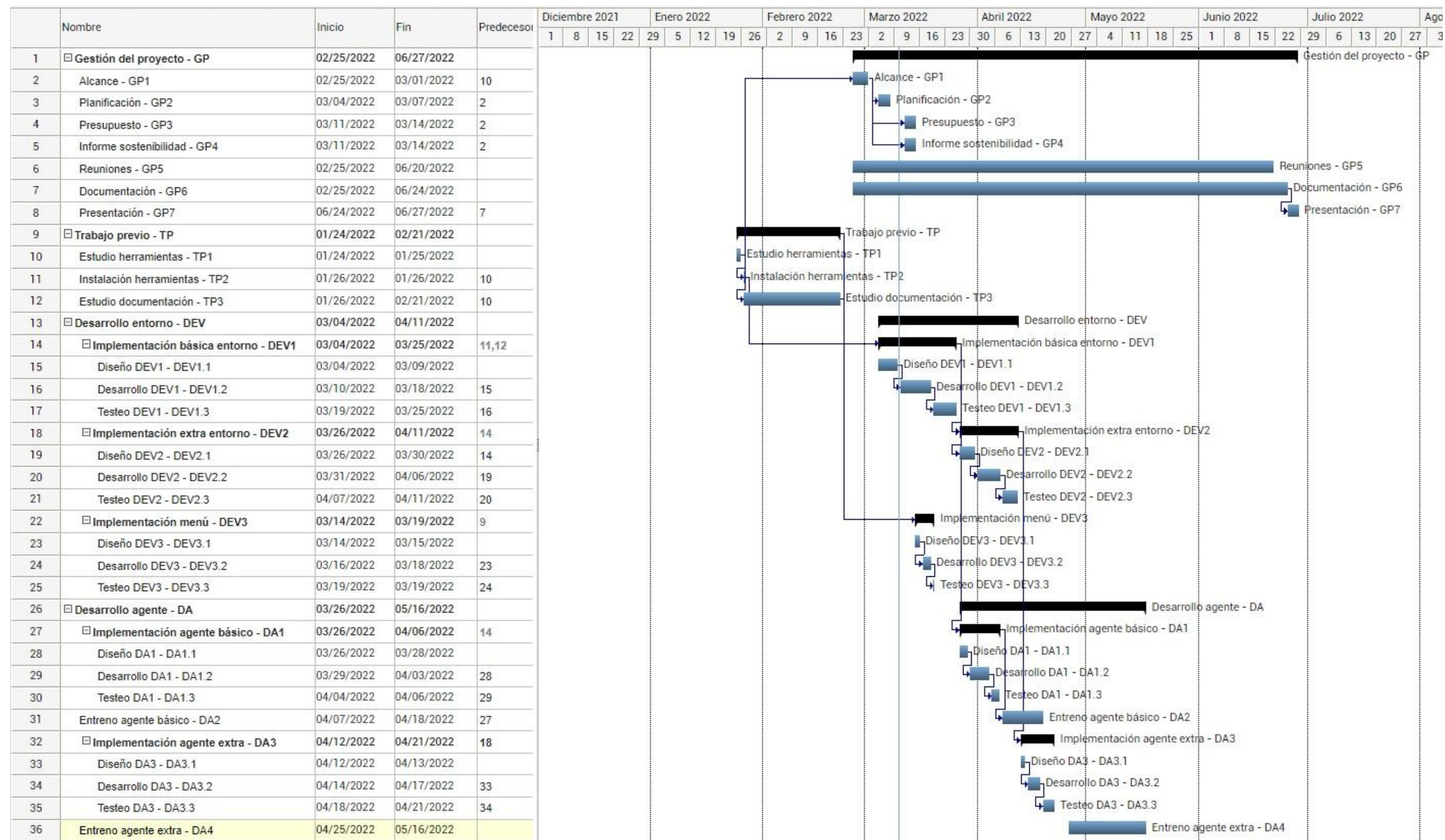
[13] PCcomponentes, "Megaport PC", visitado Marzo 11, 2022, dirección: https://www.pccomponentes.com/megaport-pc-gaming-amd-ryzen-5-5600g-16gb-2tb-rtx-2060

[14] Coworkingspain, "Coworking Girona", visitado Marzo 14, 2022, dirección: https://coworkingspain.es/espacios/coworking/girona/coworking-girona

[15] Anaconda, visitado Mayo 3, 2022, dirección: https://docs.anaconda.com/anaconda/navigator/index.html
[16] DataScience, perceptron, visitado Mayo 24, 2022, dirección: https://datascience.eu/es/aprendizaje-automatico/perceptron/
[17] Sciencesprings, Stochastic gradient descent, visitado Mayo 25, 2022, dirección: https://sciencesprings.wordpress.com/tag/stochastic-gradient-descent/
[18] Analyticsdihya, Is gradient descent sufficient for neural network?, visitado Mayo 24, 2022, dirección: https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/
[19] Github, ML-agents toolkit overview, visitado Mayo 29, 2022, dirección: https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/ML-Agents-Overview.md#deep-reinforcement-learning
[20] Arxiv, Domain Randomization for transferring deep neural networks from simulation to the real world, visitado Junio 1, 2022, dirección: https://arxiv.org/pdf/1703.06907.pdf
[21] AIstackexchange, Credit Assignment Problem, visitado Junio 5, 2022, dirección: https://ai.stackexchange.com/questions/12908/what-is-the-credit-assignment-problem
[22] Mlanctot, Use and misuse of absorbing states in multi-agent reinforcement learning, visitado Junio 5, 2022, dirección: http://aaai-rlg.mlanctot.info/papers/AAAI22-RLG_paper_32.pdf
[23] Arxiv, Multi-agent actor-critic for mixed cooperative-competitive environments, visitado Junio 5, 2022, dirección: https://arxiv.org/pdf/1706.02275.pdf
[24] Arxiv, Counterfactual multi-agent policy gradients, visitado Junio 5, 2022, dirección: https://arxiv.org/pdf/1705.08926.pdf
[25] Analyticsindiamag, Using Attention Layer, visitado Junio 8, 2022, dirección: https://analyticsindiamag.com/a-beginners-guide-to-using-attention-layer-in-neural-networks/
[26] 3DWarehouse, Padel Court, visitado Junio 20, 2022, dirección: https://3dwarehouse.sketchup.com/model/u9ddf1c34-a9c5-41fa-9a0e-eff49b81fb80/Padel-court?hl=es
[27] Tensorflow, tensorboard, visitado Junio 20, 2022, dirección: https://www.tensorflow.org/tensorboard?hl=es-419
[28] Visual studio code, visitado Junio 20, 2022, dirección: https://code.visualstudio.com
[29] Google Colab, visitado Junio 20, 2022, dirección: https://colab.research.google.com/?hl=es
[30] Seaborn, visitado Junio 20, 2022, dirección: https://seaborn.pydata.org

Anexo

Diagrama de Gantt inicial



	Nombre	Inicio	Fin	Predecesor
1	⊟ **Gestión del proyecto - GP**	**02/25/2022**	**06/27/2022**	
2	Alcance - GP1	02/25/2022	03/01/2022	10
3	Planificación - GP2	03/04/2022	03/07/2022	2
4	Presupuesto - GP3	03/11/2022	03/14/2022	2
5	Informe sostenibilidad - GP4	03/11/2022	03/14/2022	2
6	Reuniones - GP5	02/25/2022	06/20/2022	
7	Documentación - GP6	02/25/2022	06/24/2022	
8	Presentación - GP7	06/24/2022	06/27/2022	7
9	⊟ **Trabajo previo - TP**	**01/24/2022**	**02/21/2022**	
10	Estudio herramientas - TP1	01/24/2022	01/25/2022	
11	Instalación herramientas - TP2	01/26/2022	01/26/2022	10
12	Estudio documentación - TP3	01/26/2022	02/21/2022	10
13	⊟ **Desarrollo entorno - DEV**	**03/04/2022**	**04/11/2022**	
14	⊟ **Implementación básica entorno - DEV1**	**03/04/2022**	**03/25/2022**	11,12
15	Diseño DEV1 - DEV1.1	03/04/2022	03/09/2022	
16	Desarrollo DEV1 - DEV1.2	03/10/2022	03/18/2022	15
17	Testeo DEV1 - DEV1.3	03/19/2022	03/25/2022	16
18	⊟ **Implementación extra entorno - DEV2**	**03/26/2022**	**04/11/2022**	14
19	Diseño DEV2 - DEV2.1	03/26/2022	03/30/2022	14
20	Desarrollo DEV2 - DEV2.2	03/31/2022	04/06/2022	19
21	Testeo DEV2 - DEV2.3	04/07/2022	04/11/2022	20
22	⊟ **Implementación menú - DEV3**	**03/14/2022**	**03/19/2022**	9
23	Diseño DEV3 - DEV3.1	03/14/2022	03/15/2022	
24	Desarrollo DEV3 - DEV3.2	03/16/2022	03/18/2022	23
25	Testeo DEV3 - DEV3.3	03/19/2022	03/19/2022	24
26	⊟ **Desarrollo agente - DA**	**03/26/2022**	**05/16/2022**	
27	⊟ **Implementación agente básico - DA1**	**03/26/2022**	**04/06/2022**	14
28	Diseño DA1 - DA1.1	03/26/2022	03/28/2022	
29	Desarrollo DA1 - DA1.2	03/29/2022	04/03/2022	28
30	Testeo DA1 - DA1.3	04/04/2022	04/06/2022	29
31	Entreno agente básico - DA2	04/07/2022	04/18/2022	27
32	⊟ **Implementación agente extra - DA3**	**04/12/2022**	**04/21/2022**	18
33	Diseño DA3 - DA3.1	04/12/2022	04/13/2022	
34	Desarrollo DA3 - DA3.2	04/14/2022	04/17/2022	33
35	Testeo DA3 - DA3.3	04/18/2022	04/21/2022	34
36	Entreno agente extra - DA4	04/25/2022	05/16/2022	

Figura 52: Diagrama inicial de Gantt creado mediante Gantter. [9]

Diagrama de Gantt final

	Nombre	Inicio	Fin	Predecesoras
1	⊟ **Gestión del proyecto - GP**	**02/25/2022**	**06/27/2022**	
2	Alcance - GP1	02/25/2022	03/01/2022	10
3	Planificación - GP2	03/04/2022	03/07/2022	2
4	Presupuesto - GP3	03/11/2022	03/14/2022	2
5	Informe sostenibilidad - GP4	03/11/2022	03/14/2022	2
6	Reuniones - GP5	02/25/2022	06/20/2022	
7	Documentación - GP6	02/25/2022	06/24/2022	
8	Presentación - GP7	06/24/2022	06/27/2022	7
9	⊟ **Trabajo previo - TP**	**01/24/2022**	**02/21/2022**	
10	Estudio herramientas - TP1	01/24/2022	01/25/2022	
11	Instalación herramientas - TP2	01/26/2022	01/26/2022	10
12	Estudio documentación - TP3	01/26/2022	02/21/2022	10
13	⊟ **Desarrollo entorno - DEV**	**03/04/2022**	**04/11/2022**	
14	⊟ **Implementación básica entorno - DEV1**	**03/04/2022**	**03/25/2022**	11,12
15	Diseño DEV1 - DEV1.1	03/04/2022	03/09/2022	
16	Desarrollo DEV1 - DEV1.2	03/10/2022	03/18/2022	15
17	Testeo DEV1 - DEV1.3	03/19/2022	03/25/2022	16
18	⊟ **Implementación extra entorno - DEV2**	**03/26/2022**	**04/11/2022**	14
19	Diseño DEV2 - DEV2.1	03/26/2022	03/30/2022	14
20	Desarrollo DEV2 - DEV2.2	03/31/2022	04/06/2022	19
21	Testeo DEV2 - DEV2.3	04/07/2022	04/11/2022	20
22	⊟ **Desarrollo agente - DA**	**03/26/2022**	**05/16/2022**	
23	⊟ **Implementación agente básico - DA1**	**03/26/2022**	**04/06/2022**	14
24	Diseño DA1 - DA1.1	03/26/2022	03/28/2022	
25	Desarrollo DA1 - DA1.2	03/29/2022	04/03/2022	24
26	Testeo DA1 - DA1.3	04/04/2022	04/06/2022	25
27	Entreno agente básico - DA2	04/07/2022	04/18/2022	23
28	⊟ **Implementación agente extra - DA3**	**04/12/2022**	**04/21/2022**	18
29	Diseño DA3 - DA3.1	04/12/2022	04/13/2022	
30	Desarrollo DA3 - DA3.2	04/14/2022	04/17/2022	29
31	Testeo DA3 - DA3.3	04/18/2022	04/21/2022	30
32	Entreno agente extra - DA4	04/25/2022	05/16/2022	

Figura 53: Diagrama final de Gantt creado mediante Gantter. [9]

Vídeos de los resultados

Se ha demostrado el funcionamiento de los agentes mediante los diferentes conjuntos de gráficas que se han ido generando para cada uno. Pero ya que en todo momento se ha estado trabajando con un motor gráfico, se ha podido visualizar el comportamiento de los agentes mucho mejor. Por eso mismo en este apartado vamos a dejar un listado de videos con los comportamientos de los distintos agentes entrenados, y además, un vídeo mostrando cuáles han sido los pasos que se han seguido para entrenarlos.

- Vídeo demostrativo de los pasos a seguir para entrenar un agente: https://www.youtube.com/watch?v=KclB0v_wB5o
- Vídeo del agente básico: https://www.youtube.com/watch?v=EXFcdaah8XM
- Vídeo del agente con apuntado dinámico: https://www.youtube.com/watch?v=2ZOenVGUQqA
- Vídeo del agente con distintos tipos de golpes: https://www.youtube.com/watch?v=zEjAXcNwP1E
- Vídeo del agente de dos contra dos: https://www.youtube.com/watch?v=IX5KoAwl1ss