Universidad Internacional de la Rioja (UNIR)

ESIT

Máster Universitario en Inteligencia Artificial

Integración de aprendizaje por imitación y refuerzo con realidad virtual

Trabajo Fin de Máster

presentado por: Fernando González Macías

Dirigido por: Álvaro Tejeda Lorente

Ciudad:Málaga

Fecha: 15 de Julio de 2021

Índice de Contenidos

Índice de Ilustraciones

Índice de Tablas

Resumen

En este proyecto se muestra una vía para entrenar agentes inteligentes capaces de realizar tareas de manipulación en entornos complejos. Para ello se ha creado un entorno utilizando el motor gráfico Unity para, posteriormente, exportarlo y poder utilizarlo en un *script* (código) de Python donde aplicar algoritmos de aprendizaje por refuerzo. También se ha utilizado un entorno similar para obtener datos de las trayectorias realizadas por una persona operando al agente con un equipo de realidad virtual. Estos datos se han usado en los entrenamientos de aprendizaje por imitación. Los entrenamientos del agente se han realizado en Python, tanto para el aprendizaje por imitación como para el aprendizaje por refuerzo, utilizando redes neuronales convolucionales que otorgan visión artificial al agente. Todos los modelos se han creado y entrenado utilizando la librería Pytorch.

Palabras Clave: Aprendizaje por imitación, Aprendizaje por refuerzo, Realidad Virtual, Inteligencia Artificial

Abstract

This project shows a way to train intelligent agents capable of performing manipulation tasks in complex environments. To do this, an environment has been created using the Unity graphics engine to later export it and be able to use it in a Python *script* (code) to apply reinforcement learning algorithms. A similar environment has also been used to obtain data from the trajectories made by a person operating the agent with a virtual reality headset. These data have been used with imitation learning algorithms. The agent trainings have been carried out in Python, both for imitation learning and reinforcement learning, using convolutional neural networks that provide artificial vision to the agent. All models have been created and trained using Pytorch library.

Palabras Clave: Imitation learning, Reinforcement Learning, Virtual Reality, Artificial Intelligence

Capítulo 1

Introducción

1.1. Motivación

En los últimos años se ha visto un aumento en la demanda de agentes inteligentes capaces de realizar tareas complejas (Hussein y col., 2018). Algunas de estas tareas involucran la manipulación de objetos complejos y una gran coordinación mano ojo. Aun tratándose de algo realmente sencillo de realizar para los humanos, donde para realizar algunas tareas no se requiere de un mapeo completo del entorno sino que se basa únicamente en la retroalimentación de sentidos tales como la vista o el tacto (Levine y col., 2018), en robótica, estas tareas de manipulación suelen basarse en procesos de planificación y análisis del entorno que permitan realizar las trayectorias de forma estable y controladas (Levine y col., 2018). Con este paradigma, ciertas tareas de manipulación y control que los humanos basan en la visión se vuelven realmente complejas de realizar para los sistemas de control actuales.

Con el auge de la inteligencia artificial aparecen los algoritmos de aprendizaje por refuerzo. Estos algoritmos, derivados de la estadística, las teorías de control y la psicología (Q. Wang y Zhan, 2011), permiten crear agente capaces de aprender tareas complejas basándose en la experiencia propia. Permiten al agente aprender que acciones debe de realizar basándose en los sensores que dispone, recibiendo por parte del entorno una puntuación que mide la calidad de su actuación en función de la tarea que se le quiera enseñar (Vijaykumar Gullapalli, 1992), conocida como recompensa. Usando este valor, el agente modifica su política de comportamiento, adaptándose al entorno y logrando maximizar la recompensa (Q. Wang y Zhan, 2011).

Sin embargo, aun cuando estos algoritmos son realmente útiles para obtener una política óptima, su uso puede llevar a generar comportamientos muy diferentes a los que un humano realizaría (Julien Dossa y col., 2019), además de la dificultad de establecer un criterio para la recompensa que permita al agente alcanzar el objetivo deseado (Zhu y col., s.f.). Para salvar estas dificultados encontramos los algoritmos de aprendizaje por imitación. Estos algoritmos utilizan demostraciones generadas por humanos para enseñar a agentes a imitar su comportamiento (Hussein y col., 2018), lo que permite que el agente genere una política similar a la de los humanos y evita la necesidad de crear funciones de recompensa complejas.

La mezcla de estos dos paradigmas de aprendizaje es lo que permite a los humano aprender nuevas tareas, comenzando en un primer momento por observar a otras personas y, posteriormente, mejorar las habilidades aprendidas por imitación para alcanzar mejores resultados en la tarea a realizar (Hamahata y col., 2008).

1.2. Planteamiento del trabajo

En este trabajo se va a plantear un sistema que permita entrenar un agente inteligente capaz de realizar tareas motoras complejas que requieran de una gran coordinación mano ojo. Para ello se utilizarán algoritmos de aprendizaje por imitación y aprendizaje por refuerzo para simular la forma en la que los humanos aprenden.

Para poder aplicar estos algoritmos primero es necesario disponer de un entorno y un objetivo para el agente. El objetivo que aquí se plantea es crear un agente capaz de utilizar la visión para coordinar el movimiento motor de dos efectores finales que simularan las manos. Para ello se utilizará un entorno donde el objetivo del agente será acertar a una serie de dianas, colocadas en posiciones aleatorias frente a él, utilizando un arco y flechas.

El tiro con arco es un deporte olímpico bien conocido, donde el objetivo es el de disparar una flecha a un diana, intentando acertar en el centro de la misma (“Tiro con arco”, s.f.). Se trata de un deporte donde la concentración y la coordinación mano ojo son realmente fundamentales, habiéndose realizado varios estudios que avalan como la práctica de este deporte mejora sustancialmente esta coordinación (Azrul Anuar Zolkafi y col., 2018; Yadav y col., 2012).

Dada esta premisa, un agente capaz de practicar este deporte es un agente que debe

de tener una buena coordinación con ambas manos. Dado que se tratara de un agente que utilizara la visión como principal sensor, debe de ser capaz de reconocer el objeto diana, medir la distancia aproximada a la que se encuentra y apuntar correctamente con el arco. Todas estas tareas son realmente compleja, incluso para los humanos que deben de pasar varios años de preparación cuando compiten a nivel profesional (“Tiro con arco”, s.f.). Es, por tanto, una buena tarea para entrenar a un agente y comprobar su capacidad de coordinación mano ojo.

Para poder lograr este objetivo se utilizará la realidad virtual. La realidad virtual permite manipular objetos 3D dentro de un entorno virtual gracias a unos mandos y un casco que rastrea la posición de las manos y la cabeza en todo momento (Nanjappan y col., 2018). Este casco tiene en su interior dos pantallas individuales que renderizan en tiempo real el entorno 3D, mostrando una imagen estereoscópica que genera la ilusión de profundidad, aumentando la inmersión y permitiendo al usuario actuar de una forma mucho más cercana a como actuaría en un entorno físico (Nanjappan y col., 2018).

Gracias a este sistema de rastreo se podrá crear un entorno 3D virtual donde una persona pueda interactuar con un arco de una forma similar a como lo realizaría en la vida real, sin el consiguiente riesgo y coste que podría conllevar. Otra ventaja de utilizar la realidad virtual es que permite controlar el entorno al completo, pudiendo obtener mediciones que de otro modo seria extremadamente complicadas, sino imposibles de obtener.

Por último, el agente se entrenará dentro de un entorno virtual similar al que se utilizará para la toma de trayectorias. Esto tiene una serie de ventajas ya que el agente no tendrá que actuar con un arco y flechas reales, algo que podría llegar a ser peligroso y costoso.

1.2.1. Aportaciones

Con este proyecto se pretende realizar una serie de aportaciones al entrenamiento de agentes inteligentes mediante algoritmos de aprendizaje por imitación y refuerzo. Se pretende mostrar unas directrices que permitan entrenar agentes incorporando la tecnología de realidad virtual para la toma de trayectorias y lograr comportamientos similares a los de los humanos. Los puntos más importantes de estas aportaciones son:

- Creación de entornos virtuales para la toma de datos y entrenamiento del agente usando realidad virtual.

- Uso de algoritmos de aprendizaje por imitación para la inicialización de la red del agente.
- Uso de algoritmos de aprendizaje por refuerzo para mejorar el rendimiento del agente tras un primer entrenamiento con algoritmos de aprendizaje por imitación.

La tarea del agente en el entorno virtual será acertar sobre unas dianas utilizando arcos y flechas, tal como se ha expuesto en puntos anteriores. Sin embargo, los algoritmos y códigos usados son extrapolables a cualquier problema donde se la manipulación de objetos por parte del agente y se utilice visión artificial para dicha tarea.

Todos los códigos utilizados, así como los entornos creados se subirán a la plataforma *GitHub* bajo una licencia gratuita MIT para que sean accesibles a cualquiera que desee utilizarlos o mejorarlos.

1.3. Estructura de la memoria

La memoria se dividirá en distintos capítulos que a su vez estarán divididos en distintas secciones. A continuación, se exponen los distintos capítulos y secciones de manera resumida:

- **Capítulo 2 Contexto y estado del arte:** En este capítulo se describirá el estado del arte para cada una de las tecnologías que se usaran, así como algunos trabajos similares a este proyecto.
  - **Aprendizaje por refuerzo:** En esta sección se expondrán los últimos avances relacionados con los algoritmos de aprendizaje por refuerzo, mencionando algunos de los algoritmos que mejores resultados están ofreciendo y seleccionando uno de ellos para este proyecto.
  - **Aprendizaje por imitación:** Esta sección se centrara en los algoritmos de aprendizaje por imitación, exponiendo cuales son los que mejores resultados ofrecen y cuales se utilizarán para este proyecto.
  - **Integración de aprendizaje por refuerzo e imitación:** En esta sección se resumirá algunos de los trabajos donde se han integrado los algoritmos vistos en las secciones anteriores y que ventajas ofrecerá esta aproximación durante la realización del proyecto.

- **Realidad virtual y aprendizaje por imitación:** Esta sección pretende exponer algunos de los trabajos ya realizados donde se ha integrado la realidad virtual y los algoritmos de aprendizaje por imitación con el objetivo de enseñar a un agente a manipular objetos.
- **Tiro con arco:** Esta última sección del capítulo se centrara en algunos trabajos donde se enseñó a un robot a practicar el tiro con arco. También resumirán las ventajas que este deporte aporta a los humanos y porque enseñar a un agente a practicar el tiro con arco puede suponer una ventaja a la hora de mejorar la coordinación motora.

- **Capítulo 3 Objetivos y metodología de trabajo:** Este capítulo pretende explicar el objetivo general así como el sistema de evaluación que se seguirá para comprobar si se ha logrado cumplir dicho objetivo. También se expondrá la metodología de trabajo que se seguirá durante todo el proyecto. Este capítulo estará dividido en distintas secciones:
  - **Objetivo general:** Se expondrá el objetivo general y que sistema de evaluación se seguirá para comprobar si se ha alcanzado dicho objetivo.
  - **Objetivos intermedios:** Se enumerarán los distintos objetivos específicos que se deberán de alcanzar para lograr cumplir el objetivo general.
  - **Metodología de trabajo:** En esta sección se expondrá la metodología de trabajo que se seguirá para ir alcanzando los distintos objetivos intermedios.
- **Capítulo 4 Descripción del proyecto:** En este capítulo se expondrá como se ha realizado el proyecto, como se han utilizado las tecnologías mencionadas en capítulos anteriores y que proceso se ha seguido para obtener los resultados deseados. Este capítulo estará dividido en distintas secciones:
  - **Estructura de trabajo:** En esta sección de mostrará la estructura de trabajo que se seguirá para este proyecto, resumiendo las distintas partes para, en las siguientes secciones, explicar con mayor profundidad cada una de ellas.
  - **Entorno virtual:** En esta sección se mostrará el proceso seguido para crear el entorno virtual, tanto el que es accesible a través del dispositivo de realidad virtual, como el entorno donde el agente interactúa con los objetos. También se

expondrá el sistema utilizado para tomar los datos y de qué manera se efectuó dicha toma.

- **Entrenamiento del agente:** Esta sección se centrará en la planificación del entrenamiento del agente. Se expondrá que algoritmos se utilizan, como se han implementado y donde se ha realizado el entrenamiento. También se justificará como se ha evaluado el modelo y que análisis se ha efectuado para realizar dicha evaluación.

- **Capítulo 5 Análisis de resultados:** Este capítulo se centrara en los resultados obtenidos. Se realizará un análisis sobre la evolución del agente a lo largo de las diferentes épocas de entrenamiento y se efectuará un análisis final de los resultados.

  - ***Behavioral Cloning**:* En esta sección de mostrarán los resultados obtenidos para cada arquitectura durante el entrenamiento con *Behavioral Cloning*. También se agregará una tabla resumen con los resultados más importantes y se decidirá qué modelo se utilizará en la siguiente parte del entrenamiento.
  - ***Proximal Policy Optimization*:** En esta sección se mostrarán los resultados obtenidos durante el entrenamiento con *Proximal Policy Optimization* para los modelos seleccionados en la sección anterior. Adicionalmente, se entrenarán las arquitecturas de los modelos sin inicializar, con el objetivo de comparar los resultados.

- **Capítulo 6 Conclusiones y trabajos futuros:** En este último capítulo se resumirán los resultados obtenidos y se comentarán la relevancia de dichos resultados. También se expondrán posibles líneas de trabajo futuras como continuación de este proyecto. Este capítulo se dividirá en dos secciones:

  - **Conclusiones:** Se expondrán los resultados y las conclusiones obtenidas a partir de ellos.
  - **Trabajos futuros:** Se comentarán posibles vías para continuar el este proyecto, así como mejoras para el mismo.

Capítulo 2

Contexto y Estado del Arte

En este capítulo se hablará del estado del arte para cada una de las partes del proyecto. En este proyecto se hará uso de arquitecturas de aprendizaje profundo (*Deep Learning*). Estas arquitecturas forman parte de un conjunto de técnicas que se engloban dentro del Aprendizaje Maquina (*Machine Learning*), que a su vez se engloba dentro de las técnicas de Inteligencia Artificial (IA). En la figura 2.1 se muestra una representación sencilla de la situación del *Deep Learning*.

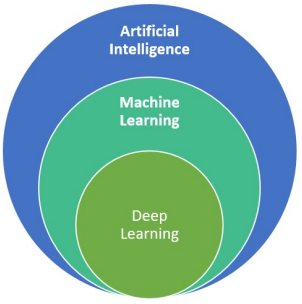


Figura 2.1: Representación del estado del Deep Learning. (“Artificial Intelligence, Machine Learning, and Deep Learning: Same context, Different concepts”, s.f.)

La Inteligencia Artificial se ha definido como la capacidad de las máquinas para ïmitarçiertas funciones cognitivas humanas. Dentro de estas funciones se encuentran el entendimiento del habla humana, la conducción autónoma, interpretación de datos complejos y la capacidad de competir en juegos de estrategia de alto nivel, entre otras (Ongsulee, 2018). El termino IA fue acuñado por primera vez en la conferencia de *Dartmounth* en 1956 y, desde entonces, ha ido evolucionando hasta convertirse en una de las ramas de las ciencias de la computación más avanzadas que existen (McCorduck y Cfe, 2004).

El Aprendizaje Maquina (*Machine Learning*) corresponde a un área de la Inteligencia Artificial y las ciencias de la computación que busca enseñar a las maquinas sin la necesidad de programar explícitamente, es decir, permitir a las maquinas aprender por si mismas (Ongsulee, 2018). Los modelos de *Machine Learning* suelen estar muy relacionadas con la estadística computacional y la optimización matemática. La estadística computacional busca obtener predicciones a partir de datos pasados, mientras que las técnicas de optimización matemáticas son las que se utilizan para relacionar los modelos con los datos estadísticos (Alzubi y col., 2018).

Dentro del *Machine Learning* encontramos múltiples áreas o tipos de problemas a resolver. Algunos de ellos son los problemas de clasificación, regresión, *clustering*, detección de anomalías y aprendizaje por refuerzo (Alzubi y col., 2018). Este tipo ultimo tipo de algoritmos, los de aprendizaje por refuerzo, son en los que se centrará este proyecto.

2.1. Aprendizaje por refuerzo

El aprendizaje por refuerzo es un área de la inteligencia artificial que estudia el comportamiento de un agente dentro de un entorno, buscando aprender un comportamiento óptimo para una tarea dada (Deepanshu Mehta, 2020). Para ello el agente recibirá una serie de observaciones como entrada, que le permitirán medir el estado actual de forma parcial. Con esas observaciones, el agente deberá de tomar una decisión (acción) que afectara al estado del entorno donde se encuentra (Kaelbling y col., 1996). La política del agente será la que rija estas decisiones, buscando maximizar una recompensa proporcionada por el entorno en función de su desempeño (Vijaykumar Gullapalli, 1992). En la figura 2.2 vemos un esquema de un modelo estándar de aprendizaje por refuerzo.

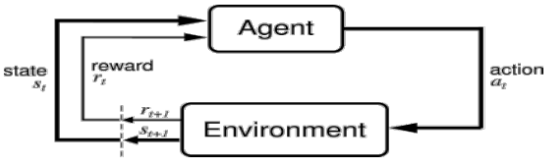


Figura 2.2: Esquema de un modelo estándar de aprendizaje por refuerzo (Deepanshu Mehta, 2020)

Los algoritmos de aprendizaje por refuerzo se basan en los procesos de Markov, donde el siguiente estado dependen únicamente del estado anterior y de la acción tomada

(Q. Wang y Zhan, 2011). Partiendo de esta premisa, el agente ira realizando acciones que le proporcionarán una recompensa inmediata en función de su desempeño, sin embargo, el agente debe de tomar en cuenta las recompensas a largo plazo para obtener un estrategia óptima (Q. Wang y Zhan, 2011).

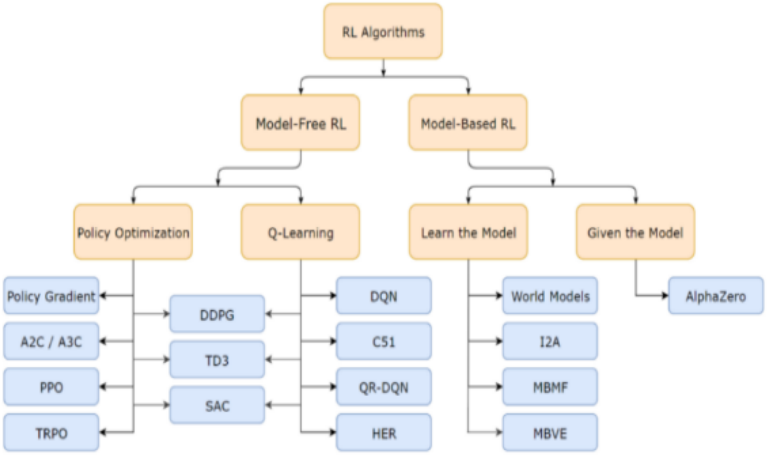


Figura 2.3: Esquema de división de los algoritmos de aprendizaje por refuerzo (Deepanshu Mehta, 2020)

Una de las primeras divisiones dentro de este tipo de algoritmos es si el agente conoce o no un modelo del entorno (*Model-Based* o *Model-Free*) que le permita predecir las transiciones de estados y las recompensas ("Part 2: Kinds of RL Algorithms — Spinning Up documentation", s.f.). Para este proyecto nos centraremos en los algoritmos donde el agente no dispone de un modelo del entorno, pues es el caso con el que nos encontramos.

Como vemos en la figura 2.3, dentro de los algoritmos *Model-Free* encontramos dos grupos:

- ***Policy-based:*** Son algoritmos buscan optimizar la política del agente maximizando la recompensa obtenida (Deepanshu Mehta, 2020).
- ***Value-based:*** Este tipo de algoritmos aproximan la función acción-valor, de forma que basan su política en las acciones que maximizan esta función (Deepanshu Mehta, 2020).

Además de estos dos grupos, existen algunos algoritmos que son un punto intermedio entre ambos grupos. En ellos se aprovechan las ventajas de los algoritmos basados en la política, como la estabilidad, y las ventajas de los algoritmos basados en la función

valor, como la eficiencia y velocidad de entrenamiento (“Part 2: Kinds of RL Algorithms — Spinning Up documentation”, s.f.).

Para este proyecto nos quedaremos con *Proximal Policy Optimization* (PPO), uno de los algoritmos más utilizados actualmente (Y. Wang y col., s.f.) gracias a los buenos resultados que ofrece junto con la facilidad de implementación (Schulman y col., s.f.).

PPO forma parte del grupo de algoritmos basados en la política (*Policy-based*). En estos algoritmos se busca la política que logre maximizar la recompensa esperada, utilizando los algoritmos de optimización basados en el gradiente (B. Liu y col., s.f.). Para ello se utiliza un estimador que permite calcular la recompensa esperada a partir de una cierta política (Schulman y col., s.f.). Este estimador puede tratarse de una función concreta cuando se trata de una tarea fácilmente parametrizable. Sin embargo, en el caso de que se trate de una tarea compleja, la estimación suele basarse en una red neuronal que se encargue de predecir la recompensa esperada (B. Liu y col., s.f.). Utilizando esta aproximación dispondremos de dos redes neuronales, una encargada de predecir la recompensa esperada y otra encargada de controlar el comportamiento del agente (B. Liu y col., s.f.).

2.2. Aprendizaje por imitación

Uno de los mayores problemas que enfrentan los algoritmos de aprendizaje por refuerzo es la función recompensa. Estos algoritmos deben de disponer de una función que guie al agente para que logre cumplir su objetivo. Sin embargo hay casos donde encontrar esta función de recompensa es realmente complejo, sino imposible (Kamyar y col., 2019). Es en este tipo de situaciones donde aparece un nuevo paradigma, el aprendizaje por imitación.

El aprendizaje por imitación pretende enseñar a un agente como cumplir una tarea específica utilizando ejemplos provistos por un experto humano (Hussein y col., 2018). Al usar estos ejemplos se evita la necesidad de describir el conocimiento necesario para realizar cierta tarea, eliminando la necesidad de disponer de restricciones que limiten el comportamiento del agente o de una función de recompensa que lo guíe hacia la tarea que se desea cumplir (Hussein y col., 2018).

Esta idea de aprender mediante imitación implica que la política del agente será similar a la política del experto que provee los ejemplos (Attia y Dayan, s.f.). Esto pone sobre

la mesa varios problemas, como la necesidad de crear una serie de ejemplos representativos de la tarea que se quiere crear y que permitan al agente aprender un política lo suficientemente genérica como para poder actuar correctamente frente a situaciones que no se encuentren dentro de los ejemplos provistos (Hussein y col., 2018).

Para este proyecto nos centraremos en el algoritmo *Behavioral Cloning* (BC) (Torabi y col., 2018), gracias a su fácil implementación y velocidad de entrenamiento.

El algoritmo BC busca enseñar a un agente como debe de interactuar con un entorno utilizando únicamente los ejemplos provistos por un experto (Torabi y col., 2018). Esto implica que el agente no necesita interactuar con el entorno para aprender una política, sino que extrae la política usada por el experto humano a partir de los datos provistos, y la imita para alcanzar el mismo objetivo.

BC tiene la ventaja de no necesitar interacción directa con el entorno por parte del agente, lo que reduce el tiempo de entrenamiento y evita la necesidad de disponer de un entorno seguro donde el agente pueda interactuar sin riesgo (Bühler y col., s.f.).

A pesar de las ventajas que tiene este algoritmo, solo puede ser aplicable cuando se tiene acceso a toda la secuencia de acciones generadas por el humano experto (Torabi y col., 2018), lo que no siempre se cumple en todas las situaciones. El caso concreto de este proyecto es una simulación virtual, lo que permite obtener esta información de manera sencilla, pudiendo usar este algoritmo sin problemas.

2.3. Integración de aprendizaje por refuerzo e imitación

Como se ha expuesto en las secciones anteriores, los algoritmos de aprendizaje por refuerzo han logrado generar estrategias de comportamiento realmente eficientes, sin embargo la eficiencia no es el único factor a tener en cuenta en usos prácticos (Julien Dossa y col., 2019). Además, este tipo de algoritmos son realmente complejos de calibrar, ya que se necesita información del entorno que puede ser complicada o imposible de obtener, además de una buena función de recompensa que permita guiar al agente hacia la meta final que se desea, lo que puede llegar a ser muy complejo (Zhu y col., s.f.).

Por otro lado, los algoritmos de aprendizaje por imitación han demostrado ser realmente eficaces a la hora de enseñar a un agente a aprender políticas similares a las usadas por los humanos (Julien Dossa y col., 2019). Sin embargo, al tratarse de algoritmos que se basan únicamente en imitar, estas políticas no alcanzar a ser óptimas y

dependen enormemente de los datos y, por tanto, de como de bien se ha realizado la tarea concreta durante la toma de datos (Julien Dossa y col., 2019).

Para vencer esta problemática se puede utilizar un sistema de entrenamiento que utilice algoritmos de ambas modalidades, es decir, integrar el aprendizaje por refuerzo y el aprendizaje por imitación. Esta integración es muy similar al sistema de aprendizaje utilizado por los humanos (Hamahata y col., 2008), donde primero se aprende a partir de un instructor que demuestra cómo realizar la tarea y, posteriormente, se mejora el rendimiento optimizando el comportamiento.

2.4. Realidad virtual y aprendizaje por imitación

La realidad virtual es una tecnología que, en los últimos años, ha ganado bastante peso, tanto desde la perspectiva de la productividad como del entretenimiento (Nanjappan y col., 2018). Esta tecnología permite que un usuario manipulo objetos 3D en entornos virtuales utilizando visión estereoscópica, gracias a un casco con dos pantallas y dos lentes situadas cada una en un ojo (Nanjappan y col., 2018). Unos controles con sistemas de seguimiento permiten conocer la posición de las manos del usuario, permitiéndole interactuar con los objetos 3D de forma similar a como lo haría de forma física (Nanjappan y col., 2018).

Esta tecnología no solo se ha utilizado para interactuar con entornos virtuales, sino que también se ha logrado utilizar para controlar robots de forma telemática, permitiendo a usuario ver el mundo a través de las cámaras del robot y controlar sus brazos (Zhang y col., 2018). Esto permite tomar control sobre el robot de forma remota, permitiendo a usuario interactuar con el entorno como lo haría si estuviera en esa misma situación, pudiendo, por tanto, crear datos de comportamiento para posteriormente utilizar algoritmos de aprendizaje por imitación (Dyrstad y col., s.f.).

El usar entornos virtuales junto con estos algoritmos no limita necesariamente su aplicabilidad en entornos físicos, el crear entorno virtual lo suficientemente cercano a la realidad física permite entrenar al agente de forma mucho más efectiva y lograr, posteriormente, buenos resultados en un entorno físico similar al virtual (Dyrstad y col., s.f.). Una buena aproximación para lograr estos resultados es utilizar cierta aleatoriedad y ruido dentro del entorno virtual (Dyrstad y col., s.f.), lo que permite que el agente aprende un comportamiento robusto frente al ruido y puedo comportarse mejor en el entorno físico

posteriormente, que presentará cierta diferencia respecto del entorno virtual.

2.5. *Deep Learning*

Deep Learning corresponde con una sub-área del *Machine Learning* cuyos modelos hacen uso de redes neuronales artificiales, inspiradas en el funcionamiento del cerebro. Este área ha obtenido un gran interés por parte de los investigadores en los últimos años debido a la mejora del rendimiento comparado con otros modelos de *Machine Learning*. Esta mejora se debe principalmente al aumento de la cantidad de datos y a la mejora computacional de los equipos modernos (DIxit y col., 2018).

Las redes neuronales artificiales profundas (Deep Neural Networks) corresponde con las redes con múltiples capas ocultas, que permiten crear relaciones no lineales entre los datos de entrada y salida. Estas redes pueden hacer uso de distintos tipos de capas, en función del objetivo (DIxit y col., 2018). Para este proyecto se va a hacer uso de redes con capas convolucionales, especializadas en visión artificial, y de redes recurrente tipo *Long-Short Term Memory* (LSTM), utilizadas para modelar secuencias de datos.

Redes Convolucionales

La visión por computador estudia el uso de los ordenadores para simular tareas de visión humanas. En los últimos años, este área impulsado el uso de modelos de *Deep Learning* para tareas de clasificación, detección y análisis de imágenes, gracias a las grandes cantidades de datos obtenidas en redes sociales y otros medios (Al-Saffar y col., 2017).

Para las tareas de visión por computador es común utilizar las llamadas redes convolucionales. Este tipo de redes aplican una operación de convolución para cada uno de los canales de entrada, utilizando un filtro. Este filtro es aprendido por la red utilizando el descenso del gradiente o algún otro algoritmos de optimización (Al-Saffar y col., 2017). En la figura 2.4 se muestra un ejemplo de convolución.

Con estas capas se extraen una serie de características de las imágenes que, posteriormente, se utilizarán para realizar la tarea de clasificación, análisis o detección correspondiente (Sainath y col., 2015).

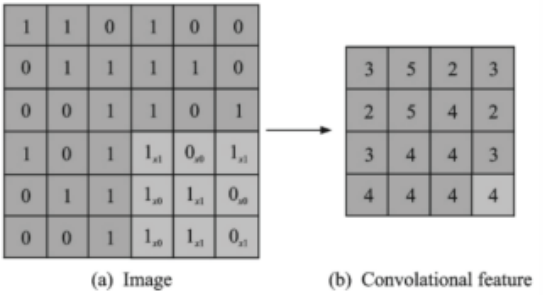


Figura 2.4: Operación de convolución (Al-Saffar y col., 2017).

Redes Recurrentes

Las redes neuronales recurrentes (RNN) son arquitecturas cuyo objetivo es encontrar patrones en datos secuenciales. Estos datos pueden ser texto o series numéricas, entre otros (Schmidt, 2019).

Para lograr modelar datos secuenciales, estas redes trasmiten información del *step* anterior a través de un estado interno llamado *Hidden State*. Este estado interno se calcular utilizando unos pesos específicos que se aplican al estado interno anterior más unos pesos aplicados a la entrada actual (Schmidt, 2019). Sin embargo, el uso de esta técnica acarrea una serie de problemas, tanto en el aumento del tiempo de entrenamiento, como problemas de *Vanishing Gradient* y *Exploding Gradient*, especialmente notables en largas secuencias (Hochreiter y Schmidhuber, 1997).

Para minimizar estos problemas, se crearon las capas *Long-Short Term Memory* (LSTM). Este tipo de capas hacen uso de una célula de memoria, con múltiples puertas, que permiten modelar un estado interno más preciso capaz de ”mantener”la información relevante durante más tiempo, olvidando aquella información que deja de ser relevante (Hochreiter y Schmidhuber, 1997).

2.6. Tiro con arco

El tiro con arco es un deporte que ha demostrado mejorar en gran medida la coordinación mano ojo (Yadav y col., 2012), incluso en periodos cortos de tiempo (Azrul Anuar Zolkafi y col., 2018). Esto implica que enseñar a un agente artificial a practicar este deporte serviría para demostrar sus capacidades de coordinación mano ojo, siempre que este agente utilice la visión para ello.

El enseñar a un agente artificial a utilizar un arco no es nuevo, se han encontrado autómatas capaces de disparar flechas con un arco creados a finales del siglo 19 (Kormushev y col., 2010). Estos autómatas efectúan los movimientos de coger la flecha, colocarla sobre el arco, tensor y disparar, pero no son capaces a apuntar a ningún objetivo concreto.

Enseñar a un robot a realizar tiro con arco es una tarea realmente compleja, que requiere de un gran cálculo matemático de trayectorias y, en el caso de utilizar visión, procesamiento de imágenes para detectar el objetivo. EL uso de estas herramientas ha logrado que un robot, el robot humanoide iCub, logre alcanzar el objetivo utilizando un arco y flechas (Kormushev y col., 2010). Sin embargo, para lograr esto la flecha debía de colocarse previamente en el arco y tensarlo, por lo que la tarea del robot era la de apuntar y liberar la flecha únicamente, debido en gran medida a la complejidad de la trayectoria y al limitado rango de movimientos del robot (Kormushev y col., 2010).

Capítulo 3

Objetivos y metodología de trabajo

3.1. Objetivo general

El objetivo principal es el de comprobar si el uso de algoritmos de aprendizaje por imitación y refuerzo, junto con tecnologías de realidad virtual, pueden ayudar a entrenar agentes capaces de adaptarse a entornos de manipulación complejos utilizando visión artificial. Con este objetivo, se entrenará a un agente virtual capaz de controlar dos efectores finales, que simularan las manos, en tareas de manipulación complejas que requieran una gran coordinación ojo mano. Para ello, el agente dispondrá de dos cámaras que renderizarán el entorno virtual, otorgándole una visión estereoscópica similar a la de los humanos, así como la capacidad de detectar la posición y orientación de las manos respecto de la cabeza y la orientación de la cabeza respecto del entorno. Estos sentidos pretenden simular las condiciones sensoriales en las que un humano se encuentra cuando interactúa con objetos utilizando un dispositivo de realidad virtual.

Utilizando las entradas provistas por los sensores antes mencionados, el agente deberá de manipular correctamente los objetos del entorno virtual para completar la tarea que se le proponga. Para ello controlará las manos y cabeza utilizando comandos de velocidad lineal y angular, así como un comando para coger o soltar el objeto que tenga en la mano.

Para controlar el comportamiento del agente se utilizará una red neuronal que, recibiendo como entrada la información de los sensores antes mencionados, deberá de generar los comandos de control para alcanzar el objetivo propuesto. Este objetivo será el de manipular un arco y flechas para lograr acertar a una serie de dianas que, de mane-

ra aleatoria, irán apareciendo a diferentes distancias frente al agente. El agente deberá de tomar una flecha de un depósito de flechas, colocarla sobre el arco, tensar, apuntar y disparar.

3.2. Objetivos intermedios

Para lograr este objetivo general, el proceso se dividirá en una serie de objetivos intermedios que pretenden generar un proceso concreto de trabajo. Estos objetivos intermedios serán:

1. **Entorno virtual:** Este primer objetivo consistirá en crear un entorno virtual accesible a través de un dispositivo de realidad virtual que permita a un humano interactuar con el entorno y grabar las trayectorias que posteriormente el agente utilizará para entrenar.

2. **Obtención de trayectorias:** En este punto, el objetivo será generar una serie de trayectorias con las que el agente pueda entrenar. Para ello se utilizará el entorno creado en el punto anterior y, mediante un dispositivo de realidad virtual, se pondrá a una serie de personas a practicar el tiro con arco dentro de este entorno, guardando las trayectorias de sus acciones.

3. **Adaptación del entorno:** El objetivo en este punto es adaptar el entorno utilizado en el punto anterior para que un agente virtual pueda interactuar con los objetos virtuales de la misma forma en la que las personas han actuado.

4. **Entrenamiento del agente:** El objetivo en este punto será el de entrenar el agente utilizando los algoritmos mencionados en puntos anteriores.

Con estos objetivos intermedios se pretende lograr que el agente aprenda a manipular un arco para alcanzar los objetivos de manera similar a como una persona lo haría.

3.3. Metodología de trabajo

La metodología que se seguirá en este proyecto consistirá en ir cumpliendo los objetivos intermedios expuestos en la sección 3.2. Cada uno de estos objetivos requiere de

herramientas diferentes para poder alcanzarlos, por lo que a continuación se enumeran las herramientas necesarias para cada objetivo, así como la metodología a utilizar.

1. **Entorno virtual:** Para crear el entorno virtual se utilizará Unity, una plataforma de desarrollo de aplicaciones interactivas de tiempo real que permite crear entornos virtuales accesibles a través de dispositivos de realidad virtual, entre otros. En esta plataforma se creará un entorno 3D capaz de grabar las trayectorias de las manos y la cabeza del sujeto que este interactuando con el entorno, así como las imágenes renderizadas de cada cámara virtual.

2. **Obtención de trayectorias:** Para la obtención de las trayectorias se utilizará el entorno 3D creado en el punto anterior, interactuando con los objetos virtuales utilizando el dispositivo de realidad virtual Oculus Quest 2, conectado mediante Oculus Link al ordenador.

3. **Adaptación del entorno:** Para adaptar el entorno se hará uso de un *toolbox* de Unity llamado ML Agents (Juliani y col., s.f.), que otorga una serie de funciones que permiten, entre otras cosas, generar un entorno similar a los GYM de OpenAI utilizando la plataforma de Unity. Este entorno se exportará como un archivo ejecutable que podrá conectarse con un *script* de Python donde, posteriormente se realizará el entrenamiento del agente.

4. **Entrenamiento del agente:** Para el entrenamiento del agente se utilizarán los algoritmos explicados en capítulos anteriores. Se realizarán varias pruebas, utilizando diferentes arquitecturas, parámetros de entrenamiento y funciones de error. En todo momento se tendrá en cuenta el tiempo de ejecución de la red, pues debe de poder ejecutarse en tiempo real para poder cumplir con la tarea propuesta.

Por último, y debido al coste computacional que los algoritmos que se utilizarán requieren, el entrenamiento se realizará utilizando los servicios de *Cloud Computing* (computación en la nube) de Microsoft Azure. Esto permitirá crear varias máquinas virtuales que puedan realizar distintos entrenamientos utilizando diferentes hiper-parámetros y, de esta forma, acelerar la investigación, al no depender de la potencia de un equipo local.

Capítulo 4

Descripción del proyecto

4.1. Estructura de trabajo

Para este proyecto se creará un entorno totalmente desde 0 utilizando el motor Unity, que permite crear entornos 3D interactivos que aplican físicas y gráficos realistas. Para ello se crearán dos entornos idénticos, uno accesible mediante un dispositivo de realidad virtual, y otro donde actuará el agente inteligente.

Tras crear los entornos virtuales se procederá a la toma de datos utilizando el entorno accesible mediante un dispositivo de realidad virtual. Los datos que se tomarán corresponden con las imágenes de cada ojo, la posición y rotación de las manos y la rotación de la cabeza. La salida corresponde con la velocidad lineal y angular de las manos y la cabeza, así como un comando que corresponde a cerrar o abrir las manos. Los datos se toman en tiempo real mientras una persona interactúa con el entorno, por lo que los datos no requieren de un etiquetado a mano.

Tras la toma de datos se realiza una normalización sobre los datos de entrada. Para los datos de salida se probarán dos estrategias, una con los datos normalizados y otra con los datos sin normalizar. La principal diferencia entre ambas estrategias será la función de activación de la capa de salida de la red, para los datos normalizados entre -1 y 1 se utilizará una función de activación tipo *tanh*, mientras que para los datos sin normalizar se usará un función de activación lineal. En la figura 4.1 se muestra un esquema del proceso de toma de datos.

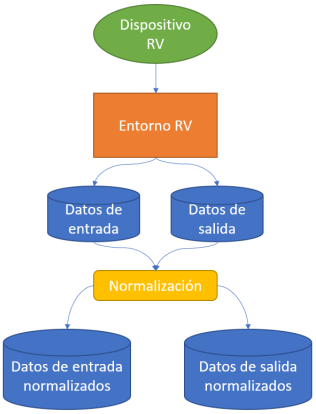
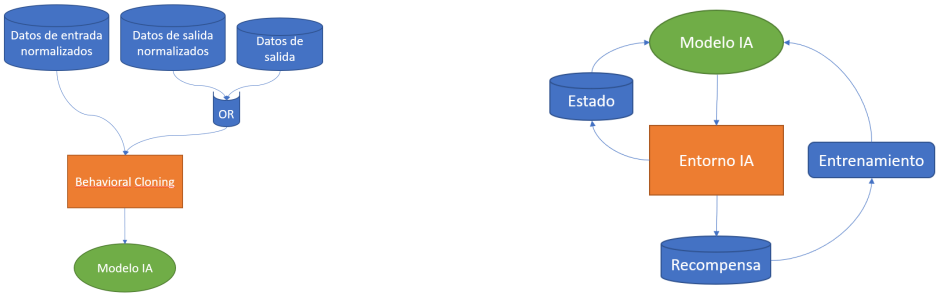


Figura 4.1: Esquema del proceso de toma y almacenamiento de datos.

Con los datos tomados y normalizados se procederá a realizar los entrenamientos de distintas arquitecturas utilizando el algoritmo *Behavioral Cloning*. Tras entrenar varias arquitecturas se seleccionarán aquellas que mejores resultados ofrezcan y se volverán a entrenar utilizando el algoritmo PPO utilizando el entorno virtual del agente. En la figura 4.2 se muestran los esquemas usados para entrenar el agente usando el algoritmo *Behavioral Cloning* (a la izquierda) y el algoritmos PPO (a la derecha).



(a) Esquema de entrenamiento BC.

(b) Esquema de entrenamiento PPO.

Figura 4.2: Esquema de entrenamiento.

4.2. Entorno virtual

En esta sección se abordará como se ha creado el entorno virtual, tanto el accesible a través del dispositivo de realidad virtual Oculus Quest 2, como el entorno donde el agente interactuará con los objetos. También se explicará el mecanismo utilizado para la toma de datos y el procesamiento previo al entrenamiento del agente.

En cuanto a los entornos, se crearán en dos proyectos diferentes, uno donde el agente es controlado mediante realidad virtual y otro donde el agente es controlado por la red neuronal. Esto se debe a que incorporar ambos entornos en uno solo genera incompatibilidades entre los *toolbox*, al estar uno de ellos, ML Agents (Juliani y col., s.f.), en fase experimental.

4.2.1. Entorno de realidad virtual

Como se expuso en capítulos anteriores, para la creación del entorno virtual se utilizará la plataforma de desarrollo de aplicaciones interactivas Unity. Esta aplicación permite crear entornos 3D interactivos donde se pueden aplicar físicas complejas para simular diferentes entornos. También dispone de un *toolbox* (herramientas extras de software) especializado para realidad virtual, gracias al cual se puede crear un entorno interactivo accesible a través de un dispositivo especializado.

Otra de las ventajas de utilizar Unity es la gran cantidad de *Assets* (modelos y texturas de objetos 3D) disponibles en su tienda online, lo que permite obtener modelos 3D con un mayor grado de realismo, permitiendo que el entorno sea más parecido a la realidad, donde suele haber más objetos presentes que pueden introducir ruido y el agente debe de aprender a ignorar.

En la figura 4.3 se puede ver una vista superior del entorno 3D creado. Como se puede comprobar, se trata de un entorno con multitud de elementos diferentes, que añaden una gran cantidad de ruido. Todos estos modelos 3D no interactivos se han obtenido a través de la tienda *online* de Unity, específicamente de (3DeLucas, s.f.).

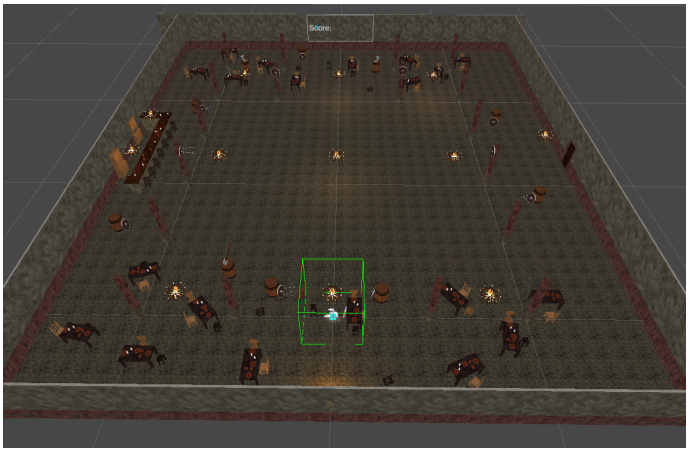


Figura 4.3: Vista superior del entorno 3D de realidad virtual.

La zona central vacía que se observa en la figura 4.3 es la zona donde las dianas aparecerán en posiciones aleatorias, pero siempre dentro de los limites dibujados por las columnas. En la figura 4.4 se encuentra una imagen del modelo de diana utilizado.

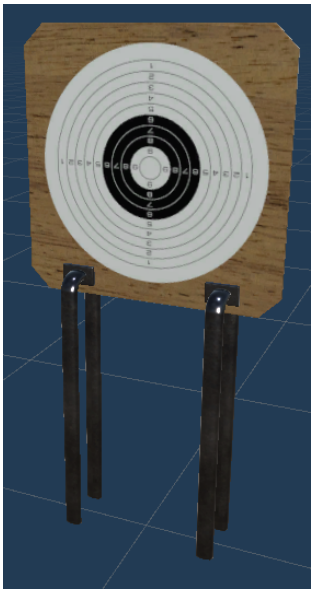


Figura 4.4: Modelo de diana usado (Yanchenkov, s.f.).

El cuadrado verde que se observa en la figura 4.3, corresponde a la zona de realidad virtual. Esta zona es el lugar donde el agente sera controlado por el dispositivo de realidad virtual y no podrá salir de él, pues corresponde con los límites de la zona de seguridad. Esta zona de seguridad marca los límites del espacio físico y sirve para evitar daños tanto a la persona que se encuentra en realidad virtual como al equipo.

En la figura 4.5 se muestran dos imágenes donde se puede apreciar el agente virtual sosteniendo el arco (imagen de la izquierda) y con el arco tensado con una flecha en el (imagen de la derecha). Estas imágenes se han obtenido durante una toma de datos donde el agente era controlado por una persona en realidad virtual.

Los modelos para el arco, las flechas y el carcaj se han obtenido de la página Scketchfab, donde diferentes personas suben sus modelos 3D. Estos modelos se han obtenido de (Panarth, s.f.).

El modelo 3D de la cabeza de robot se ha obtenido de (Designer, s.f.) y su objetivo es servir de guía para la posición de las cámaras virtuales del agente. Esto se debe a que modificar la distancia entre las cámaras del agente puede añadir una dificultad extra y al tener este modelo 3D como guía, ese error se reduce.

(a) Agente con el arco.

(b) Arco tensado.

Figura 4.5: Vistas del agente con el arco.

4.2.2. Entorno virtual del agente

En la figura 4.6 se muestra una imagen superior del entorno 3D donde se ejecutará el agente. Se puede comprobar que el entorno es exactamente el mismo que se observa en la figura 4.3, ya que el agente va a actuar en el mismo entorno donde se obtuvieron los datos.

Figura 4.6: Vista superior del entorno 3D del agente.

Sin embargo, hay una pequeña diferencia, el cuadrado verde que envolvía al agente ya no aparece. Esto es debido a que, en este entorno, el agente utilizará la red neuronal para tomar decisiones e interactuar con el entorno, por lo que la zona de seguridad necesaria para el entorno de realidad virtual ya no se incorpora en este entorno. A pesar de ello, aunque no se aprecie en la figura 4.6, el agente virtual también dispone de una zona de 3x3x3 metros a su alrededor que sirve para enseñar al agente a mantenerse dentro de la zona.

Para controlar el agente se ha creado un *script* en C# que utiliza el *toolbox* de Unity llamada ML Agents. Con este *toolbox* se crea una clase que se encargará de tomar los datos de los sensores del agente y de ejecutar los comandos de control que reciba como entrada. Esto se logra a través de un puerto de comunicación, el 5004, que permite conectar el entorno con un *script* de Python usando la librería ml-agents (Juliani y col., s.f.).

Los sensores utilizados corresponden a dos cámaras que se sitúan en la cabeza del agente, la posición y rotación de ambas manos respecto de la cabeza y la rotación de la cabeza. Estas serán las entradas que deberá de utilizar la red neuronal para determinar que comandos mandar al agente.

En cuanto a los comandos, se utilizará una aproximación similar a la utilizada en (Zhang y col., 2018), donde el efector final es controlado mediante comandos de velocidad lineal y angular. Siguiendo este ejemplo, el agente se controlará mediante comandos de velocidad lineal y angular para cada una de las manos, así como para la cabeza. Adicionalmente tendrá dos comandos extra para determinar si cierra la mano, es decir, coge un objeto, o si lo suelta. En total el agente se controlará con 20 parámetros que corresponderán con la salida de la red.

Para comparar el comportamiento del agente se utiliza primeramente el valor del error. Este error se calculará siguiendo una aproximación similar a la seguida en (Zhang y col., 2018), ya que se trata de un problema similar. Más adelante se explicará con más detalle esta función de error.

El entorno dispondrá de un sistema de puntos que servirá para medir el comportamiento del agente en función de las distintas acciones que logra cumplir:

- **Coger una flecha:** Cada vez que el agente coja una flecha del carcaj sumará 1 puntos.

- **Colocar una flecha en el arco:** Cada vez que el agente coloque una flecha en el arco sumara 5 puntos.
- **Disparar a una diana:** Cada vez que el agente acierte a una diana con una flecha sumara 10 punto.
- **Tirar la flecha fuera del entorno:** Cada vez que el agente suelte la flecha o no acierte en una diana restara 0,5 puntos al total.
- **Salir de la zona:** Cada vez que el agente salga de una zona de 3x3x3 metros a su alrededor restara 1 punto al total.

Con este sistema de puntos, cada vez que el agente complete el ciclo completo de coger una flecha, colocarla en el arco y disparar la flecha, acertando en el blanco, obtendrá un total de 16 puntos. El sistema de castigo por perder una flecha se utilizará para comprobar que el agente no desperdicie flechas y acierte a la primera sobre el objetivo, así como el sistema de castigo por salir de la zona sirve para que el agente se mantenga dentro de la misma zona que es accesible desde el entorno de realidad virtual.

Para el caso del algoritmo PPO, la función de recompensa que se utilizará será el sistema de puntos mencionado anteriormente. Este sistema de puntos permitirá el agente recibir puntos por cada acción completada de forma correcta, lo que le permitirá ir evolucionando su política para completar la tarea asignada. EL sistema de castigo por desperdiciar flechas ayudará a que el agente intente acertar a la primera sobre cada blanco y no entre en un bucle donde coja una flecha, la suelte y coja otra para sumar puntos en la parte inicial del proceso.

4.2.3. Toma de datos

Tal y como se ha expuesto en la sección anterior, el agente dispone de dos cámaras independientes, situadas una en cada ojo, lo que permite obtener una visión estereoscópica del entorno. Sin embargo, el uso de esta estrategia implica una dificultad añadida al sistema de toma de datos.

Para la toma de datos se utilizará un *script* de C# donde se guardarán las posiciones de ambas manos, así como sus rotaciones y la de la cabeza. Las posiciones y rotaciones de las manos serán relativas a la cabeza, mientras que la rotación de la cabeza será global. A la misma vez, se renderizarán las imágenes correspondientes a ambas cámaras,

se comprimirán en formato PNG y se almacenarán con un número que indicará su cronología y permitirá alinearlas con las posiciones. En la figura 4.7 vemos la renderización de ambas cámaras para un mismo instante de tiempo.

(a) Vista desde la cámara izquierda. (b) Vista desde la cámara derecha.

Figura 4.7: Vista de las cámaras del agente.

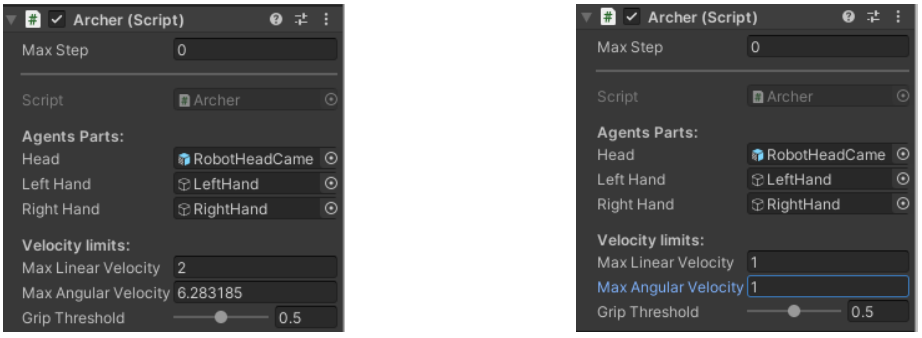
Usando el mismo *script* se almacenarán también las acciones tomadas. Estas acciones corresponden a las velocidades lineales y angulares de ambos efectores finales y de la cabeza, además de la acción de coger o no un objeto, asociado a un botón del mando de realidad virtual. Esto datos, junto con los datos de posición, se guardarán en un archivo CSV, mientras que las imágenes se guardarán en carpetas separadas para cada cámara.

Este sistema de toma de datos es computacionalmente muy costoso, ya que, no solo es necesario renderizar las dos cámaras del agente, comprimir las imágenes y guardarlas en el disco duro, sino que además es necesario renderizar las cámaras con las que la persona que se encuentra en realidad virtual ve el entorno. El motivo de que esto ocurra es que la posición de las cámaras que se utilizan para renderizar el entorno con las gafas de realidad virtual están separadas a la distancia inter pupilar de la persona que las lleve puesta, y esta distancia es distinta para cada persona. Al utilizar dos cámaras adicionales para el agente, independientemente de quien este manejando el agente durante la toma de datos, las cámaras del agente se encontrarán a la misma distancia entre ellas, algo que es realmente importante.

Para reducir la carga computacional y poder mantener una tasa de fotogramas por segunda suficiente para poder cumplir con la tarea, la frecuencia de la toma de datos se

ha reducido a 10Hz, es decir, a 10 tomas por segundo, la misma frecuencia usada por (Zhang y col., 2018) y debería ser suficiente para lograr entrenar al agente.

En la sección 4.1 se mostró un resumen de la estructura de trabajo que se seguirá. En este resumen se explica que ciertos modelos utilizarán los datos de salida normalizados, mientras que otros utilizarán los datos sin normalizar. Esto implica que es necesario crear dos entornos diferentes, uno cuya entrada sean los datos normalizados, y otro para los datos sin normalizar.



(a) Entorno normalizado. (b) Entorno sin normalizar.

Figura 4.8: Vista del panel de control del agente.

Esto se logra modificando dos variables del *script* de control del agente, la velocidad máxima lineal y la velocidad máxima angular. Estas dos variables son las que se utilizarán para obtener el vector de velocidad final que aplicará el agente para trazar las trayectorias. En la figura 4.8 se muestra el panel de control generado por el *script* de control del agente, a la derecha se puede apreciar los valores para la velocidad máxima lineal y angular cuando el entorno espera valores vectores sin normalizar, mientras que a la izquierda se muestran los valores para los vectores normalizados.

4.3. Entrenamiento del agente

El entrenamiento se va a realizar en dos partes. Primeramente se utilizar el algoritmo BC explicado en la sección 2.2. Estos primeros entrenamientos tienen como objetivo inicializar el agente para que logre cumplir parcialmente el objetivo. Posteriormente se utilizará el algoritmo PPO explicado en la sección 2.1 que, utilizando la red neuronal que mejores resultados obtenga en los primeros entrenamiento con BC, terminará de entrenar al agente en el entorno virtual buscando maximizar la puntuación, es decir, el número de

aciertos.

4.3.1. Entrenamiento con BC

El entrenamiento del agente utilizando BC (*Behavioral Cloning*) se realizará con un *script* de Python utilizando los datos recopilados con el entorno virtual donde el agente es controlado mediante un dispositivo de realidad virtual, tal como se explicó en la sección 4.2.3. Estos datos se normalizarán de tal forma que las entradas estarán entre -1 y 1. Para el caso de las imágenes de ambas cámaras, estas se normalizarán entre 0 y 1.

Los datos se almacenarán de forma separada para cada sesión. Los datos utilizados para validación y test serán los datos de dos sesiones independientes. El motivo de crear bancos de datos independientes para cada sesión es asegurar la continuidad de las trayectorias, lo que será importante para aquellas arquitecturas basadas en redes recurrentes. Para aquellas arquitecturas que no se basen en redes recurrentes se utilizara un único banco de datos para el entrenamiento, sin embargo, para la validación y test se usarán los mismos datos mencionados anteriormente.

Partiendo de esta separación entre los datos, las arquitecturas a utilizar también se separan en dos grupos: aquellas basadas con capas recurrentes y aquellas sin ellas. La arquitectura base entre ambas será la misma, con la diferencia de que una de ellas dispondrá de una o varias capas adicionales de tipo LSTM (*Long Short-Term Memory*) (Hochreiter y Schmidhuber, 1997). Al hacer esta diferenciación se podrá comparar si las redes recurrentes aportan mejores resultados que aquellas sin ellas y así comprobar si el tiempo extra necesario para entrenar estas redes Atiya y Parlos, 2000 aporta una mejora significativa. Las arquitecturas que se utilizarán se explicarán más adelante.

Para la creación y entrenamiento de las redes neuronales se utilizará el *framework* Pytorch, realizado por *Facebook AI Research* (Paszke y col., 2019). En concreto se hará uso de una API que simplificará el uso de este *framework*. Esta API se encuentra en un repositorio de GitHub (González, s.f.).

4.3.2. Arquitecturas

Se utilizarán 3 arquitecturas diferentes para el entrenamiento con el algoritmo BC. Todas las arquitecturas estarán divididas en dos partes, una parte convolucional que analizará las dos imágenes generadas por el agente, y una parte basada en capas *fully*

connected (capas donde todas las neuronas de la capa anterior se conectan con todas las neuronas de la siguiente capa) que, usando la información de las capas convolucionales y los datos de posición y rotación de las manos y cabeza, determinará los comandos de velocidad lineal y angular.

Dado que este proyecto trata de entrenar un agente capaz de actuar dentro de un entorno en tiempo real, uno de los principales requisitos para la arquitectura es que pueda ser utilizada en tiempo real. Durante la sección 4.2.3 se expuso la frecuencia a la que se tomaron los datos con los que entrenar al agente, una toma de 10 veces por segundo. Con este dato, el límite superior para considerar la red lo suficientemente rápida se situará a 10 inferencias por segundo, es decir, 10 fotogramas por segundo (fps).

***Nvidia Autonomous Car*:**

Esta arquitectura es la más ligeras de las que se usarán y está inspirada en una arquitectura publicada por Nvidia (Bojarski y col., 2016). Esta arquitectura está pensada originalmente para la conducción de vehículos autónomos, por lo que es capaz de trabajar a tiempo real, utilizando una imagen generada a partir de 3 cámaras que se encuentran sobre el vehículo. En la figura 4.10 se muestra una imagen con la arquitectura original de esta red.

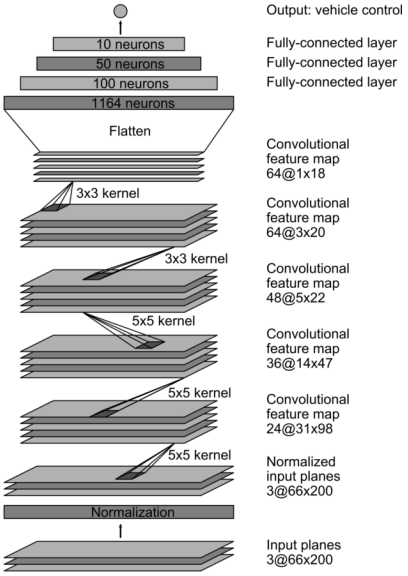


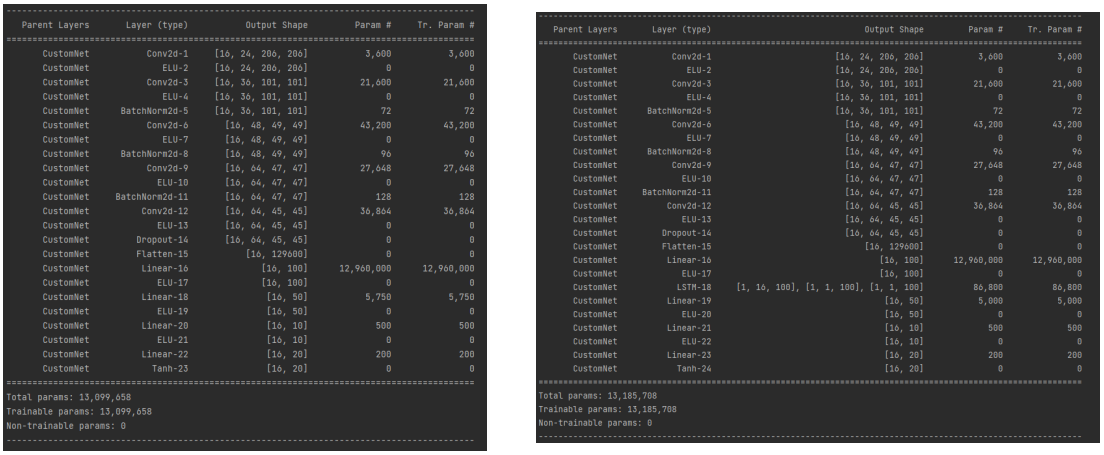
Figura 4.9: Arquitectura de la red de Nvidia (Bojarski y col., 2016).

Esta arquitectura fue adaptada de manera que la salida de la red coincidiera con el tamaño del vector de control del agente, es decir, 20 neuronas. La última capa de

la red utiliza una función de activación de tipo Tanh, para asegurar que los valores se encuentren entre -1 y 1, ya que esta arquitectura será entrenada con los datos de salida normalizados. Por último, la entrada de la red será de las dos imágenes superpuestas, por lo que será de 6x416x416.

Tal como se expuso anteriormente, se entrenarán dos redes diferentes por cada arquitectura, una con capas LSTM y otra sin ellas. Para la versión con capas de tipo LSTM se le añadirá una de estas capas tras la primera capa *Fully Connected*. Se utilizará una LSTM de una sola dirección con 115 neuronas de entrada y 100 de salida.

En la figura 4.10 se observan dos imágenes que muestran un resumen de la arquitectura utilizada. En la imagen de la izquierda se muestra la red sin la capa LSTM, mientras que la imagen de la derecha correspondo con la red que si dispone de una LSTM.



Parent Layers	Layer (type)	Output Shape	Param #	Tr. Param #
CustomNet	Conv2d-1	[16, 24, 206, 206]	3,600	3,600
CustomNet	ELU-2	[16, 24, 206, 206]	0	0
CustomNet	Conv2d-3	[16, 36, 101, 101]	21,600	21,600
CustomNet	ELU-4	[16, 36, 101, 101]	0	0
CustomNet	BatchNorm2d-5	[16, 36, 101, 101]	72	72
CustomNet	Conv2d-6	[16, 48, 49, 49]	43,200	43,200
CustomNet	ELU-7	[16, 48, 49, 49]	0	0
CustomNet	BatchNorm2d-8	[16, 48, 49, 49]	96	96
CustomNet	Conv2d-9	[16, 64, 47, 47]	27,648	27,648
CustomNet	ELU-10	[16, 64, 47, 47]	0	0
CustomNet	BatchNorm2d-11	[16, 64, 47, 47]	128	128
CustomNet	Conv2d-12	[16, 64, 45, 45]	36,864	36,864
CustomNet	ELU-13	[16, 64, 45, 45]	0	0
CustomNet	Dropout-14	[16, 64, 45, 45]	0	0
CustomNet	Flatten-15	[16, 129600]	0	0
CustomNet	Linear-16	[16, 100]	12,960,000	12,960,000
CustomNet	ELU-17	[16, 100]	0	0
CustomNet	Linear-18	[16, 50]	5,750	5,750
CustomNet	ELU-19	[16, 50]	0	0
CustomNet	Linear-20	[16, 10]	500	500
CustomNet	ELU-21	[16, 10]	0	0
CustomNet	Linear-22	[16, 20]	200	200
CustomNet	Tanh-23	[16, 20]	0	0

Total params: 13,099,658
Trainable params: 13,099,658
Non-trainable params: 0

(a) Sin LSTM.

Parent Layers	Layer (type)	Output Shape	Param #	Tr. Param #
CustomNet	Conv2d-1	[16, 24, 206, 206]	3,600	3,600
CustomNet	ELU-2	[16, 24, 206, 206]	0	0
CustomNet	Conv2d-3	[16, 36, 101, 101]	21,600	21,600
CustomNet	ELU-4	[16, 36, 101, 101]	0	0
CustomNet	BatchNorm2d-5	[16, 36, 101, 101]	72	72
CustomNet	Conv2d-6	[16, 48, 49, 49]	43,200	43,200
CustomNet	ELU-7	[16, 48, 49, 49]	0	0
CustomNet	BatchNorm2d-8	[16, 48, 49, 49]	96	96
CustomNet	Conv2d-9	[16, 64, 47, 47]	27,648	27,648
CustomNet	ELU-10	[16, 64, 47, 47]	0	0
CustomNet	BatchNorm2d-11	[16, 64, 47, 47]	128	128
CustomNet	Conv2d-12	[16, 64, 45, 45]	36,864	36,864
CustomNet	ELU-13	[16, 64, 45, 45]	0	0
CustomNet	Dropout-14	[16, 64, 45, 45]	0	0
CustomNet	Flatten-15	[16, 129600]	0	0
CustomNet	Linear-16	[16, 100]	12,960,000	12,960,000
CustomNet	ELU-17	[16, 100]	0	0
CustomNet	LSTM-18	[1, 16, 100], [1, 1, 100], [1, 1, 100]	86,800	86,800
CustomNet	Linear-19	[16, 50]	5,000	5,000
CustomNet	ELU-20	[16, 50]	0	0
CustomNet	Linear-21	[16, 10]	500	500
CustomNet	ELU-22	[16, 10]	0	0
CustomNet	Linear-23	[16, 20]	200	200
CustomNet	Tanh-24	[16, 20]	0	0

Total params: 13,185,708
Trainable params: 13,185,708
Non-trainable params: 0

(b) Con LSTM.

Figura 4.10: Resumen de las redes basadas en la arquitectura de Nvidia.

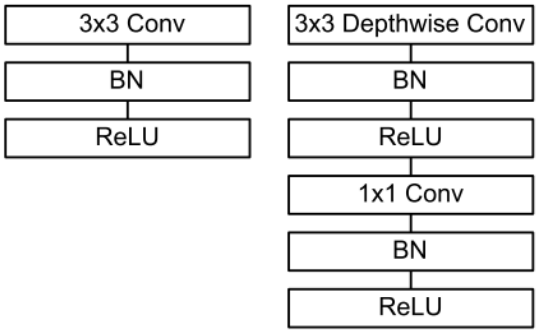
***MobilenetV1*:**

Es arquitectura está basada en la primera versión de MobileNet (Howard y col., 2017). Se trata de una arquitectura de clasificación de imágenes pensada para utilizarse en dispositivos móviles de poca potencia de cómputo, por lo que es lo suficientemente ligera como para poder utilizarse en tiempo real. En la figura 4.12 se muestran dos imágenes.

La imagen de la derecha se trata de la arquitectura original de la red MobileNet. La imagen de la izquierda corresponde a los dos tipos de bloques convolucionales utilizados en la arquitectura.

De esta arquitectura se utilizará la parte convolucional, ya que el problema que se

trata en este proyecto es de tipo regresión. Para ello, tras la parte convolucional de la arquitectura se utilizara un tipo de capa llamada *Spatial Soft Argmax*, cuya tarea será la de obtener la posición X e Y del píxel de mayor activación para cada una de los canales. Esto implica que, para la última capa de la arquitectura de la figura 4.11b, tras pasar por la capa *Spatial Soft Argmax* se obtendra una matriz de 2x1024 dimensiones que codifica la posición X e Y de máxima activación para cada uno de los 1024 canales de salida. Esta es una de las capas utilizadas por (Zhang y col., 2018) tras la parte convolucional de su red.



(a) Bloques convolucionales usados en MobileNet.

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

(b) Arquitectura de la red MobileNet.

Figura 4.11: Red MobileNet (Howard y col., 2017)

Tras la capa *Spatial Soft Argmax* se añadirá una capa *Flatten* (capa que convierte la entrada en un vector de una sola dimensión) y 3 capas *Fully Connected* (llamadas *Linear* en Pytorch), donde las dos primeras usarán una función de activación ELU, para no poner completamente a 0 los valores negativos, y la última una función de activación Tanh, ya que esta arquitectura se entrenará con los datos de salida normalizados.

Para la versión con LSTM se añadirá una capa LSTM de una sola dirección tras la primera capa Fully Connected.

```
CustomNet    SpatialSoftArgmax2d-82          [16, 1024, 2]          0          0
CustomNet                Flatten-83             [16, 2048]          0          0
CustomNet                 Linear-84             [16, 1000]  2,064,000  2,064,000
CustomNet                    ELU-85             [16, 1000]          0          0
CustomNet                 Linear-86              [16, 100]    100,100    100,100
CustomNet                    ELU-87              [16, 100]          0          0
CustomNet                 Linear-88               [16, 20]      2,020      2,020
CustomNet                   Tanh-89               [16, 20]          0          0
==========================================================================================
Total params: 5,373,960
Trainable params: 5,373,960
Non-trainable params: 0
------------------------------------------------------------------------------------------
```

(a) Sin LSTM.

```
CustomNet    SpatialSoftArgmax2d-82                [16, 1024, 2]          0          0
CustomNet                Flatten-83                   [16, 2048]          0          0
CustomNet                 Linear-84                   [16, 1000]  2,064,000  2,064,000
CustomNet                   LSTM-85  [1, 16, 1000], [1, 1, 1000], [1, 1, 1000]  8,008,000  8,008,000
CustomNet                    ELU-86                   [16, 1000]          0          0
CustomNet                 Linear-87                    [16, 100]    100,100    100,100
CustomNet                    ELU-88                    [16, 100]          0          0
CustomNet                 Linear-89                     [16, 20]      2,020      2,020
CustomNet                   Tanh-90                     [16, 20]          0          0
==========================================================================================
Total params: 13,381,960
Trainable params: 13,381,960
Non-trainable params: 0
------------------------------------------------------------------------------------------
```

(b) Con LSTM.

Figura 4.12: Resumen de las redes basadas en la arquitectura de MobileNet.

En la figura 4.12 se muestra la parte no convolucional de la arquitectura, tanto para la versión sin LSTM (a la izquierda) como para la versión con LSTM (a la derecha). La parte convolucional es igual a la mostrada en la figura 4.11b.

ResNet18:

La arquitectura ResNet aparece como una solución ante el problema de *Vanishing Gradient* en las redes neuronales profundas. Esto ocurre debido a que, conforme más profunda se vuelve la red, los gradientes que se retro propagan comienzan a volverse tan pequeños que las primeras capas de la red no logran entrenarse correctamente (Tan y Lim, 2019).

Para solventar este problema, los creadores de las redes ResNet proponen el uso de bloques residuales que, tras aplicar una serie de capas, la entrada del bloque se suma a la salida del mismo. Este logra que los gradientes al retro propagarse no se aproximen a 0 tan rápidamente y así se logre entrenar redes más profundas (He y col., 2016). En la figura 4.13 se muestra una imagen que representa un bloque residual básico.

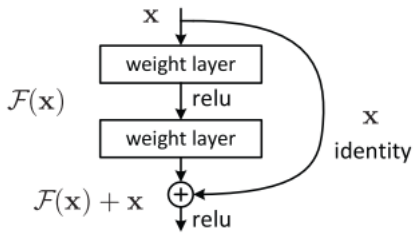
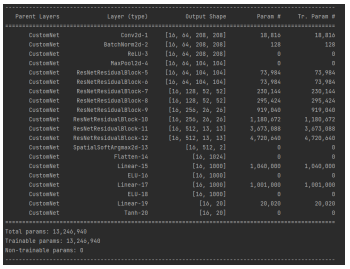


Figura 4.13: Bloque residual (He y col., 2016).

Para este proyecto se ha utilizado la arquitectura ResNet18, siendo la menos profunda de las propuestas en el artículo original He y col., 2016. Para esta arquitectura, cada bloque residual dispone de dos capas convolucionales con función de activación *ReLu* y *Batch Normalization*.

Tras la última capa convolucional, la arquitectura seguida es similar a la expuesta en la red basada en MobileNet. Usando una capa de tipo *Spatial Soft Argmax* se obtienen las posiciones X e Y de máxima activación en cada canal, en este caso 512 pues ese es el número de canales obtenidos tras la última capa convolucional.

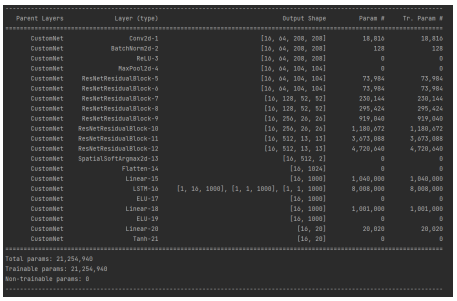
Tras la capa *Spatial Soft Argmax* se apilaran 3 capas *Fully Connected* con la misma configuración usada en la red MobileNet. En la figura 4.14a se muestra un resumen de la arquitectura utilizada.



Parent Layers	Layer (type)	Output Shape	Param #	Tr. Param #
CustomNet	Conv2d-1	[16, 64, 208, 208]	18,816	18,816
CustomNet	BatchNorm2d-2	[16, 64, 208, 208]	128	128
CustomNet	ReLU-3	[16, 64, 208, 208]	0	0
CustomNet	MaxPool2d-4	[16, 64, 104, 104]	0	0
CustomNet	ResNetResidualBlock-5	[16, 64, 104, 104]	73,984	73,984
CustomNet	ResNetResidualBlock-6	[16, 64, 104, 104]	73,984	73,984
CustomNet	ResNetResidualBlock-7	[16, 128, 52, 52]	230,144	230,144
CustomNet	ResNetResidualBlock-8	[16, 128, 52, 52]	295,424	295,424
CustomNet	ResNetResidualBlock-9	[16, 256, 26, 26]	919,040	919,040
CustomNet	ResNetResidualBlock-10	[16, 256, 26, 26]	1,180,672	1,180,672
CustomNet	ResNetResidualBlock-11	[16, 512, 13, 13]	3,673,088	3,673,088
CustomNet	ResNetResidualBlock-12	[16, 512, 13, 13]	4,720,640	4,720,640
CustomNet	SpatialSoftArgmax2d-13	[16, 512, 2]	0	0
CustomNet	Flatten-14	[16, 1024]	0	0
CustomNet	Linear-15	[16, 1000]	1,040,000	1,040,000
CustomNet	ELU-16	[16, 1000]	0	0
CustomNet	Linear-17	[16, 1000]	1,001,000	1,001,000
CustomNet	ELU-18	[16, 1000]	0	0
CustomNet	Linear-19	[16, 20]	20,020	20,020
CustomNet	Tanh-20	[16, 20]	0	0

Total params: 13,240,940
Trainable params: 13,240,940
Non-trainable params: 0

(a) Sin LSTM.



Parent Layers	Layer (type)	Output Shape	Param #	Tr. Param #
CustomNet	Conv2d-1	[16, 64, 208, 208]	18,816	18,816
CustomNet	BatchNorm2d-2	[16, 64, 208, 208]	128	128
CustomNet	ReLU-3	[16, 64, 208, 208]	0	0
CustomNet	MaxPool2d-4	[16, 64, 104, 104]	0	0
CustomNet	ResNetResidualBlock-5	[16, 64, 104, 104]	73,984	73,984
CustomNet	ResNetResidualBlock-6	[16, 64, 104, 104]	73,984	73,984
CustomNet	ResNetResidualBlock-7	[16, 128, 52, 52]	230,144	230,144
CustomNet	ResNetResidualBlock-8	[16, 128, 52, 52]	295,424	295,424
CustomNet	ResNetResidualBlock-9	[16, 256, 26, 26]	919,040	919,040
CustomNet	ResNetResidualBlock-10	[16, 256, 26, 26]	1,180,672	1,180,672
CustomNet	ResNetResidualBlock-11	[16, 512, 13, 13]	3,673,088	3,673,088
CustomNet	ResNetResidualBlock-12	[16, 512, 13, 13]	4,720,640	4,720,640
CustomNet	SpatialSoftArgmax2d-13	[16, 512, 2]	0	0
CustomNet	Flatten-14	[16, 1024]	0	0
CustomNet	Linear-15	[16, 1000]	1,040,000	1,040,000
CustomNet	LSTM-16	[1, 16, 1000], [1, 1, 1000], [1, 1, 1000]	8,008,000	8,008,000
CustomNet	ELU-17	[16, 1000]	0	0
CustomNet	Linear-18	[16, 1000]	1,001,000	1,001,000
CustomNet	ELU-19	[16, 1000]	0	0
CustomNet	Linear-20	[16, 20]	20,020	20,020
CustomNet	Tanh-21	[16, 20]	0	0

Total params: 21,254,940
Trainable params: 21,254,940
Non-trainable params: 0

(b) Con LSTM.

Figura 4.14: Resumen de las redes basadas en la arquitectura de ResNet18.

Para la red que utiliza LSTM, esta capa se ha añadido tras la primera capa *Fully Connected*, tal como se puede observar en la figura 4.14b.

Para esta arquitectura se entrenarán dos modelos extras donde la función de la última capa será una función lineal. Para este caso concreto se utilizarán los datos de salida no normalizados que se vieron en la sección 4.1.

Con este cambio se pretende comprobar si al eliminar la restricción del vector de salida de la red se pueden lograr los mismo o mejores resultados, ya que la función de activación *Tanh* genera varios problemas relacionados con la muerte del gradiente durante la retro propagación (Lau y Lim, 2017).

4.3.3. Métrica de error

Tal como se comentó en secciones anteriores, el cálculo del error se realizará de forma similar al realizado por (Zhang y col., 2018). Este cálculo se basa en la suma ponderada de 4 calculo diferentes:

- **L1 *loss*:** Media del valor absoluto del error (Pesme y Flammarion, s.f.).
- **L2 *loss*:** Media del cuadrado del error (Xie y col., s.f.).
- **Similitud coseno:** Utilizando la similitud coseno se calculara el error en dirección. Esta similitud es entre 0 y 1, donde 1 corresponde a vectores de igual dirección y 0 a vectores completamente perpendiculares (Dehak y col., s.f.). Para solventar este problema el error se calculará utilizando la ecuación $1 - SimilitudCoseno$.
- ***Sigmoid Cross Entropy*:** Esta métrica se utilizará únicamente para calcular el error del abierto y cerrado de las manos.

El error final será una suma ponderada de estos errores. Para el caso de este proyecto, el error más importante es el error de la dirección de los vectores, que se calcula con la similitud coseno. Tras ese se encuentra el error del comando de agarre, calculado con *sigmoid cross entropy*, y en última instancia los errores por magnitud de los vectores L1 y L2 *loss*. Atendiendo a esta diferenciación se determinará un peso de 1 para la similitud del coseno, un peso de 0,5 para *sigmoid cross entropy* y un peso de 0,25 para L1 y L2 *loss*.

4.3.4. Entrenamiento con PPO

Como último punto del proyecto, se realizará un entrenamiento con el algoritmo PPO utilizando la red que mejores resultados ofrezca durante el entrenamiento con BC. Para determinar la red que mejor se comporta se utilizará la medida del *loss* final, tanto de test (datos con los que la red no se ha entrenado) como de entrenamiento (datos con los que la red se entrenó).

El algoritmo PPO es un algoritmo de aprendizaje por refuerzo que requiere de dos redes, la red que controla al agente (actor) y una red encargada de aproximar la función de valor (crítico). Ambas redes se crearán utilizando Pytorch, específicamente la API del repositorio de GitHub (González, s.f.).

La red actor (encargada de controlar al agente) no comenzará un entrenamiento desde 0, sino que se utilizará la red que mejores resultados obtuviera durante el entrenamiento con BC. Este se debe a que una red ya inicializada debería de tardar menos, pues sus acciones durante los primeros episodios de entrenamiento no serán aleatorias. La red crítico (encargada de aproximar la función valor) si comenzará desde 0, aproximando con cada episodio la función valor del entorno a partir del estado del mismo.

El código de entrenamiento utilizado para el algoritmo de PPO se encuentra en la misma API con la que se crearán las redes, estando perfectamente integrado (González, s.f.).

Además de utilizar los modelos seleccionados del entrenamiento previo con BC, se realizarán dos entrenamientos utilizando las mismas arquitecturas, pero sin la inicialización previa con BC. Con esto se pretende comprobar si el haber inicializado el modelo con BC logra mejorar los resultados, tanto de puntuación máxima como de tiempo de entrenamiento, frente a un entrenamiento que parta desde 0. En la sección 5.2 se muestran los resultados obtenidos utilizando el algoritmo PPO.

Capítulo 5

Análisis de resultados

En este capítulo se procederá amostrar los resultados obtenidos para cada una de las fases del proyecto. Comenzando con los entrenamientos de BC para cada arquitectura, tanto para las redes con capas de tipo LSTM como para las que no disponen de este tipo de capas.

Posteriormente se expondrán los resultados obtenidos para los entrenamientos realizados con PPO, mostrando tanto la evolución de la recompensa como del *loss* (error) de la red actor y crítico.

5.1. *Behavioral Cloning*

Los entrenamientos utilizando BC se han realizado utilizando el servicio de *Cloud Computing* de Microsoft Azure, utilizando la cuenta de estudiante provista por la UNIR. Se ha decidido utilizar este servicio para acelerar los entrenamientos ya que la cantidad de datos utilizada hizo necesario el uso de máquinas con alta capacidad de cómputo.

Los entrenamientos, a pesar de disponer de mayor potencia de cómputo gracias al uso de este servicio, han durado una media de 16 horas para las redes sin capas LSTM y de 33 horas para aquellas con capas LSTM. Para subir los archivos a la plataforma Azure se hizo uso de la aplicación de escritorio *Microsoft Azure Storage Explorer*, con la que se pueden controlar las bases de datos que utiliza el servicio de Azure y así poder subir los datos de entrenamiento, así como los códigos.

Para los entrenamientos se tomaron un total de 46 minutos de datos en el entorno virtual controlado mediante realidad virtual. De estos 46 minutos, 42 se destinaron a entrenamiento, 2 a validación durante el entrenamiento y 2 a un test final sobre la red

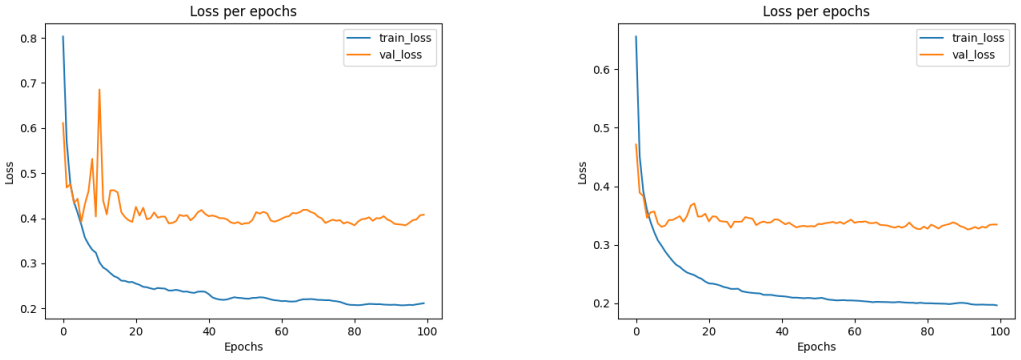
entrenada.

En cuanto a los parámetros de entrenamiento, se ha hecho uso del optimizador SGD con *momentum* (Y. Liu y col., 2020), buscando evitar que el modelo se quede atrapado en puntos de gradiente 0 y tenga un entrenamiento más estable. Como *learning rate* (tasa de aprendizaje) se ha utilizado 0.01 para todas las redes con un valor de *momentum* de 0.9. Todos los entrenamientos han durado 100 épocas y se ha utilizado la función de *loss* expuesta en la sección 4.3.3.

Es importante destacar que, gracias a que se dispone de datos de validación, el modelo final guardado no corresponderá con el modelo obtenido tras la última época de entrenamiento. Este modelo será aquel que obtenga el menor *loss* de validación durante el entrenamiento, de forma que en caso de presencia de *overfitting* (sobre entrenamiento, incapacidad de generalizar), el modelo obtenido sea siempre aquel que mejor se comporte para datos nuevos.

***Nvidia Autonomous Car*:**

La primera arquitectura entrenada fue aquella basada en el artículo publicado por Nvidia sobre la conducción de un coche autónomo basado en visión artificial (Bojarski y col., 2016). Esta arquitectura tenía la ventaja de ser bastante liviana y pensada para ser utilizada en entornos que requieren de toma de decisiones rápidas, como es el caso de la conducción autónoma.



(a) Arquitectura sin LSTM. (b) Arquitectura con LSTM.

Figura 5.1: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red de Nvidia (Bojarski y col., 2016).

En la figura 5.1 se muestran dos gráficas con la evolución del valor de *loss* tanto para

el entrenamiento como para la validación. En ambas gráficas se puede apreciar un gran *overfitting*, ya que el valor de *loss* de validación se estabiliza entre 0.4 y 0.3, mientras que el valor de *loss* de entrenamiento continúa bajando hasta 0.2.

En la figura 5.1a se muestra la evolución del valor de *loss* a lo largo de las distintas épocas de entrenamiento para la arquitectura sin LSTM, mientras que el la figura 5.1b se muestra la misma evolución pero para la red con capas LSTM. Comparando ambas figuras se puede comprobar que el modelo con capas LSTM se comporta ligeramente mejor que el modelo sin este tipo de capas, teniendo una evolución más estable durante el entrenamiento.

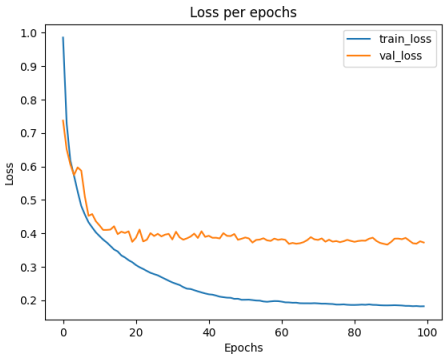
Cuando comparamos los valores de test finales de ambas redes (0.401 para la red sin LSTM y 0.362 para la red con LSTM) se confirma que la arquitectura basada en LSTM obtiene mejores resultados, por un pequeño margen, frente a aquella sin LSTM. Sin embargo, este pequeño margen puede no ser suficiente para justificar el tiempo de entrenamiento extra que se hace necesario, tal como se expuso en al inicio de esta sección.

Por último, la época donde se obtuvo el modelo que menor *loss* de validación obtuvo fue en la época 81, para la red sin LSTM, y 92, para la red con LSTM. Esto muestra que, a pesar del *overfitting* presente, el modelo pudo mejorar ligeramente los resultados gracias a no detener el entrenamiento hasta las 100 épocas. Sin embargo, el exceso de *overfitting* debe de ser tratado, ya que este modelo no logra cumplir la tarea propuesta para el agente, mostrando movimientos casi aleatorios cuando se utiliza en el entorno.

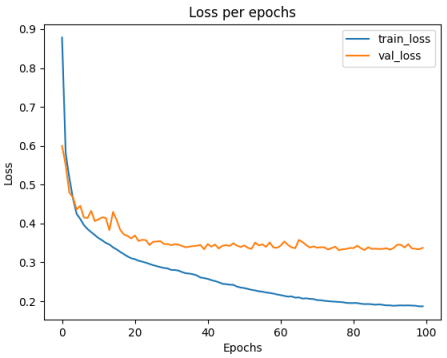
***MobileNetV1*:**

A continuación se mostrarán los resultados obtenidos con la arquitectura basada en la red MobileNet versión 1 (Howard y col., 2017). Se trata de una arquitectura de clasificación de imágenes, por lo que se ha tenido que modificar para adaptarla a este proyecto. Para más detalles sobre esta adaptación consultar la sección 4.3.2.

En la figura 5.2 se muestran las gráficas correspondientes a los entrenamientos de la redes. En estas gráficas se puede observar la evolución del *loss* tanto de entrenamiento como de validación. Tal como ocurría con la arquitectura basada en el artículo de Nvidia (Bojarski y col., 2016), se puede observar un gran *overfitting* para ambos modelos. En este caso, el *loss* de validación se queda entre 0.4 y 0.3, mientras que el *loss* de entrenamiento baja de 0.2.



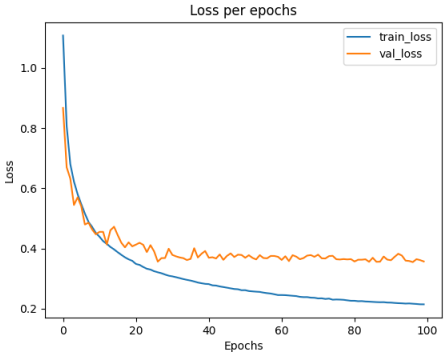
(a) Arquitectura sin LSTM.



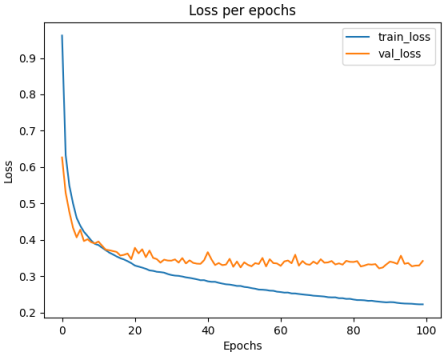
(b) Arquitectura con LSTM.

Figura 5.2: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red MobileNet (Howard y col., 2017).

Para intentar solventar este *overfitting* se realizaron dos entrenamientos nuevos donde se añadieron capas *dropout* para intentar reducir el *overfitting* en ambos modelos. En la figura 5.3 se muestran las gráficas resultantes de entrenar estos modelos.



(a) Arquitectura sin LSTM.



(b) Arquitectura con LSTM.

Figura 5.3: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red MobileNet con capas *dropout*.

Comparando las figuras 5.2 y 5.3 se puede comprobar que añadir capas de tipo *dropout* mejoran ligeramente el *overfitting* durante el entrenamiento, sin embargo no lo eliminan del todo.

Al compararlos resultados obtenidos durante la fase de test (0.387 para la red sin LSTM ni *dropout* y 0.341 para la red con LSTM pero sin *dropout*) se puede comprobar que la red basada en LSTM mejora ligeramente los resultados, tal como ocurría con la

red basada en el artículo de Nvidia, pero con una diferencia aún menor. Esta diferencia tan pequeña da a entender que los modelos basados en LSTM no aportan una mejora sustancial, por lo que parecen no ser los modelos óptimos.

Para el caso de los modelos con *dropout*, los resultados de test son prácticamente identicos (0.392 para la red sin LSTM y 0.347 para la red con LSTM), lo que sugiere que agregar capas dropout no aporta ninguna mejora para este caso.

Por último, las épocas donde se obtuvieron los mejores modelos para las redes sin *dropout* fueron las épocas 90 y 77 (sin LSTM y con LSTM), mientras que para los modelos con *dropout* fueron 97 y 88 (sin LSTM y con LSTM). En este aspecto si se observa una pequeña diferencia que puede dar a entender que los modelos con *dropout* tienden a obtener sus mejores resultados conforme más épocas ocurren, sin embargo, la diferencia vuelve a ser muy pequeña.

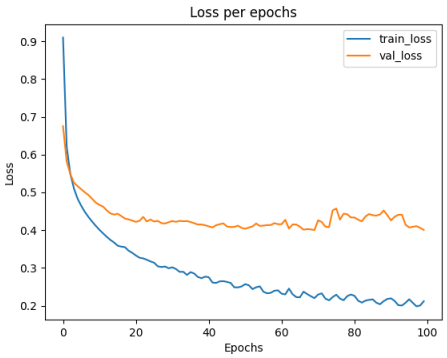
***ResNet18*:**

La última familia de arquitecturas que se utilizará son las basadas en ResNet18. Esta arquitectura utiliza bloques residuales para reducir el problema de *Vanishing Gradient* (desvanecimiento del gradiente) cuando las redes son muy profundas (He y col., 2016). Sin embargo, esta arquitectura está pensada originalmente para tareas de clasificación de imágenes, por lo que para lograr que funcione fue necesario adaptar la red para este proyecto. En la sección 4.3.2 se explican estos cambios.

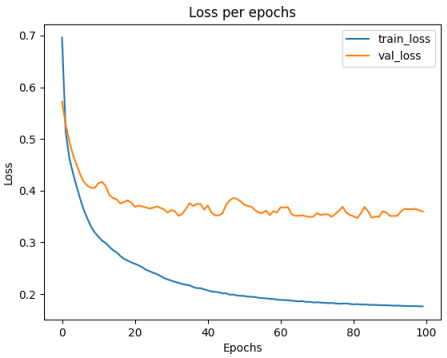
En la figura 5.4 se muestran dos gráficas con los valores de loss para entrenamiento y validación, tanto para la red con LSTM (derecha) como la red sin LSTM (izquierda). En ambas gráficas se puede apreciar claramente la presencia de *overfitting*, tal como ocurría con las otras arquitecturas nombradas anteriormente. En cuanto al entrenamiento, se puede observar que es bastante estable, estabilizándose entre 0.4 y 0.3 para la validación, y quedando por debajo de 9.2 para entrenamiento.

Para intentar reducir el *overfitting* presente se agregaron capas *dropout* al modelo, volviendo a realizar el entrenamiento. Los resultados de estos entrenamientos se pueden observar en la figura 5.5. Abas gráficas muestran aun presencia de *overfitting*, por lo que es posible concluir que agregar estas capas *dropout* no mejora el rendimiento de la red.

Al comprobar los valores de test (0.392 para el modelo sin LSTM ni *dropout* y 0.358 para el modelo con LSTM y sin *dropout*) no se puede apreciar una mejora significativa respecto de las otras dos arquitecturas. A pesar de ello, el error de entrenamiento es

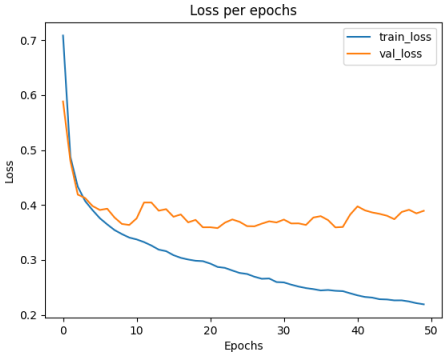


(a) Arquitectura sin LSTM.

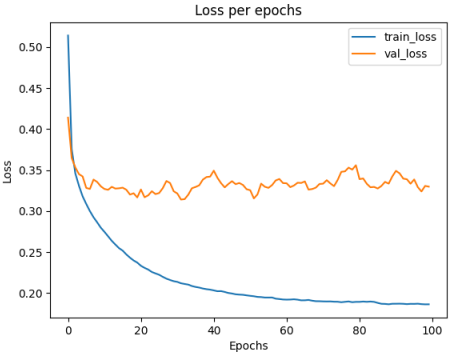


(b) Arquitectura con LSTM.

Figura 5.4: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red ResNet18 (He y col., 2016).



(a) Arquitectura sin LSTM.



(b) Arquitectura con LSTM.

Figura 5.5: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red ResNet18 con *dropout*.

ligeramente inferior, respecto de los otros modelos, por lo que parece que el *overfitting* es aún más acentuado con esta arquitectura. En cuanto a la época donde se obtuvo el mejor modelo, para la red sin LSTM fue la época 70, mientras que para la red con LSTM fue en la época 82.

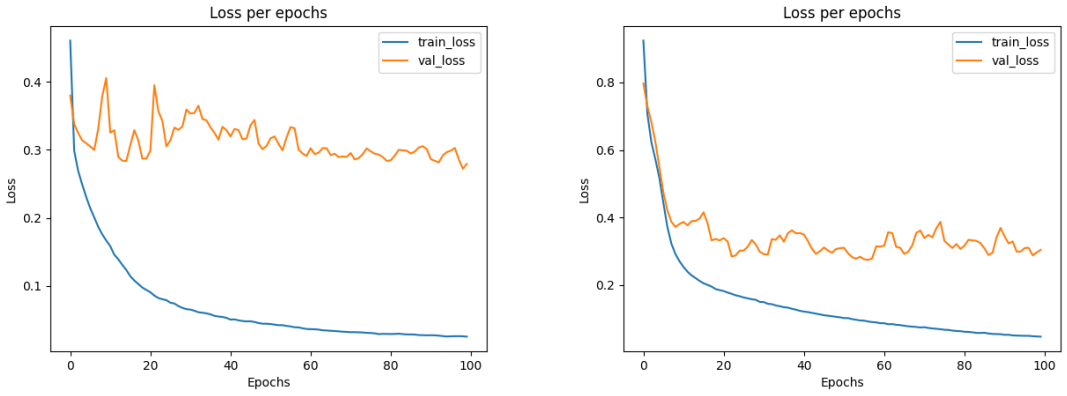
Para el caso de los modelos con *dropout*, los valores de test son realmente similares (0.36 para el modelo sin LSTM y 0.312 para el modelo con LSTM). Sin embargo, en el modelo con LSTM si se puede comprobar una mejora algo mayor, aunque igualmente pequeña, con respecto a los modelos sin *dropout*.

Donde sí se puede apreciar un gran cambio es en las épocas donde se obtuvieron los mejores modelos, quedando la red con LSTM en la época 32 y el modelo sin LSTM 22.

En esta caso se puede observar cómo los modelos dejaron de mejorar mucho antes que con el resto de arquitecturas, lo que explica porque en la figura 5.5a el entrenamiento se detuviera a las 50 épocas en vez de a las 100.

Como últimos dos modelos se entrenarán dos redes donde se modifique la función de activación de la capa final, sustituyendo la función *Tanh* actual por una función lineal. Estos dos modelos se entrenarán con los datos de salida sin normalizar, por lo que la red no solo tendrá que aprender a determinar la dirección del vector, sino también su magnitud. La idea de estos modelos es intentar eliminar el problema de la muerte del gradiente para valores cercanos a 1 o -1.

En la figura 5.6 se puede observar la evolución del *loss* tanto de entrenamiento como de validación para los modelos con función de activación lineal. A la derecha se muestra la gráfica correspondiente al modelo con LSTM, mientras que a la izquierda se muestra la gráfica del modelo sin LSTM.



(a) Arquitectura sin LSTM.

(b) Arquitectura con LSTM.

Figura 5.6: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red ResNet18 con función de activación lineal en la última capa.

Para este entrenamiento fue necesario modificar ligeramente los parámetros de entrenamiento, ya que la función de activación no limita el valor de salida entre -1 y 1, por lo que fue necesario bajar el valor de *learning rate* a 0.001.

Al observar la figura 5.6 se puede comprobar como sigue estando presente un gran *overfitting*, tanto en el modelo con LSTM como en el modelo sin LSTM. Este *overfitting* es algo mayor que los anteriores. En este caso, el *loss* de validación se queda entorno a los 0.3, y el *loss* de entrenamiento baja de 0.1. Esto demuestra que al eliminar la función de

activación *Tanh* de la red, a pesar de mejorar el *loss* de validación, el *overfitting* aumenta considerablemente.

Al comprobar los valores de test (0.279 para el modelo sin LSTM y 0.277 para el modelo con LSTM) se puede comprobar que efectivamente, los valores de *loss* han mejorado ligeramente respecto de los entrenamientos antes realizados. A pesar de esta mejora, el aumento del *overfitting* empeora el funcionamiento de la red al introducirla en el entorno.

Por último, las épocas donde se obtuvieron los mejores modelos corresponden a las épocas 99 y 57 para el modelo sin LSTM y con LSTM respectivamente. Se puede ver una gran diferencia entre el modelo con LSTM y sin LSTM, algo que no se veía con las demás arquitecturas.

Resumen:

Para concluir esta sección se va a mostrar los resultados obtenidos por cada modelo entrenado con BC. A partir de estos resultados se decidirá que modelos pasarán a ser entrenado con el algoritmo PPO, cuyos resultados se encuentran en la sección 5.2.

La decisión sobre qué modelo utilizar no dependerá únicamente del *loss* de test obtenido, sino también de la época donde se obtuvo el mejor modelo. Esto se debe a que, tal como se ha expuesto anteriormente, todos los modelos presentan un gran *overfitting*, por lo que, un modelo que logre su mejor resultado en época tempranas presentará menor *overfitting* que uno que lo logre en las últimas épocas.

En la tabla 5.1 se recogen los resultados obtenidos para todos los modelos entrenados, tanto aquellos con capas LSTM como aquellos sin dichas capas. Observando la columna ”*Loss* de test”, que corresponde con el *loss* obtenido durante la fase de test de los modelos, se puede comprobar que ha ido mejorando ligeramente conforme más profunda se hace la red, alcanzando su mínimo con las arquitecturas ResNet18 con función de activación lineal en su última capa.

Por otro lado, la columna ”Mejor época”de la tabla 5.1, correspondiente con la época donde se obtuvo el mejor modelo, muestra como casi todas las arquitecturas obtuvieron sus mejores resultados en las últimas épocas, cuando el *overfitting* era mayor. Las únicas excepciones fueron las arquitecturas basadas en ResNet18 con *dropout*, que dejaron de mejorar en épocas tempranas, cuando el *overfitting* aún no era tan pronunciado.

Teniendo en cuenta estos dos datos, los modelos más prometedores serían los modelos basados en ResNet18 con *dropout*. Eso se debe a que la diferencia entre el *loss*

Arquitectura	Loss de test	Mejor época
Nvidia	0,401	81
Nvidia LSTM	0,362	92
MobileNet	0,387	90
MobileNet LSTM	0,341	77
MobileNet con dropout	0,392	97
MobileNet LSTM con dropout	0,347	88
ResNet18	0,392	70
ResNet18 LSTM	0,358	82
ResNet18 con dropout	0,36	22
ResNet18 LSTM con dropout	0,312	32
ResNet18 Lineal	0,279	99
ResNet18 LSTM Lineal	0,277	57

Tabla 5.1: Resumen con los resultados obtenidos por cada modelo.

de test es pequeña comparado con los modelos basados en ResNet18 con función de activación lineal, pero la época donde se obtuvieron los mejores resultados es bastante más temprana. En estas épocas se puede apreciar menor *overfitting* viendo las gráficas de la figura 5.5.

Basándonos en estos datos se procederá a utilizar dos modelos para el entrenamiento con PPO. Estos dos modelos corresponden con las redes basadas en ResNet18 con *dropout*, tanto la arquitectura con capas LSTM como la que no dispone de este tipo de capas.

Utilizar dos modelos para el entrenamiento con PPO permitirá comprobar si la presencia de capas LSTM mejora el rendimiento frente a los modelos que no disponen de estas capas.

5.2. *Proximal Policy Optimization*

En esta sección se procederá a mostrar los resultados obtenidos en los entrenamiento realizados con el algoritmo *Proximal Policy Optimization* (PPO). Se trata de un algoritmo de aprendizaje por refuerzo, por lo que no requiere de ningún sistema de toma de datos,

pero si de un entorno donde el agente aprenderá.

En la sección 4.2.2 se expuso el entorno virtual donde el agente se entrenaría. Este entorno es idéntico al utilizado para la toma de datos, con la salvedad de que el agente es controlado por un modelo de IA, una red neuronal, que aprenderá con el tiempo. En la sección 4.2.2 también se mostró la función de recompensa que se utilizará durante este entrenamiento.

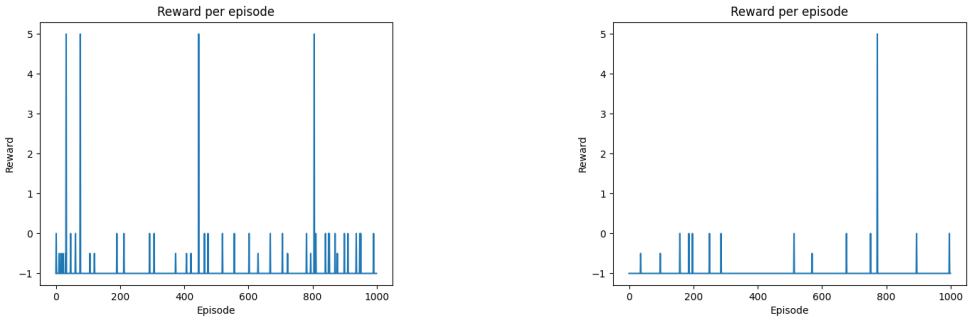
Tal como se explicó en la sección 4.3.4, se van a realizar 4 entrenamientos utilizando los mismos hiper-parámetros para todos ellos. Estos hiper-parámetros se han seleccionado realizando varios entrenamientos cortos (100 episodios) y comprobando como evolucionaba el agente con la arquitectura elegida (ResNet18 con *dropout*). A continuación, se muestra una lista con los hiper parámetros seleccionados:

- ***Learning rate***: 0.001.
- **Optimizador**: SGD.
- **Épocas**: 10.
- **Gamma**: 0.9.
- **Lambda**: 0.95.
- **Épsilon**: 0.2.
- **Beta**: 0.001.
- **Desviación típica inicial**: 0.5.
- **Desviación típica mínima**: 0.1.

Una vez seleccionados los hiper-parámetros, se procedió a entrenar los modelos. Para ello se utilizaron 1000 episodios, lo que implicó un entrenamiento de 30 horas por cada modelo aproximadamente.

Modelos sin LSTM

A continuación, se mostrarán los resultados obtenidos para los modelos sin capas LSTM dentro de sus arquitecturas. Se mostrará tanto la recompensa obtenida a lo largo de los episodios, como el *loss* obtenido tanto por parte de la red actor como de la red crítico.

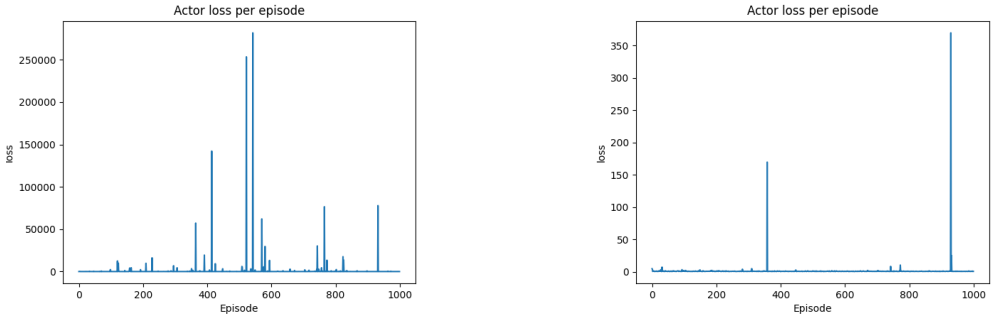


(a) Modelo inicializado con BC. (b) Modelo sin inicializar.

Figura 5.7: Gráfica con la recompensa obtenida por el agente para los modelos sin LSTM.

En la figura 5.7 se puede observar la evolución del valor de la recompensa a lo largo de los episodios. A la izquierda se muestra la recompensa obtenida por el modelo inicializado utilizando el algoritmo BC, mientras que a la derecha se muestra la recompensa obtenida por el modelo sin inicializar.

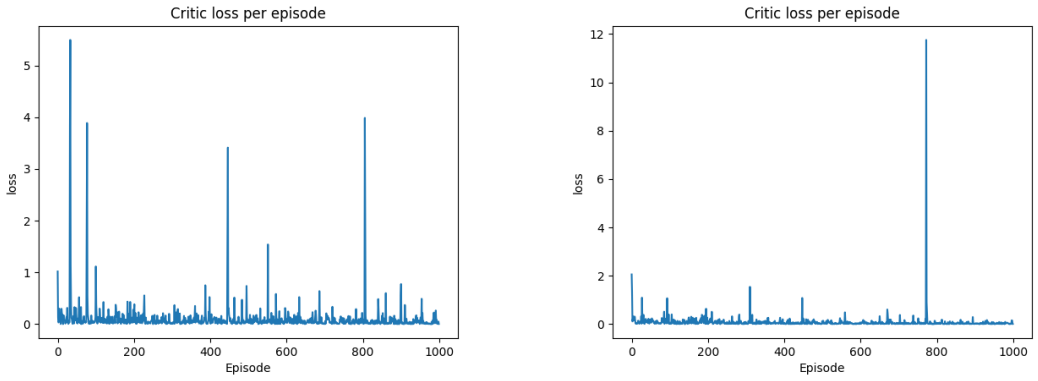
Comparando las gráficas de la figura 5.7, se puede ver una ligera diferencia en cuanta a la distribución de las recompensas. El modelo sin inicializar, en la figura 5.7b, tiende a obtener la misma recompensa a lo largo de los episodios, consiguiendo aumentarla puntualmente. Por otro lado, el modelo inicializado con BC de la figura 5.7a, muestra más cambios en la recompensa obtenida, lo que indica que desde un principio era capaz de cumplir con parte de la tarea. Esta diferencia indica que, a pesar de que el modelo inicializado mostraba un gran *overfitting*, el haber inicializado el modelo previamente permite al agente comenzar a cumplir con parte de la tarea, en este caso coger flechas de carcaj, obteniendo así una recompensa mayor a lo largo de los episodios.



(a) Modelo inicializado con BC. (b) Modelo sin inicializar.

Figura 5.8: Gráfica con el *loss* del modelo actor sin LSTM.

En la figura 5.8 se muestran las gráficas con el *loss* del modelo actor a lo largo de los episodios. Al observar estas gráficas se puede apreciar como el *loss* del modelo inicializado (izquierda) es algo más inestable frente al modelo sin inicializar (derecha). Esto parece deberse a que el cálculo del *loss* para el modelo actor depende de la predicciones realizadas por el modelo crítico, que parece ser más inestable para el modelo inicializado tal como sugiere la evolución del *loss* en la figura 5.9



(a) Modelo inicializado con BC.

(b) Modelo sin inicializar.

Figura 5.9: Gráfica con el *loss* del modelo crítico sin LSTM.

En la figura 5.9 se muestran las gráficas correspondientes al *loss* del modelo crítico, encargado de aproximar la función valor del entorno. Este modelo se basa en los ejemplos obtenidos por el modelo actor (el agente) durante a lo largo de los distintos episodios. Esto implica que, si el modelo actor no obtiene datos que hagan referencia a una mejora en la función de recompensa, el modelo crítico no podrá aproximar la función correctamente.

Es precisamente esta dificultad de aproximación lo que se observa en la figura 5.9, donde el modelo inicializado tiende a lograr distintas recompensas con mayor frecuencia que el modelo sin inicializar, que obtiene casi siempre la misma recompensa tal como se puede apreciar en la figura 5.7. Esta diferencia es lo que permite que el modelo inicializado tenga ejemplos más variados que le permitan aproximar mejor la función valor, lo que a su vez implica más picos en la gráfica debido a esos ejemplos más variados.

A pesar de que el modelo inicializado obtiene mejores resultados, ninguno de ellos es capaz de realizar toda la secuencia de pasos completa, desde coger la flecha hasta acertar en la diana, además de mantenerse dentro de la zona. Esto indica que, para realizar un entrenamiento con PPO desde cero, sería necesario modificar la función de recompensa por una que permita al agente acercarse más a su objetivo. Sin embargo,

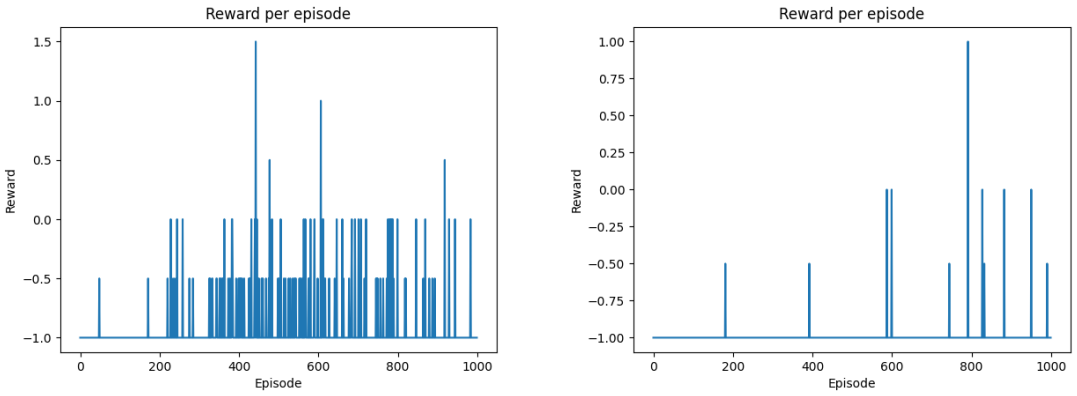
el mejor rendimiento del agente inicializado indica que, eliminando el *overfitting* presente durante los entrenamientos con BC, el agente podría obtener mejores resultados en menos tiempo.

Modelos con LSTM

A continuación, se mostrarán los resultados obtenidos a partir de los modelos con capas LSTM dentro de sus arquitecturas. Como en el caso anterior, se mostrará tanto el valor de la recompensa a lo largo de los episodios, como el *loss* de la red actor como de la red crítico.

En este punto es importante destacar una diferencia respecto de los modelos sin capas LSTM y es el hecho de que los modelos con capas LSTM tienden a tardar más en salirse de la zona de 3x3x3 marcada. Esto parece deberse a que los comandos de velocidad obtenidos de estas redes tienen una magnitud menor y parecen centrarse más en mantenerse dentro del área, en comparación con los modelos sin LSTM que tienden a salirse del área al intentar coger las flechas.

En la figura 5.10 se muestran las gráficas con la recompensa obtenida por el agente a lo largo de los episodios, tanto para el agente inicializado con BC (izquierda) como para el agente sin inicializar (derecha). Comparando ambas gráficas se puede apreciar como ocurre un fenómeno similar al visto en la figura 5.7, donde el agente inicializado obtiene una recompensa media superior al agente sin inicializar.



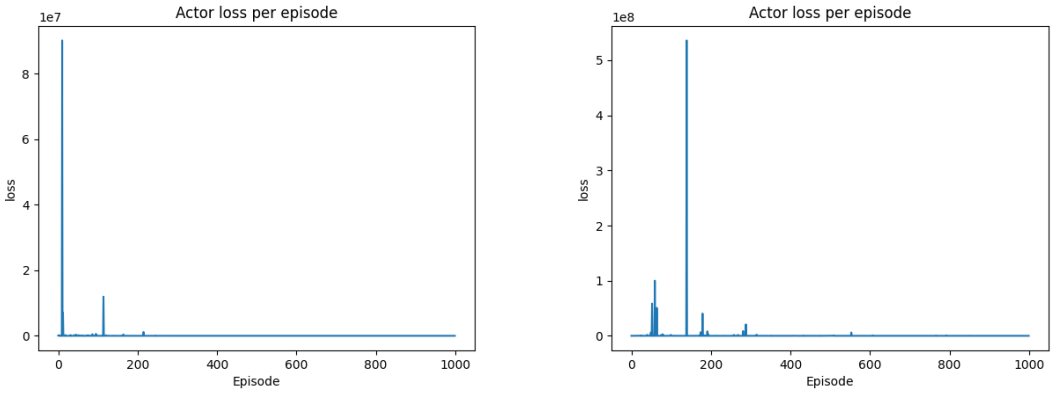
(a) Modelo inicializado con BC.

(b) Modelo sin inicializar.

Figura 5.10: Gráfica con la recompensa obtenida por el agente para los modelos con LSTM.

Al comparar la gráfica de la figura 5.10a con la de la figura 5.7a se puede apreciar una gran diferencia en la escala del eje Y. En el caso del agente sin LSTM (figura 5.7a), la recompensa máxima alcanzada es de 5 puntos, mientras que para el agente con LSTM (figura 5.10a), la recompensa máxima es de 1.5 puntos. Esto indica que el agente sin LSTM logro colocar una flecha sobre el arco, aunque no acertar con la flecha sobre la diana, mientras que el agente con LSTM no logro colocar la flecha sobre el arco, sino que saco dos flechas del carcaj sin llegar a colocar ninguna.

Otra diferencia apreciable entre el modelo con LSTM y sin LSTM es la frecuencia con la que el agente logra coger una flecha del carcaj. En este caso, la red con capas LSTM es más estable a lo largo de las épocas, ya que más frecuentemente logra coger una flecha, mientras que el modelo sin capas LSTM es menos estable, pero logra alcanzar una mejor puntuación. Esto, junto con la tendencia del modelo con LSTM a salir menos de la zona, indica que el modelo con LSTM, utilizando los mismos parámetros, es más conservador que el modelo sin LSTM.



(a) Modelo inicializado con BC. (b) Modelo sin inicializar.

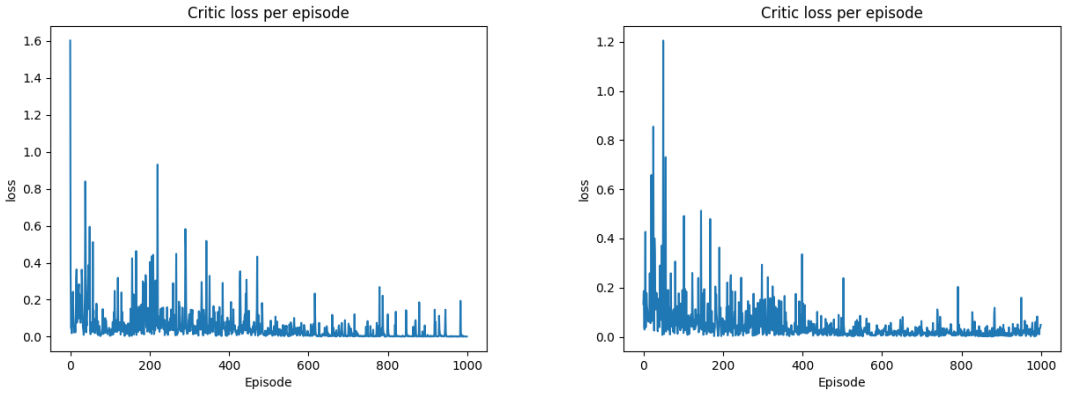
Figura 5.11: Gráfica con el *loss* del modelo actor con LSTM.

En la figura 5.11 se muestran las gráficas correspondientes al *loss* del modelo actor, tanto el inicializado con BC (a la izquierda) como el que no se ha inicializado (derecha). Ambas gráficas son bastante parecidas, teniendo de un claro *outlayer* en uno de los episodios, donde el valor aumenta enormemente. Se puede ver como este *outlayer* se encuentra al inicio del entrenamiento para el modelo inicializado, mientras que para el modelo sin inicializar se encuentra algo más adelante.

Al comparar el *loss* del modelo actor sin LSTM (5.8a) con el modelo con el *loss* del mo-

delo con LSTM (5.11a) se puede apreciar, como principal diferencia, el *outlayer* presente en el modelo con LSTM. Este *outlayer* se encuentra tanto en la red inicializada como en la red sin inicializar, por lo que parece tratarse una característica de la arquitectura con LSTM.

Por otro lado, en la figura 5.12 se muestran las gráficas con el valor del *loss* para las arquitecturas con capas LSTM. A la izquierda se encuentra la gráfica correspondiente a la red inicializada, mientras que a la derecha se encuentra la gráfica del modelo sin inicializar. Ambas gráficas son realmente parecidas, pudiendo detectar algo más de inestabilidad al principio de la gráfica de la derecha (modelo sin inicializar). Esto indica que ambas redes son capaces de aproximar la función de manera correcta, sin embargo, atendiendo a las gráficas de la figura 5.10, se puede deducir que el modelo inicializado será capaz de aproximar con mayor precisión la función valor, pues dispone de más ejemplos con distintas recompensas.



(a) Modelo inicializado con BC.

(b) Modelo sin inicializar.

Figura 5.12: Gráfica con el *loss* del modelo actor con LSTM.

Comparando los resultados de la figura 5.12a (modelo con capa LSTM), con los resultados de la figura 5.9a (modelo sin capa LSTM), se puede comprobar que el modelo con LSTM genera un entrenamiento del modelo crítico más estable. Esto puede deberse a que el modelo con LSTM es algo más conservador, tal como se explicó anteriormente, lo que implica que los ejemplos para entrenar la función valor son más parecidos entre ellos. Esta falta de ejemplos, aunque genera un entrenamiento más estable, no sería valido ya que la aproximación no será correcta.

Tal como ocurría con la arquitectura sin LSTM, los modelos basados en la arquitec-

tura con LSTM demuestran que la inicialización previa del modelo permite mejorar los resultados, logrando que el agente comience a obtener una mayor puntuación desde el principio. Estas puntuaciones iniciales permiten que el modelo crítico, encargado de aproximar la función valor, tenga una serie de ejemplos variados que le permitan obtener esta función antes, lo que se traduce en un menor tiempo de entrenamiento para el agente.

Por último, a pesar de que el modelo inicializado obtiene mejores resultados, ninguno de ellos logra cumplir una secuencia completa, ni mantenerse dentro de la zona delimitada. Las trayectorias generadas son bastante erráticas, aunque el agente tiende a acercarse al carcaj para coger una flecha. Con modelos inicializados que no presenten un *overfitting* tan fuerte es posible que el entrenamiento con PPO se pueda realizar con menos episodios y lograr que el agente complete una secuencia completa, acertando sobre la diana.

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

A continuación, se expondrán las conclusiones obtenidas tras realizar el proyecto, comenzando por la creación de los entornos y las herramientas utilizadas, hasta llegar a los resultados obtenidos.

Con este trabajo se ha mostrado una vía para entrenar agentes inteligentes capaces de realizar tareas de manipulación en entornos complejos. A pesar de no haber logrado un agente totalmente funcional, los resultados obtenidos en el capítulo 5 han demostrado la potencia de las herramientas y algoritmos utilizados. Las ventajas del uso de algoritmos de aprendizaje por imitación previo a los algoritmos de aprendizaje por refuerzo han quedado demostradas, al lograr que el comportamiento del agente comience a aproximarse a la solución con mayor velocidad. Igualmente, el uso de Unity, junto con un dispositivo de realidad virtual, permiten crear y obtener datos de un entorno complejo en poco tiempo, acelerando el proceso.

En lo referente a la creación de los entornos, el uso del motor gráfico Unity, junto con el *toolbox* Ml-Agents (Juliani y col., s.f.), ha permitido crear un entorno con alta fidelidad desde el punto de vista de la interacción de los objetos. Este realismo se complementa con una gran interfaz que simplifica su uso, pudiendo realizar entorno muy complejos y robustos de manera forma sencilla. Adicionalmente, la creciente cantidad de recursos existentes para esta plataforma ayudan a solventar los problemas de manera sencilla y lograr así un entorno robusto y con un alto grado de realismo.

El *toolbox* Ml-Agents (Juliani y col., s.f.) permite entrenar agentes utilizando el entorno

creado en Unity y algunos de los algoritmos más actuales, sin embargo, la arquitectura de los agentes tienden a ser sencillas al no poder implementar de esta forma capas convolucionales como las usadas en este proyectos. Para solventar este problema es posible exportar el entorno para poder utilizarlo en Python usando los mismos métodos que se encuentran en los *gyms* de OpenAI (Brockman y col., 2016). Estos *gyms* son los más utilizados a la hora de aplicar algoritmos de aprendizaje por refuerzo e imitación, lo que facilita el uso de los entornos creados con Unity, ya que la mayoría de los códigos de estos algoritmos están creado para usarse con este tipo de entornos.

El entorno para la toma de datos con realidad virtual también ha sido creado con Unity. La integración del dispositivo de realidad virtual con el motor gráfico es intuitiva y rápida, lo que permite adaptar el entorno fácilmente y poder tomar datos precisos. Gracias al sistema de seguimiento del dispositivo de realidad virtual, los datos obtenidos han sido precisos, tanto para la posición y rotación, como para las velocidades lineales y angulares.

Los datos se han sincronizado con la toma de las imágenes gracias a que se realizaba todo en un mismo *script*, por lo que no ha sido necesario ningún etiquetado ni proceso sincronización manual. Sin embargo, el renderizado y guardado de las imágenes durante la toma de los datos ha implicado un descenso significativo en la tasa de fotogramas por segundo (fps) del entorno de realidad virtual, por lo que fue necesario descender la frecuencia de dicha toma a 10 veces por segundo, para poder interactuar con el entorno de manera correcta. A pesar de ellos, esta frecuencia es similar a la que se muestra en el artículo (Zhang y col., 2018), por lo que es una frecuencia de datos admisible para problemas de control en tiempo real.

Tras analizar los datos de los entrenamientos realizados con el algoritmos *Behavioral Cloning* (BC), sección 5.1, se puede comprobar que todos los modelos presentan un *overfitting* muy similar, incluso el modelo basado en el artículo de Nvidia (Bojarski y col., 2016) que dispone menos parámetros que los demás. Esto parece indicar que son necesarios más datos para poder realizar el entrenamiento sin la presencia de *overfitting*, sin embargo, aumentar la cantidad de datos también aumentaría el tiempo de entrenamiento, por lo que el uso de equipos más potentes sería una necesidad.

Por otro lado, los entrenamientos realizados con el algoritmo *Proximal Policy Optimization* (PPO), sección 5.2, han mostrado resultados interesantes sobre la politica del agente. Los resultados muestran cómo, las arquitecturas que incorporan capas de tipo LSTM, tienden a generar modelos más conservadores, con velocidades de menor magni-

tud que les permiten mantenerse dentro de la zona durante más tiempo, llegando a coger varias flechas. Por otro lado, los modelos sin capas LSTM tienden a ser más agresivos, utilizando velocidades de mayor magnitud y saliendo de la zona mucho más rápido, aunque logran coger alguna flecha y colocarla en el arco, cosa que los modelos con capas LSTM no logran.

En los resultados de la sección 5.2 también muestran como los modelos inicializados con el algoritmo BC tienden a obtener recompensas más variadas que los modelos sin inicializar. Esta diferencia es especialmente notable entre los modelos con capas LSTM.

Para concluir, en este proyecto se han mostrado algunas de las tecnologías más actuales y como pueden mejorar los resultados a la hora de entrenar agentes mediante aprendizaje por refuerzo e imitación. Tanto el uso de Unity para crear un entorno como el uso de los dispositivos de realidad virtual para tomar datos han permitido simplificar el uso de estos algoritmos y poder adaptarlos a un problema concreto, lo que otorga a estas herramientas mucha potencia de cara al desarrollo de este tipo de aplicaciones. Igualmente, los resultados obtenidos han demostrado las ventajas del uso de algoritmos de aprendizaje por refuerzo e imitación juntos para mejorar y acelerar los entrenamientos para tareas de control complejas, así como las implicaciones de usar redes recurrentes para este tipo de problemas.

Todos los códigos utilizados, así como los entornos creados, se encuentran en el siguiente repositorio de GitHub (“Archery RL IL”, s.f.) bajo una licencia libre. Con esto se pretende que este trabajo pueda servir de apoyo para futuros proyectos que decidan crearse entorno a las herramientas y técnicas aquí utilizadas.

6.2. Trabajos futuros

A continuación, se nombrarán algunas de las posibles mejoras que se podrían aplicar al proyecto para mejorar el entrenamiento del agente.

Una de las primeras mejoras, ya nombradas en la sección anterior, es el uso de una mayor cantidad de datos. Con este aumento de datos se podría reducir el *overfitting* presente durante los entrenamientos con BC, pero también implicaría un mayor coste computacional, ya que con los datos utilizados los entrenamientos tardaban entre 16 y 30 horas. Este tiempo de entrenamiento se obtuvo utilizando los equipos más básicos para entrenamiento de redes neuronales del servicio de *Cloud Computing* de Azure, por lo que

utilizar equipos más potentes para acelerar el entrenamiento es una opción a considerar.

Otra posible mejora es realizar un entrenamiento previo de la parte convolucional del modelo. Con este entrenamiento previo se podría lograr que las capaz convolucionales fueran capaces de detectar los objetos del entorno antes de comenzar con el algoritmo de BC. De esta forma, durante el entrenamiento con BC se congelarían las capas convolucionales, centrando el entrenamiento en la tarea de control, no de detección de objetos. La principal desventaja de esta aproximación es la necesidad de etiquetar a mano las imágenes del agente, aunque aplicando *transfer learning* debería de ser posible entrenar esta parte de la red sin necesitar grandes cantidades de datos.

Explorar otros algoritmos, tanto para el aprendizaje por refuerzo como para el aprendizaje por imitación, podría brindar mejores resultados. Dentro de los algoritmos de aprendizaje por imitación se encuentran algoritmos como GAIL (*Generative Adversarial Imitation Learning*), que utiliza una aproximación similar a las redes generativas adversas, pero aplicado a la generación de trayectorias (Ho y Ermon, 2016).

Por otro lado, dentro de los algoritmos de aprendizaje por refuerzo encontramos múltiples opciones, tal como se expuso en la figura 2.3. Dentro de esta división, el algoritmo PPO utilizado en este proyecto se encuentra dentro de la familia de *Policy Optimization*, por lo que explorar algunos de los algoritmos de la familia *Q-Learning* podría brindar nuevos resultados y mejorar el rendimiento del agente.

Bibliografía

3DeLucas. (s.f.). Medieval Tavern Pack | 3D Furniture | Unity Asset Store. Consultado el 26 de mayo de 2021, desde https://assetstore.unity.com/packages/3d/props/furniture/medieval-tavern-pack-112546

Al-Saffar, A. A. M., Tao, H. & Talab, M. A. (2017). Review of deep convolution neural network in image classification. *Proceeding - 2017 International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications, ICRAMET 2017*, *2018-January*, 26-31. https://doi.org/10.1109/ICRAMET.2017.8253139

Alzubi, J., Nayyar, A. & Kumar, A. (2018). Machine Learning from Theory to Algorithms: An Overview. *Journal of Physics: Conference Series*, *1142*(1), 012012. https://doi.org/10.1088/1742-6596/1142/1/012012

Archery RL IL. (s.f.). Consultado el 23 de junio de 2021, desde https://github.com/fgonzalez797/Archery-RL-IL

Artificial Intelligence, Machine Learning, and Deep Learning: Same context, Different concepts. (s.f.). Consultado el 29 de junio de 2021, desde https://master-iesc-angers.com/artificial-intelligence-machine-learning-and-deep-learning-same-context-different-concepts/

Atiya, A. F. & Parlos, A. G. (2000). New results on recurrent network training: unifying the algorithms and accelerating convergence. *IEEE Transactions on Neural Networks*, *11*(3), 697-709. https://doi.org/10.1109/72.846741

Attia, A. & Dayan, S. (s.f.). *Global overview of Imitation Learning* (inf. téc.).

Azrul Anuar Zolkafi, M., Juliana Nordin, N., Abdul Rahman, H., Aien Mon Sarip, N., Islami Teng Abdullah, N. & Azmani Sahar, M. (2018). Effect of 4-Weeks Traditional Archery Intervention on Hand-Eye Coordination and Upper Limb Reaction Time Among Sedentary Youth. *The Journal of Social Sciences Research ISSN*, *6*, 1225-1230. https://doi.org/10.32861/jssr.spi6.1225.1230

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J. & Zieba, K. (2016). End to End Learning for Self-Driving Cars.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. (2016). OpenAI Gym.

Bühler, A., Gaidon, A., Cramariuc, A., Ambrus, R., Rosman, G. & Burgard, W. (s.f.). *Driving Through Ghosts: Behavioral Cloning with False Positives* (inf. téc.).

Deepanshu Mehta. (2020). State-of-the-Art Reinforcement Learning Algorithms. *International Journal of Engineering Research and*, *V8*(12). https://doi.org/10.17577/ijertv8is120332

Dehak, N., Dehak, R., Glass, J., Reynolds, D. & Kenny, P. (s.f.). *Cosine Similarity Scoring without Score Normalization Techniques* (inf. téc.).

Designer, A. (s.f.). robot emoji (Apple) - Download Free 3D model by Ambient Designer (@yokara) [f737ea8]. Consultado el 27 de mayo de 2021, desde https://sketchfab.com/3d-models/robot-emoji-apple-f737ea86d5e148df8212c60dc00efc28

DIxit, M., Tiwari, A., Pathak, H. & Astya, R. (2018). An overview of deep learning architectures, libraries and its applications areas. *Proceedings - IEEE 2018 International Conference on Advances in Computing, Communication Control and Networking, ICACCCN 2018*, 293-297. https://doi.org/10.1109/ICACCCN.2018.8748442

Dyrstad, J. S., Øye, E. R., Stahl, A. & Mathiassen, J. R. (s.f.). *Teaching a Robot to Grasp Real Fish by Imitation Learning from a Human Supervisor in Virtual Reality* (inf. téc.).

González, F. (s.f.). Keras style API for Pytorch. Consultado el 29 de mayo de 2021, desde https://github.com/fgonzalez797/api_pytorch

Hamahata, K., Taniguchi, T., Sakakibara, K., Nishikawa, I., Tabuchi, K. & Sawaragi, T. (2008). Effective integration of imitation learning and reinforcement learning by generating internal reward. *Proceedings - 8th International Conference on Intelligent Systems Design and Applications, ISDA 2008*, *3*, 121-126. https://doi.org/10.1109/ISDA.2008.325

He, K., Zhang, X., Ren, S. & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, *2016-December*, 770-778. https://doi.org/10.1109/CVPR.2016.90

Ho, J. & Ermon, S. (2016). *Generative adversarial imitation learning* (inf. téc.).

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M. & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications.

Hussein, A., Elyan, E., Gaber, M. M. & Jayne, C. (2018). Deep imitation learning for 3D navigation tasks. *Neural Computing and Applications*, *29*(7), 389-404. https://doi.org/10.1007/s00521-017-3241-z

Juliani, A., Berges, V.-P., Teng, E., Cohen, A., Harper, J., Elion, C., Goy, C., Gao, Y., Henry, H., Mattar, M. & Lange, D. (s.f.). *Unity: A General Platform for Intelligent Agents* (inf. téc.).

Julien Dossa, R. F., Lian, X., Nomoto, H., Matsubara, T. & Uehara, K. (2019). A Human-Like Agent Based on a Hybrid of Reinforcement and Imitation Learning. *Proceedings of the International Joint Conference on Neural Networks*, *2019-July*. https://doi.org/10.1109/IJCNN.2019.8852026

Kaelbling, L. P., Littman, M. L. & Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237-285. https://doi.org/10.1613/jair.301

Kamyar, S., Ghasemipour, S., Zemel, R., Gu, S. & Brain, G. (2019). *A Divergence Minimization Perspective on Imitation Learning Methods* (inf. téc.).

Kormushev, P., Calinon, S., Saegusa, R. & Metta, G. (2010). Learning the skill of archery by a humanoid robot iCub. *2010 10th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2010*, 417-423. https://doi.org/10.1109/ICHR.2010.5686841

Lau, M. M. & Lim, K. H. (2017). Investigation of activation functions in deep belief network. *2017 2nd International Conference on Control and Robotics Engineering, ICCRE 2017*, 201-206. https://doi.org/10.1109/ICCRE.2017.7935070

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J. & Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, *37*(4-5), 421-436. https://doi.org/10.1177/0278364917710318

Liu, B., Cai, Q., Yang, Z. & Wang, Z. (s.f.). *Neural Proximal/Trust Region Policy Optimization Attains Globally Optimal Policy* (inf. téc.).

Liu, Y., Gao, Y. & Yin, W. (2020). An Improved Analysis of Stochastic Gradient Descent with Momentum.

McCorduck, P. & Cfe, C. (2004). *Machines who think: A personal inquiry into the history and prospects of artificial intelligence*. CRC Press.

Nanjappan, V., Liang, H. N., Lu, F., Papangelis, K., Yue, Y. & Man, K. L. (2018). User-elicited dual-hand interactions for manipulating 3D objects in virtual reality environments. *Human-centric Computing and Information Sciences*, *8*(1), 1-16. https://doi.org/10.1186/s13673-018-0154-5

Ongsulee, P. (2018). Artificial intelligence, machine learning and deep learning. *International Conference on ICT and Knowledge Engineering*, 1-6. https://doi.org/10.1109/ICTKE.2017.8259629

Panarth. (s.f.). Newsfeed - Sketchfab. Consultado el 27 de mayo de 2021, desde https://sketchfab.com/3d-models/low-poly-medieval-weapons-free-pack-24db2937c7b947f7917674

Part 2: Kinds of RL Algorithms — Spinning Up documentation. (s.f.). Consultado el 20 de abril de 2021, desde https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, *32*. http://arxiv.org/abs/1912.01703

Pesme, S. & Flammarion, N. (s.f.). *Online Robust Regression via SGD on the ℓ 1 loss* (inf. téc.).

Sainath, T. N., Vinyals, O., Senior, A. & Sak, H. (2015). Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, *2015-August*, 4580-4584. https://doi.org/10.1109/ICASSP.2015.7178838

Schmidt, R. M. (2019). Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Openai, O. K. (s.f.). *Proximal Policy Optimization Algorithms* (inf. téc.).

Tan, H. H. & Lim, K. H. (2019). Vanishing Gradient Mitigation with Deep Learning Neural Network Optimization. *2019 7th International Conference on Smart Computing and Communications, ICSCC 2019*. https://doi.org/10.1109/ICSCC.2019.8843652

Tiro con arco. (s.f.). Consultado el 18 de abril de 2021, desde https://tokyo2020.org/es/deportes/tiro-con-arco/

Torabi, F., Warnell, G. & Stone, P. (2018). *Behavioral Cloning from Observation* (inf. téc.).

Vijaykumar Gullapalli. (1992). *REINFORCEMENT LEARNING AND ITS APPLICATION TO CONTROL* (inf. téc.).

Wang, Q. & Zhan, Z. (2011). Reinforcement learning model, algorithms and its application. *Proceedings 2011 International Conference on Mechatronic Science, Electric Engineering and Computer, MEC 2011*, 1143-1146. https://doi.org/10.1109/MEC.2011.6025669

Wang, Y., He, H. & Tan, X. (s.f.). *Truly Proximal Policy Optimization* (inf. téc.). https://github.com/

Xie, H., Lau, R. Y. K., Zhen, W., Mao, X., Li, Q. & Wang, Z. (s.f.). *Multi-class Generative Adversarial Networks with the L2 Loss Function* (inf. téc.).

Yadav, S. K., Dudhale, S. & Vats, M. M. (2012). Sports and Yogic Sciences Editor in Chief Editor. *International Journal of Physical Education*, *1*(2).

Yanchenkov, N. (s.f.). Military target | 3D Environments | Unity Asset Store. Consultado el 26 de mayo de 2021, desde https://assetstore.unity.com/packages/3d/environments/military-target-136071

Zhang, T., Mccarthy, Z., Jow, O., Lee, D., Chen, X., Goldberg, K. & Abbeel, P. (2018). *Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation* (inf. téc.).

Zhu, Y., Wang, Z., Merel, J., Rusu, A., Erez, T., Cabi, S., Tunyasuvunakool, S., Kramár, J., Hadsell, R., De Freitas, N. & Heess, N. (s.f.). *Reinforcement and Imitation Learning for Diverse Visuomotor Skills* (inf. téc.).

Apéndice A

Articulo

Integración de aprendizaje por imitación y refuerzo con realidad virtual

Fernando González Macías

Universidad Internacional de la Rioja, Logroño (España)

15 de Julio de 2021

RESUMEN

En este proyecto se pretende mostrar el uso de algunas de las ultimas tecnologías y herramientas para entrenar agentes virtuales utilizando algoritmos de aprendizaje por imitación y aprendizaje por refuerzo. Se creará un entorno utilizando el motor gráfico Unity para, posteriormente, exportarlo y poder utilizarlo en un *script* de Python para aplicar algoritmos de aprendizaje por refuerzo. También se utilizará un entorno similar para obtener datos de las trayectorias realizadas por una persona operando al agente con un equipo de realidad virtual. Estos datos se usarán en los entrenamientos de aprendizaje por imitación. Los entrenamientos del agente se realizarán en Python, tanto para el aprendizaje por imitación como para el aprendizaje por refuerzo, utilizando redes neuronales convolucionales que otorgarán visión artificial al agente. Todos los modelos se crearán y entrenarán utilizando la librería Pytorch.

PALABRAS CLAVE

Aprendizaje por imitación, Aprendizaje por refuerzo, Realidad Virtual, Inteligencia Artificial

I. INTRODUCCIÓN

En los últimos años, con el auge de la robótica, la demanda de agente inteligentes capaces de realizar tareas de manipulación complejas ha aumentado [1]. La complejidad de estas tareas ha ido en aumento, hasta el punto donde los procesos basados en planificación y análisis exhaustivo del entorno, pierden eficiencia [2]. Es por ello que, con este proyecto, se pretende mostrar como el uso de algoritmos de IA avanzados, junto con la tecnología de realidad virtual, pueden ayudar al desarrollo de agentes inteligentes para manipulación en entornos complejos.

Para lograr esta meta se hará uso de algoritmos de aprendizaje por imitación y aprendizaje por refuerzo, utilizando un entorno de manipulación complejo y se le otorgará al agente una visión artificial estereoscópica del entorno. El entorno elegido ha sido uno basado en tiro con arco, ya que se trata de un deporte que requiere de la manipulación de herramientas complejas y, según sugieren varios estudios [3] [4], mejora la coordinación mano ojo.

Se trata de un entorno realmente complejo, donde el agente deberá de aprender a reconocer objetos utilizando visión artificial y tomar decisiones de control basadas en velocidades. La idea del proyecto es dar una primera aproximación hacia este tipo de problemas complejos y, a pesar de no lograr resultados definitivos, mostrar como el uso de estas tecnologías puede ayudar a crear agentes capaces de realizar tareas de manipulación complejas.

Para loas algoritmos de IA que se utilizarán en este proyecto, se ha decidido integrar un algoritmos de aprendizaje por imitación y otro de aprendizaje por refuerzo. Para el algoritmo de aprendizaje por imitación se utilizará *Beha-*

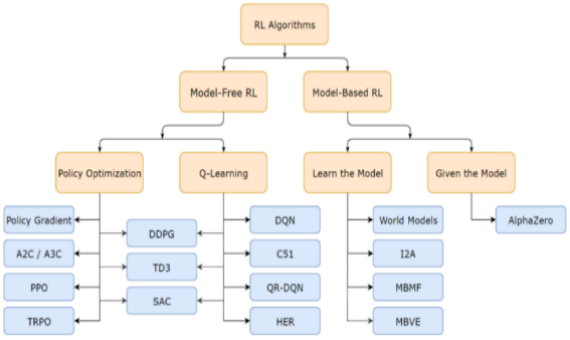


Figura 1: Esquema de división de los algoritmos de aprendizaje por refuerzo [7]

vioral Cloning (BC) [5]. Para el algoritmo de aprendizaje por refuerzo se utilizará *Proximal Policy Optimization* (PPO) [6].

II. ESTADO DEL ARTE

A. Aprendizaje por refuerzo

El aprendizaje por refuerzo es un área de la inteligencia artificial que estudia el comportamiento de un agente dentro de un entorno, buscando aprender un comportamiento óptimo para una tarea dada [7]. Este tipo de algoritmos utilizan una función, llamada función de recompensa, para medir la bondad de la política del agente ante una tarea dada [8]. Se basan en los procesos de Markov, donde el siguiente estado dependen únicamente del estado anterior y de la acción tomada [9].

En la figura 1 se muestra un esquema con algunos de los algoritmos de aprendizaje por refuerzo más utilizados [7]. En esa figura se puede apreciar donde se encuentra el algoritmo que se utilizará en este proyecto, el algoritmo *Proximal Policy Optimization* (PPO). PPO entra dentro de la familia de algoritmos llamada *Policy Optimization* y se basa en la idea de optimizar una función que otorga a cada estado un valor (*Value Function*) [10]. Al tratarse de un entorno difícil de parametrizar, la *Value Function* se sustituye por una red neuronal encargada de aproximar esta función [11].

B. Aprendizaje por imitación

Uno de los mayores problemas que enfrentan los algoritmos de aprendizaje por refuerzo es la función recompensa. Estos algoritmos deben de disponer de una función que guié al agente para que logre cumplir su objetivo. Sin embargo hay casos donde encontrar esta función de recompensa es realmente complejo, sino imposible [12]. Para salvar esta dificultad, los algoritmos de aprendizaje por imitación enseñan a un agente como cumplir una tarea especifica utilizando ejemplos provistos por un experto humano [1]. Al usar estos ejemplos se evita la necesidad de describir el conocimiento necesario para realizar cierta tarea, eliminando la necesidad de disponer de restricciones que limiten el comportamiento del agente o de una función de recompensa que lo guíe hacia la tarea que se desea cumplir [1].

De entre los algoritmos existentes, uno de los mas eficientes es el algoritmo *Behavioral Cloning* (BC). Este algoritmo busca enseñar a un agente como debe de interactuar con un entorno utilizando únicamente los ejemplos provistos por un experto [5]. Al no necesitar interacción directa con el entorno se reduce el tiempo de entrenamiento y se evita la necesidad de disponer de un entorno seguro donde el agente pueda interactuar sin riesgo [13]. Sin embargo, este algoritmos presenta varias desventajas, como la alta dependencia de los datos, que puede llevar a la presencia de *overfitting*, o a no lograr las trayectorias óptimas [14].

C. Integración de aprendizaje por refuerzo e imitación

Como se ha explicado anteriormente, tanto el aprendizaje por refuerzo como el aprendizaje por imitación presentan una serie de desventajas. Para vencer esta problemática se puede utilizar un sistema de entrenamiento que utilice algoritmos de ambas modalidades, es decir, integrar el aprendizaje por refuerzo y el aprendizaje por imitación. Esta integración es muy similar al sistema de aprendizaje utilizado por los humanos [15], donde primero se aprende a partir de un instructor que demuestra como realizar la tarea y, posteriormente, se mejora el

rendimiento optimizando el comportamiento.

D. Realidad virtual y aprendizaje por imitación

La realidad virtual es una tecnología que, en los últimos años, ha ganado bastante peso, tanto desde la perspectiva de la productividad como del entretenimiento, permitiendo a un usuario manipular objetos 3D en entornos virtuales utilizando visión estereoscópica, gracias a un casco con dos pantallas y dos lentes situadas cada una en un ojo, y a unos controles con sistemas de seguimiento [16].

También se ha utilizado para controlar robots de forma telemática, permitiendo a usuario ver el mundo a través de las cámaras del robot y controlar sus brazos [17]. Esto permite tomar control sobre el robot de forma remota, lo que implica la posibilidad de crear datos de comportamiento para posteriormente utilizar algoritmos de aprendizaje por imitación [18].

III. OBJETIVOS Y METODOLOGÍA

El objetivo principal es el de comprobar si el uso de algoritmos de aprendizaje por imitación y refuerzo, junto con tecnologías de realidad virtual, pueden ayudar a entrenar agentes capaces de adaptarse a entornos de manipulación complejos utilizando visión artificial. Para ello se utilizará un agente virtual dentro de un entorno donde se simularán las físicas de un arco y flechas. EL agente deberá de coger una flecha de un carcaj cercano, colocarla sobre el arco, tensar y disparar, acertando sobre una diana que se posicionará delante suya en una posición aleatoria.

Para lograr esta objetivo se crearán una serie de objetivos intermedios:

1. **Entorno virtual:** Este primer objetivo consistirá en crear un entorno virtual accesible a través de un dispositivo de realidad virtual que permita a un humano interactuar con el entorno y grabar las trayectorias que posteriormente el agente utilizará para entrenar.
2. **Obtención de trayectorias:** En este punto, el objetivo será generar una serie de trayectorias con las que el agente pueda entrenar. Para ello se utilizará el entorno creado en el punto anterior y, mediante un dispositivo de realidad virtual, se pondrá a una serie de personas a practicar el tiro con arco dentro de este entorno, guardando las trayectorias de sus acciones.
3. **Adaptación del entorno:** El objetivo en este punto es adaptar el entorno utilizado en el punto anterior para que un agente virtual pueda interactuar con los objetos virtuales de la misma forma en la que las personas han actuado.
4. **Entrenamiento del agente:** El objetivo en este punto será el de entrenar el agente utilizando los algoritmos mencionados en puntos anteriores.

Cada objetivo intermedio dispondrá de una serie de herramientas propias con las que se desarrollará. Aquellos objetivos relacionados con la creación del entorno virtual (objetivos 1 y 3) se desarrollarán enteramente en el motor gráfico Unity. La obtención de trayectorias (objetivo 2) se realizará con el dispositivo de realidad virtual Oculus Quest 2, conectado al ordenador mediante un cable USB tipo C llamada Oculus Link. Estas trayectorias de almacenarán en archivos CSV, guardando las imágenes correspondientes en carpetas separadas y numeradas.

Por ultimo, el entrenamiento del agente se realizará en Python, creando *scripts* diferentes para los entrenamientos con *Behavioral Cloning* y *Proximal Policy Optimization*. Los modelos de redes neuronales se crearán utilizando la librería Pytorch [19]. Para estos entrenamientos se hará uso del servicio de *Cloud Computing* de Microsoft Azure, ya que se tratan de algoritmos computacionalmente muy costosos, con tiempos de entrenamiento de 16 a 35 horas.

IV. CONTRIBUCIÓN

Con este proyecto se pretende demostrar como el uso de las tecnologías de realidad virtual,

el conjunto con algoritmos de aprendizaje por imitación y aprendizaje por refuerzo, pueden ayudar a crear agentes inteligentes capaces de realizar tareas de manipulación complejas.

Todos los códigos y entornos creados, así como los datos obtenidos, se subirán bajo una licencia gratuita MIT a GitHub y otras plataformas, con el objetivo de que sirva para futuros trabajos que pretendan profundizar más en estas tareas.

V. RESULTADOS

Los modelos utilizados se basan en 3 arquitecturas diferentes. La primera de ellas se basa en un modelo propuesto por Nvidia [20], el segundo se basa en la red MobileNet de primera generación [21] y el tercero se basa en la arquitectura ResNet18 [22].

La arquitectura propuesta por Nvidia es especifica para tareas de control, específicamente para coches autónomos [20], por lo que la adaptación de la arquitectura consistió únicamente en modificar la capa de salida para obtener los vectores de las velocidades.

Para las arquitecturas MobileNet y ResNet18, ambas se utilizan para clasificación de imagenes, por lo que para adaptarlas se introdujo una capa tipo *Spatial Soft Argmax* [17], que se encarga de obtener las coordenadas X e Y de máxima activación para cada canal de salida de la imagen. Tras esta capa se añade una capa *Flatten* y 3 capas *Fully Connected* (llamadas *Linear* en Pytorch), donde las dos primeras usarán una función de activación ELU y y la última una función de activación Tanh. Para el caso de ResNet 18, se crearán dos modelos, uno con la función de activación Tanh y otro con una función de activación lineal.

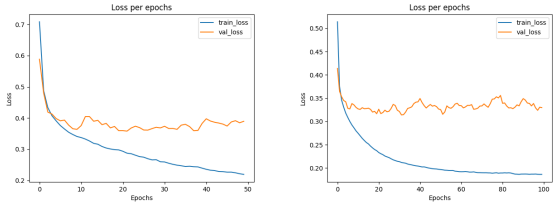
Todas las arquitecturas tendrán dos variantes, una con capas recurrentes tipo LSTM y otra sin este tipo de capas. Con esto se pretende comprobar como este tipo de capas afectan al comportamiento del agente, al disponer de una ”memoria”de lo ocurrido en pasos anteriores.

Arquitectura	Loss de test	Mejor época
Nvidia	0,401	81
Nvidia LSTM	0,362	92
MobileNet	0,387	90
MobileNet LSTM	0,341	77
MobileNet con dropout	0,392	97
MobileNet LSTM con dropout	0,347	88
ResNet18	0,392	70
ResNet18 LSTM	0,358	82
ResNet18 con dropout	0,36	22
ResNet18 LSTM con dropout	0,312	32
ResNet18 Lineal	0,279	99
ResNet18 LSTM Lineal	0,277	57

Tabla 1: Resumen con los resultados obtenidos por cada modelo.

A. *Behavioral Cloning*

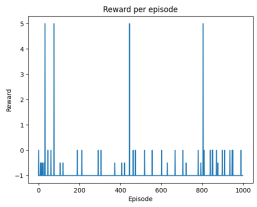
En la tabla 1 se muestran los resultados obtenidos en los entrenamientos de cada modelo. Estos resultados corresponde con el valor de *loss* obtenidos con los datos de test y la época donde se obtuvo el mejor modelo dado el *loss* de validación. El motivo de mostrar la época donde se obtuvo el mejor modelo es por el gran *overfitting* presente en los entrenamientos, tal como se puede comprobar en la figura 2.



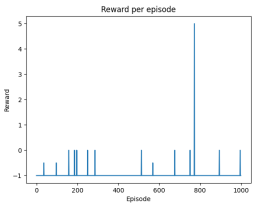
(a) Arquitectura sin LSTM. (b) Arquitectura con LSTM.

Figura 2: Gráfica con el *loss* de entrenamiento y validación por cada época para los modelos basados en la red ResNet18 con *dropout*.

Teniendo en cuenta los valores de la tabla 1, el modelo que mejores resultados da en el modelo ResNet18 con dropout. Se utilizará tanto el modelo con LSTM como el modelo sin LSTM para el entrenamiento con *Proximal Policy Optimization*. El motivo de utilizar este modelo es que, todos los entrenamientos han presentado *overfitting*, sin embargo esta arquitectura alcanzo su mejor desempeño en épocas cercanas al inicio, mientras que el resto lo alcanzaron en épocas posteriores. Esto implica que el modelo con *dropout*, además de ser uno de los que

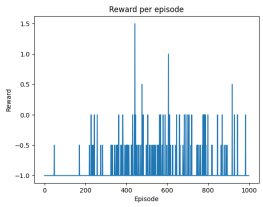


(a) Modelo inicializado con BC.

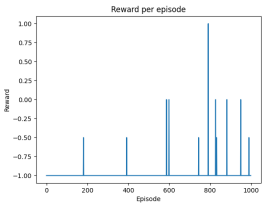


(b) Modelo sin inicializar.

Figura 3: Gráfica con la recompensa obtenida por el agente para los modelos sin LSTM.



(a) Modelo inicializado con BC.



(b) Modelo sin inicializar.

Figura 4: Gráfica con la recompensa obtenida por el agente para los modelos con LSTM.

menor *loss* tienen, también alcanzo su máximo desempeño de validación en épocas cercanas, prestando por tanto menos *overfitting* que el resto. En la figura 2 se muestran las gráficas correspondientes al entrenamiento de estos modelos.

B. *Proximal Policy Optimization*

Para los entrenamientos con *Proximal Policy Optimization* se hará uso de los modelos que mejores resultados han obtenido en el entrenamiento previo. En esta caso se tratan de los modelos ResNet18 con *dropout*, tanto la arquitectura con LSTM como la arquitectura sin LSTM.

Se harán 4 entrenamientos, dos para los modelos inicializados con BC y otros dos para los modelos sin inicializar. Con esto se pretende comprobar si inicializar los modelos, a pesar del *overfitting* encontrado, permite que los entrenamientos con PPO obtengan mejores resultados.

En la figura 3 se muestran los resultados para los modelos sin capas LSTM. A la derecha se muestra la recompensa obtenida por el modelo sin inicializar, mientras que a la izquierda se muestra la recompensa del modelo inicializado. Ambos modelos alcanzan el máximo de la recompensa en 5 puntos, lo que implica que han logrado coger una flecha y colocarla sobre el arco antes de salir de la zona de seguridad. A pesar de tener la misma recompensa máxima, el modelo inicializado logra alcanzarla antes y más veces, comparado con el modelo sin inicializar.

En la figura 4 se muestran las gráficas correspondientes a las recompensas obtenidas por los modelos con capas recurrentes tipo LSTM, tanto el modelo inicializado (izquierda) como el modelo sin inicializar (derecha). En esta caso, el modelo inicializado alcanza un máximo de recompensa de 1,5, lo que indica que es capaz de coger dos flechas, soltando una de ellas, pero no coloca ninguna sobre el arco. En el caso del modelo sin inicializar, el máximo lo alcanza en 1, lo que indica que es capaz de coger una flecha, pero no colocarla sobre el arco.

VI. ANÁLISIS DE RESULTADOS

Atendiendo a los resultados obtenidos en los entrenamientos con *Behavioral Cloning*, específicamente en la figura 2, se puede comprobar como los modelos presentan un gran *overfitting*. A pesar de aplicar técnicas para intentar reducirlo, tales como *dropout* y *batch normalization*, el *overfitting* sigue presente en todos los modelos. Esto parece indicar que la cantidad de datos tomadas no son suficientes para realizar un entrenamiento sin *overfitting*.

Por otro lado, atendiendo a los gráficos de las figuras 3 y 4, es posible comprobar como, durante el entrenamiento con *Proximal Policy Optimization*, los modelos previamente inicializados con *Behavioral Cloning* logran mejores resultados que los modelos sin inicializar. Esto lo podemos ver al comprobar la frecuencia con la que los modelos no inicializados obtienen cambios en las recompensas, que es mucho menor que los modelos inicializados. Esta

variabilidad permite entrenar mejor el modelo crítico, encargado de aproximar la función valor, ya que obtiene datos más variados, lo que se traduce en entrenamientos más rápidos.

Durante los entrenamientos con PPO, los modelos con redes recurrentes mostraban comportamientos diferentes a los modelos sin este tipo de capas. Ambos modelos con capas LSTM, tanto el modelo inicializado como el modelo sin inicializar, mostraban un comportamiento mas conservador, buscando mantenerse dentro de los limites de la zona de seguridad realizando movimientos pequeños a bajas velocidades. Por otro lado, los modelos sin capas LSTM tendían a realizar movimientos más bruscos, saliéndose antes de la zona de seguridad, pero explorando más el entorno y logrando alcanzar algunos de los objetivos más rápidamente.

VII. CONCLUSIONES

Los resultados obtenidos son realmente prometedores, tanto desde la perspectiva de las herramientas utilizadas como de los entrenamientos realizados.

En cuanto a los herramientas utilizadas, el uso del motor gráfico Unity, junto al *toolbox* Ml-Agents [23], aporta un gran valor al uso de este tipo de algoritmos, al permitir crear entornos 3D con físicas realistas donde poder entrenar a los agentes de forma segura.

En cuanto a los entrenamientos con *Behavioral Cloning*, todos ellos mostraron un gran *overfitting*, a pesar de utilizar técnicas para intentar reducirlo. Este *overfitting* parece deberse a la falta de mayores cantidades de datos durante los entrenamientos. Se hizo uso de 46 minutos de datos, lo que equivale a tiempos de entrenamiento de 16 a 35 horas, dependiendo de la arquitectura, por lo que aumentar la cantidad de datos también implicaría aumentar el tiempo de entrenamiento, o la necesidad de mejorar los equipos utilizados.

Los entrenamientos con *Proximal Policy Optimization* han mostrado algunos resultados interesantes. El primero de ellos es la diferencia presente entre los modelos inicializados y los no inicializados. A pesar de que ninguno de ellos logrará una política válida, los modelos inicializados demostraron una mayor velocidad de convergencia y unas puntuaciones mejores con la misma cantidad de episodios. Esto se logro a pesar del gran *overfitting* que presentaban los modelos inicializados con BC, lo que indica que una mejor inicialización podría llevar a un entrenamiento mucho más rápido.

Por otro lado, la diferencia en las políticas entre los modelos con capas recurrentes y los modelos sin estas capas es muy notable. Los modelos con capas recurrentes tipo LSTM muestran un comportamiento mucho más conservador, centrándose más en mantenerse dentro de la zona de seguridad y no tanto en coger las flechas. Por el contrario, los modelos sin capas LSTM tendían a realizar movimientos más bruscos, pero lograban coger y colocar algunas flechas sobre el arco, lo que les daba mayor información para la aproximación de la función valor.

A partir de este punto, es posible continuar el proyecto siguiendo diferentes lineas. Una de ellas es, tal como se ha mencionado anteriormente, aumentar la cantidad de datos para el entrenamiento del algoritmos BC, utilizando equipos con mayor potencia de computo, buscando reducir el *overfitting* presente.

Otra continuación podría darse al intentar utilizar otros algoritmos de aprendizaje por imitación, como GAIL (*Generative Adversarial Imitation Learning*) [24], así como otros algoritmos de aprendizaje por refuerzo, como los algoritmos de la familia *Q-Learning* que se pueden ver en la figura 1.

Todas estas continuaciones se pueden realizar utilizando los mismos códigos y datos usados en este proyecto, que se encuentra aquí[1] bajo una licencia libre. Con esto se pretende que este trabajo pueda servir de apoyo para futuros proyectos que decidan crearse entorno a las herramientas y técnicas aquí utilizadas o basados en algunas de las ampliaciones aquí expuestas.

[1] https://github.com/fgonzalez797/Archery-RL-IL

Referencias

[1] Ahmed Hussein, Eyad Elyan, Mohamed Medhat Gaber, and Chrisina Jayne. Deep imitation learning for 3D navigation tasks. *Neural Computing and Applications*, 29(7):389–404, apr 2018.

[2] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, apr 2018.

[3] Mohd Azrul Anuar Zolkafi, Norsham Juliana Nordin, Hayati Abdul Rahman, Noor Aien Mon Sarip, Nur Islami Teng Abdullah, and Mohd Azmani Sahar. Effect of 4-Weeks Traditional Archery Intervention on Hand-Eye Coordination and Upper Limb Reaction Time Among Sedentary Youth. *The Journal of Social Sciences Research ISSN*, 6:1225–1230, 2018.

[4] S K Yadav, Sunil Dudhale, and Mr Manish Vats. Sports and Yogic Sciences Editor in Chief Editor. *International Journal of Physical Education*, 1(2), 2012.

[5] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral Cloning from Observation. Technical report, 2018.

[6] Yuhui Wang, Hao He, and Xiaoyang Tan. Truly Proximal Policy Optimization. Technical report.

[7] Deepanshu Mehta. State-of-the-Art Reinforcement Learning Algorithms. *International Journal of Engineering Research and*, V8(12), jan 2020.

[8] Vijaykumar Gullapalli. REINFORCEMENT LEARNING AND ITS APPLICATION TO CONTROL. Technical report, 1992.

[9] Qiang Wang and Zhongli Zhan. Reinforcement learning model, algorithms and its application. In *Proceedings 2011 International Conference on Mechatronic Science, Electric Engineering and Computer, MEC 2011*, pages 1143–1146, 2011.

[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov Openai. Proximal Policy Optimization Algorithms. Technical report.

[11] Boyi Liu, Qi Cai, Zhuoran Yang, and Zhaoran Wang. Neural Proximal/Trust Region Policy Optimization Attains Globally Optimal Policy. Technical report.

[12] Seyed Kamyar, Seyed Ghasemipour, Richard Zemel, Shixiang Gu, and Google Brain. A Divergence Minimization Perspective on Imitation Learning Methods. Technical report, 2019.

[13] Andreas Bühler, Adrien Gaidon, Andrei Cramariuc, Rares Ambrus, Guy Rosman, and Wolfram Burgard. Driving Through Ghosts: Behavioral Cloning with False Positives. Technical report.

[14] Rousslan Fernand Julien Dossa, Xinyu Lian, Hirokazu Nomoto, Takashi Matsubara, and Kuniaki Uehara. A Human-Like Agent Based on a Hybrid of Reinforcement and Imitation Learning. In *Proceedings of the International Joint Conference on Neural Networks*, volume 2019-July. Institute of Electrical and Electronics Engineers Inc., jul 2019.

[15] Keita Hamahata, Tadahiro Taniguchi, Kazutoshi Sakakibara, Ikuko Nishikawa, Kazuma Tabuchi, and Tetsuo Sawaragi. Effective integration of imitation learning and reinforcement learning by generating internal reward. In *Proceedings - 8th International Conference on Intelligent Systems Design and Applications, ISDA 2008*, volume 3, pages 121–126, 2008.

[16] Vijayakumar Nanjappan, Hai Ning Liang, Feiyu Lu, Konstantinos Papangelis, Yong Yue, and Ka Lok Man. User-elicited dual-hand interactions for manipulating 3D objects in virtual reality environments.

Human-centric Computing and Information Sciences, 8(1):1–16, dec 2018.

[17] Tianhao Zhang, Zoe Mccarthy, Owen Jow, Dennis Lee, Xi Chen, Ken Goldberg, and Pieter Abbeel. Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation. Technical report, 2018.

[18] Jonatan S Dyrstad, Elling Ruud Øye, Annette Stahl, and John Reidar Mathiassen. Teaching a Robot to Grasp Real Fish by Imitation Learning from a Human Supervisor in Virtual Reality. Technical report.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems*, 32, dec 2019.

[20] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars. apr 2016.

[21] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. apr 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-December, pages 770–778. IEEE Computer Society, dec 2016.

[23] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. Technical report.

[24] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. Technical report, 2016.