Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

Trabajo Fin de Máster

Aprendizaje automático para conducción autónoma

Machine learning for autonomous driving

Autora: Beatriz Salvador Ramos

Directora: Ana Cristina Murillo Arnal

Co-director: Rubén Martínez Cantín

Máster en Ingeniería Electrónica
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Noviembre 2020

Resumen

Actualmente, se está produciendo un fuerte desarrollo de los vehículos autónomos, ya que pueden ayudar en diversos aspectos tanto cotidianos como medioambientales. La conducción autónoma se sustenta en diferentes disciplinas, entre las que se encuentra el aprendizaje automático. Concretamente el aprendizaje profundo o *deep learning*, está logrando los avances más importantes en este ámbito hoy en día. Por ello, este proyecto se centra en el estudio de algunos de los métodos más utilizados en el estado del arte y en la implementación de un sistema de aprendizaje automático para esta aplicación, tras analizar los modelos obtenidos con distintos algoritmos y variaciones de entre los estudiados.

El objetivo de este trabajo es la implementación de un sistema de *deep learning* para aprender el control de velocidad y dirección de un vehículo autónomo, utilizando un simulador de conducción autónoma. El uso de un simulador realista como el utilizado y la conexión con él durante el entrenamiento, es parte importante cuando se utilizan algoritmos de aprendizaje por refuerzo como los implementados en este trabajo por su característica de aprender de la experiencia, mediante prueba y error, convirtiéndose en esencial en una aplicación como la estudiada, ya que hacer esto en el mundo real sería inviable (costoso e inseguro). Por ello, aunque la puesta en marcha no sea sencilla, es fundamental lograr conectar estos algoritmos con los escenarios realistas del entorno de simulación. Dicho simulador, también se utilizará como plataforma para evaluar en distintos escenarios los modelos obtenidos de manera segura. La idea es implementar varias versiones del sistema, con diferentes algoritmos y modificaciones de ellos, y evaluar los resultados obtenidos con cada una de estas versiones comparando su rendimiento y generalización en distintos entornos.

En este proyecto, primero se aborda el estudio de las diferentes técnicas utilizadas en conducción autónoma y del *software* necesario para el desarrollo del sistema, así como la elección del simulador a utilizar, su instalación y aprendizaje de su manejo. En este trabajo se ha decidido utilizar el simulador realista de conducción autónoma *Airsim*, y entornos estándar de *deep learning* como *Keras* y *TensorFlow*. Otra parte importante después del estudio ha sido la puesta en marcha y el desarrollo de las distintas versiones y variaciones propuestas, que proporcionan los diversos modelos a evaluar, con los que se implementa el sistema final.

Como resultado de este trabajo, se ha desarrollado un sistema de conducción autónoma que consta de dos partes, aunando así dos técnicas de aprendizaje automático distintas, siguiendo las propuestas de la literatura existente. Por un lado, el aprendizaje por refuerzo, parte principal del sistema implementado, y por otro lado, el aprendizaje supervisado, ya que se utiliza una red neuronal convolucional (CNN) para obtener un modelo preentrenado que proporcionar a los algoritmos de *reinforcement learning* como base para no partir desde cero su entrenamiento.

Los algoritmos, resultados y entorno de simulación y evaluación de este proyecto son interesantes dentro del grupo de investigación en el que se ha realizado, ya que hay pocas soluciones completas que incluyan la interacción con el simulador realista que se utiliza en este proyecto. Los resultados obtenidos son comparables o de mejor calidad que los ejemplos encontrados disponibles públicamente, por lo cual este trabajo es un gran punto de partida para líneas de investigación que continúan en este sentido.

Índice general

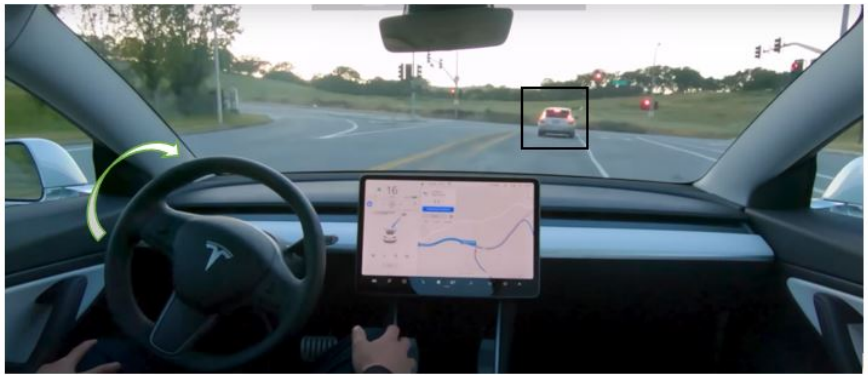
Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este proyecto se centra en el estudio de diferentes técnicas de aprendizaje automático utilizadas en conducción autónoma, explicando su fundamento y analizando los retos que enfrentan y la manera en la que estos se abordan en la literatura, y en la implementación de un sistema de *deep learning* para dicha aplicación tras la evaluación de dos técnicas diferentes. A continuación se va a exponer la motivación y el contexto que han impulsado la realización de este trabajo, así como los objetivos que se esperan obtener y las tareas llevadas a cabo en el mismo.



fuente:
`https://www.youtube.com/watch?time_continue=13&v=tlThdr3O5Qo&feature=emb_title`

Figura 1.1: Coche autónomo circulando por una carretera.

1.1. Motivación

Tanto los avances en el desarrollo de los vehículos autónomos como el creciente número de investigaciones y éxitos del aprendizaje automático, sobre todo del *deep learning* cada vez más en auge, han motivado la realización de este proyecto y se van a explicar a continuación.

Avances en conducción autónoma. Cada vez son más los avances en tecnología, comunicaciones y robótica que se producen actualmente y que tienen influencia en diversos ámbitos de nuestra vida cotidiana. En el transporte, dichos avances han dado lugar a multitud de investigaciones sobre conducción autónoma en la última década. El objetivo es el desarrollo de vehículos autónomos[3], de los que se puede ver un ejemplo en la Figura 1.1, que pueden ayudar a disminuir la contaminación, el consumo de energía y el coste de los trayectos, reducir los choques y la congestión en las carreteras, al mismo tiempo que aumenta la accesibilidad del transporte a todo el mundo (por ejemplo a personas con problemas de movilidad), siendo así de gran ayuda en distintos aspectos tanto cotidianos como medioambientales.

Aprendizaje automático y *deep learning*. La conducción autónoma se sustenta en diferentes disciplinas, como por ejemplo las comunicaciones, la ingeniería de *hardware*, de *software* o las tecnologías de *Big Data* e internet de las cosas (IoT). Entre ellas se encuentra el aprendizaje automático, que gracias al uso de varias técnicas basadas en *deep learning*, está logrando los avances más importantes en este ámbito hoy en día. El *deep learning* no solo está presente en conducción autónoma, se utiliza en muchos aspectos de la sociedad moderna, desde filtrado de búsquedas o recomendaciones en la web, identificación de objetos en imágenes, hasta en dispositivos como cámaras inteligentes y *smartphones*. Este está haciendo grandes avances en problemas muy variados como pueden ser el reconocimiento de voz, de texto o incluso en el ámbito de la medicina en temas como la genética y la predicción de enfermedades.

La gran cantidad de aplicaciones en las que se utiliza el *deep learning* y el gran número de investigaciones y rápidos avances dentro de este campo, han sido una de las principales motivaciones a la hora de elegir un trabajo relacionado con el aprendizaje automático, centrado además en una aplicación que puede ayudar en diferentes aspectos tanto de la sociedad como de la vida cotidiana y muy en auge actualmente, como lo es la conducción autónoma.

1.2. Contexto de realización del trabajo

Este proyecto se ha realizado en el grupo de investigación *Robotics, Perception and Real Time (RoPeRT)*, dentro del Instituto de Ingeniería e Investigación de Aragón (I3A). Se ha desarrollado bajo la supervisión de la directora de este grupo y la ayuda y experiencia de otros integrantes de él.

El objetivo es implementar un sistema de *deep learning* basado en visión para el control de un vehículo autónomo, obteniendo los datos de entrenamiento de un simulador. A pesar de que dentro del grupo no se había utilizado un simulador de este tipo y partía desde cero en su instalación y estudio, sí contaba con la experiencia en visión por computador y aprendizaje automático de los miembros del grupo para poder iniciar un aprendizaje básico en esas materias.

Desde un punto de vista personal, gracias a trabajar en este laboratorio he aprendido algunos de los fundamentos básicos en visión por computador y *deep learning*, campos que están muy presente en diversas aplicaciones en la actualidad, y además también he visto como se trabaja dentro de un equipo, algo muy importante de cara al futuro.

1.3. Objetivos y tareas

El objetivo principal de este proyecto es la implementación de un sistema de *deep learning* basado en visión para aprender el control de la velocidad y dirección de un vehículo autónomo, obteniendo los datos de entrenamiento de un simulador. Para ello, se propone utilizar un entorno de simulación realista de vehículos autónomos como AirSim[4], estudiar al menos una técnica de aprendizaje supervisado y otra de aprendizaje por refuerzo para la aplicación planteada, y finalmente, tras el análisis de los algoritmos más utilizados en la literatura en el ámbito de la conducción autónoma y la implementación de alguna variación, desarrollar un sistema de conducción autónoma.

Los problemas abordados son, en primer lugar, la obtención de datos de entrenamiento a partir de simulación (elección del simulador, instalación y uso del mismo) y, en segundo lugar, el estudio, implementación y evaluación de diferentes técnicas de *deep learning* para conducción autónoma (aprendiendo a utilizar un entorno de programación típico de Keras-Tensorflow con *Python* y los fundamentos de las técnicas utilizadas).

Dentro de las tareas realizadas se encuentran las siguientes:

- **Tarea 1**. Instalación y estudio del software necesario, tanto del entorno de desarrollo *Anaconda* como del simulador *Airsim* (basado en *Unreal Engine*[1]), y realización de tutoriales para la familiarización con ellos y el aprendizaje de su funcionamiento.
- **Tarea 2**. Estudio y análisis de diferentes técnicas de *deep learning* para utilizarlas en el ámbito de la conducción autónoma, viendo cuáles son las más usadas en las investigaciones actualmente.
- **Tarea 3**. Puesta en marcha y comparación de dos sistemas de aprendizaje de parámetros esenciales de la conducción autónoma, como pueden ser la velocidad o el control de la dirección del volante, utilizando entornos estándar de *deep learning como Keras y Tensorflow*. Evaluación de su rendimiento y generalización en distintos entornos.
- **Tarea 4**. Propuesta e implementación de modificaciones o adaptaciones y evaluación de estos resultados respecto a los previos.
- **Tarea 5**. Elaboración de la documentación del proyecto realizado.

Distribución temporal. La distribución temporal de cada una de las tareas llevadas a cabo durante la realización del trabajo se puede ver en la Figura 1.2 de manera aproximada. En la imagen cada fila representa una tarea y cada columna un mes. Las columnas están divididas en dos de forma que quede separado en cada mes la primera quincena de la segunda. Se observan cinco tareas principales que se corresponden con las detalladas en el párrafo anterior, con su duración aproximada.

[1] `https://www.unrealengine.com/en-US/`

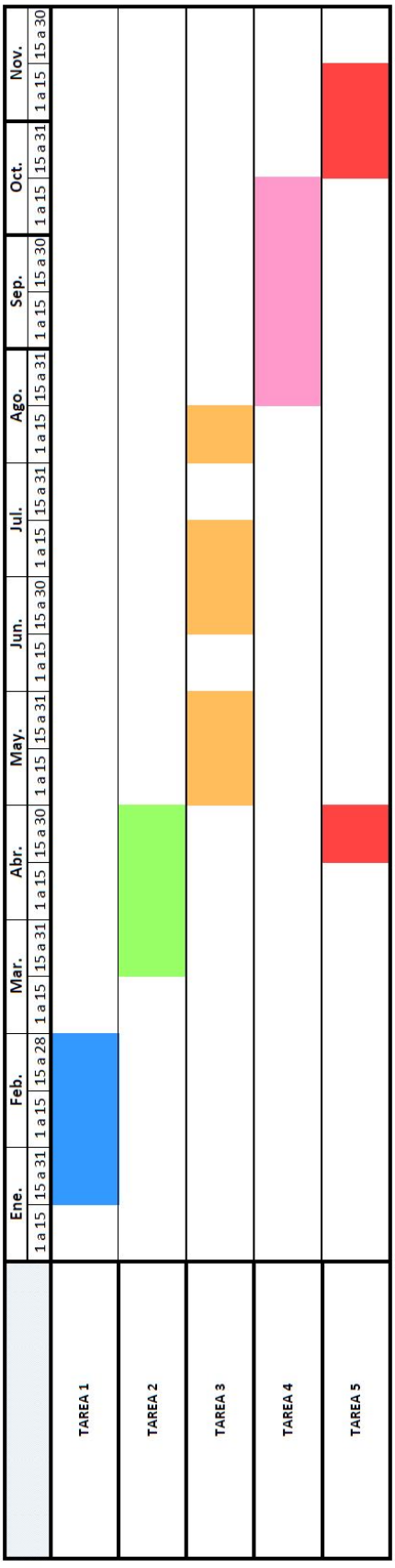


Figura 1.2: Diagrama de Gantt aproximado de la distribución de tareas llevadas a cabo en el proyecto.

1.4. Contenido de la memoria

En el capítulo 2 se explican los fundamentos del aprendizaje automático, los tres tipos que existen, las diversas aplicaciones en las que se utiliza, los algoritmos estudiados y ejemplos de estos en el ámbito de la conducción autónoma. El capítulo 3 describe los motivos por los que el uso de los simuladores de vehículos autónomos es adecuado y muy utilizado hoy en día, analizando las alternativas y centrándose en la opción elegida y en los datos utilizados y obtenidos con él. En el capítulo 4 se detallan los pasos seguidos para la implementación del sistema de *deep learning*, explicando en primer lugar la red neuronal convolucional (CNN) para clasificación y posteriormente los algoritmos de *reinforcement learning* y sus modificaciones. En el capítulo 5 se describen los experimentos llevados a cabo en este trabajo y la evaluación de los resultados obtenidos en ellos. Finalmente, el capítulo 6 recoge las conclusiones extraídas durante la realización del proyecto.

Como material adicional se incluye un anexo A en el que se detalla la configuración de los entrenamientos de todos los modelos desarrollados y se explican los resultados obtenidos en cada entrenamiento (concretamente la *loss* y las recompensas obtenidas durante el tiempo de entrenamiento).

Capítulo 2

Machine Learning en conducción autónoma

Machine learnng. El aprendizaje automático o *machine learning*[5] es una disciplina relacionada con la inteligencia artificial, que sirve para lograr que un sistema, máquina o *software*, sea capaz de aprender automáticamente respecto a una entrada dada, mediante la adaptación de ciertos algoritmos. Se utiliza en muchos aspectos de la sociedad moderna, desde filtrado de búsquedas o recomendaciones en la web, hasta en dispositivos como cámaras inteligentes y *smartphones*. Dentro del *machine learning* se distinguen tres tipos diferentes de aprendizaje, el supervisado, el no supervisado y el aprendizaje por refuerzo o *reinforcement learning*. Estos se distinguen por la información que recibe el agente o sistema durante el aprendizaje, para saber lo que es correcto o incorrecto. Los más utilizados en vehículos autónomos, el tema que atañe a este proyecto, son el aprendizaje supervisado y por refuerzo. Por esta razón, a continuación en la sección 2.1, estos dos se explicarán más en detalle.

Deep learnng. Sin embargo, dado que los métodos convencionales de *machine learning* tienen limitaciones para procesar datos en su forma original y requieren de gran experiencia para diseñar extractores de características que proporcionen una representación interna adecuada de la entrada, se utilizan métodos de *deep learning*[6] cada vez en mayor medida. El *deep learning* permite que los modelos computacionales que se componen de múltiples capas de procesamiento aprendan automáticamente, y con diversos niveles de abstracción, las representaciones necesarias para la detección o clasificación de datos sin procesar. Con la composición de suficientes transformaciones, se pueden aprender funciones muy complejas, por esto las redes neuronales profundas (con una gran cantidad de capas ocultas) encajan dentro de esta técnica.

Avances y aplicaciones. El *deep learning* está haciendo grandes avances en los últimos años en algunos de los problemas que se han resistido con las técnicas clásicas de visión por computador, tratadas en [7], y *machine learning*. Algunos ejemplos de ello son las mejoras logradas con el uso de redes neuronales convolucionales profundas en reconocimiento y detección de objetos en imágenes[8] y en reconocimiento de voz y audio[9], o los avances conseguidos

gracias a las redes recurrentes en el procesamiento de datos secuenciales como el texto[10] y sonido. Gracias al *deep learning* se han logrado buenos resultados incluso en dominios como el descubrimiento de fármacos[11], en traducción de idiomas[12] y comprensión del lenguaje natural[13], en genética[14] y predicción de enfermedades[15], así como también en el ámbito en el que se centra este artículo, la conducción autónoma[16]. Estos avances cada vez se dan más rápido debido a que el *deep learning* requiere muy poca ingeniería manual, por lo que puede aprovechar los aumentos en la cantidad de cómputo y datos disponibles, lo que lo convierte en un foco de investigación muy importante actualmente.

2.1. Tipos de Machine Learning

Como se ha dicho anteriormente, el *machine learning* se divide en tres tipos de aprendizaje automático: el supervisado, el no supervisado y el *reinforcement learning*.

En ámbito de la conducción autónoma, la mayor parte de investigaciones actualmente estudian técnicas basadas en *deep learning*, como se ve en [17], tanto de aprendizaje supervisado como de *reinforcement learning*, por ello en esta sección se van a desarrollar más estos dos.

2.1.1. Aprendizaje supervisado

La forma más común de *machine learning* es el aprendizaje supervisado. En este caso, el sistema aprende de un conjunto proporcionado de datos etiquetados, es decir, de datos que tienen ejemplos de entrenamiento que enseñan cual es la entrada, con su correspondiente etiqueta que se corresponde con su valor de salida deseado.

Dentro de él, los algoritmos de *deep learning* más utilizados para aplicaciones de conducción autónoma son las redes neuronales convolucionales (CNN) y en menor medida las redes neuronales recurrentes (RNN). Por esta razón, en la sección 2.2, se van a explicar los fundamentos de cada uno de ellos.

Además de estos algoritmos, existe una forma de aprendizaje supervisado, que es utilizada en conducción autónoma, denominada aprendizaje por imitación o *Imitation learning* (IL)[18]. La idea básica que sustenta esta técnica es la del entrenamiento a través de la imitación del comportamiento de un experto. Con esta idea de aprendizaje, se distinguen dos categorías: la clonación de comportamiento y el aprendizaje por refuerzo inverso. Debido a que la clonación de comportamiento de extremo a extremo ha atraído interés dentro del ámbito de la conducción autónoma recientemente, es la categoría que se va a analizar. En esta forma de aprendizaje el sistema necesita recibir datos de entrenamiento con imágenes u observaciones de la entrada asociadas con acciones del demostrador o experto. De esta manera la red neuronal profunda, utilizada como clasificador o regresor, aprende a reconocer patrones con los que asocia la entrada a los parámetros de control que permitan replicar la acción del experto.

La clonación de comportamiento es una de las técnicas más ampliamente utilizadas en conducción autónoma, como se ve en el ejemplo de [19], junto con algoritmos de *deep reinforcement learning*, de los que se estudiará un ejemplo en el siguiente apartado 2.2. El aprendizaje por imitación tiene ventajas como la capacidad de imitar al demostrador sin necesidad de interactuar con el entorno,

a diferencia del aprendizaje por refuerzo que se basa en la prueba y error, lo cual es algo a tener en cuenta en el entrenamiento de vehículos autónomos. Aunque también tiene ciertos inconvenientes o limitaciones como por ejemplo la dificultad para generalizar lo aprendido a nuevas tareas o escenarios como se analiza en [1].

2.1.2. Aprendizaje no supervisado

En el aprendizaje no supervisado, sin embargo, se proporcionan al sistema datos de entrenamiento sin etiquetar, es decir, sin el valor deseado de salida, por lo que habitualmente se utilizan técnicas como el *clustering* o agrupamiento, como se ve en el ejemplo «*Deep clustering for unsupervised learning of visual features*»[20], para encontrar patrones en los datos.

2.1.3. Aprendizaje por refuerzo

Entre los dos tipos de aprendizaje anteriores, se encuentra el aprendizaje por refuerzo o *reinforcement learning*, en el que el agente aprende patrones gracias a la experiencia, interaccionando con el entorno, realizando acciones y viendo si obtiene un resultado positivo o negativo. En este tercer tipo, hay un enfoque de prueba y error, ya que lo que controla el aprendizaje no son los datos etiquetados introducidos, sino una función de recompensa que le indica si las acciones llevadas a cabo son correctas o no, de manera que el agente adapte su estrategia en función de esta recompensa.

Este método ha sido ampliamente utilizado en tareas de control durante mucho tiempo, y actualmente, la mezcla de *reinforcement learning*, del que hay una extensa descripción en [21], con *deep learning* es uno de los enfoques más prometedores en tareas como la conducción autónoma, como se expone en «*Deep reinforcement learning framework for autonomous driving*»[22]. Esta combinación ha dado lugar a una técnica llamada *deep reinforcement learning* o *deep Q learning* , dentro de la que existen diferentes algoritmos entre los que se encuentran las *deep Q networks* que se van a analizar en el apartado 2.2.

A pesar de ser uno de los tipos de *machine learning* más utilizados en conducción autónoma, tiene ciertos inconvenientes como los problemas de reproducibilidad que son habituales en esta técnica, estudiados en [23]. Esto se debe a su necesidad de explorar el escenario realizando distintas acciones para obtener la recompensa correspondiente, ya que el entorno objetivo puede variar cada vez y un modelado explícito de cada escenario posible no es una solución realista. Por último y en relación también a esto surge otro problema, en el *deep reinforcement learning* se requieren millones de pruebas y errores en el entorno objetivo que, en el ámbito de la conducción autónoma, son imposibles de ensayar en el mundo real ya que por coste e inseguridad es algo inviable. Además existen limitaciones al pasar de los resultados de los entrenamientos *off line* a las pruebas en un sistema real , que aparecen en todos los métodos de *deep learning* en el ámbito del desarrollo de vehículos autónomos.

2.2. Algoritmos estudiados

En este apartado se van a explicar tres de los algoritmos comúnmente utilizados en el ámbito de la conducción autónoma mencionados en la sección anterior y se va a poner un ejemplo de su uso en dicha aplicación.

2.2.1. Redes neuronales convolucionales (CNN)

Las redes neuronales convolucionales (CNN)[24] son un tipo de red neuronal artificial que se utiliza principalmente para el procesamiento de imágenes y que se clasifica dentro del aprendizaje supervisado. Se pueden ver como extractores de características y aproximadores de funciones no lineales muy complejas, que logran identificar patrones y objetos en los datos de entrada a dicha red. Para ello, son necesarias múltiples capas ocultas de procesamiento que aprenden con distintos niveles de abstracción, ya que se van especializando cada vez más. En cada capa se filtra la imagen con máscaras que la recorren y se obtiene una salida o mapa de características que pasa a la siguiente capa, hasta lograr reconocer o clasificar esa entrada. Las CNN están parametrizadas por un conjunto de pesos, $\mathbf{W}$, y por unos valores de sesgo o *bias*, $\mathbf{b}$. Su objetivo es encontrar durante el entrenamiento los valores de dichos parámetros que hacen que el error sea mínimo, gracias al algoritmo de *backpropagation* y al gradiente descendente, capturando así las características más discriminantes de la imagen. Las redes neuronales profundas explotan la propiedad de que muchas señales naturales son jerarquías de composición, es decir, que las características de nivel superior se obtienen componiendo las de nivel inferior. En las imágenes, las combinaciones locales de bordes forman motivos, los motivos se ensamblan en partes y las partes forman objetos reconocibles. Ocurre algo similar en el habla y el texto, desde sonidos, fonemas, sílabas, palabras hasta oraciones. La agrupación permite que las representaciones varíen muy poco cuando los elementos de la capa anterior varían en posición y apariencia lo que permite generalizar, por ejemplo a la hora de detectar un coche lo hace de igual manera independientemente de su color, tamaño, forma o posición.

En el ámbito de la conducción autónoma, se utiliza habitualmente tomando como entrada imágenes de carreteras vistas desde un vehículo y como etiqueta la acción que debería realizar el automóvil si estuviera en ese entorno (por ejemplo, el ángulo de posición del volante), de lo que se ve un ejemplo en [25]. Además las CNN pueden identificar tanto la carretera como obstáculos en ella, por ejemplo otros coches o peatones. También es común verlo como extractor de características que después se utilizará como base para realizar un entrenamiento con métodos de *reinforcement learning*, haciendo así que este segundo se realice más rápido. Esta última aplicación de las CNN es la que se va a utilizar en el sistema implementado del que se hablará en el capítulo 4.

2.2.2. Redes neuronales recurrentes (RNN)

Las redes neuronales recurrentes (RNN)[26] son el tipo de red neuronal artificial que, de entre las técnicas de *deep learning*, obtiene los mejores resultados en el procesamiento de datos que tienen una secuencia temporal como el habla o el texto, por lo que se utiliza típicamente para aplicaciones como la traducción automática. También se clasifican dentro del aprendizaje supervisado pero

a diferencia de otros tipos de redes, estas tienen un ciclo de realimentación, que se puede desplegar para generar una arquitectura de red sin bucle para su mejor comprensión (esto no se hace en la práctica), como se observa en la Fig. 2.1, viendo así que comparte los mismos pesos aprendidos en cada capa. Estas redes procesan un elemento de la secuencia de entrada cada vez, manteniendo en sus unidades ocultas un "vector de estado"que contiene implícitamente información sobre la historia de todos los elementos de la secuencia pasados. Las RNN tradicionales no son adecuadas para datos secuenciales muy largos, ya que si la red es muy profunda su gradiente de salida tiene dificultades para propagarse hacia las capas anteriores. Esto se soluciona en las redes de memoria a corto y largo plazo, LSTM, gracias a la incorporación de tres puertas que controlan el estado de entrada, de salida y de memoria.

En el desarrollo de vehículos autónomos se aprovecha el buen rendimiento de las RNN, más concretamente de la arquitectura LSTM, para aprender la dinámica temporal en series de datos secuenciales. Se utiliza, por ejemplo, en sistemas de predicción de trayectorias de automóviles o peatones eficientes, como se propone en «*Probabilistic vehicle trajectory prediction over occupancy grid map via recurrent neural network*»[27], ya que en conducción autónoma se debe garantizar un alto grado de seguridad incluso en entornos inciertos y dinámicamente cambiantes.

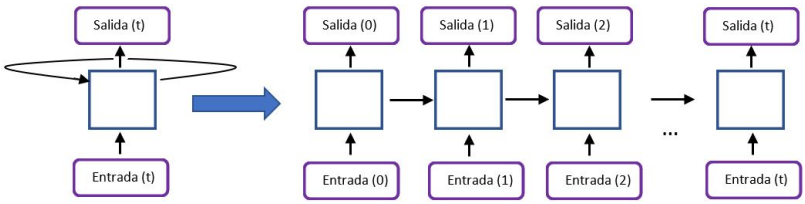


Figura 2.1: Red neuronal recurrente con su bucle (izquierda) y esa misma red desplegada (derecha).

2.2.3. *Deep Q networks* (DQN)

El *deep reinforcement learning* es una técnica que aprovecha los buenos resultados del *deep learning* y de redes neuronales profundas como las CNN, sobre todo a la hora de extraer características para clasificar y detectar objetos en imágenes, y lo junta con el enfoque de prueba y error del *reinforcement learning*. Dentro de ella, uno de los algoritmos más utilizados son las *deep Q networks* (DQN)[28], que se pueden observar en la Fig. 2.2, en las que la salida de la red no son clases, son valores Q dados a las acciones que el agente ha aprendido a llevar a cabo en función del estado del entorno que ha identificado a la entrada. El objetivo es encontrar los pesos óptimos de la red que hagan que se maximice el valor de Q en cada par estado-acción que haya recibido una recompensa positiva del entorno.

Esta técnica ya ha tenido importantes éxitos como por ejemplo en los juegos clásicos de Atari 2600, como se ve en los resultados de «*Playing atari with deep reinforcement learning*»[29], donde se demostró que en una DQN el *rein-*

forcemnet learning es el responsable de la parte de planificación de la acción, mientras que el *deep learning* es el responsable de la parte de representación del entorno. También se ha probado, en [28], que con este tipo de algoritmos se puede lograr un nivel de control humano e incluso mejor. En el ámbito de la conducción autónoma hay numerosas investigaciones que estudian está técnica, como «*Autonomous driving system based on deep Q learnig*»[30], en las que se están obteniendo buenos resultados. Se espera lograr avances prometedores gracias al *deep reinforcement learning*, que se está convirtiendo en el método más utilizado en este campo, por ello es en el que más se va a profundizar en este proyecto.

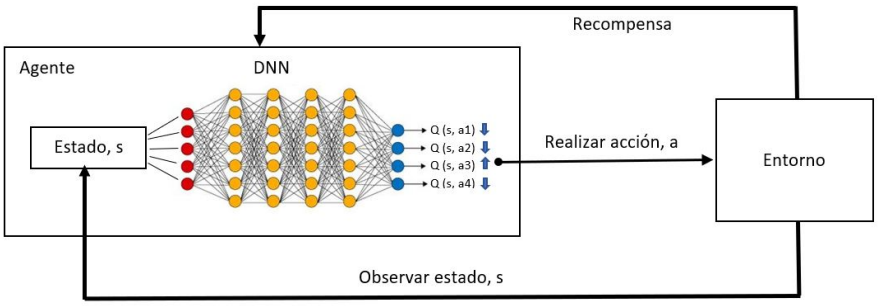


Figura 2.2: Arquitectura de las *deep Q networks* utilizadas en *deep reinforcement learning*.

Capítulo 3

Simulación para *autonomous driving*

En este capítulo se van a exponer los motivos por los que es adecuado utilizar un simulador de conducción autónoma en esta aplicación, algunos de los simuladores existentes más utilizados, el simulador elegido para la realización de este proyecto y los datos utilizados y obtenidos de él.

3.1. Simulador de conducción autónoma

En conducción autónoma se dan ciertos retos y limitaciones, algunos de ellos relacionados con el uso de técnicas de *deep learning* y otros con la propia aplicación, para los que puede resultar útil el uso de simuladores.

Retos abordados en conducción autónoma. Por un lado, en cuanto a los retos ligados a la conducción autónoma, la mayoría radican en su carácter de aplicación crítica para la seguridad y en las limitaciones que hay para pasar de los modelos entrenados *off line* a la implementación en un vehículo autónomo real. Estos han de tener la capacidad de lidiar con situaciones impredecibles y tomar la mejor decisión y la más segura para todos los implicados, incluso en escenarios complejos de conducción y con muchos elementos dinámicos. Para ello, actualmente se tiende al *reinforcement learning* de extremo a extremo, en el que se analizan experiencias reales de conducción considerando el proceso completo, sin enfocarse en los detalles de los componentes individuales del escenario ya que para el objetivo final no se necesita conocer cada parte del escenario o de otro automóvil, únicamente la posición de la carretera y de los obstáculos relevantes en ella, tanto peatones como otros vehículos. Se centra solo en las mejores decisiones para cada situación, por lo que es la mejor alternativa para esta aplicación como ya se ha dicho anteriormente. Sin embargo, el uso esta técnica implica la necesidad de realizar millones de pruebas y errores en el entorno objetivo, incluyendo casos extremos y fuera de lo normal que, en esta aplicación debido a su naturaleza, son imposibles de ensayar y explorar con un vehículo en el mundo real, como ya se ha mencionado en la sección 2.1, por lo que el uso de un simulador puede resultar de gran ayuda.

Limitaciones ligadas al uso de técnicas basadas en *deep learning*. Por otro lado, el uso de técnicas basadas en *deep learning* conlleva la necesidad de poseer una gran cantidad de datos, o bien etiquetados o bien obtenidos mediante la exploración del entorno, con los que poder entrenar el sistema. Esta es su mayor limitación, ya que a pesar de que el creciente desarrollo de las tecnologías IoT y *Big Data* supone un gran avance en el reto de la adquisición de los datos y en el almacenamiento y procesamiento de *petabytes* de ellos, respectivamente, en algunas aplicaciones siguen sin ser suficientes para lograr unos buenos resultados. Además existen dos enfoques a la hora de implementar el sistema: el tradicional con distintos bloques secuenciales para cada proceso, en los que se puede utilizar *deep learning* o técnicas clásicas de visión de forma independiente en cada etapa, y el enfoque *end to end* o de extremo a extremo, ambos mostrados en la Fig. 3.1. Actualmente se tiende a las arquitecturas *end to end* en diversas aplicaciones, entre las cuales está la conducción autónoma, ya que puede reemplazar las múltiples etapas del enfoque tradicional, que en ocasiones resuelve problemas por separado más complejos que el del objetivo final, por una sola red neuronal profunda, lo cual puede simplificar el sistema extrayendo solo la información relevante. Esta arquitectura requiere una cantidad aún mayor de datos para lograr un funcionamiento óptimo y con mejores resultados que en el enfoque modular. Por ello, su uso agrava el problema de la necesidad de una gran suma de datos, necesarios para que no se produzcan problemas de generalización a la hora de reconocer nuevos objetos, situaciones o escenarios, lo cual es grave en aplicaciones críticas para la seguridad como lo es la conducción autónoma. Este problema se acrecenta en entornos de conducción complejos, con tráfico denso y muchos agentes dinámicos.

Ventajas del uso de un simulador. Por todo ello, hoy en día en vez de analizar experiencias reales de conducción, se tiende cada vez más a extraer datos de simuladores de vehículos autónomos para el entrenamiento sobre todo de sistemas de *deep reinforcement learning* de extremo a extremo, como es el caso de [31]. Con el uso de estos simuladores se consiguen cantidades casi ilimitadas de datos, que pueden ser tanto de escenarios comunes en la conducción autónoma

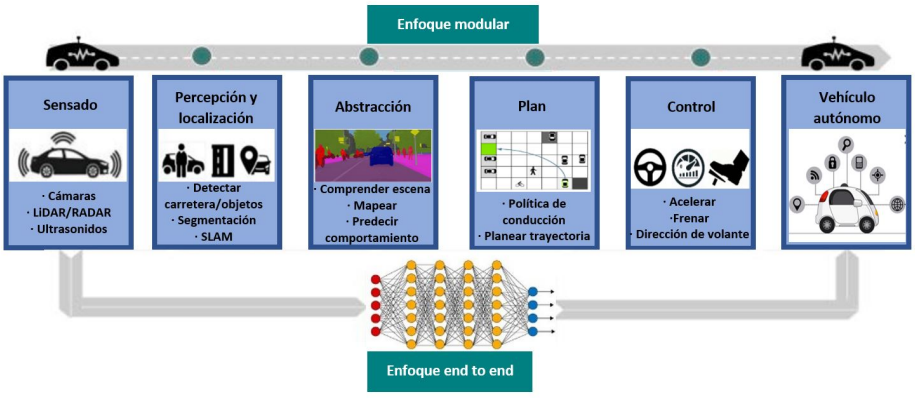


Figura 3.1: Sistema de conducción autónoma con un enfoque modular frente a otro *end to end* o de extremo a extremo.

como de situaciones extremas de una manera totalmente segura, atajando así algunos de los problemas relativos a esta aplicación así como de limitaciones ligadas al uso de técnicas de *deep learning* mencionados anteriormente. El uso de simuladores permite obtener la gran cantidad de datos requerida, siendo una fuente adecuada para algoritmos de *deep learning* de extremo a extremo, que permiten que una red neuronal profunda asigne la entrada sin formato (en bruto) a la salida directa, como los utilizados en conducción autónoma y en este proyecto. La posibilidad de obtener una gran suma de datos mejora problemas de reproducibilidad y disminuye en buena medida los problemas de generalización al llevar a cabo comportamientos de conducción más realistas y desafiantes durante el entrenamiento que los que se podrían realizar con un coche en el mundo real. También a la hora de realizar las pruebas y errores necesarios en *reinforcement learning* de manera segura, que son inviables en un entorno objetivo real, es útil el uso de simuladores así como lo es para validar los modelos entrenados.

3.1.1. Simuladores existentes

Respecto a dichos simuladores de conducción autónoma, hay de dos tipos, unos que ofrecen un aspecto similar al de un videojuego, más virtual e irreal, y otros que tienen un aspecto mucho más realista, más similar a los escenarios que se puede encontrar un vehículo autónomo en el mundo real.

Simuladores irreales. Ejemplo de los primeros son Javascript racer utilizado en [31] o TORCS (*The open racing car simulator*)[32], que es más conocido y utilizado en multitud de investigaciones como se puede ver en «*Deepdriving: Learning affordance for direct perception in autonomous driving*»[16], «*Deep reinforcement learning framework for autonomous driving*»[22] o «*End-to-End Autonomous Driving Decision Based on Deep Reinforcement Learning*»[33]. Este último está diseñado como un juego de carreras pero se utiliza comúnmente como plataforma de investigación.

Simuladores realistas. En cuanto a los segundos, son simuladores desarrollados más recientemente con el propósito claro de producir avances en el ámbito de la conducción autónoma, ya que probar algoritmos en el mundo real es caro, inseguro y requiere mucho tiempo. Los más utilizados actualmente son CARLA[34], que es de código abierto para apoyar el desarrollo, entrenamiento y validación de sistemas autónomos de conducción en entornos urbanos, y AirSim[4], que está construido sobre *Unreal Engine* y ofrece simulaciones realistas tanto física como visualmente en variedad de condiciones y entornos y con posibilidad de adaptarse a otro tipo de vehículos o robots, como por ejemplo drones. Se pueden ver resultados y validaciones de modelos que aprovechan el potencial de CARLA en «*Behavioral cloning from observation*»[18], «*Exploring the limitations of behavior cloning for autonomous driving*»[1] o «*Controllable imitative reinforcement learning for vision-based self-driving*»[35], mientras que en «*Federated transfer reinforcement learning for autonomous driving*»[36] o «*Autonomous driving via deep reinforcement learning*»[37] exploran con el reciente AirSim. El aspecto de los escenarios en los cuatro simuladores mencionados se puede ver en la Fig. 3.2.

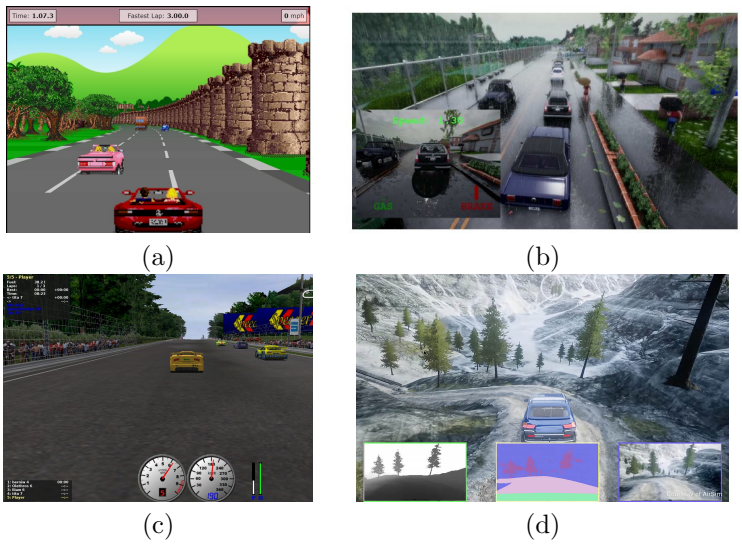


Figura 3.2: Simuladores de conducción autónoma: (a) Javascript racer, (b) CARLA (fuente: [1]), (c) TORCS y (d) AirSim.

A pesar de las ventajas que ofrecen estos simuladores, con ellos aparece el problema de adaptar los resultados obtenidos en simulación al mundo real, aunque utilizando los más realistas este problema mejora en cierta medida. Otros estudios como « *Virtual to real reinforcement learning for autonomous driving*»[2], analizan la posibilidad de utilizar un simulador más sencillo (como TORCS) cuyas imágenes de escenarios pasen, gracias a una red neuronal profunda, de virtuales e irreales a imágenes sintéticas con escenas realistas manteniendo la misma estructura del entorno. Con estas últimas se entrena posteriormente el sistema utilizando *reinforcement learning*, como se observa en la Fig. 3.3.

Sin embargo, aún con todos los avances que se están produciendo y con el uso de simuladores, la ampliación al espectro completo de los comportamientos dados en la conducción sigue siendo un problema sin resolver. Al igual que lo es el paso de los modelos entrenados en entornos virtuales con las técnicas de *deep learning* aprendidas a sistemas y vehículos en el mundo real, aunque trabajos como [38] y [39] han mostrado resultados alentadores para el futuro respecto a esto.

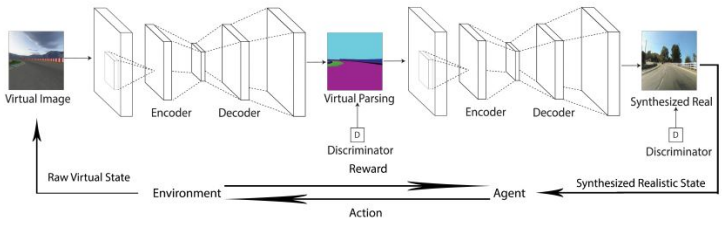


Figura 3.3: Paso de imágenes virtuales del simulador TORCS a imágenes sintéticas realistas para entrenar un sistema de conducción autónoma utilizando *reinforcement learning*, propuesto en [2].

3.1.2. Airsim

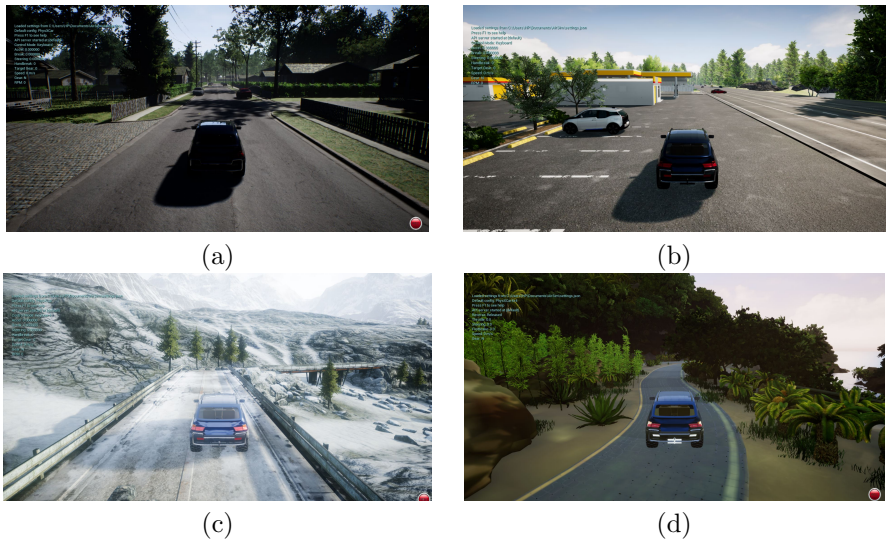
Para este proyecto, se ha decidido utilizar el simulador de conducción autónoma en entornos realistas *Airsim*[4], dados los buenos resultados de este tipo de simuladores en otras investigaciones. Este es uno de los desarrollados más recientemente y se ha elegido por la gran variedad de escenarios y condiciones realistas que posee. Otras razones para su elección son, la existencia de una amplia documentación sobre él debido a que se ha utilizado en múltiples trabajos, también de versiones más reducidas del simulador que permiten entrenar modelos de manera más sencilla y con menos requerimientos *hardware* para su uso y, además, ya se poseía experiencia en el manejo del motor gráfico *Unreal Engine* en el que está basado.

Airsim es una plataforma de código abierto que permite recolectar un gran número de datos de entrenamiento etiquetados en distintos entornos. Ofrece simulaciones realistas y está diseñado para ser extensible a nuevos tipos de vehículos, plataformas *hardware* y protocolos de *software*. Este simulador tiene como objetivo reducir la brecha entre la simulación y la realidad para ayudar al desarrollo de vehículos autónomos, ya que desarrollar y probar estos algoritmos en el mundo real es un proceso largo, caro y a menudo inseguro, siendo necesario transferir el aprendizaje que se lleva a cabo en simulación a la realidad. Por ello, es importante desarrollar modelos precisos del entorno y de la dinámica del sistema, con la mayor riqueza y diversidad posible, para que el comportamiento simulado imite de cerca el mundo real. Esto es logrado en gran medida por *Airsim*, que ofrece simulaciones de alta fidelidad, además de que sus componentes principales incluidos el motor de física, los modelos de vehículos, los modelos de entorno y los modelos de sensores, están diseñados para ser utilizados independientemente con dependencias mínimas fuera de *Airsim*.

Extracción de datos AirSim posee APIs (interfaz de programación de aplicaciones), descritas en su documentación[1], para que se pueda interactuar con el vehículo en la simulación desde un entorno de programación. Con ellas se pueden obtener imágenes del simulador, validar la conexión con él, obtener el estado del vehículo (parámetros como su velocidad, posición dentro del escenario o información sobre colisiones con elementos del entorno), controlar el vehículo (tanto su velocidad como el ángulo del volante para realizar giros), incluso se pueden activar efectos de tiempo meteorológico (como lluvia), etc. Gracias a esto, se puede lograr una gran cantidad de datos etiquetados y muy variados, ya que además de existir varios escenarios, es posible cambiar las condiciones de estos (como el tiempo o la iluminación), algo que en el mundo real podría llevar días o incluso meses. De este modo con *Airsim* podemos lograr, en primer lugar obtener imágenes del simulador etiquetadas para formar un *dataset*. En segundo lugar utilizar los escenarios para las pruebas y errores durante el entrenamiento de algoritmos de *reinforcement learning* que solo se pueden entrenar en simulación. En tercer lugar, se puede realizar clonación de comportamiento guardando una trayectoria realizada por el usuario a replicar por el agente, extrayendo del simulador las posiciones objetivo y los parámetros de conducción que realiza el vehículo para realizar la trayectoria. Por último, también proporciona un banco de pruebas seguro para validar y evaluar modelos ya entrenados.

[1] `https://microsoft.github.io/AirSim/`

Escenarios de *Airsim*. Se ha utilizado un paquete compilado del entorno de simulación *Airsim*, independiente y más reducido que el simulador completo, del repositorio de *GitHub* «*AutonomousDrivingCookbook*»[2]. Este contiene cuatro escenarios distintos con características muy diferentes: *neighborhood* en el que se puede conducir por las calles de un vecindario, *city* en el que se ve una carretera de ciudad por la que circulan otros vehículos y hay más elementos dinámicos, *landscape* donde aparece una carretera de montaña con diversas curvas y un camino nevado y escarpado y, por último, *coastline* en el cual se observa una carretera con unos alrededores llenos de vegetación y el mar. Se puede ver una imagen de cada uno de los entornos descritos en la Figura 3.4. Se ha decidido utilizar este paquete reducido debido a que con él se requieren menos requisitos *hardware* y menos espacio y memoria. Además, el tiempo de preparación previo a la utilización del simulador es menor, ya que no es necesario compilar cada escenario por separado, basta con ejecutar el paquete en la *PowerShell* de *Windows* y darle como parámetro el nombre del escenario que se desee abrir, siendo así más sencillo su uso para la familiarización con él pero manteniendo a la vez una buena variedad de entornos y condiciones de conducción.



(a) (b) (c) (d)

Figura 3.4: Escenarios existentes en el paquete de *Airsim* utilizado: (a) *Neighborhood*, (b) *city*, (c) *landscape* y (d) *coastline*.

Dataset. Además de esto para la red neuronal convolucional que se ha entrenado, para posteriormente utilizar ese modelo como *encoder* (modelo que haya extraído y codificado las características del entorno de conducción de manera previa a otro entrenamiento) o base para el algoritmo de *reinforcement learning* y así reducir el tiempo de entrenamiento necesario en este último, se ha utilizado un *dataset* del mismo repositorio de GitHub[3] mencionado en el párrafo anterior.

[2] `https://github.com/microsoft/AutonomousDrivingCookbook/tree/master/DistributedRL`

[3] `https://github.com/microsoft/AutonomousDrivingCookbook/tree/master/AirSimE2EDeepLearning`

Las imágenes de este *dataset* pertenecen al escenario *landscape* de *Airsim* y en ellas se observa el entorno con una única cámara frontal para la entrada visual, como se ve en el ejemplo de la Figura 3.5.

El *dataset* se divide en dos partes, por un lado las imágenes y por otro ficheros .tsv en los que para cada imagen contiene la etiqueta del ángulo de dirección del volante adecuado en ese momento. Además las imágenes se dividen en dos estrategias de conducción, la normal y la de desvíos, en las primeras el ángulo etiquetado es pequeño (cercano a cero) para que el coche vaya recto por la carretera, mientras que en las segundas los ángulos son bruscos, llevando al coche lado a lado de la carretera para que pueda corregirse si se está saliendo o evitar un obstáculo en la carretera, proporcionándole así suficientes ejemplos de las distintas situaciones que se pueden dar en la conducción para el entrenamiento. Todos estos datos se tratan, en primer lugar tomando solo la parte de interés de cada imagen, la mitad de abajo en la que se encuentra la carretera, reduciendo así el tiempo de entrenamiento y evitando que el modelo se enfoque en características irrelevantes del entorno. Posteriormente se unen todos los datos de los ficheros .tsv en un solo marco de datos, combinándolos en archivos .h5, adecuados para grandes conjuntos de datos. Este conjunto final, tiene cuatro partes: una matriz con los datos de la imagen, una matriz con el estado anterior del coche que contiene la dirección del volante, el freno, el acelerador y la velocidad, una matriz con las etiquetas correspondientes al ángulo de dirección normalizado en el rango de -1 a 1 y una matriz con datos sobre los archivos. El marco final de datos se divide un 70 % para *train*, un 20 % para *eval* y un 10 % para *test*.

(a) (b)

Figura 3.5: Ejemplo de dos imágenes del *dataset* utilizado para entrenar la CNN.

Capítulo 4

Sistema Implementado

En este capítulo se detallan los pasos seguidos para llevar a cabo la implementación del sistema de conducción autónoma. Explicando las partes y el esquema general del sistema desarrollado, desde la red neuronal convolucional utilizada para obtener el modelo que servirá de *encoder* para dicho sistema, hasta el algoritmo de *reinforcement learning* que se va a utilizar como base y las variaciones implementadas en él para dar lugar al sistema final.

4.1. Resumen general y partes del sistema

En este trabajo se plantea un sistema de conducción autónoma que aúna dos técnicas de aprendizaje automático distintas, como se ve en la Figura 4.1, por un lado el aprendizaje supervisado ya que se utiliza una red neuronal convolucional y por otro lado el aprendizaje por refuerzo, la parte principal del sistema desarrollado. Por lo tanto, dicho sistema consta de dos partes diferenciadas. En primer lugar se entrena una red convolucional con los datos del escenario *landscape* de *Airsim*, descritos en el apartado *dataset* de la subsección 3.1.2. Esta red aprende únicamente un parámetro de conducción, el ángulo del volante adecuado para la situación de cada imagen, y el modelo obtenido con ella servirá como *encoder* para que el algoritmo de *reinforcement learning* parta desde ahí. Su estructura y funcionamiento se va a explicar en más detalle en la sección 4.2. En segundo lugar, se utiliza un algoritmo de aprendizaje por refuerzo, que es el que realiza la mayor parte del trabajo a la hora de aprender la tarea de conducción, ya que como se ha estudiado en capítulos anteriores, es la técnica más usada y que mejores resultados está obteniendo para esta aplicación. Se analizan, implementan y evalúan distintos algoritmos y variaciones típicos en este tipo de aprendizaje, explicados en la sección 4.3. En este trabajo, los algoritmos de *reinforcement learning* se entrenan en su mayoría en un escenario diferente al del *encoder*, en este caso en *neighborhood*, realizando el coche acciones y pruebas *online* en el simulador durante el entrenamiento, y ajustando a este nuevo entorno los pesos del modelo proporcionado por la CNN, aunque también se entrena uno de los algoritmos en el escenario *landscape* para así analizar su funcionamiento en distintos entornos. Estos algoritmos, no aprenden únicamente el ángulo de giro del volante como la CNN, aprenden también a controlar la velocidad (control de acelerador y freno del vehículo), una conducción más completa.

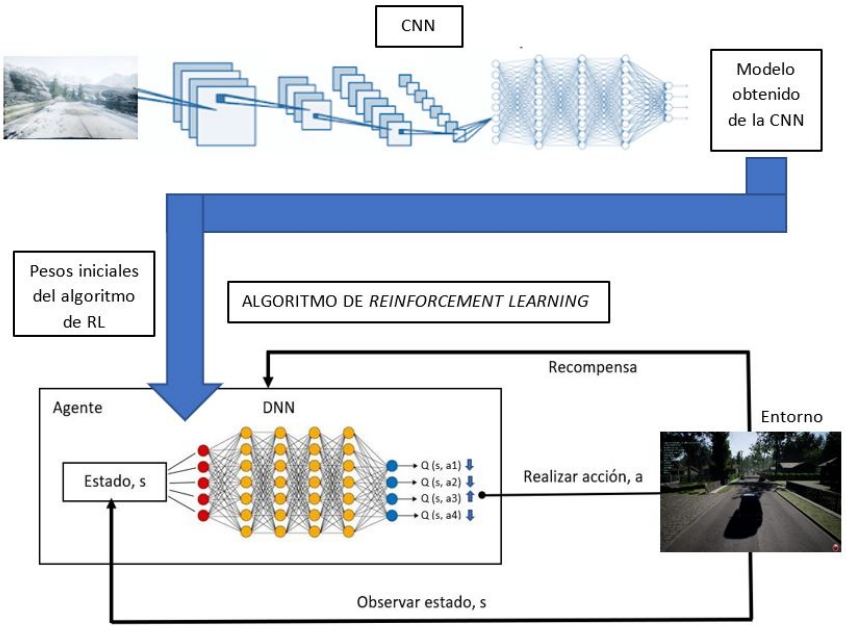


Figura 4.1: Esquema general del sistema implementado.

4.2. Algoritmo de aprendizaje supervisado implementado

Como se ha descrito en la sección anterior, en el sistema implementado se utiliza como base para los posteriores algoritmos de aprendizaje por refuerzo un modelo proporcionado por una red neuronal convolucional que predice el ángulo del volante dada una imagen. Dicha red es la que se va a explicar a continuación.

Red neuronal convolucional (CNN). Se utiliza una red neuronal convolucional tomada de un repositorio de *GitHub* de *Microsoft*, desarrollador de *Airsim*, llamado «*AutonomousDrivingCookbook*». El algoritmo «*TrainModel*» utilizado se encuentra en la carpeta «*AirSimE2EDeepLearning*»[1]. Está implementada utilizando *Keras* y *TensorFlow*, marcos de desarrollo típicos en *deep learning*. Esta red usa los datos de *train* y *eval* en ficheros .h5 separados, ya tratados como se ha descrito en el apartado *dataset* de la subsección 3.1.2, para resolver un problema de regresión y devolviendo así el ángulo adecuado de giro del volante. La arquitectura de esta red consta de la capa de entrada, a la cual se le proporciona la imagen de entrada con su tamaño esperado, diversas capas ocultas convolucionales en las que su entrada es la salida de la capa anterior y finalmente se añaden varias capas *fully connected*, sin embargo, en la primera de ellas la entrada ya no es únicamente la salida de las capas convolucionales, sino que además se le concatena el último estado conocido del coche.

[1] `https://github.com/microsoft/AutonomousDrivingCookbook/blob/master/AirSimE2EDeepLearning/TrainModel.ipynb`

A continuación se va a explicar dicha arquitectura, que se puede ver en la Figura 4.2, en más detalle. En primer lugar, se cuenta con tres capas convolucionales (*Conv2D*), con 16, 32 y 32 filtros respectivamente de con un tamaño de 3x3 cada uno de ellos, las cuales utilizan la función de activación ReLu (típica en CNNs), y el relleno *same* en el argumento *padding* para solucionar el problema de efecto de borde. Tras cada una de las capas de convolución, se encuentra una capa *MaxPooling2D* (también hay tres capas de este tipo) que agrupan un área de 2x2 píxeles (filtro 2x2) y lo sustituyen por el valor máximo dentro de ella, para así reducir el tamaño de los mapas de características que salen tras las convoluciones a la mitad. Después de esto hay un *dropout*, que lo que hace es desactivar diferentes neuronas de la red para así que se reduzca el *overfitting* o sobreajuste durante el entrenamiento. En la red utilizada el parámetro de esta capa, que indica el porcentaje de neuronas desactivadas en cada iteración, era de 0.2 pero se decidió cambiarlo a 0.5 ya que seguía habiendo algo de sobreajuste. Ahora se pasa a añadir las capas *fully connected*, tres en total, con un *dropout* de 0.5 como el descrito anteriormente entre ellas (en total dos *dropout* más). Pero como ya se ha dicho anteriormente, como entrada a la primera de ellas se concatena la salida tras el *dropout* que hay después de las convoluciones con el estado del coche. Son capas *dense*, las dos primeras con 64 y 10 neuronas respectivamente, mientras que la última, la capa de salida, tiene una única neurona que es la que predice el valor numérico del ángulo del volante. Finalmente, a la hora de compilar el modelo se utiliza el optimizador *Nadam* con un *learning rate* diferente al dado en el repositorio (en vez de 0.0001 se utiliza uno un poco más grande, de 0.0005) y para la loss se utiliza la métrica *mse* (media de los errores cuadráticos). El modelo obtenido se guarda para utilizarlo posteriormente y partir desde él.

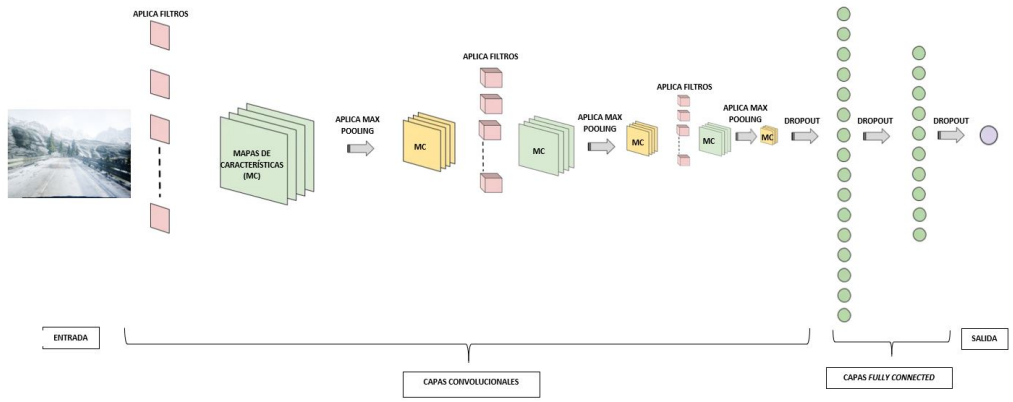


Figura 4.2: Arquitectura de la CNN utilizada.

4.3. Algoritmos de *deep reinforcement learning* implementados

Como parte fundamental del sistema implementado, se tiene el algoritmo de *reinforcement learning*. Se parte desde el algoritmo de una *deep Q network* (DQN) que utiliza *Airsim* del mismo repositorio de GitHub mencionado en la sección anterior, dentro de la carpeta *DistributedRL*[2]. Se utilizan como base los códigos encontrados en dicho repositorio de *distributed_agent.py*, *rl_model.py* y *airsim_client.py* y los datos de *road_lines.txt* y *reward_points.txt*, en los cuales se encuentran puntos aleatorios del escenario en los que iniciar la simulación y las posiciones de una trayectoria correcta dentro del escenario *neighborhood* que sirven para calcular la recompensa obtenida en cada episodio respectivamente. Primero se realiza un entrenamiento sin realizar ninguna modificación en este algoritmo que se explica en el siguiente párrafo y posteriormente, sobre él se implementan distintas variaciones que se detallan más adelante.

Deep Q network (DQN). Para entrenar se lanza el algoritmo *distributed_agent.py*, en el que se importa *airsim_client.py* para posibilitar la conexión y extracción de datos del simulador durante el entrenamiento y *rl_model.py* en el cual se encuentra la DQN.

Este código en *distributed_agent.py*, como su nombre indica, estaba pensado para entrenarse de manera distribuida en *Azure* mientras que en este trabajo se va a hacer de manera local, por lo que el primer cambio realizado es quitar las partes del código que se utilizan para el entrenamiento distribuido, dejando únicamente la parte local. En este *script*, lo primero que se hace es inicializar la clase agente con varios parámetros proporcionados como argumentos por la terminal como por ejemplo el *batch size*, el tamaño del *buffer* de memoria para guardar diferentes acciones en cada episodio, la épsilon mínima para el algoritmo, el número de iteraciones tras el cual se guarda un *checkpoint* del modelo en ese momento o distintos *paths*, entre otros. A continuación, se crea un objeto modelo de la clase *RLModel* cargando en él los pesos de la CNN explicada en la sección 4.2 y se realiza la conexión con *Airsim* para explorar el entorno y comenzar el aprendizaje. En el simulador se realizan varias acciones, hasta completar el *buffer* de memoria, y con los datos obtenidos del simulador al completar el episodio se actualizan los pesos del modelo entrenando el algoritmo DQN. Dichas acciones al principio son todas aleatorias, ya que el parámetro épsilon tiene valor 1, pero conforme van realizándose iteraciones este valor va bajando lentamente hasta llegar al mínimo establecido, en este caso de 0.1, donde el 10 % de acciones son aleatorias para que el vehículo pueda seguir explorando, encontrando acciones que generen mejores recompensas que las aprendidas mejorando así la conducción y evitando que el agente se atasque durante el aprendizaje y pueda seguir mejorando a lo largo del entrenamiento. Esto es un método denominado *epsilon-greedy*, en el cual se puede ir disminuyendo épsilon de distintas maneras, pero en este caso está implementado de forma que el valor de épsilon se obtiene restando en cada *epoch* un 0.003 al valor actual de épsilon hasta llegar al mínimo donde se queda constante.

[2] `https://github.com/microsoft/AutonomousDrivingCookbook/tree/master/DistributedRL/Share/scripts_downpour/app`

Para que la red aprenda es muy importante un cálculo adecuado de la recompensa o *reward* de cada episodio. En este caso, se calcula obteniendo del simulador la posición del coche en cada momento y hallando la distancia de esta a los segmentos formados al tomar las coordenadas de un punto y el siguiente de la trayectoria guardada en el archivo *reward_points.txt*. A dicha distancia se la multiplica después por una tasa a la cual decae la recompensa y se hace la exponencial, obteniendo así el valor de la recompensa correspondiente a la posición. En la recompensa, además de la posición del coche, influyen la velocidad y las colisiones, asignándoles en este caso una recompensa de cero tanto si hay un choque como si el vehículo está parado (velocidad menor que 2). En la primera versión del sistema implementado esto no se ha modificado, se realizará en variaciones posteriores, lo que sí se ha hecho es añadir un pequeño código para guardar los valores de la recompensa acumulada en cada episodio durante el entrenamiento, para posteriormente poder representarla.

En cuanto a la arquitectura de la red utilizada y el algoritmo DQN propiamente dicho, se encuentran en el *script rl_model.py* que se va a detallar a continuación. El algoritmo *deep Q network* combina el algoritmo *Q-learning* con las redes neuronales profundas (DNN), las cuales utiliza para aproximar la función Q. Se utilizan dos redes neuronales para estabilizar el proceso de aprendizaje. La primera, la red neuronal principal (en este caso *action_model*), está representada por los parámetros θ y se utiliza para estimar los valores Q del estado, s, y acción, a, actuales. La segunda, la red neuronal objetivo (en este caso *target_model*), parametrizada por θ', tiene la misma arquitectura que la red principal pero se usa para aproximar los valores Q del estado siguiente s' y la siguiente acción a'. El aprendizaje ocurre en la red principal, ya que la red objetivo se congela (sus parámetros no se cambian) durante varias iteraciones, y después los parámetros de la red principal se copian a la red objetivo, transmitiendo así el aprendizaje de una a otra, haciendo que las estimaciones calculadas por la red objetivo sean más precisas. La arquitectura de dichas redes, es muy similar a la utilizada en la CNN descrita en la sección anterior. La parte convolucional consta de igual modo de tres capas *Conv2D*, seguidas cada una de ellas de una capa de *MaxPooling* y tras todo ello un *dropout*. La parte de capas *fully connected* difiere algo más, teniendo en este caso únicamente dos capas *Dense*, entre las cuales aparece un *dropout*. La primera tiene 128 neuronas mientras que la segunda, que se corresponde con la capa de salida, consta de 5 neuronas, una por cada acción posible, siendo cada acción un ángulo de giro de volante (ángulos normalizados entre -1 y 1: -1, -0.5, 0, 0.5 y 1). A la hora de compilar el modelo se utiliza el optimizador Adam con un *learning rate* de 0.001 y para el argumento *loss* la métrica de la media de los errores cuadráticos.

Una vez se conoce la arquitectura de las redes utilizadas se pasa a implementar el cálculo de los *targets* en las DQN, ecuación 4.1. Para ello, se realiza una predicción con el *target_model* de los valores Q correspondientes a todas las acciones posibles de llevar a cabo en el estado siguiente y se toma el máximo valor de Q obtenido tras dicha predicción. A esto se le multiplica γ, un factor de descuento entre cero y uno que determina la importancia de las recompensas futuras (si es cercano a cero solo tendrá en cuenta las recompensas actuales mientras que si es cercano intentará que la recompensa sea alta a más largo plazo) y finalmente se le suman las recompensas. El valor obtenido de esta operación debe ser igual al valor de Q en el estado actual para las acciones elegidas, por lo tanto se actualizan los valores Q en el *action_model*.

$$Q(s, a; \theta) = r + \gamma max_{a'} Q(s', a'; \theta') \tag{4.1}$$

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}[((r + \gamma max_{a'} Q(s', a'; \theta')) - Q(s, a; \theta))^2] \tag{4.2}$$

Tras esto, se implementa la ecuación de Bellman con la que se calcula la función de coste o *loss*, ecuación 4.2. En ella, se realiza el cuadrado de la diferencia entre ambos lados del igual de la ecuación 4.1. Finalmente, en este código se ha añadido a la hora de realizar el *fit* del modelo un *callback* de *Tensorboard* que permite la visualización de la *loss* durante el tiempo de entrenamiento.

Double deep Q network (DDQN). Se implementa también un algoritmo distinto de *reinforcement learning*, utilizando ahora una *double deep Q network* (DDQN) en lugar de una DQN simple. Una DDQN[40] es un algoritmo típico en *deep reinforcement learning* y el siguiente paso tras una DQN para intentar obtener un modelo mejor. Se decide implementar para comparar los resultados con los obtenidos anteriormente en la aplicación de conducción autónoma, ya que se ha visto que es una variación muy utilizada en otras investigaciones. Se usa para reducir las sobreestimaciones que se producen en una DQN, ya que el algoritmo piensa que la recompensa que va a obtener con los valores Q que aprende será mayor de lo que realmente obtendrá. La solución propuesta en las DDQN consiste en separar la selección de la acción de su evaluación, primero decidiendo cuál es la mejor acción de todas las posibles con la red principal (*action_model*) y después evaluando dicha acción en la red objetivo (*target_model*) para conocer su valor Q, tal y como se observa en la ecuación 4.3.

En este algoritmo no se cambia ni el cálculo de la recompensa, ni la arquitectura de las redes, únicamente se sustituye la parte del código en *rl_model.py* que se encarga del cálculo de la ecuación 4.1 por la ecuación 4.3 como se ha explicado.

$$Q(s, a; \theta) = r + \gamma Q(s', argmax_{a'} Q(s', a'; \theta); \theta') \tag{4.3}$$

4.4. Modificaciones implementadas en la DQN

Cambios en la *reward*. Una vez estudiado y analizado el algoritmo DQN de base desde el que se parte en este proyecto, se procede a realizar algunas variaciones en él con el fin de comparar los resultados de varios modelos e intentar encontrar una implementación que logre mejores resultados que dicho algoritmo base.

Como primeras modificaciones se han propuesto dos variaciones en el cálculo de la recompensa:

- **Variación de la recompensa, v1 (colisiones).** En primer lugar se decide dar una recompensa negativa si se produce una colisión con algún elemento del entorno, en este caso de -10, en lugar de dar una recompensa de cero. Esto se debe a que tanto un choque como que el vehículo estuviera parado tenían, en el algoritmo de base, la misma recompensa de cero asignada, siendo una colisión un escenario mucho peor en la conducción de un vehículo, una situación que se debe evitar a toda costa.

- **Variación de la recompensa, v2 (colisiones + acelerones).** Al cambio realizado en la modificación anterior se decide añadirle además una recompensa en función de la aceleración. Esta consiste en darle -5 cuando el vehículo tiene un valor de aceleración muy alto, con el objetivo de que el coche vaya a una velocidad más o menos constante y sin dar acelerones. De este modo, el coche obtiene una recompensa negativa con la realización de dos acciones, los acelerones bruscos y las colisiones, siempre asignándole la menor recompensa al peor caso que se debe evitar, en conducción a los choques.

Entrenamiento en distinto escenario. Con el algoritmo de DQN que mejores resultados se obtienen, en este caso con la implementación de la variación de la recompensa v2 como se va a analizar en capítulo 5, se decide realizar un entrenamiento en otro de los escenarios de *Airsim*. Se considera interesante, ver que resultados se obtienen si se utiliza el mismo escenario que se usa para entrenar el *encoder* (CNN), por ello se elige el escenario *landscape*.

Sobre el código implementado en la variación de la recompensa v2, en el *script distributed_agent.py*, se precisan las siguientes modificaciones. En primer lugar, a la hora de realizar la conexión con *Airsim* se debe cambiar el argumento que selecciona el escenario de *neighborhood* a *landscape*. Después es necesario obtener unos nuevos ficheros de *reward_points.txt* y *road_lines.txt* para este escenario, ya que los proporcionados por el repositorio de GitHub utilizados en el resto de algoritmos se corresponden con los puntos por los que pasa el coche en una trayectoria correcta y coordenadas del escenario *neighborhood*, respectivamente. Para la obtención de dichos archivos, la parte más importante para esta modificación, se ha escrito un pequeño programa llamado *guarda_posicion.py*.

En este programa lo primero que se hace es la conexión con *Airsim*, pero en lugar de darle el control del vehículo a los pesos de los modelos entrenados como en la evaluación, es el usuario el que mediante las flechas del teclado puede mover el coche por el escenario, creando así la trayectoria de referencia deseada en dicho entorno con la que se calculará la recompensa. Después de conectarse con el simulador se crea un bucle en el que se obtiene información de *Airsim*, en concreto las coordenadas x e y del coche cada dos segundos. Con ellas, tomando un punto y el siguiente se crea la variable segmento y cada segmento es escrito en una línea del nuevo archivo creado llamado *reward_points_landscape.txt*, para lograr un fichero de texto con el mismo formato que el dado para *neighborhood*.

Las puntos de *road_lines.txt*, sin embargo, están en un sistema de coordenadas diferente, el de *Unreal Engine*. Para pasar del punto de inicio en este escenario en coordenadas del vehículo a coordenadas de *Unreal* hay que sumarle al primero un *offset* que está definido en el código *distributed_agent.py* como *car_start_coords*. Además es necesario un pequeño ajuste más, en el escenario *neighborhood* se comenzaba la simulación en cada iteración en una coordenada de *road_lines.txt* diferente con un ángulo de inicio distinto (o de frente o girado 180 grados de manera aleatoria, ya que es un vecindario en el que ambos lados de la carretera tienen salida). Sin embargo, el nuevo escenario es un camino muy estrecho, con vallas a los lados y solo se puede avanzar, ya que detrás tiene una pared de rocas, por lo que no tendría sentido iniciar la simulación con un ángulo aleatorio, por ello se deja el ángulo adecuado fijo para que el vehículo comience enderezado en la carretera.

Capítulo 5

Experimentación y evaluación

En este capítulo se va a detallar el *set up* utilizado para todos los experimentos realizados y la configuración concreta con los parámetros determinados de cada uno de ellos. Posteriormente se van a evaluar los resultados de los modelos obtenidos, analizando y discutiendo su funcionamiento y el comportamiento del vehículo dentro de distintos escenarios del simulador *Airsim*.

5.1. Configuración de los experimentos

Set up. Para realizar los experimentos ha sido necesario utilizar un ordenador potente del laboratorio en el que se trabaja cuyas características son: un procesador *Intel Core i7-6700* CPU @3.40GHz x 8, una memoria RAM de 32 GB y una GPU GeForce GTX 1070/PCIe/SSE2.

Los códigos del repositorio de GitHub[1] explicados en el capítulo 4 y utilizados como base para emprender este trabajo estaban desactualizados, por ello en el entorno virtual creado en *Anaconda* se han instalado unas versiones, tanto de *Python* como de las librerías esenciales, bastante anteriores a la última que existe. Esto dificultó y alargó la puesta en marcha más de lo previsto inicialmente porque no se sabía cuales eran las versiones exactas de todas las librerías utilizadas, lo que dio problemas de compatibilidad en su instalación. Concretamente, se utiliza la versión 3.6.12 de *Python*, la versión 2.3.1 de *Keras* y la versión 1.14.0 de *TensorFlow* para poner en marcha los programas de base y que funcione también el paquete reducido del simulador utilizado, que al igual que los códigos tampoco usa la última versión de *Airsim*.

Los datos y escenarios utilizados en todos los algoritmos ya se han explicado tanto en el apartado *dataset* como en el de escenarios de *Airsim* del capítulo 3 respectivamente.

Dado que la velocidad del vehículo es constante, como métrica de calidad del modelo obtenido se va a medir el tiempo que aguanta el coche en la carretera sin chocar con nada, viendo como evoluciona este tiempo en distintos puntos del entrenamiento.

[1] `https://github.com/microsoft/AutonomousDrivingCookbook`

En particular se calcula como la media de n ejecuciones para calcular el valor esperado con respecto a las variaciones ambientales y perturbaciones:

$$t_{driving} = \sum_{i=1}^{n}(t_i)/n \tag{5.1}$$

donde t_i es el tiempo (en segundos) que el vehículo logra *conducir* de manera autónoma, es decir, permanecer en la carretera sin colisionar ni salirse de ella y en este caso $n = 10$.

Además también se analiza su comportamiento en diferentes escenarios de manera cualitativa durante la ejecución de la evaluación. Todos los modelos se han entrenado aproximadamente durante 48 horas y los detalles de los entrenamientos de cada uno de ellos, como la *reward* o recompensa acumulada por episodio, la *loss* durante el entrenamiento y los hiperparámetros elegidos, se pueden ver en detalle en el Anexo A.

Modelos utilizados para los experimentos. Se han entrenado los siguientes modelos con los algoritmos y variaciones detallados en el capítulo 4 para realizar los experimentos de la sección 5.2:

1. **Base DQN con *Reward_base***. DQN versión 0: Se lanza el algoritmo de base del repositorio *AutonomousDrivingCookbook*[2] sin cambios.

2. **Base DQN con *Reward_v1***. DQN versión 1: Se utiliza el algoritmo DQN con la variación de la recompensa v1 detallada en 4.3, dando más penalización a las colisiones en el cálculo de la *reward*.

3. **Base DQN con *Reward_v2***. DQN versión 2: Se utiliza el algoritmo DQN con la variación de la recompensa v2 detallada en 4.3, donde a versión 1 se le añade una recompensa también negativa a las aceleraciones bruscas.

4. ***Double* DQN con *Reward_base***. Cambio del algoritmo DQN por el DDQN, utilizando el resto del código, incluido el cálculo de la *reward*, como en el experimento DQN versión 0.

5. **Base DQN con *Reward_v2* entrenado en otro escenario**. Mismo código que en el experimento 3 pero utilizando el escenario *landscape* durante el entrenamiento en lugar del *neighborhood*.

5.2. Experimentos realizados y evaluación de los resultados

5.2.1. Análisis de las distintas variaciones implementadas

Objetivo. En el primer experimento se pretende evaluar, utilizando el simulador, el comportamiento de los distintos algoritmos implementados, para determinar cuál de los modelos obtenidos logra mejores resultados y tiene una mejor actuación en el mismo escenario en el que se ha entrenado (*neighborhood*).

[2] `https://github.com/microsoft/AutonomousDrivingCookbook/tree/master/DistributedRL/Share/scripts_downpour/app`

Descripción del experimento. En este experimento, en primer lugar se conecta con el escenario *neighborhood* para evaluar los modelos desarrollados en el mismo entorno que el utilizado para su entrenamiento. Al haber guardado varios *checkpoints* del modelo en diferentes puntos de su entrenamiento, se tienen sus pesos en distintos momentos del aprendizaje. Esto sirve para poder ver la evolución de la conducción durante el entrenamiento, además de analizar los resultados del modelo final obtenido. Para cada variación implementada, se llevan a cabo cinco pruebas diferentes, correspondientes a cinco de los *checkpoints* guardados aproximadamente en los momentos en los que se llevaba un tiempo de entrenamiento de 10h, 20h, 30h, 40h y 50h (al finalizar los dos días aproximados de entrenamiento). Cada una de dichas pruebas, consiste en realizar 10 ejecuciones del algoritmo de evaluación para obtener una media del tiempo (en segundos), $t_{driving}$, que el vehículo logra permanecer en la carretera sin colisionar ni salirse de ella. Los tiempos $t_{driving}$ obtenidos como resultado son los que permiten analizar el desempeño de cada modelo, además de las valoraciones cualitativas que el usuario puede realizar mientras se ejecuta la evaluación, ya que se puede ver el comportamiento del coche en el escenario de *Airsim* durante la evaluación.

Resultados. La Tabla 5.1 muestra en primer lugar el tiempo medio, $t_{driving}$, y en segundo lugar la desviación típica entre paréntesis, tras las 10 ejecuciones realizadas y la evolución de este en cinco puntos distintos del aprendizaje del modelo.

Modelo evaluado	**10h entrenamiento**	**20h entrenamiento**	**30h entrenamiento**	**40h entrenamiento**	**Modelo final**
DQN base (v0)	3.57s (0.48s)	3.89s (0.54s)	9.90s (1.93s)	10.27s (2.48s)	**11.99s** (4.21)
DQN *reward* v1	7.61s (2.23s)	8.39s (2.30)s	16.52s (3.69s)	16.45s (4.68s)	**18.72s** (4.84s)
DQN *reward* v2	7.93s (1.30s)	9.34s (2.48s)	10.49s (3.98s)	13.00s (3.88s)	**18.85s** (5.88s)
DDQN base	4.31s (0.55s)	5.12s (2.01s)	5.78s (2.49s)	7.12s (3.48s)	**10.74s** (4.41s)

Tabla 5.1: Tiempo de conducción ($t_{driving}$) medio en 10 ejecuciones y desviación estándar con entrenamiento y validación en el mismo escenario (*neighborhood*).

Como se puede ver en la tabla 5.1, el modelo que mejor resultado obtiene al finalizar el entrenamiento es el del algoritmo DQN con la versión 2 de la *reward*, aunque con tiempos muy similares a los de la DQN con la versión v1 de la *reward*. Esto se debe a que en ambos modelos, la colisión está penalizada con una recompensa negativa, por lo que el vehículo ha aprendido a intentar evitar esta situación de manera más rápida que el DQN base. Sin embargo, de manera cualitativa se ha observado que esto lleva también a que el coche decida la mayor parte de las veces seguir recto en lugar de girar, en los cruces de carretera que se encuentra. En cuanto a la DDQN no se han obtenido los resultados pensados, se esperaba lograr un tiempo mejor que en la DQN base, sin embargo es algo menor. Esto se debe a que la mejora de la DDQN respecto al DQN se suele ver

más adelante, con un número de *epoch* y por tanto un tiempo de entrenamiento del algoritmo mayor al realizado en este trabajo.

Además, la evolución del tiempo en la carretera en cada uno de los cuatro modelos se puede ver de manera más clara en las gráficas mostradas en la Figura 5.1. Tanto en esta figura como en la tabla anterior se observa como en el DQN con *reward* v1 y en el DQN con *reward* v2 la conducción mejora de manera más rápida que en el DQN base, aguantando más tiempo en la carretera incluso en los primeros momentos del entrenamiento. En cambio, en la evolución del tiempo en la carretera de la DDQN, se ve como tiene un comportamiento similar a la DQN base en un principio pero sin llegar a alcanzar el tiempo de esta en el modelo final, aunque se observa una tendencia más creciente en los puntos de la DDQN que en los de la DQN básica.

Con estos resultados, se elige como mejor modelo de entre los analizados la DQN base con *reward* v2.

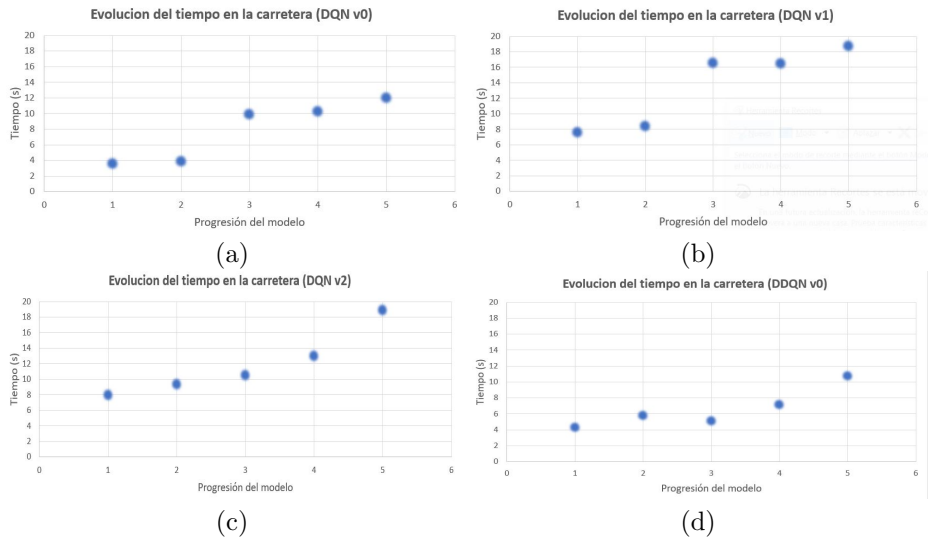


Figura 5.1: Evolución del tiempo en la carretera, $t_{driving}$, durante diferentes puntos del entrenamiento de los cuatro modelos analizados. (a) DQN base (v0), (b) DQN reward v1, (c) DQN reward v2 y (d) DDQN.

5.2.2. Análisis de un modelo entrenado en el mismo escenario que el *encoder*

Objetivo. Analizar la influencia de los datos que se utilizan en el *encoder*, entrenando el algoritmo con mejores resultados del experimento anterior (DQN con *reward* v2) en el mismo escenario del que se han sacado las imágenes del *dataset* para la CNN (*landscape*), y comprobando si el aprendizaje es más rápido y la conducción en este entorno mejor que la lograda entrenando el *encoder* y el algoritmo de *reinforcement learning* en escenarios distintos.

Descripción del experimento. En este experimento, se conecta con el escenario *landscape* para evaluar el modelo DQN con *reward* v2 entrenado en

dicho escenario, realizando así un análisis de su rendimiento utilizando el mismo entorno para el entrenamiento del *encoder*, el entrenamiento de la DQN y la evaluación. Al igual que en el experimento anterior, se han guardado diversos *checkpoints* durante el entrenamiento, por lo que se puede analizar tanto el proceso de aprendizaje como el comportamiento al finalizar, viendo así si difiere del caso en el que *encoder* y DQN están entrenados en un escenario distinto y la influencia que tiene en el aprendizaje de la conducción los datos utilizados en el *encoder*. Como en el experimento anterior, se calcula el $t_{driving}$ medio de diez ejecuciones en cinco momentos diferentes del entrenamiento. Los tiempos obtenidos son los que se van a analizar en el apartado de resultados, así como las valoraciones cualitativas del usuario sobre el comportamiento del vehículo durante la evaluación.

Resultados. La Tabla 5.2 muestra en primer lugar el tiempo medio, $t_{driving}$, y en segundo lugar la desviación típica entre paréntesis, tras las 10 ejecuciones realizadas y la evolución de este en cinco puntos distintos del aprendizaje del modelo.

Modelo evaluado	**10h entrenamiento**	**20h entrenamiento**	**30h entrenamiento**	**40h entrenamiento**	**Modelo final**
DQN v2	2.94s	3.18s	3.94s	5.19s	**5.34s**
(*landscape*)	(0.21s)	(0.32s)	(0.16s)	(1.22s)	(0.65s)
DQN	7.93s	9.34s	10.49s	13.00s	**18.85s**
reward v2	(1.30s)	(2.48s)	(3.98s)	(3.88s)	(5.88s)

Tabla 5.2: Tiempo de conducción ($t_{driving}$) medio en 10 ejecuciones y desviación estándar con entrenamiento y validación en el mismo escenario que el *encoder* (*landscape*).

Como se observa en la tabla 5.2, el tiempo obtenido con el modelo final entrenado en *landscape* es mucho menor que el del mismo algoritmo (DQN con *reward* v2) entrenado en *neighborhood*. Esto se debe a que la carretera existente en *landscape* es estrecha, con vallas, muchas curvas, subidas y bajadas del terreno y cubierta de nieve (por lo tanto es un camino resbaladizo ya que *Airsim* es realista en la física del escenario), a diferencia de las grandes rectas llanas de *neighborhood*, como se ve en la Figura 3.4, siendo por tanto el primero un entorno más complicado.

Atendiendo ahora a la evolución del $t_{driving}$ en el modelo entrenado en *landscape*, que también se muestra en la Figura 5.2, se puede ver que a lo largo del aprendizaje va aumentando dicho tiempo, pero no lo hace de manera más rápida que en los casos en que el entrenamiento del *encoder* y del algoritmo de *reinforcement learning* se realiza en un entorno distinto. Durante la evaluación se ve que no comienza ya con una trayectoria más adecuada aunque el *encoder* utilice datos del mismo escenario, por ello se extrae, que el entorno del que procedan los datos utilizados en la CNN no tienen una gran influencia sobre el entrenamiento posterior. Esto se puede deber a que en el *encoder* se tenía en cuenta solo la mitad inferior de las imágenes del *dataset*, fijándose así en la dirección de la carretera únicamente y sin tener en cuenta el resto de aspectos del entorno (existencia de árboles, edificios, montañas...) y por lo tanto afectando del

mismo modo a los algoritmos de *reinforcement learning* entrenados en cualquier escenario.

Lo que también se puede ver es que en los dos últimos puntos de análisis del $t_{driving}$ en el modelo (con 40h de entrenamiento y modelo final), este aumenta más notablemente (tendencia hacia arriba) que en los *checkpoints* anteriores, algo que también se ha podido apreciar viendo la mejora del comportamiento del coche en el simulador durante la evaluación, lo que lleva a pensar que con un mayor tiempo de entrenamiento se podrían alcanzar unos resultados mejores.

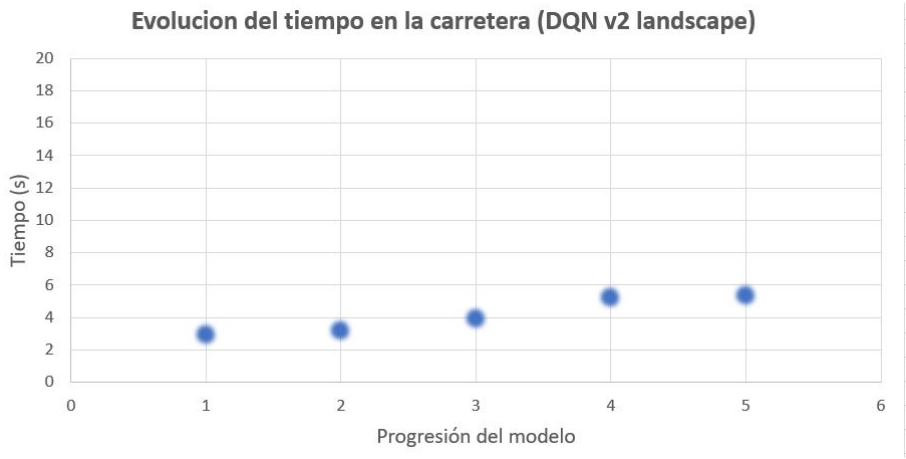


Figura 5.2: Evolución del tiempo en la carretera, $t_{driving}$, durante diferentes puntos del entrenamiento del modelo DQN con *reward* v2 entrenado en *landscape*

5.2.3. Actuación en distintos escenarios

Objetivo. Analizar el comportamiento del coche evaluando los cinco modelos propuestos en distintos escenarios, para estudiar así la generalización a distintos entornos de cada uno de ellos.

Descripción del experimento. En este experimento se realiza el análisis del rendimiento de los modelos obtenidos al final del entrenamiento de cada versión en distintos escenarios. En este caso, como en los experimentos anteriores, también se calcula el $t_{driving}$ de diez ejecuciones, pero sin analizar puntos intermedios del entrenamiento, únicamente con el modelo final. Esto se lleva a cabo en tres entornos distintos, en los que se incluye el escenario en el que ha sido entrenado el algoritmo de *reinforcement learning* (*neighborhood* en todos los casos excepto en el último modelo que ha sido entrenado en *landscape*), el del *encoder* (*landscape*) y otro totalmente nuevo (*coastline*, aunque en el caso del último modelo entrenado en *landscape* tanto *coastline* como *neighborhood* son nuevos). Los tiempos $t_{driving}$ obtenidos y las valoraciones cualitativas realizadas por el usuario durante la evaluación van a permitir el análisis de la generalización de los modelos desarrollados y su rendimiento en nuevos entornos.

Resultados. La Tabla 5.3 muestra el tiempo medio, $t_{driving}$, tras 10 ejecuciones, y su desviación típica entre paréntesis, evaluando el modelo final en distintos escenarios. En las cuatro primeras filas se muestran los resultados de los 4 modelos entrenados en *neighborhood* mientras que la última corresponde al modelo entrenado en *landscape*.

En ella se puede observar, en cuanto a los cuatro primeros modelos, que el $t_{driving}$ en los escenarios de *landscape* y *coastline* es notablemente menor que en el escenario *neighborhood* de entrenamiento, siendo la modificación de la DQN con *reward* v2 la que mayores tiempos alcanza. Esto se puede deber a que son tres escenarios con características muy diferentes, como se muestra en la Figura 3.4 y se ha explicado en el capítulo 3, y por lo tanto resulta difícil la generalización de un entorno a otro. Además cualitativamente durante la evaluación se aprecia que en los cuatro casos, el modelo desarrollado ha aprendido que ir recto en el centro de la carretera es la opción con la que obtiene recompensas más altas (ya que en *neighborhood* hay grandes rectas de carreteras anchas y los giros en cruces son más complicados de realizar y hay más posibilidades de chocar con algo), por lo tanto es la acción que decide llevar a cabo en todos los escenarios, lo cual muestra que no se está generalizando bien. Las características de cada entorno, explican que el $t_{driving}$ en *coastline* sea mayor que en *landscape* en estos cuatro casos, ya que en *landscape* apenas hay rectas (la carretera ya tiene al comienzo una gran curva cuesta abajo), mientras que en *coastline* la carretera es algo más ancha y no tiene un gran giro hasta varios metros más adelante.

En la última fila, se ven los resultados del modelo entrenado en *landscape*, en el que se observa que los tiempos medios de conducción en los otros dos escenarios tampoco son altos, como ocurre con el resto de modelos. También durante la evaluación se ha apreciado que el comportamiento del modelo en todos los escenarios es el adecuado para la trayectoria de *landscape* (avanzar recto entre dos y tres segundos y girar a la derecha después), por lo que este modelo tampoco generaliza correctamente a los diferentes entornos. Del mismo modo que en los otros modelos, los tiempos en *neighborhood* y *coastline* se explican con las características de cada entorno. Mientras que en *neighborhood* hay una gran carretera recta y un giro a la derecha provoca un choque rápido contra un edificio en ese lado, en *coastline* la carretera no tiene edificios con los que chocar a sus laterales por lo que aguanta más tiempo hasta chocar.

Modelo evaluado	***Neighborhood***	***Landscape***	***Coastline***
DQN base (v0)	11.99s (4.21s)	4.46s (0.90s)	8.18s (1.88s)
DQN *reward* v1	18.72s (4.84s)	4.84s (0.51s)	14.76s (0.21s)
DQN *reward* v2	18.85s (5.88s)	5.11s (1.08s)	15.30s (1.76s)
DDQN base	10.74s (4.41s)	4.16s (0.43s)	6.54s (0.47s)
DQN v2 (*landscape*)	3.51s (0.42s)	5.34s (0.65s)	7.93s (0.80s)

Tabla 5.3: Tiempo de conducción ($t_{driving}$) medio de 10 ejecuciones y desviación estándar con entrenamiento y validación en distintos escenarios.

Capítulo 6

Conclusiones

Este capítulo presenta las conclusiones extraídas durante la realización de este proyecto, tanto técnicas como personales, así como los problemas abordados en él. También se exponen algunas alternativas de trabajo futuro que se pueden realizar partiendo de este proyecto.

6.1. Conclusiones técnicas

El objetivo principal de este proyecto era implementar y poner en marcha un sistema de conducción autónoma basado en técnicas de *deep learning*, utilizando un simulador para la extracción de los datos necesarios, el entrenamiento de los algoritmos y la evaluación de los modelos desarrollados de manera segura y fácil. Este objetivo se ha cumplido, logrando aprender los fundamentos de distintas técnicas de aprendizaje automático, profundizando sobre todo en algunos algoritmos de *reinforcement learning*. También conocer el manejo de un simulador realista de conducción como es *Airsim*, algo muy importante de cara al entrenamiento y evaluación de dichos algoritmos ya que necesitan que el agente explore el entorno y realice muchas pruebas en él, algo que en el mundo real sería inviable en el caso de vehículos autónomos. Como conclusión de ello, se ha comprobado la gran importancia y potencial que tienen este tipo de simuladores en esta aplicación así como la adecuación de las técnicas de *deep learning* estudiadas, sobre todo de *deep reinforcement learning* como apuntaban otras investigaciones actuales, para una tarea como la conducción autónoma.

Tras la realización de los experimentos planteados, que se muestran en el capítulo 5, se ha analizado que las modificaciones propuestas logran unos resultados comparables o incluso de mejor calidad que los obtenidos con el ejemplo del repositorio de GitHub «*AutonomousDrivingCookbook*» que se ha tomado como base para comenzar este proyecto. Se consigue así un resultado satisfactorio, dando lugar un trabajo que sirva de punto de partida para seguir investigando en este ámbito como se expone en la sección 6.4 ya que hay muchos caminos para seguir mejorando en este campo.

Además, como conclusión más general, decir que el sistema implementado, los resultados obtenidos y entorno de simulación utilizado en este proyecto servirán dentro del grupo de investigación como base para continuar con otras investigaciones en esta línea.

6.2. Conclusiones personales

Personalmente, estoy satisfecha con el trabajo ya que me ha servido como experiencia para ver como se trabaja en un laboratorio de investigación, rodeada de personas con más experiencia de las que he podido aprender y las cuales me han ayudado y aconsejado.

Elegí este trabajo debido a que los vehículos autónomos son una aplicación que actualmente se está investigando mucho y que se sustenta en un campo tan en auge como el aprendizaje automático y *deep learning*, en el cual estoy interesada y del que me gustaría aprender más. Esta decisión ha sido adecuada ya que me ha permitido tanto aprender los fundamentos básicos de un algoritmo de aprendizaje supervisado ampliamente utilizado para diferentes aplicaciones como son las CNNs como de aprender qué es el aprendizaje por refuerzo e implementar alguno de sus algoritmos más típicos, estudiando y analizando otros trabajos de investigación.

Además he mejorado en la programación con *Python*, aprendiendo a utilizar librerías típicas de *deep learning* como son *keras* y *TensorFlow*, dentro de un entorno virtual en la plataforma *Anaconda*. También me he dado cuenta de la utilidad de los simuladores de conducción autónoma, comprobando por mi misma el potencial que tienen al haber utilizado *Airsim* en este proyecto.

Pienso que todo lo aprendido, tanto la parte técnica como la de trabajo en equipo, me puede servir en un futuro.

6.3. Problemas encontrados

Durante la realización de este proyecto se han tenido que abordar diversos problemas. El primero fue encontrar documentación y publicaciones científicas de calidad en el ámbito de la investigación académica y comprenderlos y analizarlos de manera crítica. Además se tuvo que aprender los fundamentos básicos de muchos algoritmos nuevos ya que no se tenía ningún conocimiento previo de *reinforcement learning*.

Una vez adquiridos los conocimientos básicos en el campo de aprendizaje automático y la conducción autónoma necesarios para empezar, aparecieron problemas en el *set up* de un entorno adecuado en el que se pudieran implementar y evaluar los experimentos. Por un lado, el portátil del que se disponía no era lo suficientemente potente para la realización de los entrenamientos con el simulador, por lo que era necesaria la utilización de uno más potente del laboratorio en el que se trabajaba con el que debía establecer conexión remota dada la situación sanitaria, la cual se logró tras varios intentos con diferentes métodos y programas, utilizando finalmente *TeamViewer*. Por otro lado, el código del repositorio de GitHub *AutonomousDrivingCookbook* utilizado como base para comenzar este trabajo, estaba desactualizado y requería el uso de versiones más antiguas de *Python*, *keras*, *TensorFlow* y el resto de librerías necesarias, por lo que se encontraron algunos problemas de compatibilidad en las versiones a la hora de crear un entorno virtual adecuado en *Anaconda* e instalar dichos paquetes, los cuales finalmente se lograron solucionar pero hicieron que la puesta en marcha del proyecto fuera más costosa de lo esperado. También el paquete reducido del simulador utilizado era antiguo, por lo que no tenía las últimas actualizaciones, aunque esto no ha afectado demasiado al desarrollo del trabajo.

Finalmente, el último reto encontrado está relacionado con la dificultad de la aplicación estudiada, la conducción autónoma, ya que debido a esto, es complicado obtener unos grandes resultados a la primera, ya que aún disponiendo de un ordenador potente y con una buena tarjeta gráfica necesitan de mucho tiempo de entrenamiento. Por un lado, porque son necesarias miles de *epoch* (incluso se realizan millones en algunos trabajos) para que el algoritmo empiece a aprender y converja y, por otro lado, porque la simulación de la conducción se realiza en tiempo real, por lo que cada episodio de simulaciones para llevarse a cabo necesita un tiempo que va aumentando conforme el algoritmo aprende ya que el vehículo va aguantando un mayor tiempo en la carretera. El tiempo de entrenamiento es un problema ya que se pueden llegar a necesitar días, incluso más de una semana, para extraer buenos resultados.

6.4. Trabajo Futuro

Este proyecto es una base que se puede continuar desarrollando y profundizar más en él, abriendo el camino a diferentes alternativas en las que se puede seguir trabajando en un futuro.

El primer paso, puede ser con los mismos algoritmos estudiados en este trabajo, realizar entrenamientos en un ordenador más potente y durante más tiempo para comprobar si los resultados que se pueden llegar a lograr con ellos son mejores. También se puede entrenar el *encoder* de base (CNN) con una mayor cantidad y más variada de datos, utilizando imágenes de otros escenarios del simulador, para analizar de qué manera afecta esto al modelo final y si ayuda a la generalización en distintos escenarios.

Otra alternativa, es seguir investigando algoritmos de *deep reinforcement learning* estudiados en otras investigaciones y que hayan logrado buenos resultados. Se pueden implementar muchos algoritmos. Un ejemplo es la *dueling DQN*[41], en el que los valores Q se dividen en dos partes distintas, la función de valor, V(s), que dice cuánta recompensa obtendremos desde el estado actual y la función de ventaja, A(s, a) que dice cuánto mejor es una acción respecto a las demás, de manera que se propone que la red neuronal utilizada divida su capa final en dos que estimen estas dos partes por separado y en la capa de salida juntar ambas partes estimando en ella los valores Q. Otro ejemplo interesante es el algoritmo *rainbow*[42], que combina los avances y ventajas de muchos algoritmos de *deep reinforcement learning* y según algunas investigaciones obtiene los mejores resultados.

Para finalizar, una última posibilidad puede ser explorar los retos que se abordan en el paso de un modelo entrenado en simulación al mundo real, utilizando para ello alguno de los robots móviles disponibles en el laboratorio en el que se utilice uno de los modelos obtenidos viendo así su funcionamiento en un sistema real.

Apéndice A

Detalles y resultados adicionales de los entrenamientos

A.1. Configuración general de los entrenamientos

Encoder. Para todos los entrenamientos se utiliza un mismo *encoder* entrenado con datos del escenario *landscape*.

Escenarios. En los cuatro primeros modelos se utiliza el escenario *neighborhood* del simulador durante el entrenamiento del algoritmo de *reinforcement learning*. En cambio, el último se entrena con el escenario *landscape*.

Hiperparámetros. Los hiperparámetros elegidos son un *batch size* de 32, un tamaño de *replay memory* (*buffer* de memoria que guarda las acciones del coche durante cada episodio) de 50, una *batch update frequency* (que concreta cada cuantos *batches* se guarda un *checkpoint* del modelo) de 500, se define una épsilon inicial de 1 que baja 0.003 en cada iteración y una épsilon mínima de 0.1, un γ o factor de descuento de 0.9 y un *learning rate* de 0.001 con para *Adam*.

Tiempo de entrenamiento. Todos los entrenamientos han tenido una duración aproximada de dos días y por lo tanto se han realizado aproximadamente 10.000 *epoch* en cada uno (esto varía un poco en cada experimento concreto).

Resultados obtenidos en el entrenamiento. Se obtiene, en primer lugar, un archivo CSV con los valores "en crudo" de la *reward*, guardado durante el entrenamiento con todos los valores de recompensa obtenidos, viendo en cada línea las de cada episodio completo. Este sirve para obtener la gráfica de la *reward* acumulada total por episodio, en la que se muestra el valor de esta en el eje de ordenadas y las *epoch* en el de abscisas. Y por último, una gráfica de la *loss* que se obtiene con *Tensorboard*, en la que se muestra su valor en el eje de ordenadas y el tiempo de entrenamiento en horas en el eje de abscisas.

A.2. Base DQN con *reward* base (v0)

Gráfica de la *loss*. Como se ve en la Figura A.1, la *loss* disminuye en un primer momento del entrenamiento, subiendo después ligeramente. En algoritmos de aprendizaje por refuerzo, esto se suele dar en las partes iniciales del entrenamiento sobre todo, cuando el modelo aún necesita más tiempo de entrenamiento y no ha convergido, por lo que se puede dar un aumento en ella, a diferencia de en aprendizaje supervisado.

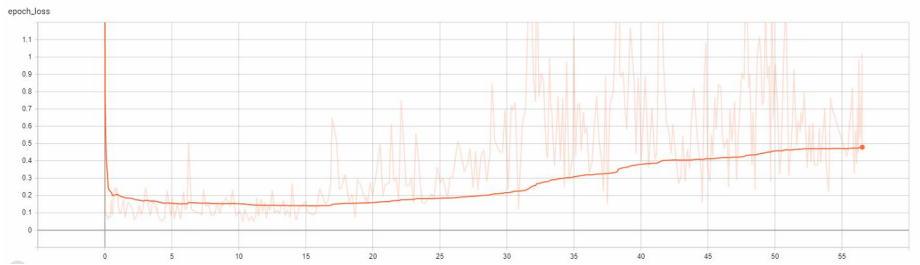


Figura A.1: Gráfica de evolución de la *loss* durante el tiempo de entrenamiento (en horas) del modelo 1.

Gráfica de la *reward*. Se observa en la Figura A.2, donde se ve que los valores de la recompensa, a partir de la *epoch* 4000, se van haciendo más altos, rondando entre valores de 100 y 150, llegando en algunos casos incluso a valores de 200. Se observa a lo largo de toda la gráfica que hay muchos valores bajos de recompensa, lo que es debido a que la épsilon mínima está constante en un valor de 0.1, por lo que el 10 % de las acciones que se realizan son aleatorias de manera que no se estanque el aprendizaje y el agente siga explorando. Lo que también se puede apreciar es que las recompensas siguen sin llegar a un valor máximo en el que se queden más o menos fijas, lo que significa que aún se necesita un mayor tiempo de entrenamiento para que el modelo converja y aprenda la mejor política de conducción posible con este algoritmo.

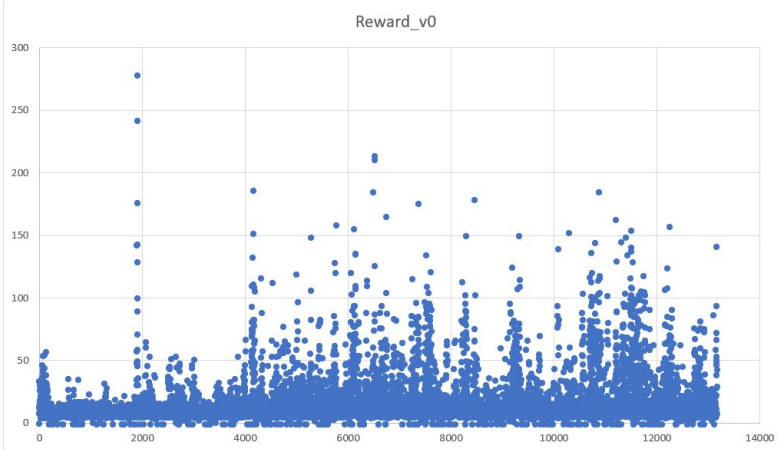


Figura A.2: Gráfica del valor de la *reward* acumulada del episodio en cada iteración del modelo 1.

A.3. Base DQN con *reward* v1

Gráfica de la *loss*. Se muestra en la Figura A.3, donde se ve que la *loss* disminuye en un primer momento del entrenamiento, subiendo después ligeramente, de la misma manera que en el entrenamiento del modelo anterior, aunque llegando a valores algo más altos.

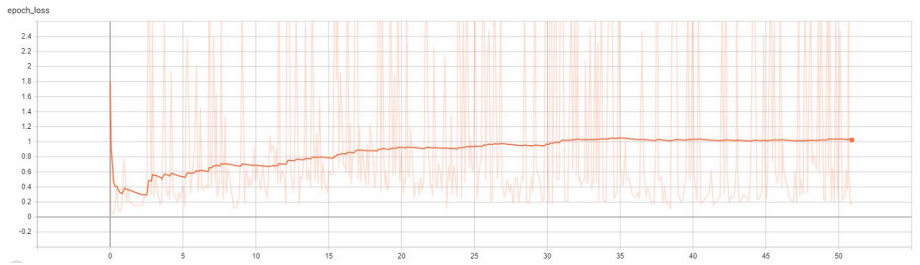


Figura A.3: Gráfica de evolución de la *loss* durante el tiempo de entrenamiento (en horas) del modelo 2.

Gráfica de la *reward*. Se observa en la Figura A.4. En este caso, se ve que las recompensas empiezan a ser más altas antes que en el modelo anterior, sobre la *epoch* 1000, siendo el aprendizaje más rápido que en el experimento 1. Además se observa que ya a partir de la *epoch* 5000, las *rewards* obtenidas en el episodio se encuentran entre los valores de 200 y 250, en lugar de estar entre 100 y 150 como en el entrenamiento anterior, aunque también se aprecia que con más tiempo de entrenamiento puede llegar a seguir aumentando. Por lo tanto, a la vista de estos resultados de entrenamiento se puede decir que esta modificación del código de base mejora en cierta medida la conducción, aunque es necesario realizar la evaluación del modelo para tener una conclusión definitiva.

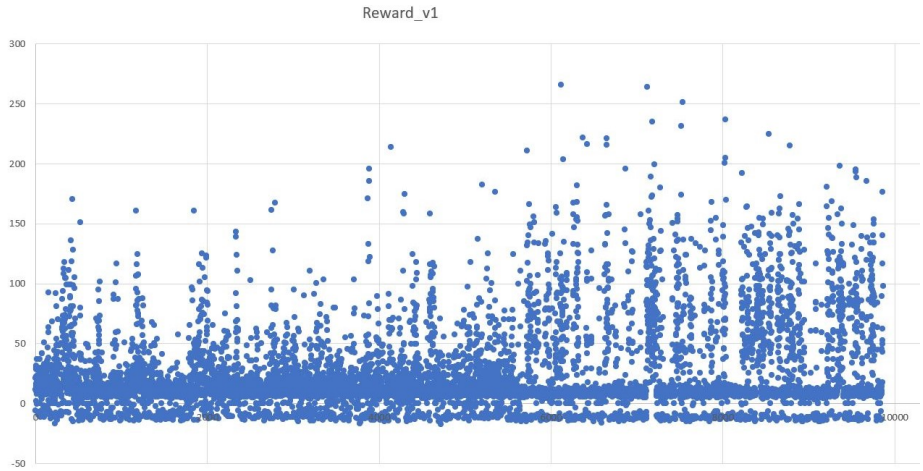


Figura A.4: Gráfica del valor de la *reward* acumulada del episodio en cada iteración del modelo 2.

A.4. Base DQN con *reward* v2

Gráfica de la *loss*. Se muestra en la Figura A.5, donde se ve que la *loss* disminuye en un primer momento del entrenamiento, subiendo después, del mismo modo que ocurría en el entrenamiento del primer modelo detallado en la sección A.2, obteniendo sin embargo unos valores sobre 0.8 similares a los de la sección A.3.

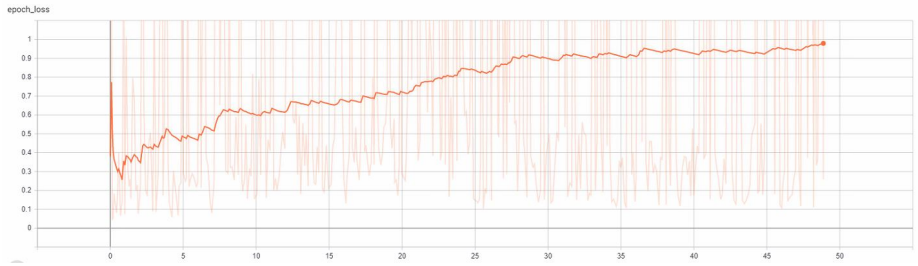


Figura A.5: Gráfica de evolución de la *loss* durante el tiempo de entrenamiento (en horas) del modelo 3.

Gráfica de la *reward*. Se muestra en la Figura A.6. En este caso, se ve que las recompensas son más altas antes que en el entrenamiento de la sección A.3, sobre todo apreciándose a partir de la *epoch* 6000. Se observa que a partir de ahí, las *rewards* obtenidas en el episodio se encuentran entre los valores de 250 y 300, en lugar de estar entre 200 y 250 como en el caso anterior, aunque también se aprecia, de igual modo, que con más tiempo de entrenamiento puede llegar a seguir aumentando. Por lo tanto, esta modificación del código de base mejora los valores de las recompensas, siendo el mejor resultado de estas hasta el momento, aunque con valores muy similares a los de la DQN con la *reward* v1 de la sección A.3.

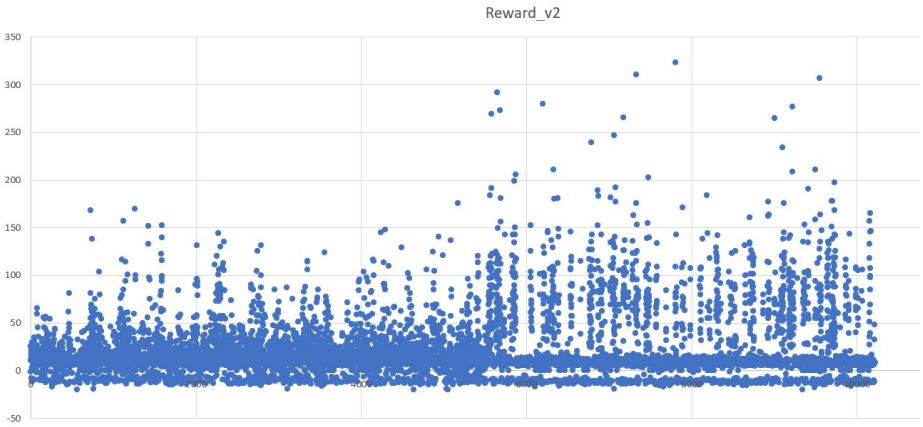


Figura A.6: Gráfica del valor de la *reward* acumulada del episodio en cada iteración del modelo 3.

A.5. *Double* DQN con *reward* base

Gráfica de la *loss*. Se muestra en la Figura A.7, donde se ve que la *loss* disminuye en un primer momento del entrenamiento, subiendo después ligeramente, de la misma manera que en el entrenamiento del modelo DQN base de la sección A.2. Sin embargo, en este caso se obtiene un valor de *loss* menor, sobre 0.2, apreciándose así la diferencia en el cálculo de los valores Q entre la DQN y la DDQN, siendo en esta segunda los predichos más aproximados a los obtenidos realmente.

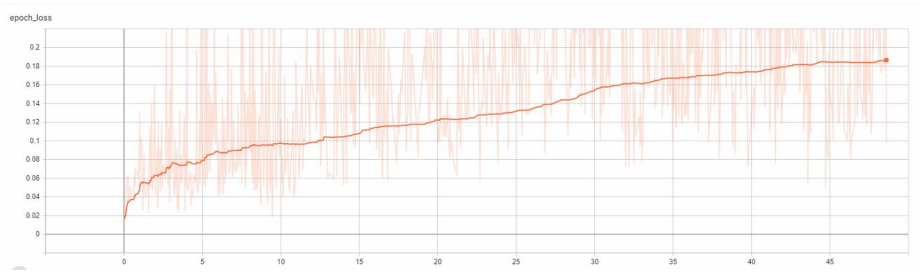


Figura A.7: Gráfica de evolución de la *loss* durante el tiempo de entrenamiento (en horas) del modelo 4.

Gráfica de la *reward*. Se observa en la Figura A.8, donde se ve que los valores de la recompensa, a partir de la *epoch* 3000, se van haciendo más altos, en un momento del entrenamiento más temprano que en el DQN base de la sección A.2. A partir de aquí, la *reward* acumulada por episodio ronda entre valores de 150 y 200, llegando en algunos casos incluso a valores puntuales superiores a 250, recompensas ligeramente superiores a las del DQN base. Lo que también se puede apreciar es que las recompensas siguen sin llegar a un valor máximo en el que se queden más o menos fijas, lo que significa que aún se necesita un mayor tiempo de entrenamiento para que el modelo converja y aprenda la mejor política de conducción posible con este algoritmo.

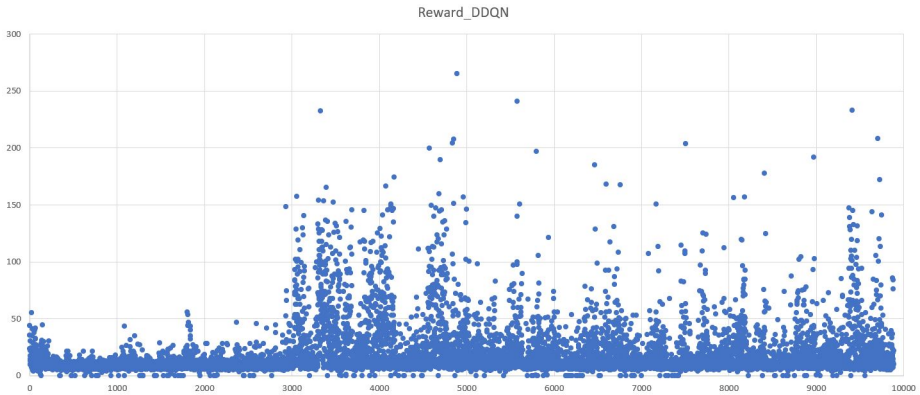


Figura A.8: Gráfica del valor de la *reward* acumulada del episodio en cada iteración del modelo 4.

A.6. DQN con *reward* v2 entrenado en *landscape*

Gráfica de la *loss*. Se muestra en la Figura A.9, donde se ve que la *loss* disminuye en un primer momento del entrenamiento, subiendo después y quedándose en un valor sobre 0.8, al igual que en su versión entrenada en *neighborhood* detallada en la sección A.4.

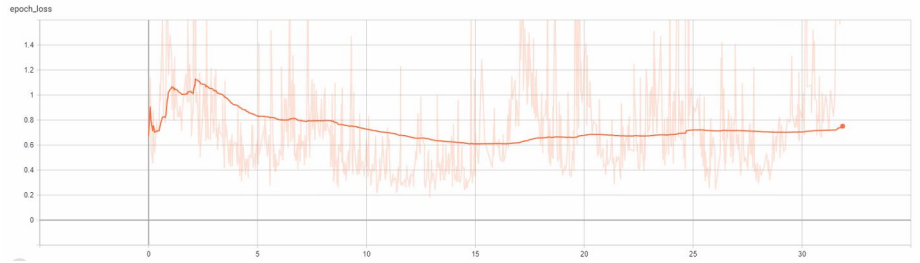


Figura A.9: Gráfica de evolución de la *loss* durante el tiempo de entrenamiento (en horas) del modelo 5.

Gráfica de la *reward*. Se observa en la Figura A.10, donde se ve que los valores de la recompensa acumulada por episodio ronda entre 100 y 150, llegando en algunos casos incluso a valores puntuales superiores a 300 y superiores. Lo que también se puede apreciar es que las recompensas oscilan continuamente entre estos valores, siguen sin llegar a un valor máximo en el que se queden más o menos fijas, lo que significa que aún se necesita un mayor tiempo de entrenamiento para que el modelo converja y la conducción sea más adecuada.

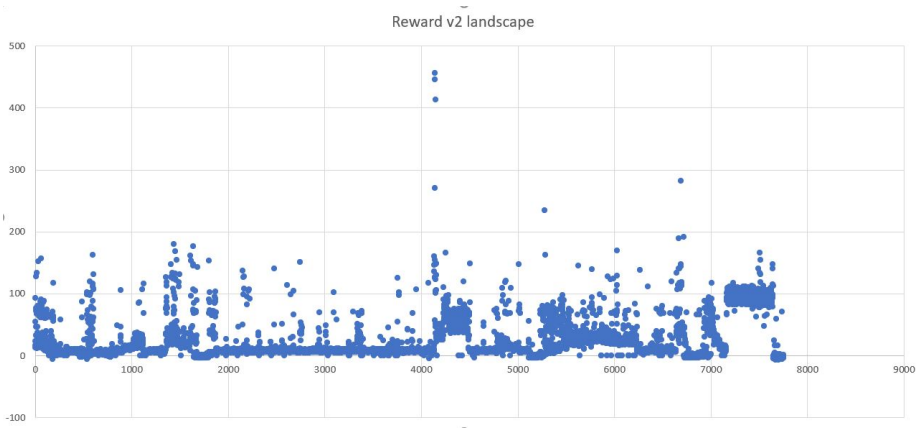


Figura A.10: Gráfica del valor de la *reward* acumulada del episodio en cada iteración del modelo 5.

Bibliografía

[1] Felipe Codevilla, Eder Santana, Antonio M López, and Adrien Gaidon. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 9329–9338, 2019.

[2] Xinlei Pan, Yurong You, Ziyan Wang, and Cewu Lu. Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952*, 2017.

[3] Saeed Asadi Bagloee, Madjid Tavana, Mohsen Asadi, and Tracey Oliver. Autonomous vehicles: challenges, opportunities, and future implications for transportation policies. *Journal of modern transportation*, 24(4):284–303, 2016.

[4] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[5] Rene Y. Choi, Aaron S. Coyner, Jayashree Kalpathy-Cramer, Michael F. Chiang, and J. Peter Campbell. Introduction to machine learning, neural networks, and deep learning. *Translational Vision Science and Technology*, 9(2):14–14, 2020.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[7] Richard Szeliski. *Computer Vision: Algorithms and Applications*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[8] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[9] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[10] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101*, 2016.

[11] Junshui Ma, Robert P Sheridan, Andy Liaw, George E Dahl, and Vladimir Svetnik. Deep neural nets as a method for quantitative structure–activity relationships. *Journal of chemical information and modeling*, 55(2):263–274, 2015.

[12] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[13] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[14] James Zou, Mikael Huss, Abubakar Abid, Pejman Mohammadi, Ali Torkamani, and Amalio Telenti. A primer on deep learning in genomics. *Nature genetics*, 51(1):12–18, 2019.

[15] Hui Y Xiong, Babak Alipanahi, Leo J Lee, Hannes Bretschneider, Daniele Merico, Ryan KC Yuen, Yimin Hua, Serge Gueroussov, Hamed S Najafabadi, Timothy R Hughes, et al. The human splicing code reveals new insights into the genetic determinants of disease. *Science*, 347(6218):1254806, 2015.

[16] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[17] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 2019.

[18] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[19] Felipe Codevilla, Matthias Miiller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–9. IEEE, 2018.

[20] Mathilde Caron, Piotr Bojanowski, Armand Joulin, and Matthijs Douze. Deep clustering for unsupervised learning of visual features. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 132–149, 2018.

[21] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[22] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

[23] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[24] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[25] Bichen Wu, Forrest Iandola, Peter H Jin, and Kurt Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 129–137, 2017.

[26] Mikael Boden. A guide to recurrent neural networks and backpropagation. *the Dallas project*, 2002.

[27] ByeoungDo Kim, Chang Mook Kang, Jaekyum Kim, Seung Hi Lee, Chung Choo Chung, and Jun Won Choi. Probabilistic vehicle trajectory prediction over occupancy grid map via recurrent neural network. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 399–404. IEEE, 2017.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[30] Takafumi Okuyama, Tad Gonsalves, and Jaychand Upadhay. Autonomous driving system based on deep q learnig. In *2018 International Conference on Intelligent Autonomous Systems (ICoIAS)*, pages 201–205. IEEE, 2018.

[31] April Yu, Raphael Palefsky-Smith, and Rishi Bedi. Deep reinforcement learning for simulated autonomous vehicle control. *Course Project Reports: Winter*, pages 1–7, 2016.

[32] Bernhard Wymann, Eric Espié, Christophe Guionneau, Christos Dimitrakakis, Rémi Coulom, and Andrew Sumner. Torcs, the open racing car simulator. *Software available at http://torcs. sourceforge. net*, 4(6):2, 2000.

[33] Zhiqing Huang, Ji Zhang, Rui Tian, and Yanxin Zhang. End-to-end autonomous driving decision based on deep reinforcement learning. In *2019 5th International Conference on Control, Automation and Robotics (ICCAR)*, pages 658–662. IEEE, 2019.

[34] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[35] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 584–599, 2018.

[36] Xinle Liang, Yang Liu, Tianjian Chen, Ming Liu, and Qiang Yang. Federated transfer reinforcement learning for autonomous driving. *arXiv preprint arXiv:1910.06001*, 2019.

[37] Roy Amante Salvador and Maria Isabel Saludares. Autonomous driving via deep reinforcement learning. 2019.

[38] Matthias Müller, Alexey Dosovitskiy, Bernard Ghanem, and Vladlen Koltun. Driving policy transfer via modularity and abstraction. *arXiv preprint arXiv:1804.09364*, 2018.

[39] Luona Yang, Xiaodan Liang, Tairui Wang, and Eric Xing. Real-to-virtual domain unification for end-to-end autonomous driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 530–545, 2018.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[41] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003, 2016.

[42] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.