Escola Tècnica Superior d'Enginyeria
Electrònica i Informàtica La Salle

Treball Final de Màster

Máster Universitario en Ingenieria de Telecomunicaciones

Aprendizaje por refuerzo profundo para la navegación de vehículos autónomos.

Alex Godó Alberch

Rosa María Alsina Pagès

ACTA DEL EXAMEN
DEL TRABAJO FINAL DE MÁSTER

Reunido el Tribunal calificador en la fecha indicada, el alumno

D. Alex Godó Alberch

expuso su Trabajo Final de Máster, titulado:

Aprendizaje por refuerzo profundo para la navegación de vehículos autónomos.

Acabada la exposición y contestadas por parte del alumno las objeciones formuladas por los Sres. miembros del tribunal, éste valoró dicho Trabajo con la calificación de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENTE DEL TRIBUNAL

Abstract

La inteligencia artificial ya es parte de nuestras vidas, de hecho, se usa lo suficiente como para que ni siquiera nos demos cuenta en nuestro día a día. El mundo de la automoción no es ninguna excepción, junto con una gran variedad de sensores instalados en el vehículo, este será capaz de conducir autónomamente en cualquier tipo de circunstancia. Este tipo de tecnologías están avanzando drásticamente ofreciendo al conductor una experiencia de conducción cómoda y segura. Pero, ¿hasta qué punto esta tecnología podría ser implementada y no suponer un riesgo para las vidas de los pasajeros y los que los rodean?.

El objetivo de este proyecto es estudiar uno de los métodos de Machine Learning más utilizados para desarrollar vehículos autónomos: Reinforcement Learning. Se compararán algunos de los algoritmos más populares y novedosos hasta la fecha.

Cada algoritmo se implementará en un simulador basado en una autopista con 5 carriles, donde un coche deberá aprender a desplazarse sin colisionar con los vehículos adyacentes a la máxima velocidad posible utilizando únicamente una cámara frontal situada en la parte delantera del mismo.

Aunque la mayoría de los algoritmos han dado resultados satisfactorios aun se debe investigar más en profundidad para poder aplicarlos en simuladores más complejos o en el mundo real.

Keywords: Reinforcement Learning, Machine Learning, Autonomous driving, DQN, A2C, PPO.

Índice

Acrónimos

AV: *Autonomous Vehicle.*

ADAS: *Advanced Driver Assistance System.*

RL: *Reinforcement Learning*.

ML: *Machine Learning*.

V2V: Vehicle to Vehicle.

V2I: Vehicle to Interface.

DSRC: *Dedicated Short Range Communication.*

DQN: Deep Q Network.

A2C: *Advantage Actor Critic*.

PPO: Proximal Policy Optimization.

ANN: Artificial Neural Network.

CNN: Convolutional Neural Network.

Figuras

Tablas

Ecuaciones

1 Introducción

Desde el primer vehículo producido en masa, el Ford T de 1908, muchas han sido las evoluciones que han cambió las características de los vehículos. En esos años, tener un automóvil era un lujo reservado para unos pocos privilegiados. Pero hoy en día, se ha convertido en algo muy corriente, y la industria automotriz sigue avanzando a diario con nuevas tendencias: mejorar, por ejemplo, la seguridad, eficiencia y conectividad. Una de las mejoras más importantes esta relacionada con la movilidad. La movilidad, es uno de los puntos más desafiantes que la sociedad enfrentará en los próximos años. No solo conlleva un desafío tecnológico, sino también un cambio social debido al cambio de paradigma de modos de transporte y sus mejoras en asuntos públicos.

Por esta razón, los fabricantes han posicionado el foco de atención en el desarrollo de vehículos autónomos (AV). Hasta hace poco tiempo, imaginarse coches que circularan por sí solos sin necesidad de conductor parecía reservado a la ciencia ficción. Sin embargo, teniendo en cuenta la evolución del sector de la automoción, es evidente que el aterrizaje de los automóviles de conducción autónoma está más cerca de lo que parece. Son múltiples los beneficios de los coches autónomos: reducen la congestión del trafico, los accidentes, los costes de movilidad, mejoran la utilización de carreteras, combaten el cambio climático y las emisiones C02 entre muchas otras más.

Incluso con todo este progreso, los accidentes y las muertes causadas por automóviles sin conductor aún representan una amenaza real. En siguientes apartados, se investigan las tecnologías que se utilizan el vehículo autónomo y lo impulsan hacia este progreso. Dada la tasa actual de desarrollo tecnológico, la llegada de los automóviles autónomos es inevitable. Si bien existen preocupaciones de seguridad y políticas que deben abordarse en la fase de desarrollo, los beneficios para la seguridad vial y la calidad de vida prevalecerán a largo plazo.

Por motivos de seguridad, antes de incorporarse a las carreteras, los vehículos autónomos primero deberán avanzar a través de 6 niveles de asistencia al conductor. La SAE (Sociedad de Ingenieros Automotrices), una organización profesional en el campo de la ingeniería, que creó un estándar para los niveles de automatización. Cada nivel de automatización se relaciona con dos métricas clave: las capacidades autónomas del automóvil y el tipo de interacción entre el automóvil y el conductor. Los cinco niveles de automatización son:

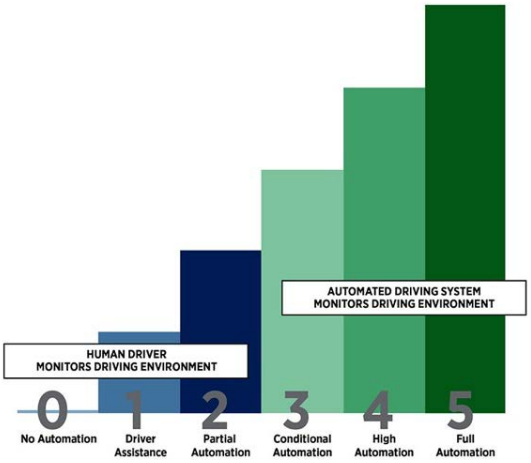


Figura 1. Niveles de autonomía.

- Nivel 0 - Sin automatización de conducción: El conductor humano es totalmente responsable de operar el vehículo.

- Nivel 1 - Asistencia para conducir: El vehículo está controlado en todo momento por el conductor, pero puede disponer de algunas características de asistencia, como por ejemplo, el control de crucero adaptativo, donde el vehículo se puede mantener a una cierta distancia detrás del vehículo de delante.

- Nivel 2 - Automatización de conducción parcial: El vehículo de nivel 2 tiene sistemas internos que se encargan de todos los aspectos de la conducción: dirección, aceleración y frenado. Sin embargo, el controlador debe poder intervenir si falla algún aspecto del sistema.

- Nivel 3 - Automatización de conducción condicional: Los vehículos de nivel 3 son aquellos que realmente pueden considerarse autónomos. Permitiendo al conductor sentarse y relajarse, ya que el automóvil puede encargarse de todo mientras conduce por la carretera. Los conductores pueden usar su teléfono de manera segura o mirar películas, aunque aún deben estar disponibles para intervenir si es necesario, por lo que quedarse dormido no es una opción.

- Nivel 4 - Conducción altamente automatizada: El vehículo puede conducir autónomamente en ciertas condiciones, controlar el entorno y no requiere intervención humana.

- Nivel 5 - Automatización completa: El vehículo es totalmente autónomo en todas las condiciones, no requiere intervención humana y puede conducir en carreteras y en diversas condiciones ambientales adversas.

Esta tecnología se implementa en los vehículos para comprender el entorno y tomar conciencia de las posibles situaciones de una manera semiautónoma para una conducción más segura se conoce como Sistema Avanzado de Asistencia al Conductor (ADAS). ADAS permite equipar los

automóviles y otros vehículos con sistemas de ingeniería de conducción autónoma como sensores basados en LiDAR, cámara de alta resolución, sistemas automáticos de frenado de emergencia o sistemas de advertencia de colisión frontal. Algunos ejemplos de este tipo de tecnologías son:

- Control de crucero adaptativo.
- Advertencia de colisión frontal.
- Sistema de asistencia de aparcamiento.
- Frenado de emergencia autónomo.
- Advertencias de cambio de carril.

Este proyecto se ha focalizado de una manera en que, para lograr una navegación autónoma, se utiliza la combinación de las funciones ADAS existentes. El vehículo deberá saber coordinar los sensores laterales, longitudinales y frontales para navegar con seguridad sin intervención humana. El objetivo de utilizar las funciones ADAS ya existentes es intentar acercar el desarrollo al de un vehículo autónomo de nivel 4 o nivel 5. Con el estado del arte de ADAS. Debido a que la mayoría de las funciones ADAS responsables del movimiento lateral y longitudinal de los vehículos solo pueden funcionar en carreteras , la segunda parte del proyecto será una investigación de los diferentes simuladores de vehículos autónomos, y especialmente, los que simulan una carretera [1].

Finalmente, el cerebro del vehículo, responsable de elegir la función ADAS correcta en cada estado, se basará en un algoritmo basado en aprendizaje profundo (DL) llamado aprendizaje por refuerzo (RL). Una cámara RGB será la responsable de decidir el estado del vehículo en cada momento.

El aprendizaje profundo tiene propiedades muy interesantes, y puede utilizar en vehículos autónomos para procesar datos sensoriales y tomar decisiones. Algunos ejemplos son: la detección de carril, detección de peatones, reconocimiento de señales de tráfico, detección de semáforos, reconocimiento de rostros (por ejemplo: si un automóvil autónomo necesita detectar y reconocer el rostro del conductor o de otras personas en el interior), detección de coches, detección de obstáculos, reconocimiento del entorno, predecir acciones humanas…

La lista continúa, y no cabe duda que estos sistemas son herramientas muy poderosas, pero hay algunas propiedades que pueden afectar su practicidad, especialmente cuando se trata de vehículos autónomos.

Muchos algoritmos de aprendizaje automático son en realidad muy impredecibles, sí, los humanos también lo son, pero la imprevisibilidad de estos sistemas es peor que la de los humanos y, por lo tanto, esto hace que de alguna forma sea inseguro aplicar hoy en día estos sistemas en algo tan complejo como es el mundo de la automoción, donde vidas humanas corren peligro. El sistema DL realmente puede predecir una acción de forma errónea, especialmente si las condiciones son bastante nuevas. Un humano, por otro lado, puede usar varios métodos para tomar decisiones adecuadas en circunstancias imprevistas.

1.1 Estado del arte

La industria automotriz ha contribuido a la innovación y al crecimiento económico. Actualmente el mundo está en la cúspide de la mayor revolución tecnológica en este sector: los vehículos autónomos. El objetivo principal es mantener a las personas fuera del control del vehículo y liberarlos de las tareas de conducción manteniendo siempre la seguridad de los pasajeros. Los elementos fundamentales en este tipo de tecnologías son los sensores ADAS para obtener información del entorno, un procesador que permita el procesamiento de dichos sensores y los actuadores responsables de convertir las señales eléctricas de la unidad de control en una acción. A continuación, se describirán algunos sensores más utilizados actualmente y un enfoque a otras tecnologías que podrían ser implantadas en un futuro cercano como son la comunicación de vehículo y vehículo a vehículo (V2V) y la comunicación de vehículo a infraestructura (V2I).

La tecnología ADAS hace posible la existencia de los vehículos sin conductor. Sin la capacidad de detectar el entorno alrededor de un automóvil, los sistemas informáticos no tendrían forma de saber qué velocidad establecer, dónde girar, cuándo cambiar de carril o cuándo frenar si hay una emergencia. El sistema de detección debe detectar todos los movimientos posibles dentro de su rango para tomar decisiones en una fracción de segundo.

Hoy en día, los vehículos sin conductor, interactúan con el entorno de 4 formas principales. El primero es el Radar para detectar objetos a largo y corto alcance, parecido al radar, algunas compañías optan por implementar un LiDAR que dispone de características similares. Otro sistema, usa una serie de cámaras RGB o infrarrojas situadas alrededor del vehículo para proporcionar una visión 3D del entorno. Por ultimo, los sensores de ultrasonidos también situados alrededor del vehículos, ayudan a detectar obstáculos a corto alcance y son utilizados usualmente por el sistema de aparcamiento. Como se puede apreciar, hay multitud de diferentes tecnologías instaladas en el vehículo, todas orientadas a una función especifica.

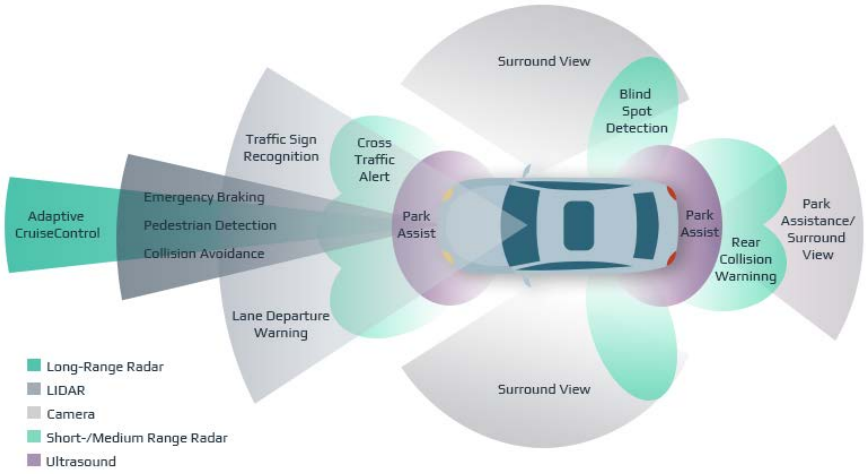


Figura 2. Sensores ADAS.

- Radar: Esta tecnología esta basada en ondas de radio para medir la distancia entre objetos, así como su velocidad y ángulo. El radar envía constantemente señales de radio que rebotan en los obstáculos. Comparan la señal transmitida con la señal recibida para

comprender el entorno. Además, estos sensores son relativamente pequeños, livianos y baratos. Todo esto hace que la tecnología sea muy atractiva para los fabricantes de automóviles. Como puntos negativos, el radar no puede determinar la forma de un objeto de manera precisa. Es por eso que un sensor de radar instalado en un vehículo autónomo no permitirá distinguir todos los objetos en la carretera. En otras palabras, no habría ninguna diferencia, para un radar, si hay un peatón o un árbol por delante.

- LiDAR: Esta tecnología utiliza luz infrarroja en vez de ondas de radio para escanear objetos y crear un mapa 3D del entorno. Un sensor LiDAR transmite rayos láser que rebotan en los objetos y regresan al sensor. Según la información recibida, un sistema LiDAR crea una nube de puntos que refleja la forma y el tamaño del objeto. El primer inconveniente es que los sensores LiDAR dependen de las condiciones climáticas. No pueden entregar imágenes precisas de los alrededores en entornos con niebla, nieve o polvo. En realidad, esto significa que la tecnología del sistema LiDAR siempre debe combinarse con sensores secundarios. Además, el precio de estos sensores es mucho más elevado al de los sensores radar. Como puntos positivos, las imágenes del LiDAR, en comparación con las del Radar, tienen una mayor precisión y resolución.

- Cámara: Las cámaras son una tecnología mucho más madura y ampliamente comprendida. También, son confiables y relativamente baratas ofreciendo la forma más precisa de crear una representación visual del mundo que nos rodea. Ahora bien, las cámaras a menudo enfrentan las mismas limitaciones que se encuentran con el ojo humano: La imagen no siempre es nítida o confiable cuando hace mal tiempo, especialmente cuando llueve o nieva intensamente. Por la noche, se depende de los faros delanteros del vehículo par iluminar el entorno, lo que limita en mayor modo la precisión. Además, una cámara no es capaz de calcular distancias entre objetos por si sola y depende de un procesador potente para hacer estos cálculos.

- Ultrasonidos: Por ultimo, similar al radar, el sensor ultrasonido funciona emitiendo ondas, pero en este caso, son ondas de sonido de alta frecuencia. Los sensores ultrasónicos han sido comunes en los automóviles desde la década de 1990 para su uso como sensores de estacionamiento, y son muy económicos. Su alcance puede limitarse a solo unos pocos metros en la mayoría de las aplicaciones, pero son ideales para proporcionar capacidades de detección adicionales para casos de uso en baja velocidad.

Históricamente, la mayoría de las compañías de vehículos autónomos han apostado por el sistema LiDAR ya que, hasta hace poco, las redes neuronales no eran lo suficientemente potentes como para manejar múltiples entradas de cámara. Tesla es una de las compañías más notables que ha apostado mucho por las cámaras, integrando ocho de ellas en cada vehículo, junto con una poderosa computadora de red neuronal que se entrena con datos provenientes de radares para dar profundidad a la imagen captada por la cámara. Aseguran que ofrece resultados iguales o incluso mejores a los sistemas de LiDAR a un precio mucho más reducido.

Para que esta revolución tenga lugar, el entorno que rodea al vehículo también debe evolucionar. En primer lugar, la infraestructura de la ciudad (como por ejemplo los semáforos) y deberá soportar tanto la visión humana como la artificial. La infraestructura vial debe pasar de mensajes analógicos diseñados para los ojos humanos a mensajes digitales diseñados para que la tecnología en los vehículos autónomos pueda interpretar el entorno y responder rápidamente. No solo la comunicación entre vehículo y entorno (V2I) debe evolucionar, sino que también la comunicación entre los distintos vehículos entre si (V2V).

Las comunicaciones de vehículo a vehículo (V2V) comprenden una red inalámbrica donde los automóviles se envían mensajes entre sí con información sobre lo que está sucediendo. Estos datos incluirían velocidad, ubicación, dirección de desplazamiento, frenada ... La tecnología de vehículo a vehículo utiliza comunicaciones dedicadas de corto alcance (DSRC) que utilizan frecuencias de 5.9Ghz, un estándar establecido por organismos como FCC e ISO. El alcance es de hasta 300 metros o aproximadamente 10 segundos a velocidades de autopista. V2V se implementaría como una red de malla, lo que significa que cada nodo o automóvil podría enviar, capturar y retransmitir señales. Con cinco a diez saltos en la red se podrán saber con precisión las condiciones del tráfico a 1 kilometro más adelante [2].

1.2 Motivación

No cabe duda que el tema de los vehículos sin conductor ya no se limita únicamente al mundo puramente teórico. Con compañías como, Uber, Intel, Tesla, Waymo y Google, todas desarrollando vehículos autónomos, la discusión está ahora en el espacio público. Y aunque una sociedad sin conductores puede parecer una utopía, se está acelerando hacia ella mucho más rápido de lo que se piensa.

Ciertamente, el futuro es apasionante y es increíble los cambios tecnológicos que se están produciendo en este siglo. Por esta razón, mi motivación personal es: desarrollar un sistema de vehículo autónomo utilizando los algoritmos de RL más modernos.

Aunque la implementación y toda la teoría que hay detrás del aprendizaje profundo parece complicada a primera vista, se pueden aplicar los conocimientos aprendidos durante el master y la experiencia laboral adquirida hasta la fecha para conseguir un proyecto funcional con resultados satisfactorios.

1.3 Objetivos

El objetivo principal de este trabajo es desarrollar y comparar 4 algoritmos de aprendizaje reforzado (RL) sobre un simulador basado en Unity que permitan a un coche aprender a conducir a través de una autopista de 5 carriles sin colisionar utilizando solo una cámara RGB situada en la parte frontal del mismo. Para ello, se estudiarán conceptos previos que establecerán un pilar fundamental para después profundizar en los algoritmos más complejos que se implementarán en el simulador escogido. Además, se estudiarán las ventajas y desventajas de cada algoritmo con ejemplos y graficas.

- Revisión de los simuladores actuales:
    - Explorar los diferentes simuladores que existen actualmente.
    - Investigar el funcionamiento del simulador utilizado.

- Desarrollo de software:
    - Utilización de las API/librerías para los análisis.

- Análisis de los resultados obtenidos
    - Estudio de los resultados de los 4 diferentes algoritmos.

- Conclusiones
    - Mencionar otras aplicaciones en las que se pueda aplicar RL. Investigando también el futuro de está tecnología.

Todo el aprendizaje adquirido para realizar este proyecto se ha basado en el libro: “Deep Reinforcement Learning Hands-On, Maxim Lapan”. El libro proporciona una introducción a los conceptos básicos de RL, brindando los conocimientos para programar agentes de aprendizaje inteligentes para asumir una gran variedad de tareas prácticas. El libro también ofrece decenas de ejemplos y explicaciones detalladas para cada algoritmo.

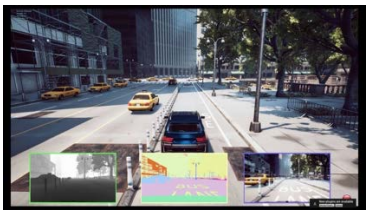
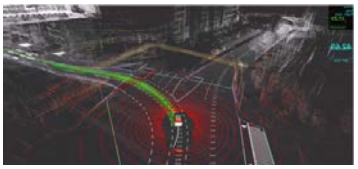
2 Simuladores

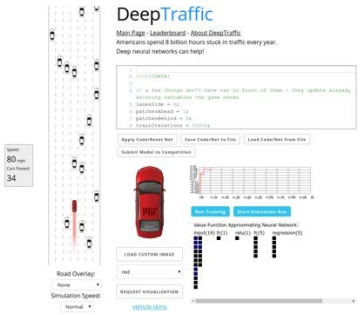
Es de vital importancia la elección de un buen simulador que se adapte al proyecto. Ya que este determinará el comportamiento y parámetros del vehículo. Actualmente, debido al auge de los algoritmos de RL, hay simuladores de todo tipo enfocados en infinidad de áreas relacionadas con la robótica. Además, un buen simulador tiene que ser rápido, ya que para implementar estos tipos de algoritmos es necesario iterar sobre varias partidas diferentes capturando un gran flujo de datos para poder entrenar al vehículo y llegar una solución correcta.

En este capítulo se detallaran algunos de los simuladores más utilizados para el desarrollo de aplicaciones de RL orientadas a vehículos autónomos. La siguiente investigación esta basada en el proyecto "*AV navigation with Deep Reinforcement Learning, Abril 2019, Àlex Cabañeros*" [3] añadiendo, además mi experiencia personal al probando algunos de ellos.

2.1 Comparación de simuladores

Se presenta una tabla resumen con el nombre del simulador, una breve descripción y los puntos positivos y negativos. Teniendo en cuenta que los puntos positivos y negativos están enfocados a los objetivos a realizar mencionados anteriormente.

Simulador	Imagen	Puntos positivos	Puntos negativos
AirSim		• Ofrece simulaciones 3D y físicas muy realistas. • Documentación extensa y de calidad. • Posibilidad de crear escenarios personalizados.	• Complejo y se escapa de los objetivos del proyecto. • Se necesita gran potencia de computación para correr el simulador. • Ofrece compatibilidad con todo tipo de sensores ADAS.
Autoware		• Ofrece simulaciones 3D y físicas muy realistas. • Puede ejecutarse en línea.	• Excesivamente complejo y se escapa de los objetivos del proyecto. • Basado en C++ y ROS. • Ofrece compatibilidad con todo tipo de sensores ADAS.

Carla		• Ofrece simulaciones 3D y físicas muy realistas. • Documentación extensa y de calidad. • Posibilidad de crear escenarios personalizados.	• Complejo y se escapa de los objetivos del proyecto. • Se necesita gran potencia de computación para correr el simulador. • Ofrece compatibilidad con todo tipo de sensores ADAS.
DeepTraffic		• Simple de utilizar pero no se pueden implementar algoritmos complejos.	• No ofrece visión de la cámara, solo la visión global de la autopista. • Solo permite al usuario ajustar los parámetros del algoritmo, no implementar uno propio. • Basado en Javascript.
Udacity AV Simulator		• Simulaciones 3D • Posibilidad de crear escenarios personalizados.	• No es de código abierto. • Poca documentación. • Basado en C.
Udacity Highwaypath-planning		• Simulaciones 3D • Posibilidad de crear escenarios personalizados.	• No es de código abierto. • Poca documentación. • Basado en C. • Basado solo en una autopista.
Robotbenchmark		• Funciona en el navegador. • Simulaciones 3D.	• No permite hacer un reset el entorno por software (solo manualmente). Para los algoritmos RL es importante tener esta funcionalidad. • Basado en Python.

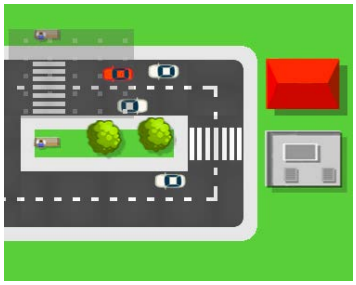
Metacar		• Creación de mapas personalizados. • No es necesario muchos recursos para poder ejecutarlo ya que corre sobre el navegador.	• Está basado en JavaScript y corre sobre el navegador. Aunque existe un proyecto de otro desarrollador que lo adapta para Python, no es estable y complica la implementación de algoritmos más complejos. • No ofrece visión de la cámara, solo ofrece un radar. • No ofrece ningún tipo de documentación por lo que investigar su funcionamiento reside en la prueba y error.
Unity ML-Agents Highway Simulator		• No es rápido a la hora de entrenar los algoritmos.	• Basado en Python. • Ejemplos de uso en el repositorio del creador. • Ofrece sensores ADAS de cámara y radar.

Tabla 1. Comparación de diversos simuladores.

En conclusión, el simulador que mejor se adapta para la implementación del proyecto es "Unity ML-Agents Highway Simulator". Ofrece los requisitos necesarios para una correcta implementación manteniendo una complejidad moderada respecto a otros simuladores. Cabe destacar que el objetivo principal de esta tesis es investigar sobre los algoritmos de RL, y utilizar un simulador excesivamente complejo aumentaría de forma exponencial los tiempos de entrenamiento, pudiendo llegar a converger en soluciones incorrectas debido al gran numero de parámatelos que hay que configurar.

2.2 Detalles del Simulador: Unity ML-Agents Highway Simulator

Finalmente, respecto a lo detallado en el apartado anterior, el simulador utilizado es el siguiente: "*Unity ML-Agents Highway Simulator*" [4].

Este simulador desarrollado por *Kyushik* y esta basado en el entorno Unity, un motor de desarrollo de videojuegos. Este motor es especialmente popular ya que contiene una infinidad

de características y es lo suficientemente flexible como para crear casi cualquier juego que se pueda imaginar.

Una vez ejecutado el simulador aparece la siguiente interfaz:

Figura 3. Captura del simulador.

La anterior captura muestra en el centro de la pantalla al coche que se quiere entrenar. A la derecha se muestran diferentes variables que servirán para monitorizar los resultados de cada simulación:

- Current Lane: En que carril está actualmente el vehículo.
- Distance: Distancia recorrida en cada partida. Cuando se llega a 2600 metros se reinicia el simulador y todos los parámetros.
- Reward: Una puntuación o recompensa por cada acción que haga el vehículo a entrenar. Más adelante se detallará con más detalle el valor de cada puntuación.
- Current Speed: Velocidad actual del vehículo. La velocidad máxima es de 80 Km/h mientras que la mínima es de 40 Km/h.
- Front Warning: Indica si hay un coche cerca en la dirección delantera.
- Left Warning: Indica si hay un coche cerca en la dirección izquierda.
- Right Warning: Indica si hay un coche cerca en la dirección derecha.
- Front Distance: Distancia en metros con el vehículo de delante.
- Num Overtakes: Numero total de adelantamientos en la partida actual.

El vehículo a entrenar solo puede realizar 5 acciones:

- Acelerar
- Frenar
- Girar a la derecha para cambiar de carril
- Girar a la izquierda para cambiar de carril
- No hacer nada

De esta forma, en este proyecto en concreto no ha sido necesario desarrollar ningún script para que el coche mantenga la posición en el centro del carril ya que de por si solo lo hace. Además, este simulador ofrece otras ayudas como: 3 sensores de proximidad que no permitirán al vehículo hacer un cambio de carril cuando haya otro vehículo en esa misma dirección. Es decir, solo se producirá una colisión si al cambiar de carril el coche de detrás acelera de manera

repentina o el de delante frena bruscamente. También se podría dar el caso de colisión cuando otro vehículo cambia de carril bruscamente cuando el vehículo a entrenar esta acelerando y este, no tiene tiempo de frenar.

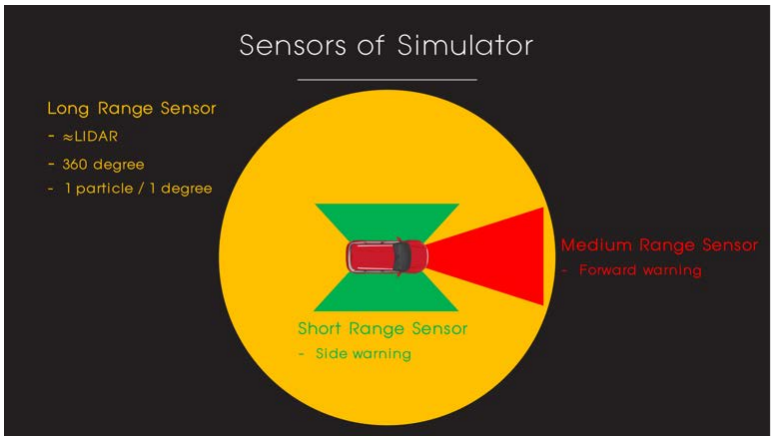


Figura 4. Sistemas ADAS del simulador.

Cada vez que enviamos una acción de movimiento al vehículo (acelerar, frenar, girar o no hacer nada) el simulador retorna un vector de 373 posiciones con la siguiente información:

- 0 - 359: Datos del LIDAR Data (1 por cada grado hasta 360).
- 360 - 362: Left warning, Right Warning, Forward Warning.
- 363: Distancia actual al vehículo de delante.
- 364: Velocidad actual del vehículo de delante.
- 365: Velocidad actual del vehículo.
- 366: Número total de adelantamientos.
- 367: Numero total de cambios de carril.
- 368 - 372: Puntuación o Recompensa.

La puntuación o recompensa también referido como *reward* en inglés, es un valor que retorna el simulador cuando ejecutamos una acción, está puede ser positiva o negativa. Este simulador, ofrece 5 tipos de recompensa diferentes:

- Recompensa longitudinal: Se retorna una recompensa según la siguiente fórmula. Cuando el vehículo se esté desplazando a máxima velocidad (80 Km/h) retornará 1 y cuando se esté desplazando a la velocidad mínima (40 Km/h) un 0.

$$\text{Recompensa longitudinal} = \frac{Velocidad\ actual\ - Velocidad\ mínima}{Velocidad\ máxima - Velocidad\ mínima}$$

Ecuación 1. *Recompensa longitudinal*.

- Recompensa lateral: Se otorga una recompensa de - 0.5 cuando el vehículo cambia de carril.

- Recompensa por adelantamiento: Se otorga una recompensa según la siguiente fórmula

$$\text{Recompensa por adelantamiento} = 0.5 * \textit{Número de adelantamientos}$$

Ecuación 2. *Recompensa por adelantamiento*.

- Recompensa por violación: Se otorga una recompensa de -0.1 cuando se infringe una temeridad. Por ejemplo, si el vehículo cambia al carril izquierdo con otro vehículo cerca en ese mismo carril.

- Recompensa por colisión: Se otorga una recompensa de -10 cuando el vehículo colisiona con otro.

Para realizar esta tesis, se ha optado por ignorar los datos provenientes del radar LiDAR para focalizarse únicamente en la imagen captada por la cámara frontal del vehículo.

Figura 5. Imagen captada por la cámara del vehículo.

Por lo tanto, del vector de observaciones anterior, únicamente se aprovecharán: la velocidad actual del vehículo, el numero de adelantamientos, el numero de cambios de carril y la recompensa. Los 3 primeros para monitorizar los resultados y comprobar que efectivamente el algoritmo esta aprendiendo correctamente y la recompensa para poder entrenar el modelo. En futuros capítulos se detallará el cómo.

Una vez explicado el funcionamiento del simulador se ha programado un pequeño script para verificar su funcionamiento y comprobar que las acciones se ejecuten de forma prevista.

```
1.  import numpy as np
2.  from mlagents.envs import UnityEnvironment
3.
4.  os_ = "Linux"
5.
6.  env_name = "../environment/" + os_ + "/Driving"
7.
8.  env = UnityEnvironment(file_name=env_name)
9.
10. default_brain = env.brain_names[0]
11. brain = env.brains[default_brain]
12. num_actions = brain.vector_action_space_size[0]
13.
14. env_info = env.reset(train_mode=False)[default_brain]
15.
16. for iteration in range(2000):
17.     action = np.random.randint(0, num_actions)
18.     env_info = env.step(action)[default_brain]
```

Este pequeño código, escoge en cada iteración una acción aleatoria de las 5 posibles. Realizará este proceso 2000 veces. De este modo se tendrá como base un algoritmo sin entrenar, es decir, tomando solo acciones aleatorias para después poderlo comprar con los otros mucho más inteligentes.

3 Conceptos previos

En este capítulo se estudiarán 2 conceptos de vital importancia: las redes neuronales y las redes convolucionales. Ambas se utilizaran más adelante, por lo que es necesario entender de forma adecuada su funcionamiento antes de avanzar con el tema principal de esta tesis.

3.1 Red Neuronal Artificial

Las Redes Neuronales Artificiales también conocidas como ANN (*Artificial Neural Network*), son sistemas informáticos inspirados en las redes neuronales biológicas que constituyen los cerebros humanos. Dichos sistemas aprenden a realizar tareas generalmente sin ser programados con una regla específica. Por ejemplo, pueden agrupar datos no etiquetados de acuerdo con diversos patrones y similitudes entre ellos. Las redes neuronales artificiales, también pueden extraer características que alimentan a otros algoritmos; por lo tanto, se puede pensar en estas como componentes de otras aplicaciones de aprendizaje como por ejemplo Reinforcement Learning.

3.1.1 Historia de las Redes Neuronales Artificiales

Aunque el estudio del cerebro humano tiene centenares de años. El primer paso hacia las redes neuronales tuvo lugar en 1943, cuando Warren McCulloch, un neurofisiólogo y Walter Pitts, un joven matemático, escribieron un artículo sobre cómo podrían funcionar las neuronas. Modelaron una red neuronal simple utilizando solo circuitos eléctricos.

En 1958, Frank Rosenblatt, neurobiólogo de Cornell, comenzó a trabajar en lo que actualmente se conoce como Perceptrón. El Perceptrón, que resultó de esta investigación, fue construido totalmente en hardware y es la red neuronal más antigua y que sorprendentemente todavía se utiliza en la actualidad. El perceptrón, calcula una suma ponderada de las entradas, resta un umbral y pasa uno de los dos valores posibles como resultado.

Un año más tarde, en 1959, Bernard Widrow y Marcian Hoff de Stanford desarrollaron modelos que llamaron ADALINE y MADALINE. MADALINE fue la primera red neuronal que se aplicó a un problema del mundo real. Esta red, se utiliza como un filtro adaptativo que elimina los ecos en las líneas telefónicas y todavía está en uso comercial.

Desafortunadamente, estos éxitos anteriores, hicieron que las personas exageraran el potencial de las redes neuronales. Esta exageración excesiva, acompañada con la pobre eficacia de la redes creadas hasta la fecha hicieron que la gente perdiera el interés por este tipo de tecnologías. Además, en el mismo período de tiempo, se escribió un documento que sugería que no podía haber una extensión de la red neuronal de una sola capa a una red de varias capas. Asimismo, muchas personas en el campo estaban usando una función de perdida que no era válida ya que no era diferenciable. Como resultado, la investigación y la financiación disminuyeron drásticamente.

En 1985, el Instituto Americano de Física comenzó lo que se ha convertido en una reunión anual: Redes neuronales para la informática. Para 1987, la primera Conferencia Internacional del Instituto de Ingenieros Eléctricos y Electrónicos (IEEE) sobre redes neuronales atrajo a más de 1,800 asistentes, lo que hizo que la financiación volviera a fluir de nuevo [5].

Con el auge de Internet a principios de la década de 2000, los investigadores tuvieron acceso a más datos (en particular, imágenes) que podrían usar para entrenar estas redes. Las redes neuronales necesitan muchos datos de entrenamiento para llegar a soluciones optimas y con una base de datos casi infinita como es internet, ayudó mucho a mejorar los resultados obtenidos hasta la fecha. Surgiendo así, nuevos algoritmos y optimizaciones que prometían

3.1.2 Funcionamiento

Deep Neural Network o Red Neuronal profunda, es el termino que se utiliza para describir un conjunto de Redes Neuronales puestas en serie, es decir, redes compuestas de varias capas.

Cada capa está formada por nodos. Un nodo es donde ocurre la toda computación, este se activa cuando encuentra estímulos suficientes. Un nodo combina la entrada de los datos con un conjunto de coeficientes, o pesos, que amplifican o amortiguan esa entrada, asignando así importancia a las entradas. Estos productos de peso de entrada se suman y luego el resultado se pasa a través de la llamada función de activación de cada nodo, para determinar si esa señal debería progresar aún más a través de la red y, en qué medida, afectar el resultado final, es decir, un acto de clasificación. Si las señales pasan, la neurona se ha "activado".

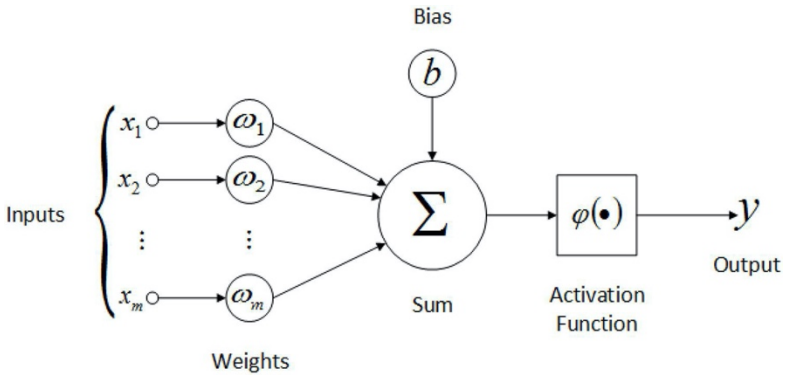


Figura 6. Nodo de una red neuronal profunda.

Una red puede estar formada de diversas capas que contengan una o más neuronas. De este modo, las posibilidades y tamaños de una red pueden ser casi ilimitados. La siguiente figura muestra una red con 1 capa de entrada con 3 neuronas, 2 capas intermedias de 4 neuronas y una capa de salida con 1 neurona [6].

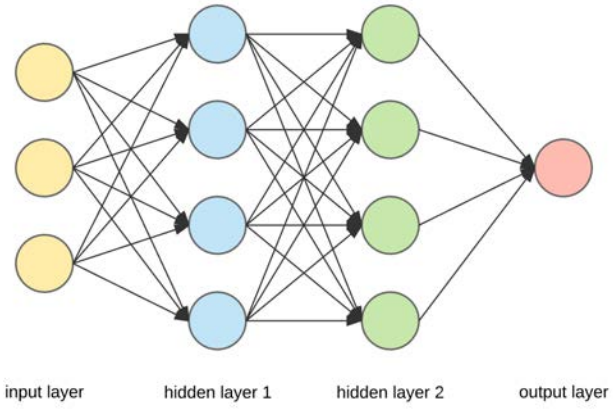


Figura 7. Red Neuronal.

Por lo tanto la operación que realiza cada neurona es la suma de cada una de sus entradas multiplicadas por sus pesos correspondientes sumando a este resultado el valor del bias, todo ello pasado por la función de activación:

$$y_{j=f(b+\sum x_i w_i)}$$

Ecuación 3. ***Operación de una neurona*****.**

3.1.3 Función de activación

Las funciones de activación son ecuaciones matemáticas que determinan la salida de una red neuronal. La función de activación está unida a cada neurona de la red y determina si esta debe activarse o no. Las funciones de activación, también ayudan a normalizar la salida de cada neurona en un rango entre 1 y 0 o entre -1 y 1.

Un aspecto adicional de estas funciones es que deben ser computacionalmente eficientes porque se calculan a través de miles o incluso millones de neuronas para cada muestra de datos.

Hay 3 tipos principales de funciones de activación: binarias, lineales y no lineales.

- Binarias: Si el valor de entrada está por encima o por debajo de cierto umbral, la neurona se activa y envía exactamente la misma señal a la siguiente capa. El problema con una función de este tipo es que no permite salidas de valores múltiples; por ejemplo, no es capaz de clasificar las entradas en diversas categorías en la salida.

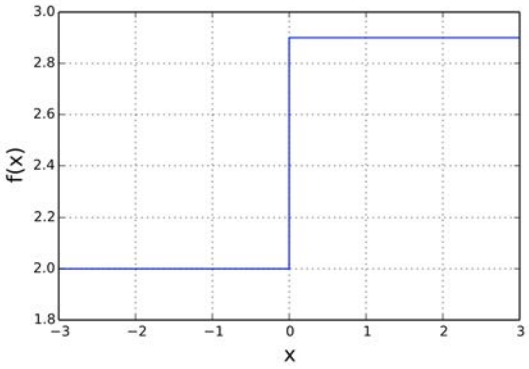


Figura 8. Función de activación binaria.

- Lineales: Toma las entradas, multiplicadas por los pesos de cada neurona, y crea una señal de salida proporcional a la entrada. Tiene dos principales problemas. Al ser una función lineal, la derivada de la función es una constante y no tiene relación con la entrada. Por lo tanto, como se describirá en el siguiente apartado, la red no se puede entrenar para optimizar sus pesos . Además, todas las capas de la red neuronal colapsan en una sola, es decir, con este tipo de funciones de activación, no importa cuántas capas contenga la red neuronal, la última capa será una función lineal de la primera capa.

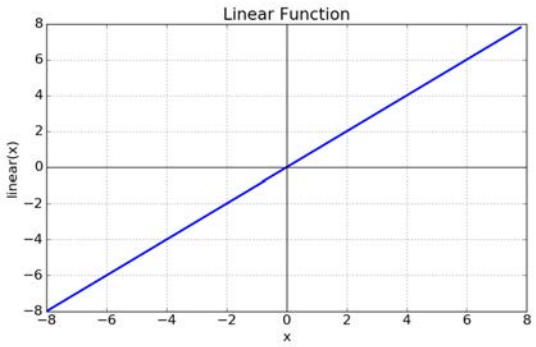


Figura 9. Función de activación lineal.

- No lineales: Los modelos modernos de redes neuronales utilizan funciones de activación no lineales. Permiten que el modelo cree asignaciones complejas entre las entradas y salidas de la red, que son esenciales para aprender y modelar datos complejos. A diferencia de la lineales, su función derivada está relacionada con las entradas. También, permiten apilar en serie múltiples capas de neuronas.

  Hay muchos tipos de activaciones no lineales pero para la realización de esta tesis se han utilizado 2. Estas, se van a explicar con más detalle a continuación.

  - ReLu: Matemáticamente, se define como:

$$f(x) = \max(0, x)$$

Ecuación 4. Ecuación *ReLU*.

ReLU es lineal para todos los valores positivos y cero para todos los valores negativos. Esto implica un coste de cálculo muy reducido. Por lo tanto, el modelo toma menos tiempo para entrenar o correr y por consiguiente, converge más rápido.

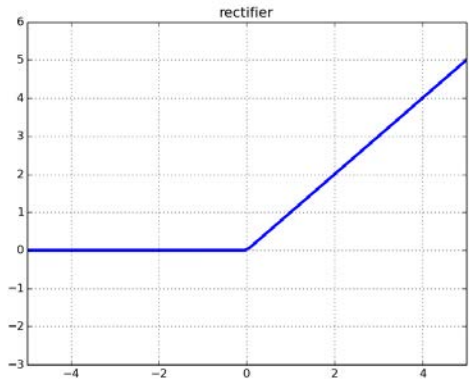


Figura 10. Función de activación ReLU.

  - Softmax:
    La función Softmax es una función de activación que convierte números en probabilidades que suman uno. La función Softmax genera un vector que

representa las distribuciones de probabilidad de una lista de resultados potenciales. Softmax realiza la transformación de n números $x_1, x_2, \dots x_n$

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Ecuación 5. Ecuación *Softmax*.

Las salidas de están función están comprendidas entre 0 y 1 y entre todas ellas suman 1. Forman una distribución de probabilidad.

3.2 Optimización y aprendizaje

La tarea de optimización o aprendizaje de una red neuronal es un proceso laborioso, independientemente del tamaño de la entrada y el número de neuronas que contenga cada capa de la red. Para que este sea exitoso, hay que preparar correctamente el conjuntos de datos en la entrada, calcular la desviación respecto a las salidas exactas y seleccionar los coeficientes de peso para cada una de las neuronas. El entrenamiento consiste en la selección de coeficientes para cada neurona de las capas, de manera que con determinadas señales de entrada se obtiene el conjunto necesario de señales de salida.

Generalmente, una red neuronal artificial suele ser entrenada con métodos supervisados. Esto significa que hay un conjunto de datos de entrenamiento que contiene ejemplos con valores reales.

Las redes neuronales se entrenan en dos etapas: propagación hacia adelante y propagación de errores hacia atrás. Durante la propagación hacia adelante, se hace una predicción de la respuesta. La entrada viaje a través de las neuronas multiplicándose por los pesos y sumándose con el bias. A la salida de cada neurona se le aplicará la función de activación. Al principio del entrenamiento, se inicializan los pesos de cada neurona de forma pseudoaleatoria, por lo que el resultado de la primera propagación hacia delante no será del todo correcta y el error respecto al valor de salida esperado será grande. Para ello se debe entrenar la red. Para conseguirlo, se utiliza lo que se conoce como función de pérdida. Una función de pérdida dice cuán bueno es el modelo que se está entrenando. Esta función, tiene su propia curva y sus propios gradientes. La pendiente de esta curva indica cómo actualizar los parámetros (pesos y bias de cada neurona) para hacer el modelo más preciso. Este proceso es conocido como descenso por gradiente.

Aunque hay multitud de diferentes funciones de perdidas, a modo de ejemplo para esta explicación, se utilizará una de las más empleadas: el error cuadrático medio entre el valor esperado de salida y, menos el valor predicho por la red.

$$f(w, b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (wx_i + b))^2$$

Ecuación 6. *Error cuadrático medio*.

Donde $wx_i + b$ es la salida de la neurona, N es el numero de datos de entrada, e y son las salidas que se esperan obtener una vez entrenada entrenar la red.

Matemáticamente, se puede representar el gradiente como la derivada parcial de la función de perdida respecto los pesos w y la derivada parcial de la función de perdida respecto al bias b.

$$\frac{df}{dw} = \frac{1}{N}\sum -2x_i(y_i - (mx_i + b))$$

$$\frac{df}{db} = \frac{1}{N}\sum -2(y_i - (mx_i + b))$$

Ecuación 7. ***Error cuadrático medio diferenciado*.**

Se itera a través de los datos utilizando los nuevos valores w y b. Cada derivada parcial, indica la pendiente de la función de perdida en la posición actual. Lo que es lo mismo, la dirección en la que se debe mover para actualizar los parámetros. El grado de la actualización de los pesos y el bias está controlado por el factor de aprendizaje. Un factor de aprendizaje demasiado bajo implica que la tasa de error se mantiene alta y no desciende lo suficientemente rápido, por lo que se tendría que esperar meses o años para obtener un buen rendimiento. En cambio, una factor de aprendizaje demasiado alto implica que el rendimiento divergiera.

Hay diversos algoritmos que optimizan los pesos de la red a partir del calculo del gradiente. Entre los más utilizados destacan: SGD, RMSProp y Adam.

- SGD: Está basado en el descenso de gradiente. El algoritmo empieza en un punto aleatorio de la función de perdida y avanza con pasos (tan grandes como el factor de aprendizaje indique) en la dirección opuesta al gradiente en cada iteración hasta idealmente encontrar el mínimo de la función de perdida. La optimización SGD elige aleatoriamente un punto de datos del conjunto de datos completo en cada iteración para reducir enormemente los cálculos y mejorar la rapidez de entrenamiento.

- RMSProp: SGD tiene un problema en que las tasas de aprendizaje tienen que escalar con 1 / T para lograr la convergencia, donde T es el número de iteración. Es decir, después de un tiempo se dan pasos muy pequeños y no se avanza demasiado hacia el mínimo de la función. RMSProp intentan evitar esto ajustando automáticamente el tamaño del gradiente: a medida que el gradiente promedio se hace más pequeño, el coeficiente en la actualización SGD se hace más grande para compensar. Se consigue dividiendo el factor de aprendizaje entre un promedio exponencialmente decreciente de gradientes cuadrados.

- Adam: Adam añade, el concepto “impulso” algoritmo RMSProp. Adam acumula el gradiente de los gradientes pasados para determinar la dirección a seguir.

Los criterios de utilización de cada optimizador no están definidos y se besan en métodos empíricos de prueba y error. Para cada conjunto de datos diferentes, se deberá probar cual es el optimizador que mejor satisface las necesidades [7].

3.3 Red Neuronal Convolucional

Las redes neuronales convolucionales son redes neuronales artificiales en las cuales las neuronas corresponden a campos receptivos casi de la misma manera que las neuronas en la corteza visual primaria de los cerebros biológicos. Este tipo de red es una variante de un perceptrón multicapa, pero debido a que su aplicación se realiza en una matriz bidimensional, son muy efectivos para tareas de visión artificial como la clasificación y segmentación de imágenes.

3.3.1 Historia

En 1959, David Hubel y Torsten Wiesel describieron el concepto de "células simples" y "células complejas" en la corteza visual humana. Propusieron que ambos tipos de células se utilizan en el reconocimiento de patrones cuando el ojo humano capta una imagen.

En la década de los 80, el Dr. Kunihiko Fukushima se inspiró en el trabajo de Hubel y Wiesel sobre células simples y complejas, y propuso el modelo conocido como: *neocognitron*. El modelo *neocognitron* incluye componentes denominados "células S" y "células C". Estas ya no son células biológicas, sino más bien operaciones matemáticas. La idea general era capturar el concepto propuesto por Hubel y Wiesel y convertirlo en un modelo computacional para el reconocimiento de patrones visuales.

El primer trabajo sobre redes neuronales convolucionales modernas (CNN) se produjo en la década de los 90, inspirado en el *neocognitron*. Yann LeCun demostró que un modelo CNN puede usarse con éxito para el reconocimiento de caracteres escritos a mano.

A lo largo de la década de 1990 y principios de 2000, los investigadores llevaron a cabo más trabajos sobre el modelo CNN. Alrededor de 2012, las CNN disfrutaron de un gran aumento de popularidad después de que una CNN llamada AlexNet lograra etiquetar y clasificar todo tipo de imágenes con un rendimiento sobresaliente.

A lo largo de los últimos años, las CNN han logrado un excelente trabajo en la clasificación de imágenes, realizar reconocimiento facial y analizar imágenes médicas [8].

3.3.2 Funcionamiento

Las Redes Neuronales Convolucionales (CNN) tienen una arquitectura diferente a las redes neuronales normales. A diferencia de las Redes Neuronales Artificiales explicadas en el apartado anterior, las capas en una Red Neuronal Convolucional se organizan en 3 dimensiones: ancho, alto y profundidad. Además, las neuronas en una capa no se conectan a todas las neuronas de la siguiente capa, sino solo a una pequeña región.

Las CNN tiene 2 bloques claramente diferenciados:

- Boque de extracción: En este bloque, la red realizará una serie de convoluciones y operaciones de agrupación durante las cuales se detectarán ciertas características en la imagen de entrada.

- Bloque de clasificación: Aquí, las capas están completamente conectadas y servirán como clasificador sobre estas características extraídas.

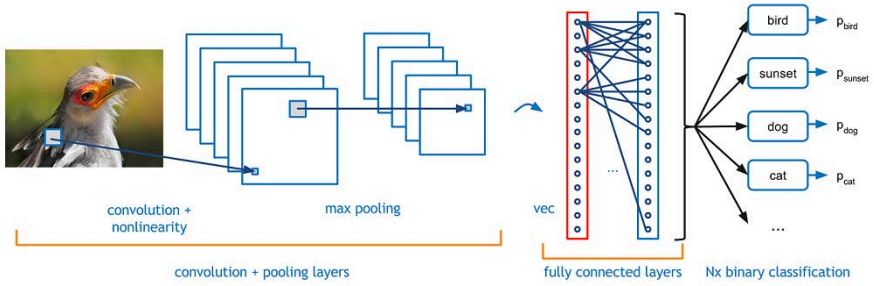


Figura 11. Esquema red neuronal convolucional.

3.3.2.1 Bloque de extracción

La convolución es uno de los principales actores en este bloque de extracción. La operación matemática de convolución transforma dos funciones de forma lineal y continua para producir una nueva tercera función.

En el caso de una CNN, la convolución se realiza en los datos de entrada con el uso de un filtro o *Kernel* para luego producir un mapa de características. Hay muchos tipos de filtros que sirven para detectar características especificas de cada imagen.

Se ejecuta una convolución deslizando este filtro sobre los valores de la entrada. En cada posición, se realiza una multiplicación matricial y se suma el resultado en el mapa de características. Se Realizan numerosas convoluciones en la entrada, donde cada operación utiliza un filtro diferente. Esto da como resultado diferentes mapas de características. Al final, se toman todos estos mapas de características y se juntan como la salida final de la capa. Al igual que cualquier otra red neuronal, se utiliza una función de activación para que la salida no sea lineal.

Otro termino importante es el *stride*. El *stride*, son los pasos que el filtro de convolución se mueve cada vez. Por ejemplo, cunado el *stride* vale 1, significa que el filtro se desliza píxel por píxel. Al aumentar el tamaño, el filtro se desliza sobre la entrada con un intervalo mayor y, por lo tanto, hay una menor superposición entre las celdas. La siguiente imagen muestra la convolución utilizando un filtro 3x3.

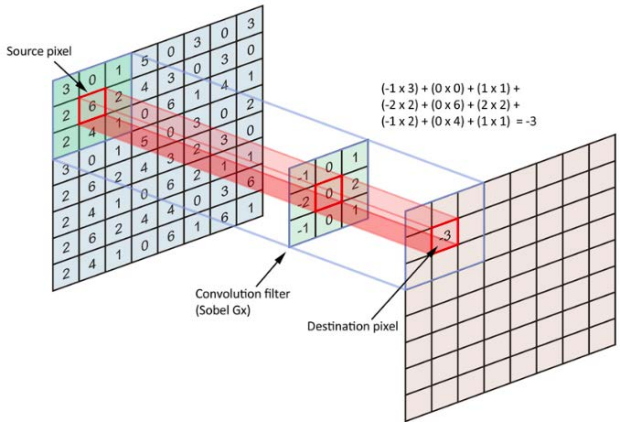


Figura 12. Convolución.

Después de una capa de convolución, es común agregar una capa de *pooling*. Su función es la de reducir el número de parámetros y, por consecuencia, disminuir también el cálculo en la red. El tipo más utilizado es el *max-pooling*, que como su nombre indica, toma el valor máximo en cada ventana. Los tamaños de ventana deben especificarse de antemano [9].

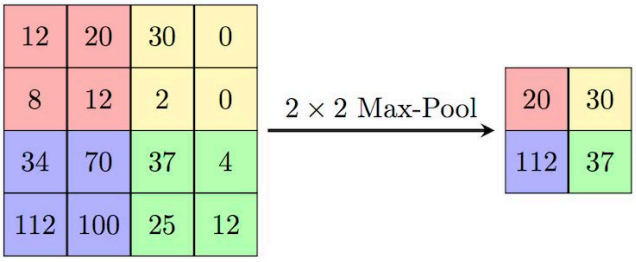


Figura 13. Max-Pooling.

3.3.2.2 Bloque de clasificación

Después de las capas de convolución y *pooling*, el bloque de clasificación consta de unas pocas capas completamente conectadas. Sin embargo, estas capas completamente conectadas solo pueden aceptar datos de 1 dimensión. Para convertir nuestros datos 3D a 1D, utilizamos la función *flatten()* en Python. Esta bloque es esencialmente lo mismo que una Red Neuronal Artificial comentada en el capítulo anterior. Por lo tanto, el proceso de entrenamiento es el mismo también.

4 Reinforcement Learning

Machine learning (ML) o aprendizaje automático es una parte de la informática donde la eficiencia de un sistema se mejora al realizar repetidamente las tareas mediante el uso de datos. Existen tres técnicas de aprendizaje automático claramente diferenciadas: aprendizaje supervisado, no supervisado y de refuerzo.

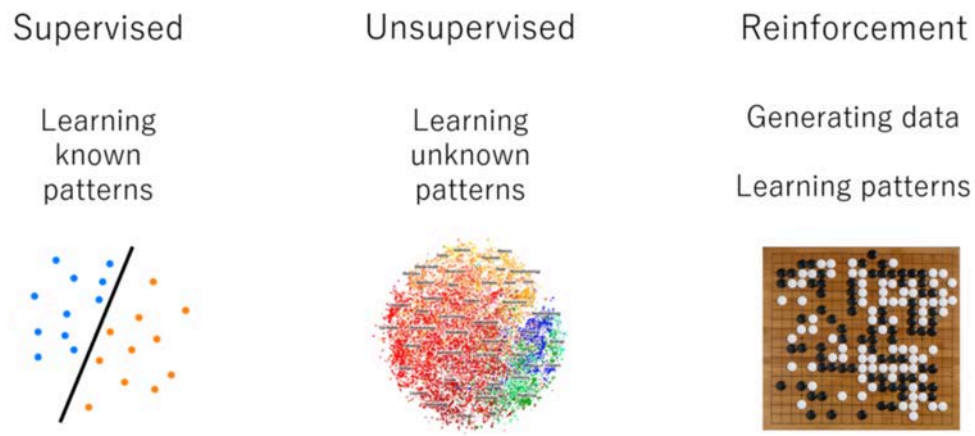


Figura 14. Campos de ML.

- Aprendizaje supervisado: La información en el aprendizaje supervisado se proporciona como un conjunto de datos etiquetado, un modelo puede aprender de él para proporcionar el resultado del problema fácilmente. Es frecuentemente utilizado en problemas de clasificación de imágenes o problemas de regresión.

- Aprendizaje no supervisado: Este algoritmo de aprendizaje es completamente opuesto al aprendizaje supervisado. En resumen, no hay un conjunto de datos etiquetados en el aprendizaje no supervisado. Su objetivo principal es explorar los patrones subyacentes y predecir la salida. Aquí, básicamente, se proporcionan datos a la máquina y se busca localizar características ocultas y agrupar los datos de una manera que tenga sentido.

- Aprendizaje por refuerzo: Aquí los algoritmos aprenden a reaccionar a un entorno por sí mismos.

Es en este último en los que se focalizará la investigación de esta tesis. Más adelante en el capítulo, se expondrán los conceptos y definiciones básicas de RL para después focalizarse en la teoría de los 3 algoritmos principales que se implementarán.

4.1 Introducción

El aprendizaje por refuerzo o Reinforcement Learning (RL) es un área del aprendizaje automático o Machine Learning (ML) basada en el entrenamiento de modelos de ML para tomar una secuencia de decisiones. Un agente aprende a lograr un objetivo en un entorno incierto y potencialmente complejo. En el RL, una inteligencia artificial se enfrenta a una situación de juego. La inteligencia artificial emplea prueba y error para encontrar una solución al problema.

Esta, obtiene recompensas o sanciones por las acciones que realiza. Su objetivo es maximizar la recompensa total.

Aunque el diseñador establece la política de recompensas, es decir, las reglas del juego, no le da al modelo pistas ni sugerencias sobre cómo resolverlo. Depende del propio modelo descubrir cómo realizar la tarea para maximizar la recompensa, comenzando con pruebas totalmente aleatorias y terminando con tácticas sofisticadas. El aprendizaje por refuerzo es actualmente la forma más efectiva de potenciar la creatividad de una máquina. A diferencia de los seres humanos, la inteligencia artificial puede acumular experiencia de miles de juegos paralelos si se ejecuta un algoritmo en una infraestructura suficientemente potente.

La siguiente figura muestra dos entidades principales de RL: agente y entorno, y sus canales de comunicación: acciones, recompensas y estado.

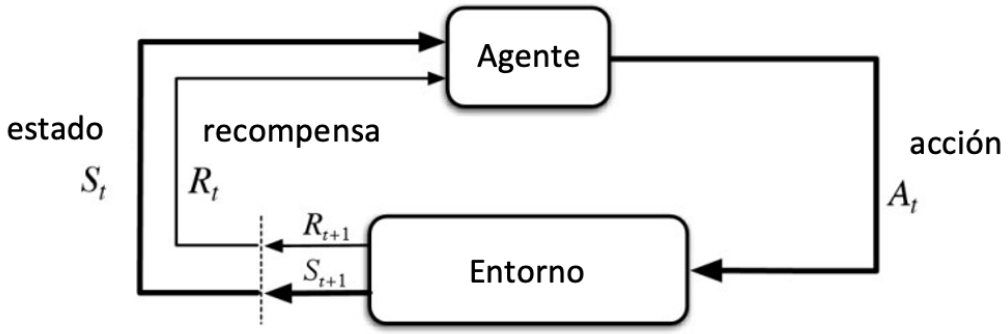


Figura 15. Entidades principales RL.

4.1.1 Agente

Un agente es algo que interactúa con el entorno, este ejecuta ciertas acciones, hace observaciones y recibe recompensas por ello. En la mayoría de los escenarios de RL, el agente es el software. Ya que se supone que resuelve algún problema de una manera más o menos eficiente.

4.1.2 Entrono

El entorno es todo lo que se comprende fuera del agente. En el caso de esta tesis, el entorno sería el simulador de conducción. La comunicación del agente con el entorno se limita a la recompensa (obtenida del entorno), acciones (ejecutadas por el agente y entregadas al entorno) y observaciones (información adicional de la recompensa que el agente recibe del entorno).

4.1.3 Acción

Las acciones son actuaciones que un agente puede hacer sobre el entorno. Las acciones en esta tesis, por ejemplo, son los 5 movimientos permitidos por las reglas del simulador (acelerar, frenar, girar para cambiar al carril derecho o izquierdo y no hacer nada).

Se distinguen entre dos tipos de acciones: discretas o continuas. Las acciones discretas forman el conjunto finito de cosas mutuamente excluyentes que un agente puede hacer, como moverse

hacia la izquierda o hacia la derecha. Las acciones continuas tienen algún valor asociado, como la acción de un automóvil de girar la rueda con un ángulo y dirección. Ángulos diferentes podrían conducir a un escenario diferente un segundo después, así que simplemente girar la rueda no es suficiente

4.1.4 Recompensa

Es un valor escalar que se obtiene cuando se ejecuta una acción sobre el entorno. Como se explicó anteriormente, la recompensa puede ser positiva o negativa, grande o pequeña. El propósito de la recompensa es decirle al agente qué tan bien se ha comportado.

El término "refuerzo" proviene del hecho de que la recompensa obtenida por un agente debe reforzar su comportamiento de manera positiva o negativa. La recompensa suele ser local, lo que significa que refleja el éxito de la actividad reciente del agente y no todos los éxitos logrados por el agente hasta ahora. Por supuesto, obtener una gran recompensa por alguna acción no significa que un segundo después no enfrentará consecuencias dramáticas como resultado de sus decisiones anteriores. Por otro lado, la recompensa no debe verse como algo secundario o sin importancia ya que la recompensa es la fuerza principal que impulsa el proceso de aprendizaje del agente. Si una recompensa es incorrecta, ruidosa o ligeramente desviada del objetivo principal, entonces hay una posibilidad de que el entrenamiento vaya en una dirección equivocada.

4.1.5 Estado

El estado son observaciones que el entorno proporciona al agente para informarle lo que sucede alrededor suyo. Las observaciones pueden ser relevantes para la próxima recompensa o pueden no serlo. En el caso a desarrollar, la observación es un *frame* captado por la cámara instalada en el vehículo.

4.2 Fundamentos RL previos

En esta sección, se expone la representación matemática y la notación de los formalismos que se acaban de discutir en el apartado anterior. Luego, utilizando estos conceptos como base, se explorarán otros conceptos más avanzados de RL, incluyendo estado, episodio, política, valor y ganancia.

4.2.1 Procesos de decisión de Markov

Para llegar a entender bien los Procesos de decisión de Markov, este capitulo está estructurado desde los conceptos más básicos, empezando con la definición de proceso de Markov (MP), luego extendiéndola aplicando recompensas, lo que lo convertirá en un proceso de recompensa de Markov (MRP). Para finalmente, agregar acciones, lo que llevará a un Proceso de decisión de Markov (MDP).

4.2.1.1 Proceso de Markov (MP)

El proceso de Markov, también conocido también como la cadena de Markov. Se utiliza para modelar probabilidades utilizando información que pueden codificarse en el estado actual. Algo pasa de un estado a otro de forma pseudoaleatoria. Cada estado tiene una cierta probabilidad de transición al otro estado, por lo que cada vez que se encuentre en un estado y se desee

realizar una transición, una cadena de Markov puede predecir resultados basados en datos de probabilidad preexistentes.

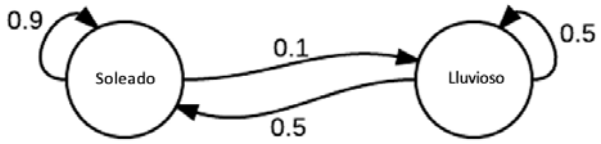


Figura 16. Cadena de Markov.

Un ejemplo para entender el funcionamiento del proceso de la cadena de Markov es el que se muestra en la anterior imagen. En este ejemplo, se puede observar que hay dos estados: "soleado" y "lluvioso". Si se quiere saber cual es la probabilidad que el día siguiente sea lluvioso sabiendo que el día actual es soleado, se puede observar que la cadena de Markov indica que hay una probabilidad de 0.1 que esto pase. Siguiendo este patrón, se puede ver que probablemente habrá muchos días soleados agrupados, seguidos de una cadena más corta de días lluviosos. Obviamente, este no es un ejemplo del mundo real.

Un modelo de Markov es un modelo estocástico con la propiedad de que los estados futuros están determinados solo por el estado actual; en otras palabras, el modelo no tiene memoria; solo sabe en qué estado se encuentra ahora, no ninguno de los estados que ocurrieron anteriormente.

Este proceso se puede modelar utilizando una matriz de probabilidad donde cada fila de la matriz representa la probabilidad de pasar del estado original a cualquier estado sucesor.

4.2.1.2 Proceso de recompensa de Markov (MRP)

Hasta ahora se ha visto cómo la cadena de Markov define la dinámica de un entorno utilizando un conjunto de estados y una matriz de probabilidad de transición. Pero el objetivo del aprendizaje por refuerzo es el de maximizar la recompensa, por ello, se le introduce la recompensa a la cadena de Markov.

La recompensa se puede representar de varias formas. La forma más general es tener otra matriz cuadrada, similar a la matriz de transición, con una recompensa dada por la transición del estado i al estado j, que reside en la fila i y la columna j.

Para cada transición en una cadena MP, se añade una cantidad extra: la recompensa. Así que ahora, todas las observaciones tienen un valor de recompensa asociado a cada transición del sistema. Para cada episodio, se define el retorno en el tiempo, t, como:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Ecuación 8. *Retorno*.

También se agrega al modelo un factor de descuento γ, que es un número único de 0 a 1. Este valor da más o menos importancia a las recompensas futuras. Si γ es igual a 1, entonces el retorno, es igual a la suma de todas las recompensas posteriores y al agente tiene una visibilidad

perfecta de cualquier recompensa posterior. Si γ es igual a 0, el retorno será solo una recompensa inmediata sin ningún estado posterior.

Otro termino importante que se utilizará más adelante es el valor de estado, se calcula como la esperanza matemática del retorno para cualquier estado:

$$V(s) = \mathrm{E}[G|S_t = s]$$

Ecuación 9. *Valor de estado*.

Para cada estado, s, el valor de estado, $\mathrm{V}(s)$, es el retorno promedio que se obtiene siguiendo el proceso de recompensa de Markov.

4.2.1.3 Proceso de decisión de Markov (MDP)

Para convertir el proceso de recompensa de Markov (MRP) en un MDP, se debe agregar un conjunto de acciones A, que debe ser finito. Este es el espacio de acción del agente. En segundo lugar, se necesita condicionar la matriz de transición con acciones, lo que básicamente significa que la matriz de transiciones necesita una dimensión de acción adicional, que la convierte en un cubo.

Finalmente, se necesita agregar acciones a la matriz de recompensa de la misma manera que con la matriz de transición. La matriz de recompensas dependerá no solo del estado sino también de la acción. En otras palabras, la recompensa que obtiene el agente ahora dependerá no solo del estado en el que termina, sino también de la acción que conduzca a este estado.

4.2.2 Política

La política es un conjunto de reglas que controla el comportamiento del agente. En el caso del simulador de conducción de esta tesis, tres posibles políticas podrían ser:

- Moverse de forma aleatoria.
- Solo acelerar.
- No hacer nada.

Como se ha comentado anteriormente, el objetivo principal del agente en RL es obtener la mayor cantidad de recompensa posible. Entonces, las diferentes políticas pueden dar diferentes cantidades de retorno, lo que hace importante encontrar una buena política.

Formalmente, la política se define como la distribución de probabilidad sobre las acciones para cada estado posible:

$$\pi(a|s) = P[A_t = a|S_t = s\,]$$

Ecuación 10. *Política*.

4.2.3 Ecuación de Bellman

Para entender la ecuación de Bellman, se propone el siguiente ejemplo: un agente observa el estado S_0 y tiene N acciones disponibles. Cada acción conduce a otro estado, $S_1 \dots S_N$, con una recompensa respectiva, $r_1 \dots r_n$. Además, se supone que se conocen los valores, V_i, de todos los estados conectados al estado S_0.

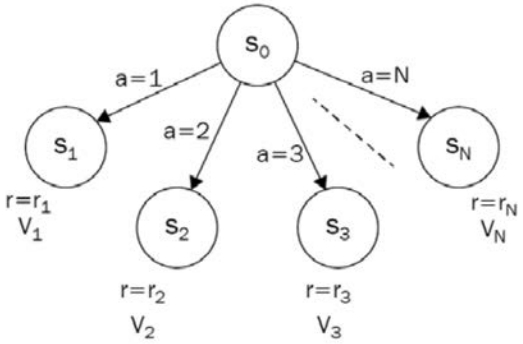


Figura 17. Ejemplo 1.

Para tomar la mejor decisión, el agente necesita calcular los valores para cada acción y elegir el resultado máximo equivalente a la acción con mayor recompensa. Por lo tanto el valor del estado S_0 es:

$$V_0 = max_{a\epsilon 1\dots N}(r_a + \gamma V_a)$$

Ecuación 11. ***Valor de estado S0 en ejemplo 1*****.**

Se extiende el concepto anterior en un entorno en la que una misma acción tiene la probabilidad de acabar en un estado o en otro. Este tipo de entornos se conocen como entorno estocásticos y se utilizan cuando el entorno es incierto. Lo que se debe hacer es calcular el valor esperado para cada acción, en lugar de simplemente tomar el valor del siguiente estado.

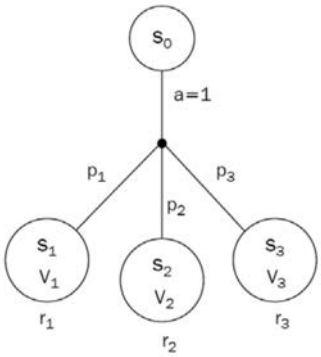


Figura 18. Ejemplo 2.

$$V_0 = \sum_{s\in S} p_{a,0\to s}(r_{s,a} + \gamma V_s)$$

Ecuación 12. ***Valor del estado S0 en ejemplo 2*****.**

Donde $p_{a,i\to j}$ es la probabilidad de ejecutar una acción a en el estado i y transicionar hacia el estado j. Combinando las dos ecuaciones anteriores, se llega a la siguiente ecuación general:

$$V_0 = max_{a\epsilon A} \sum_{s\in S} p_{a,0\to s}(r_{s,a} + \gamma V_s)$$

Ecuación 13. *Valor de estado S0*.

El valor del estado es igual a la acción que da la máxima recompensa inmediata esperada posible más la recompensa descontada a largo plazo. Esta ecuación de Bellman es la base no solo en RL sino también en una programación dinámica mucho más general, que es un método ampliamente utilizado para resolver problemas prácticos de optimización.

Por ultimo, queda por definir el valor de acción o $\mathrm{Q}(\mathrm{s}, \mathrm{a})$. $\mathrm{Q}(\mathrm{s}, \mathrm{a})$ es igual a la recompensa total que se puede obtener al ejecutar la acción a en el estado s. Se puede definir de forma recursiva análogamente a las ecuaciones anteriores como:

$$Q(s,a) = r(s,a) + \gamma max_{a' \in A} Q(s',a')$$

Ecuación 14. *Valor de acción*.

Donde s' es el estado futuro y a' es la acción que se ha realizado para llegar a ese estado.

Los valores Q son mucho más convenientes en la práctica, ya que para el agente, es mucho más fácil tomar decisiones sobre acciones basadas en Q que en V. En el caso de Q, para elegir la acción basada en el estado, el agente solo necesita calcular Q para todas las acciones disponibles y elegir la acción con el mayor valor de Q. Para hacer lo mismo usando los valores de los estados, el agente necesita conocer no solo los valores, sino también las probabilidades de las transiciones. En la práctica, rara vez se conocen de antemano, por lo que el agente necesita estimar las probabilidades de transición para cada acción.

4.3 Cross-Entropy

El método de entropía cruzada se centra principalmente en la política. Una red neuronal predice una política específica, que básicamente dice para cada observación qué acción debe tomar el agente.

Esta abstracción hace que el agente sea relativamente simple: se necesita pasar una observación del entorno a la NN, obtener una distribución de probabilidad sobre las acciones y realizar un muestreo aleatorio utilizando una distribución de probabilidad para llevar a cabo una acción. Este muestreo aleatorio agrega aleatoriedad al agente, lo cual es bueno, ya que al comienzo del entrenamiento, cuando los pesos son aleatorios, el agente se comporta de manera aleatoria.

Durante la vida del agente, su experiencia es representada con episodios. Cada episodio es una secuencia de observaciones que el agente ha recibido del entorno. En el caso del simulador utilizado para la investigación, un episodio son todas las observaciones que hace el vehículo hasta que colisiona con otro o haya recorrido 2600 metros. Para cada acción que ha ejecutado, recibe una recompensa. Entonces, para cada episodio o partida, se puede calcular la recompensa total que el agente ha acumulado.

Los pasos se han seguido para implementar el método de entropía cruzada son los siguientes:

1. Jugar N cantidad de episodios usando el modelo y entorno actual (al principio con pesos aleatorios).
2. Calcular la recompensa total para cada episodio y decidir un límite de recompensa.
3. Desechar todos los episodios con una recompensa por debajo del límite establecido.

4. Entrenar con los episodios no desechados utilizando las observaciones como entrada y acciones emitidas como salida deseada.
5. Repetir desde el paso 1.

Este método selecciona los mejores episodios y entrena sobre ellos para ir mejorando poco a poco para conseguir cada vez mayores recompensas. Este método es bastante fácil de implementar gracias a su sencillez y ofrece resultados bastante robustos en entornos sencillos.

Además, tiene como particularidad la utilización de la función de perdida de la entropía cruzada para entrenar la red que predice la política. A diferencia de la función de error cuadrático medio explicada en el capítulo 3.2, está se define como:

$$f = -\frac{1}{N}\sum_{x}[y\ln(a) + (1-y)\ln(1-a)]$$

Ecuación 15. *Entropía cruzada*.

Donde a es la salida de la neurona defienda anteriormente, N es el numero de datos de enteramiento, la suma se realiza sobre las entradas, x e y son las salidas que se esperan obtener una vez entrenada entrenar la red.

La entropía cruzada es siempre positiva y tiende a cero a medida que la neurona mejora en el cálculo de la salida deseada. Por esta razón, esta función de perdida es utilizada comúnmente en problemas de clasificación mientras que el error cuadrático medio se utiliza en problemas de regresión.

4.4 Algoritmos basados en valor

Este tipo de algoritmos, como su nombre indica, se centrar en los estados de valor, o lo que es lo mismo en los valores Q.

4.4.1 Deep Q-Learning (DQN)

En DQN, se utiliza una red neuronal para aproximar la función Q. El estado o observación se proporciona a la red como entrada y el valor de todas las acciones posibles se genera como salida.

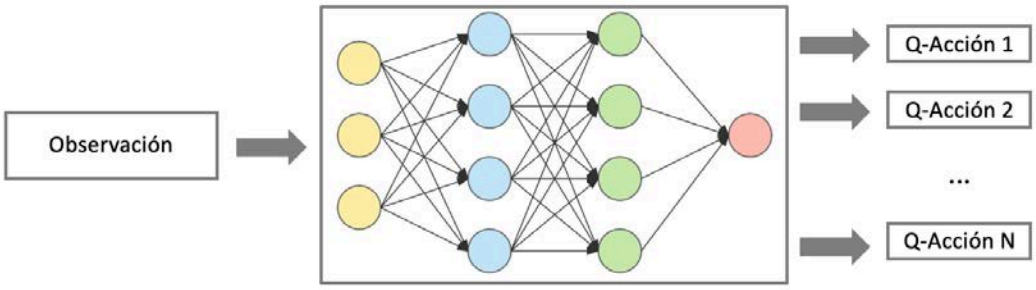


Figura 19. Red DQN.

La misma red está calculando el valor de Q objetivo (utilizando la ecuación de Bellman con el estado futuro) y el valor de Q predicho (utilizando el estado actual). Se necesitan estos dos

valores de Q para poder entrenar la red, pero utilizando una misma red para predecirlos, podría haber mucha divergencia entre ambos. Entonces, en lugar de usar una red neuronal para aprender, se utilizan dos.

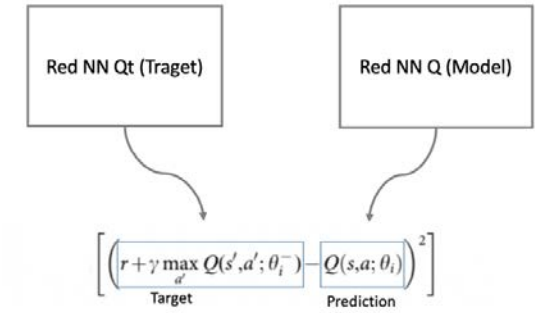


Figura 20. Uso de dos redes en DQN.

La imagen muestra la función de perdida del error cuadrático medio para entrenar la red. Donde la red Qt estima el valor objetivo pero con parámetros fijos. Es decir, cada N episodios, se actualizarán los pesos de la red Qt con los pesos de la red Q. Esto lleva a un entrenamiento más estable porque mantiene la red que calcula objetivo con menos variaciones.

Finalmente, en lugar de ejecutar este proceso para cada par consecutivo de estado/acción, el agente almacena en cada paso los datos de transición: estado, acción, recompensa y estado futuro o siguiente. Para entrenar el modelo, se muestrea un número aleatorio de transiciones de este búfer. Esto dará un subconjunto dentro del cual la correlación entre las muestras es baja ya que este muestreo contendrá transiciones distintitas que corresponden a N distintas transiciones espaciadas en el tiempo.

En modo resumen, los pasos a seguir para implementar el algoritmo DQN son los siguientes:

1. Se inicializa los parámetros para las dos redes $Q(s, a)$ y $Qt(s, a)$ con pesos aleatorios.
2. Con probabilidad ε, seleccionar una acción aleatoria, a; de lo contrario, seleccionar la acción como: $a = argmax_a Q(s, a)$.
3. Ejecutar la acción a y observar la recompensa r, y el siguiente estado, s '.
4. Almacene la transición (s, a, r, s ') en el búfer de reproducción.
5. Muestrear un conjunto de transiciones del búfer de reproducción.
6. Para cada transición en el búfer, calcular $y = r + \gamma max_{a' \in A} Qt(s', a')$. Si el episodio ha terminado, calcular $y = r$ (ya que no hay recompensa futura posible).
7. Calcular la pérdida utilizando el error cuadrático medio: $L = (y - Q(s, a))^2$.
8. Actualizar la red Q(s, a) utilizando el algoritmo ADAM.
9. Cada N episodios, copiar los pesos de Q a Qt.
10. Repetir desde el paso 2 hasta que la red converja.

4.5 Algoritmos basados en política

El tema central en el aprendizaje por valores Q del anterior apartado, es el valor del estado (V) o el valor de acción (Q). Si se conoce el valor, la decisión en cada paso es simple: solo se actúa basándose en el valor, y eso garantiza una buena recompensa total al final del episodio. Pero los

algoritmos basados en la política, como su nombre indica, están basados en la política como un ente propio, es decir, la política es lo que se está buscando cuando se está resolviendo un problema de RL. En los métodos basados en políticas, en lugar de aprender una función de valor que dice cuál es la suma esperada de recompensas dado un estado y una acción, se aprende directamente la política que asigna el estado a la acción. Es decir, se seleccionan acciones sin utilizar una función de valor.

Esto significa que se trata directamente de optimizar una función de política π sin preocuparse por una función de valor. Directamente se parametriza π. Como se verá más adelante, se puede usar una función de valor para optimizar los parámetros de la política. Pero en este tipo de algoritmos, la función de valor nunca será usada para directamente seleccionar una acción.

4.5.1 Advantage Actor Critic (A2C)

Se parte de una política π que tiene parámetros θ. Esta π genera una distribución de probabilidad de acciones.

$$\pi_\theta(a|s) = P[a|s]$$

Ecuación 16. ***Política*****.**

Para medir qué tan buena es esta política, se utiliza una función llamada función objetivo o puntuación $J(\theta)$ que calcula la recompensa esperada de la política. En entornos estocásticos (cuando una única acción tiene la probabilidad de acabar en un estado o en otro) se define como:

$$J_{avg\ R}(\theta) = E_\pi(r) = \sum_s d(s) \sum_a \pi_\theta(s,a) R_a^s$$

Ecuación 17. ***Función puntuación o objetivo A2C*****.**

Donde:

- $\sum_a \pi_\theta(s,a)$ es la probabilidad de que se tome la acción a desde el estado s bajo esta política.
- R_a^s es la recompensa inmediata que se obtendrá.
- $d(s)$ es la probabilidad de estar en el estado s y se define como la división entre el numero da apariciones de este estado entre el numero de apariciones de todos los estados.

$$d(s) = \frac{N(s)}{\sum_{s'} N(s')}$$

Ecuación 18. ***Probabilidad de estar en el estado s*****.**

Una vez definida la función $J(\theta)$ que dice qué tan buena es la política. Ahora, se quiere encontrar un parámetro θ que maximice la función objetivo. La función objetivo se define como la suma de la recompensa esperada utilizando la política.

$$\theta^* = argmax_\theta\, J(\theta) = argmax_\theta\, E_{\pi\theta}[\sum_t R(s_t, a_t)]$$

Ecuación 19. ***Función objetivo maximizando θ .***

La función de puntuación también puede ser definida como la esperanza de la función recompensa siguiendo la política:

$$J(\theta) = E_{\pi}[R(\tau)]$$

Ecuación 20. ***Función objetivo como esperanza*****.**

Donde:

- τ son secuencias de estado, acción.

Maximizar la función objetivo significa encontrar la política óptima, para ello, se utilizará el ascenso de gradiente en los parámetros de la política. Al contrario que el descenso de gradiente, el ascenso de gradiente indica la dirección del aumento más pronunciado de la función, lo que llevará a maximizar la función objetivo. El gradiente de la función J(θ) será:

$$\nabla_{\theta} J(\theta) = \sum_{\tau} \nabla_{\theta} \pi(\tau; \theta) R(\tau)$$

Ecuación 21. ***Gradiente de la función objetivo*****.**

En un entorno estocástico, la política genera una distribución de probabilidad $\pi(\tau; \theta)$. Dados los parámetros actuales de θ, se obtiene la probabilidad de tomar la serie de pasos (s0, a0, r0 …).

Diferenciar está función de probabilidad es difícil, por está razón, para que sea mucho más sencilla de diferenciar, se transforma en un logaritmo. Se utiliza la siguiente igualdad:

$$\nabla \log x = \frac{\nabla x}{x}$$

Ecuación 22. ***Igualdad del gradiente de log(x)*****.**

Se puede expresar el gradiente de la política como:

$$\nabla_{\theta} \pi(\tau; \theta) = \frac{\nabla_{\theta} \pi(\tau; \theta)}{\pi(\tau; \theta)} \pi(\tau; \theta) = \nabla_{\theta} \log(\pi(\tau; \theta)) \pi(\tau; \theta)$$

Ecuación 23. ***Gradiente de la política*****.**

Sustituyendo el valor anterior en la función objetivo se obtiene:

$$\nabla_{\theta} J(\theta) = \sum_{\tau} \nabla_{\theta} \log(\pi(\tau; \theta)) \pi(\tau; \theta) R(\tau)$$

Ecuación 24. ***Función objetivo con el nuevo gradiente de la política*****.**

Convirtiendo la sumación de nuevo a una esperanza:

$$\nabla_{\theta} J(\theta) = E_{\pi}[\nabla_{\theta}(\log \pi(\tau|\theta)) R(\tau)]$$

Ecuación 25. ***Función objetivo convertido en sumación*****.**

Se puede concluir sobre los gradientes de política:

$$\text{Gradiente de la política} = E_{\pi}[\nabla_{\theta}(\log \pi(s, a, \theta)) R(\tau)]$$

Ecuación 26. ***Gradiente de la política*****.**

Donde:

- $\pi(s, a, \theta)$ es la función de política.
- $R(\tau)$ es la función recompensa.

$$\text{Regla de actualización} = \nabla\theta = \alpha\nabla_\theta\big(\log\pi(s, a, \theta)\big)R(\tau)$$

Ecuación 27. ***Regla de actualización*****.**

Donde:

- $\nabla\theta$ es el cambio de parámetros.
- α es el factor de aprendizaje.

Analizando las igualdades anteriores, se puede observar que el gradiente de la política, indica cómo se debería cambiar la distribución de la política a través de parámetros θ si se quiere maximizar la puntuación.

La función $\mathrm{R}(\tau)$ es un valor escalar que puntúa el valor de las acciones tomadas:

- Si $\mathrm{R}(\tau)$ es alto querrá decir que, en promedio, se han tomado acciones que conducen a recompensas altas. Se quiere aumentar la probabilidades de estas acciones.
- Si $\mathrm{R}(\tau)$ es bajo querrá decir que, en promedio, se han tomado acciones que conducen a recompensas bajas, y por lo tanto, reducir la probabilidad de estas acciones.

Con lo anteriormente explicado se ha definido las características de un algoritmo llamado Vainilla o Montecarlo. Este tiene un problema: como solo se calcula la recompensa $R(\tau)$ al final de cada episodio, y se promedian todas las acciones (incluso si algunas de las acciones tomadas fueron muy malas), para tener una buena política, se necesitan infinidad de muestras lo que resulta en un aprendizaje excesivamente lento y más en el simulador utilizado, ya que la duración de cada episodio es relativamente larga.

Para solucionar esto, se utiliza el método A2C que combina los algoritmos basados en el valor (como el DQN) y los algoritmos basados en la política. La solución reside en actualizar la función $R(\tau)$ en cada paso y no solo al final de cada episodio utilizando los valores Q. La nueva actualización de política sustituyendo la recompensa por el valor de Q es:

$$\nabla\theta = \alpha\nabla_\theta\big(\log\pi(s_t, a, \theta)\big)Q(s_t, a_t)$$

Ecuación 28. ***Regla de actualización con valores Q*****.**

Los métodos basados en valores tienen una gran variabilidad. Para reducir este problema, se utiliza la función de ventaja en lugar de la función de Q en la ecuación anterior. La ventaja se define como:

$$A(s, a) = Q(s, a) - V(s)$$

Ecuación 29. ***Ventaja*****.**

Donde:

- $Q(s, a)$ es la recompensa total descontada al ejecutar la acción a en el estado s.
- $V(s)$ el valor del estado s.

La ventaja indica en que medida es mejor la acción que se ha tomada basándose en la acción que se espera que ocurra en ese estado. En otras palabras, esta función calcula la recompensa adicional que se obtiene si se realiza esta acción.

El problema de implementar esta función de ventaja como se ha definido, es que requiere dos funciones de valor: $\mathrm{Q(s, a)}$ y $\mathrm{V(s)}$. Afortunadamente, se puede usar el error TD para aproximar la función de ventaja. Con este truco, se puede cambiar el valor de acción Q y usar solo una red que predice los dos valores de estado. De lo contrario se necesitarían 2 redes para calcular la ventaja (una para el valor de acción y otra para el valor de estado).

$$A(s, a) = r + \gamma V(s') - V(s)$$

Ecuación 30. ***Ventaja expresada con los valores de estado*.**

En resumen y para finalizar este apartado, la idea principal de este algoritmo es dividir el modelo en dos: La red de políticas (que devuelve una distribución de probabilidad de acciones) se llama Actor y otra red llamada Crítico, que dicta cuán de buenas fueron las acciones tomadas.

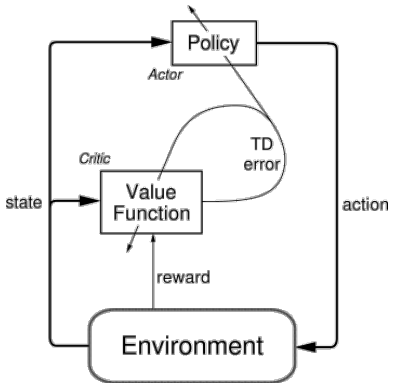


Figura 21. Esquema A2C.

La red neuronal del Actor toma como entrada el estado y estima la mejor acción según la política. Por otro lado, la red neuronal del Crítico, evalúa la acción calculando la función de valor. Estos dos modelos mejoran simultáneamente su propio papel a medida que pasa el tiempo. El resultado es que la arquitectura general aprenderá de manera más eficiente que los dos métodos por separado [10].

1. Inicializar los parámetros de red, θ, con valores aleatorios.
2. Reproducir N pasos en el entorno utilizando la política actual, π_θ y guardando s_t , a_t , r_t
3. R = 0 si se alcanza el final del episodio.
4. Para $i = \ t - 1 \ldots t_{start}$
1. $R \leftarrow r_i + \gamma R$
2. Acumular el gradiente de política: $\partial\theta_\pi \leftarrow \partial\theta_\pi + \nabla_\theta \log \pi_\theta(a_i|s_i)(R - V_\theta(s_i))$
3. Acumular el gradiente de valor: $\partial\theta_v \leftarrow \partial\theta_v + \frac{\partial(R - V_\theta(s_i))^2}{\partial\theta_v}$

4. Actualizar los parámetros de la red utilizando los gradientes acumulados, moviéndose en la dirección de los gradientes de política, $\partial\theta_\pi$, y en la dirección opuesta de los gradientes de valor, $\partial\theta_v$.
5. Repetir desde el paso 2 hasta alcanzar la convergencia.

4.5.2 Proximal Policy Optimization (PPO)

El artículo que primero menciona este algoritmo data de finales de 2017. *Proximal Policy Optimization Algorithms* (arXiv:1707.06347) [11].

La idea principal de PPO es evitar tener una actualización de política demasiado grande, como sucede en A2C. Para conseguirlo, solo se modificará la función objetivo respecto al algoritmo A2C. Esta nueva función objetivo, utiliza una relación que proporciona la diferencia de la política nueva y la antigua. Hacer esto, asegurará que la actualización de la política no sea demasiado grande.

Recordar que la función de puntuación de A2C es la siguiente:

$$J(\theta) = \mathrm{E}[\log\pi_\theta(a_t|s_t)A_t]$$

Ecuación 31. ***Función puntuación o objetivo*****.**

La idea de esta función, utilizada en A2C, es hacer un ascenso de gradiente (equivalente a tomar un descenso de gradiente negativo) de esta forma el agente se ve obligado a tomar acciones que conducen a mayores recompensas y evitar malas acciones. Sin embargo, no es del todo estable y hay demasiada variabilidad en cada paso.

La idea detrás de PPO es mejorar la estabilidad del entrenamiento de la red Actor al limitar la actualización de la política en cada paso. Para poder limitar la actualización de política, PPO introduce una nueva función objetivo llamada: *Clipped surrogate objective function*, que restringirá el cambio de política en un rango pequeño usando un *clip*.

Si se define el ratio entre las dos políticas como:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

Ecuación 32. ***Ratio de políticas*****.**

La nueva función objetivo incluyendo el *clip* será:

$$J_\theta^{clip} = E[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)]$$

Ecuación 33. ***Función puntuación PPO*****.**

Este función, limita la relación entre la política antigua y la nueva para estar en el intervalo [1 - ε, 1 + ε], por lo que al variar ε, se puede limitar el tamaño de la actualización. Si se toma el mínimo del objetivo *clipped* y no *clipped*, el objetivo final será un límite más pequeño del objetivo no recortado. Hay dos casos a considerar:

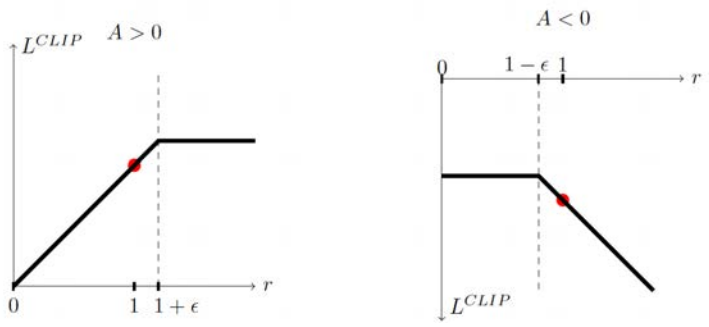


Figura 22. Clip.

- $A_t > 0$: Significa que la acción es mejor que el promedio de todas las acciones en ese estado. Por lo tanto, se debe incitar a la política a aumentar la probabilidad de tomar esa acción en ese estado. Sin embargo, debido al *clip*, $r_t(\theta)$ solo crecerá hasta $1 + \varepsilon$. Esto significa que esta acción no puede ser cien veces más probable en comparación con la política anterior (ya que el *clip* lo limita). Esto se hace porque no se quiere actualizar demasiado la política. Tomar medidas en un estado específico es solo un intento, no significa que siempre dará lugar a una recompensa positiva en el futuro, por lo que no se quiere ser demasiado codiciosos porque también puede conducir a una mala política.

- $A_t < 0$: En este caso la acción no ha tenido una buena recompensa. En consecuencia, , $r_t(\theta)$ disminuirá (porque la acción es menos probable para la política actual que para la anterior) pero debido al clip, $r_t(\theta)$ solo disminuirá hasta $1 - \varepsilon$. Además, análogamente al caso anterior, no se hará un gran cambio en la política reduciendo por completo la probabilidad de tomar esa acción, ya que a veces, una mala acción puede conducir a una secuencia de buenas acciones (por ejemplo, frenar unos segundos el coche para dejar pasar al coche del carril adyacente, para así, poder cambiar de carril y avanzar al coche que obstruía delante).

4.6 Valor contra política

En este apartado se comentaran las ventajas y desventajas de estos dos tipos de algoritmos basados en la valor y los basados en la política.

- Espacio de acción continuo
  Los algoritmos basados en políticas son más efectivos en espacios de acción continuos. En DQN, se asigna una recompensa para cada acción posible, en cada paso de tiempo, dado el estado actual. Pero, en estados de acción continuos se pueden tener infinidad de acciones. Por ejemplo, un coche autónomo puede girar las ruedas en una infinidad de grados posibles (10°, 10.5°, 45.8°, 50.2°…) lo que supondría calcular un valor de Q para cada ángulo de giro posible. Por otro lado, en los métodos basados en políticas, simplemente se ajustan los parámetros directamente.

- Convergencia
  Generalmente, los algoritmos basados en políticas tienen mejor convergencia. Es decir, llegan a una solución optima más rápidamente que los basados en valor. Con los métodos basados en valor se puede tener una gran variación durante el entrenamiento.

Esto se debe a que la elección de la acción puede cambiar drásticamente por un mínimo cambio en los valores de acción estimados. Por otro lado, con el gradiente de políticas, solo se sigue el gradiente para encontrar los mejores parámetros. Ahora bien, los gradientes de políticas tienen una gran desventaja. Muchas veces, convergen en un máximo local en lugar de en el máximo global. En lugar de DQN, que siempre trata de alcanzar el máximo. Además, los gradientes basados en la política suelen converger más lentamente y pueden tardar más tiempo para entrenar.

- Entornos estocásticos
  No se necesita exploración explícita. En aprendizaje por valores Q, se utiliza una estrategia de épsilon para explorar el entorno y evitar que el agente se atasque con una política no óptima. En A2C, con las probabilidades devueltas por la red, la exploración se realiza automáticamente. Al principio, la red se inicializa con pesos aleatorios y devuelve una distribución de probabilidad uniforme. Esta distribución corresponde al comportamiento aleatorio del agente. Los algoritmos basados en el gradiente de política pueden aprender mejor una política estocástica.

- Búfer de memoria
  Los métodos basados en valor como DQN necesitan guardar las transiciones en un búfer de memoria. Esto significa que, en lugar de ejecutar Q-learning en pares de estado / acción a medida que ocurren durante la simulación, el agente almacena los datos en un búfer para después ir cogiendo muestras aleatorias de este. Para su correcto funcionamiento, este búfer suele ser de un tamaño grande, como ya se ha comentado, en el caso de esta tesis, se están guardando 90000 transiciones conteniendo cada una: 8 imágenes de 80x80 (4 correspondientes al estado presente y 4 al estado futuro), un valor de recompensa y un valor para representar la acción. Si se hacen los cálculos, teniendo en cuenta que cada numero se ha codificado como float32 (4 Bytes), el *array* de 90000 posiciones abarcará 18Gb de la memoria RAM.

  En cambio, los métodos de gradiente de políticas no pueden ser entrenarnos con datos obtenidos de la política anterior. Esto es bueno y malo al mismo tiempo. Lo bueno es que tales métodos generalmente convergen más rápido. El lado negativo es que generalmente requieren mucha más interacción con el entorno que los métodos basados en valor, como DQN.

- Una sola red para cada modelo
  No se necesita una red de destino Qt. En A2C, se utilizan valores Q, pero se obtienen de la experiencia en el entorno. En cambio, en DQN, se utiliza una red Qt para romper la correlación en la aproximación de los valores Q.

5 Desarrollo

En este capítulo se describirá el desarrollo más en detalle de cada algoritmo. Cabe destacar que no todo el código escrito se muestra en las siguientes explicaciones, sólo las partes más relevantes. Además, algunas partes del código se han editado sutilmente para reducirlo y facilitar a mejorar su mejor comprensión. El código completo de los 3 algoritmos se adjuntará junto a la memoria de la tesis.

5.1 Librerías y entrono de desarrollo

Para desarrollar el código se ha utilizado el entrono de Anaconda junto con Jupyter lab con Python 3.7. La ventaja que aporta un entorno estilo del tipo *Notebook* que utiliza Jupyter Lab respecto a un script convencional es que se puede ejecutar bloques de código por separado y observar el output en tiempo real. Lo cual, ayuda mucho para depurar el código y encontrar errores más fácilmente. Además, cuando se comparte un *Notebook*, este también guarda la salida y gráficos que se hayan generado por cada bloque de código. Por esta y muchas razones más, Jupyter Lab es uno de los entornos de desarrollo más populares actualmente.

Antes de empezar con el desarrollo del código se mostrará un resumen tanto de las librerías utilizadas como del sistema operativo utilizado para entrenar al simulador.

Librerías utilizadas:

- Keras: Esta librería se utiliza para crear y utilizar las redes neuronales a muy alto nivel. Ofrece APIs consistentes y simples, minimiza la cantidad de acciones del usuario requeridas y proporciona mensajes de error claros y procesables. También cuenta con una amplia documentación y guías para desarrolladores [12].

- Numpy: Esta librería consiste en funciones matemáticas para operar con vectores o matrices a alto nivel [13].

- Cv2: OpenCV es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Se ha utilizado para convertir en blanco y negro las imágenes captadas por la cámara [14].

- TQDM: Se utiliza para mostrar una barra de progreso durante el entrenamiento. También muestra el tiempo transcurrido y una estimación del tiempo restante para la finalización del entrenamiento [15].

- Random: Librería que permite generar un número aleatorio [16].

- Mlagents: Librería que permite establecer la comunicación con el entrono Unity [17].

- Tensorflow: Se utiliza Tensorboard librería contenida dentro de Tensorflow para mostrar graficas de los resultados. Tanto del simulador (velocidad, número de adelantamientos, recompensa y cambios de carril) como de algunos parámetros de la propia red neuronal [18].

- Collections: Este módulo implementa tipos de datos de contenedores especializados que ofrecen alternativas a los contenedores integrados de uso general de Python, *dict*, *list*, *set* y *tuple* [19].

- Datetime: El módulo proporciona clases para manipular fechas y horas. Se utilizaría principalmente para escribir la fecha en el nombre de los *logs* para poderlos reconocer más fácilmente [20].

Aunque las especificaciones del ordenador utilizado no son las más punteras del mercado, son suficientes para poder soportar a carga de trabajo. Detalles del sistema utilizado:

- Sistema operativo: Ubuntu 20.04
- CPU: AMD FX8350
- GPU: GTX 970
- RAM: 16Gb DDR3

5.2 Optimizaciones

En este apartado se definirán 2 técnicas que harán que la red aprenda y converja más rápido y con mejores resultados. Estas, se implementarán de igual forma en los 4 algoritmos que se explicarán a más adelante.

5.2.1 Frame Stacking

Se conoce como *Frame Stacking* el proceso en el que se colocan N *frames* individuales uno encima del otro. La siguiente figura muestra 16 *frames* consecutivas capturadas por el simulador. El proceso consiste en escoger 1 de cada 4 imágenes separadas por 3 *frames* entre ellas hasta tener 4 imágenes que servirán para entrenar la red. Con esto, se consigue dar una perspectiva de "movimiento" a la red neuronal, dicho de otro modo, la red a entrenar recibe 4 imágenes a la vez en vez de 1. Ya que con una sola imagen, la red no puede saber si por ejemplo, el coche de delante está acelerando o frenando, es decir, no puede percibir movimiento. Aplicando esta técnica, la red es capaz de entender mejor el entorno y aprender mejor que acciones tomar en cada momento.

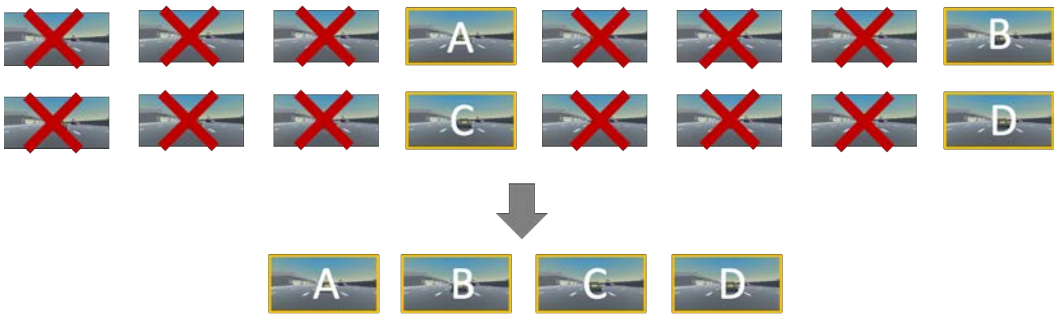


Figura 23. Fotogramas por muestra de entrenamiento.

Así pues, cada vez que el vehículo realice una acción, la nueva imagen captada por la cámara del coche, se añadirá a un *array* de 16 posiciones. Siempre que se quiera entrenar a la red, se

escogerán 4 imágenes no consecutivas del *array*, en este caso, se escogerá una imagen cada 4 imágenes, lo que supondrá 4 imágenes en total (ya que el *array* tiene 16 posiciones). Al inicializar o hacer un *reset* al simulador, se llenará esta *array* con la misma imagen en todas las posiciones; a falta de imágenes futuras.

```
1.  def frame_stack(total_obs, obs_set):
2.      obs_stack_obs = np.zeros([OBS_SIZE, OBS_SIZE, 1])
3.      frame = get_image(total_obs)
4.      obs_stack_obs[:,:,0] = frame
5.      obs_set.append(obs_stack_obs)
6.      obs_stack = np.zeros((OBS_SIZE, OBS_SIZE, 4))
7.
8.      for i in range(4):
9.          obs_stack[:,:,i:(i+1)] = obs_set[-1 - (4 * i)]
10.
11.     del obs_set[0]
12.
13.     obs_stack = np.uint8(obs_stack)
14.     obs_stack = np.float32(normalize_image(obs_stack))
15.
16.     return obs_stack, obs_set
```

Esta técnica, esta basada en el articulo: “Human-level control through deep reinforcement learning, Volodymyr Mnih”. Este articulo escrito en el 2015 es aparentemente “viejo” para el mundo de RL pero, fue el primero en implementar esta técnica [21].

5.2.2 Compresión de la imagen

La imagen captada por el simulador es una imagen en color RGB. Por lo general, para una red neuronal, una imagen en color no agregara ninguna información nueva sustancial respecto a un imagen en blanco y negro. La información contenida en una imagen en escala de grises es suficiente para la clasificación. Por ejemplo, una imagen en escala de grises de la cara de una persona ya es suficiente para poder identificarla.

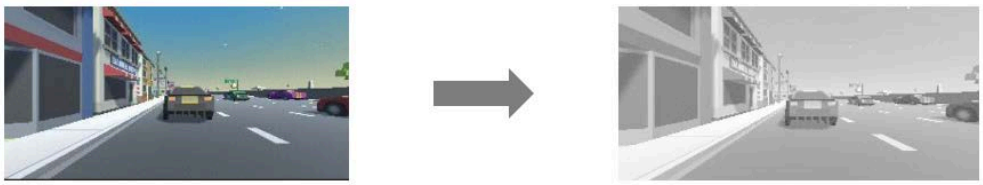


Figura 24. Conversión de imagen a blanco y negro.

Además, el uso de una imagen en color aumentará el número de entradas por 3, correspondientes a los valores de color R, G, B. Esto tiene 2 problemas:

La red necesitará muchos más parámetros a entrenar debido al aumentado su tamaño (para un vector de entrada más grande). Entrenar un mayor número de parámetros es difícil y computacionalmente más caro, se necesitará muchos más datos de entrenamiento y número mucho mayor de iteraciones.

A pesar de todos los esfuerzos de regularización, posiblemente la red tenderá a sobreajustarse. Se conoce como sobreajuste cuando un modelo aprende los detalles y el ruido en los datos de entrenamiento en la medida en que impacta negativamente el rendimiento del modelo en los nuevos datos de entrada.

```
1.  def get_image(total_obs):
2.      frame = 255 * total_obs.visual_observations[0]
3.      frame = np.uint8(frame)
4.      frame = np.reshape(frame, (frame.shape[1], frame.shape[2], 3))
5.      frame = cv2.resize(frame, (OBS_SIZE, OBS_SIZE))
6.      frame = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
7.      frame = np.reshape(frame, (OBS_SIZE, OBS_SIZE))
8.      return frame
```

El código muestra como se captura la imagen RGB de dimensiones 80x80 (con una forma de (80,80,3)) del entorno y se utiliza la librería CV2 para convertirla a blanco y negro (a una forma de (80,80)) donde cada posición de la matriz es un valor de 1 a 255 indicando la escala de gris. Después se normalizan los valores de cada uno de los pixeles para que los resultados estén entre 0 y 1.

```
1.  def normalize_image(frame):
2.      return (frame - (255.0/2)) / (255.0/2)
```

5.3 Cross-Entropy

Este apartado describe el código del algoritmo de Entropía Cruzada. Resaltando las partes del código más relevantes. A continuación, se muestra un esquema que resume el código de una forma más visual.

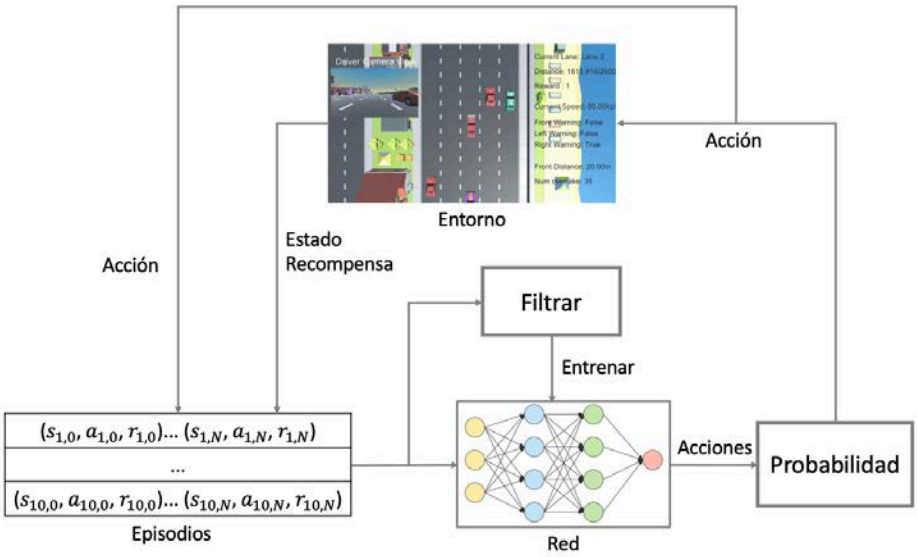


Figura 25. Esquema entropía cruzada.

Las variables para este algoritmo son las siguientes:

```
#Training parameters
BATCH_SIZE = 10
PERCENTILE = 70
MAX_EPISDOES = 40
OBS_SIZE = 80

#Logging parameters
LOG_DIR = "logs_cross_entropy/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=LOG_DIR)

Episode = namedtuple("Epsiode", field_names=["reward","steps","vehicle_speed","overtakes","lane_changes"])
EpisodeStep = namedtuple("EpisodeStep",field_names=["observation","action"])
```

- BATCH_SIZE: Número de episodios para cada iteración.
- PERCENTILE: Porcentaje para filtrar los mejores episodios.
- MAX_EPISODES: Número de iteraciones.
- OBS_SIZE: Tamaño de la imagen, en este caso es de 80x80.
- LOG_DIR: Directorio donde se guardarán los logs.
- Tensorboard_callback: Se utiliza para mostrar las variables en Tensorboard.
- Episode: *Tuple* que guarda episodios.
- EpisodeStep: *Tuple* que guarda los pasos de cada episodio.

El bucle principal, itera sobre los 10 episodios con la función *iterate_batches*. Una vez completados los 10 episodios, se filtran los que han dado mejores recompensas con la función *filter_batch*. Se entrena la red con estos episodios "elite" y se registran los resultados para ver la evolución durante el entrenamiento. Finalmente, el proceso anterior se repite hasta que se haya llegado a 40 iteraciones (10 episodios * 40 iteraciones = 400 episodios). Llegadas a las 40 iteraciones se guardan los parámetros de la red neuronal convolucional para poder probar el rendimiento del modelo después.

```
logging()
episode_count = 1
for iter_no, batch in enumerate(iterate_batches(env, default_brain, net, BATCH_SIZE)):
    obs, acts, reward, reward_m, speed_m, overtakes_m, lane_changes_m = filter_batch(batch, PERCENTILE)

    #Train the net with the best episodes (observations + actions)
    net.fit(np.array(obs),np.array(acts))

    print("%d: reward_mean=%.1f speed_mean=%.1f, overtakes_mean=%.1f, lane_changes_mean=%1.f"  % (iter_no, reward_m, speed_m, overtakes_m, lane_changes_m))
    episode_count += 1

    #Log results for Tensorboard display
    tf.summary.scalar("Speed", speed_m, step=iter_no)
    tf.summary.scalar("Overtake", overtakes_m, step=iter_no)
    tf.summary.scalar("Lane Changes", lane_changes_m, step=iter_no)
    tf.summary.scalar("Reward", reward_m, step=iter_no)

```

```
    if episode_count == MAX_EPISDOES:
        print("Finished training!")
        net.save_model()
        env.close()
        break
```

A continuación, la función *iterate_batches* predice las probabilidades de ejecutar cada acción dependiendo de la observación actual. Se escoge la acción respeto a dichas probabilidades y se ejecuta en el simulador, guardando el par observación-acción en EpisodeSteps. Cuando el episodio ha acabado, se añaden los valores de: recompensa, EpisodeSteps, velocidad del vehículo, adelantamientos y cambios de carril en la variable Episode. Se repite este proceso para 10 episodios. Una vez completados, se vuelve al bucle principal con la función *yield*.

```
def iterate_batches(env, default_brain, net, batch_size):
    batch = []
    overtake_list = []
    lanechange_list = []
    episode_reward = 0.0
    episode_vehicle_speed = []
    episode_steps = []
    total_obs = env.reset(train_mode=True)[default_brain]
    obs_stack, obs_set = reset_frame_stack (total_obs)

    while True:
        action_probabilities = net.get_pred(np.array(obs_stack).reshape(-1,80,80,4))
        action = np.random.choice(5, p=action_probabilities)

        #Get info from enviroment and ad it to the frame stack
        total_actual_obs = env.step(action)[default_brain]
        new_obs_stack, obs_set = frame_stack (total_actual_obs, obs_set)

        reward = total_actual_obs.rewards[0]
        episode_reward += reward
        done = total_actual_obs.local_done[0]

        #We save the observation that was chosen to take the action
        episode_steps.append(EpisodeStep(observation = new_obs_stack, action = action))

        obs_stack = new_obs_stack

        if not done:
            episode_overtakes = total_actual_obs.vector_observations[0,-7]
            episode_lane_changes = total_actual_obs.vector_observations[0,-6]
            episode_vehicle_speed.append(100 * total_actual_obs.vector_observations[0,-8])

        else:
            batch.append(Episode(reward = episode_reward, steps = episode_steps, vehicle_speed = np.mean(episode_vehicle_speed), overtakes = episode_overtakes, lane_changes = episode_lane_changes))

            episode_steps, episode_reward, obs_stack, obs_set = end_of_episode_reset()

            if len(batch) == batch_size:
                yield batch
```

```
                batch = []
```

Finalmente, la función *filter_batch*, como su nombre indica, se encarga de filtrar los mejores episodios de los 10 jugados. Para ello, se utiliza la función de la librería numpy: percentile. El percentil es una medida de posición usada en estadística que indica, una vez ordenados los datos de menor a mayor, el valor de la variable por debajo del cual se encuentra un porcentaje dado de observaciones en un grupo. De estos 10 episodios se obtiene un umbral de recompensa equivalente al percentil 70. Es decir, se busca un umbral donde solo un 30% de las recompensas están por encima del percentil 70. Con este umbral se filtran los mejores episodios y se retornan al bucle principal para entrenar la red.

```
def filter_batch(batch, percentile):
    #Filter the best episodes for training
    rewards = list(map(lambda s: s.reward, batch))
    speed = list(map(lambda s: s.vehicle_speed, batch))
    overtakes = list(map(lambda s: s.overtakes, batch))
    lane_changes = list(map(lambda s: s.lane_changes, batch))

    #Calculate the percentile of the rewards
    reward_bound = np.percentile(rewards, percentile)
    reward_mean = float(np.mean(rewards))
    speed_mean = float(np.mean(speed))
    overtakes_mean = float(np.mean(overtakes))
    lane_changes_mean = float(np.mean(lane_changes))

    train_obs = []
    train_act = []

    for example in batch:
        #If epsidoe is not "elite" return back to main loop
        if example.reward < reward_bound:
            continue
        train_obs.extend(map(lambda step: step.observation, example.steps))

        train_act.extend(map(lambda step: step.action, example.steps))

    return train_obs, train_act, reward_bound, reward_mean, speed_mean, over
takes_mean, lane_changes_mean
```

El bloque de clasificación de la red neuronal convolucional se ha definido con los siguientes parámetros y se mantendrá constante en los 4 algoritmos de esta tesis:

Las 3 capas que constituyen el bloque son:

1. 32 filtros con un tamaño de 8x8 , un *Stride* de 4 y activación Relu.
2. 64 filtros con un tamaño de 4x4 , un *Stride* de 2 y activación Relu.
3. 64 filtros con un tamaño de 3x3 , un *Stride* de 1 y activación Relu.

```
def create_model(self):
        model = Sequential()
        model.add(Conv2D(filters=32, kernel_size=8, strides = 4, input_shape
 = (OBS_SIZE, OBS_SIZE,4), activation='relu'))
        model.add(Conv2D(filters=64, kernel_size=4, strides = 2, activation=
'relu'))
```

```
5.          model.add(Conv2D(filters=64, kernel_size=3, strides = 1, activation=
    'relu'))
6.          model.add(Flatten())
7.          model.add(Dense(units=512, activation='relu'))
8.          model.add(Dense(units=5, activation='softmax')) #Softmax to avoid ne
    gative probability and make all values sum 1
9.
10.         model.compile(loss="sparse_categorical_crossentropy", optimizer=Adam
    (lr=0.0025))
11.         model.summary()
12.         return model
```

En la primera capa se especifica el tamaño de entrada. En este caso es de (80,80,4). Keras es capaz de detectar automáticamente (si se ha dimensionado correctamente la entrada) el número de muestras de entrenamiento. Es decir, el tamaño del vector de entrada de la red tiene 4 dimensiones: (numero de muestras de entrenamiento, 80, 80, 4). Donde 80 es el tamaño de la imagen y 4 es el numero de imágenes por muestra de enteramiento. De este modo, no hace falta especificar las muestras de entrenamiento y así ,con una misma red se puede utilizar para predecir una muestra y ser entrenada con por ejemplo 32 muestras.

Por otro lado, la capa de clasificación contiene una entrada con las salidas del bloque de extracción, una capa oculta con un número de neuronas que ira variando para cada algoritmo y tantas neuronas en la salida como número de acciones. En este caso se compone de 5 salidas, con una activación Softmax.

Para compilar el modelo se debe especificar la función de pérdida y el factor de aprendizaje que se utilizarán para entrenar la red. La función de pérdida para este algoritmo es: entropía cruzada. Esta es una buena función pérdida cuando se está trabajando en tareas de clasificación. Destacar que la función de activación de la ultima capa es *Softmax* por la misma razón, ya que se está buscando tener en la salida una distribución de probabilidad de acciones, es decir clasificar las acciones de mejor a peor.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
conv2d_1 (Conv2D) (None, 19, 19, 32) 8224
_________________________________________________________________
conv2d_2 (Conv2D) (None, 8, 8, 64) 32832
_________________________________________________________________
conv2d_3 (Conv2D) (None, 6, 6, 64) 36928
_________________________________________________________________
```

Figura 26. Parámetros red convolucional.

Se ha escogido esta red basándose en el articulo: *Playing Atari with Deep Reinforcement Learning by V. Mnih.* Este articulo que se ha utilizado como base para este algoritmo, enseña a un agente a jugar el juego de PONG de la consola Atari utilizando DQN.

5.4 DQN

En esta sección, se describe el algoritmo DQN para controlar el vehículo del simulador. Para explicar mejor este apartado, la siguiente figura muestra un esquema del proceso a seguir.

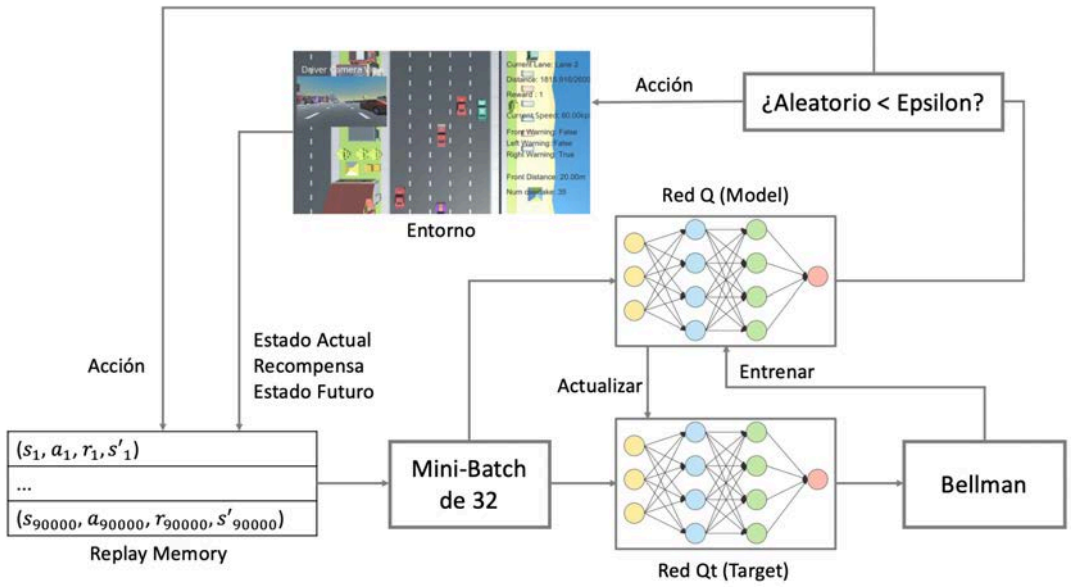


Figura 27. Esquema DQN.

Las constantes para este algoritmo son las siguientes:

```
#Training parameters
DISCOUNT = 0.99 #Gamma
REPLAY_MEMORY_SIZE = 90_000  # Maximum size of the deque of steps (explodes my RAM)
MIN_REPLAY_MEMORY_SIZE = 50_000
MINIBATCH_SIZE = 32
ACTION_SIZE = brain.vector_action_space_size[0]
OBS_SIZE = 80
UPDATE_TARGET_EVERY = 5
epsilon = 1.00
EPSILON_DECAY = 0.995
MIN_EPSILON = 0.1
EPISODES = 400

#Logging parameters
LOG_DIR = "logs_DQN/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=LOG_DIR)
```

- DISCOUNT: El parámetro gamma utilizado en la ecuación de Bellman para calcular la recompensa futura esperada.
- REPLAY_MEMORY_SIZE: El tamaño del *array* (*deque*) de experiencia.
- MIN_REPLAY_MEMORY_SIZE: El tamaño mínimo de muestras para empezar a entrenar la red.
- MINIBATCH_SIZE: El tamaño de muestreo de transiciones del *array* de experiencia que se utilizará para entrenar la red.

- ACTION_SIZE: Las acciones que se pueden ejecutar en el simulador (5 acciones).
- OBS_SIZE: Tamaño de la imagen, en este caso es de 80x80.
- UPDATE_TARGET_EVERY: Constante que indica cada cuantos episodios se actualizará la red Qt con los parámetros de la red Q.
- Epsilon: Variable que indica si se debe tomar una acción aleatoria o una acción proveniente de la red.
- EPSILON_DECAY: Ratio de reducción de épsilon.
- MIN_EPSILON: El valor mínimo de épsilon que se puede alcanzar.
- EPSIODES: Numero de episodios que jugará el agente.
- LOG_DIR: Directorio donde se guardarán los logs para después mostrarlos en el panel de Tensorboard.
- Tensorboard_callback: Se utiliza para mostrar las variables en Tensorboard.

Para cada episodio o partida, se inicializa las variables a 0 y se hace un *reset* al simulador poniendo el vehículo en la posición inicial. A continuación, se selecciona una acción aleatoria o una acción proveniente de la red dependiendo del valor de épsilon. Esta aleatoriedad, permite al agente explorar el entrono de forma que, al principio toma acciones aleatorias para después progresivamente, cuando la red esté mejor entrenada, ir cogiendo los valores predichos por la red. El valor de épsilon se reduce al final de cada episodio con un ratio de 0.995. Si se tiene en cuenta que se han simulado 400 episodios, el valor de la caída de épsilon tendrá la siguiente forma.

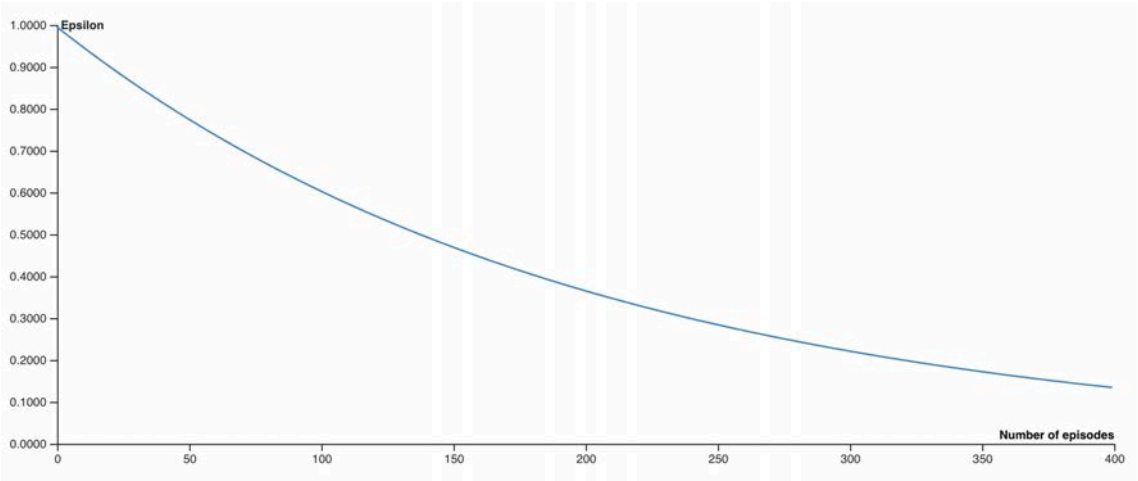


Figura 28. Épsilon para cada episodio.

Donde, por ejemplo, en el episodio 350, un 20% de las acciones ejecutadas serán aleatorias y un 80% provendrán de la salida de la red. Con esto se consigue que aunque la red esté entrenada, siempre haya una posibilidad de tomar una acción aleatoria para explorar el entrono y así tener la posibilidad de poder seguir mejorando el modelo.

```
1.  for episode in tqdm(range(1, EPISODES + 1), unit='Episodes'):
2.      episode_vehicle_speed = []
3.      episode_reward = 0
4.      episode_overtakes = []
5.      episode_lane_changes = []
6.      step = 1
7.
```

```
    # Reset environment and get initial state
    total_obs = env.reset(train_mode=True)[default_brain]
    obs_stack, obs_set = reset_frame_stack(total_obs)

    done = False
    while not done:
        if np.random.random() > epsilon:
            # Get action from Q table
            action = np.argmax(agent.get_qs(obs_stack))
        else:
            # Get random action
            action = np.random.randint(0, ACTION_SIZE)
```

Una vez seleccionada la acción a ejecutar. Se debe enviar la acción al simulador para que la ejecute, y devuelva: el nuevo estado (una nueva imagen), la recompensa obtenida al realizar dicha acción y si el episodio ha terminado o no. Después, se ejecutará la función *frame_stack* explicada anteriormente para apilar 4 imágenes que servirán para entrenar la red.

```
        total_obs = env.step(action)[default_brain]
        reward = total_obs.rewards[0]
        done = total_obs.local_done[0]
        new_obs_stack, obs_set = frame_stack(total_obs, obs_set)
```

En cada paso del simulador, se actualiza el búfer de memoria con: las 4 imágenes observadas antes de tomar la acción, la acción tomada, la recompensa por tomar dicha acción, las nuevas 4 imágenes obtenidas al realizar la acción y por último, si el episodio ha acabado o no. Se ejecuta la función *agent.train* para entrenar el modelo DQN y se actualizan las nuevas variables. Se asigna a la observación actual la nueva observación y se suma 1 a los pasos realizados por el simulador.

```
        agent.update_replay_memory((obs_stack, action, reward, new_obs_stack
    , done))
        agent.train(done, step)
        obs_stack = new_obs_stack
        step += 1
        step_track +=1
        Episode_reward += reward
```

A mediada que va avanzando el episodio, se va guardando la velocidad media, los adelantamientos, la recompensa total y los cambios de carril. Cuando el episodio haya finalizado, se muestran en Tensorboard con la función tf.summary.scalar.

```
        episode_vehicle_speed.append(100 * total_obs.vector_observations[0,-
    8])
        episode_overtakes.append(total_obs.vector_observations[0,-7])
        episode_lane_changes.append(total_obs.vector_observations[0,-6])

        if done:
            tf.summary.scalar("Speed", np.mean(episode_vehicle_speed), step=
    episode)
```

```
            tf.summary.scalar("Overtake", episode_overtakes[-
2], step=episode)
            tf.summary.scalar("Lane Changes", episode_lane_changes[-
2], step=episode)
            tf.summary.scalar("Reward", episode_reward, step=episode)


    print("Episode: " + str(episode) + ", Reward: " + str(episode_reward) +
", Epsilon: " + str(epsilon) + ", Overtakes: " + str(episode_overtakes[-
2]) + ", Lane Changes: " + str(episode_lane_changes[-
2]) + ", Speed: " + str(np.mean(episode_vehicle_speed)) + ", Steps: " + str(
step) + ", Total steps: " + str(step_track))
```

Cuando el episodio haya terminado por completo, se actualiza el valor de épsilon multiplicándolo por el porcentaje de reducción. Una vez finalizado el número total de episodios, el entrenamiento habrá finalizado, se guardan los parámetros y pesos de la red en local para poder utilizarlos más adelante para probar el rendimiento del modelo.

```
    if epsilon > MIN_EPSILON:
        epsilon *= EPSILON_DECAY
        epsilon = max(MIN_EPSILON, epsilon)

# When training is finished save model
agent.save_model()
```

A continuación, se explicará la función *agent.train* que entrena y gestiona el modelo DQN. Antes de empezar a entrenar se debe asegurar que hay las transiciones necesarias en el búfer para poder entrenar. En este caso 50.000. Esto se debe a que si muestrean 32 muestras en cada paso y un episodio tiene de media entre 1700 y 2000 pasos, con menos de 50.000 posiciones en el búfer se estarían cogiendo muchas imágenes repetidas para entrenar la red.

Seguidamente, se muestrean 32 muestras aleatorias de este búfer utilizando la función *random.sample*. Se predice el valor de Q para cada acción con la red Q (model) utilizando 32 muestras de 4 imágenes (con la cuales se ha decido tomar la acción) de 80x80 de tamaño como entrada. Con la red Qt (target), se predice el valor de Q futuras utilizando las nuevas imágenes observadas al ejecutar la acción en el simulador.

Para cada muestra, se calcula la ecuación de *Bellman* y se actualizan los valores de Q para la acción tomada. Junto con estos valores y la observación con la cual se ha decido tomar dicha acción se entrena la red Q.

Por ultimo, cada 5 episodios se actualizará los parámetros red Qt con los parámetros de la red Q.

```
def train(self, terminal_state, step):
        # Start the training when replay_memory greater than MIN_REPLAY_MEMO
RY_SIZE
        if len(self.replay_memory) > MIN_REPLAY_MEMORY_SIZE:

            minibatch = random.sample(self.replay_memory, MINIBATCH_SIZE)
```

```
6.
7.              # Get current states (posicion 0 of transition array)
8.              batch_obs_stack = np.array([transition[0] for transition in mini
    batch])
9.
10.             current_qs_list = self.model.predict(batch_obs_stack)
11.
12.             # Get future states (posicion 3 of transition array)
13.             batch_new_obs_stack = np.array([transition[3] for transition in
    minibatch])
14.
15.             future_qs_list = self.target_model.predict(batch_new_obs_stack)

16.
17.             x,y = [],[] #Define x as the inputs and y as outputs of NN
18.
19.             for index, (obs_stack, action, reward, new_obs_stack, done) in e
    numerate(minibatch):
20.                 if not done:
21.                     max_future_q = np.max(future_qs_list[index])
22.                     new_q = reward + DISCOUNT * max_future_q
23.                 else:
24.                     new_q = reward
25.
26.                 # Update Q value for given state
27.                 current_qs = current_qs_list[index]
28.                 current_qs[action] = new_q
29.                 x.append(obs_stack)
30.                 y.append(current_qs)
31.
32.             # Trains the model every step with the current_qs and current_st
    ate which are obtained from the Bellman eq.
33.             self.model.fit(np.array(x), np.array(y), batch_size=MINIBATCH_SI
    ZE, verbose=0, shuffle=False)
34.
35.             if terminal_state:
36.                 self.target_update_counter += 1
37.
38.             # If counter reaches set value, update target network with weigh
    ts of main network
39.             if self.target_update_counter > UPDATE_TARGET_EVERY:
40.                 self.target_model.set_weights(self.model.get_weights())
41.                 self.target_update_counter = 0
42.         else:
43.             return
```

Por último, las dos redes neuronales convolucionales utilizadas tienen los siguientes parámetros. Donde la capa convolucional será la misma para los 4 algoritmos.

```
1.  def create_model(self):
2.      model = Sequential()
3.      model.add(Conv2D(filters=32, kernel_size=8, strides = 4, input_shape = (
    OBS_SIZE, OBS_SIZE,4), activation='relu'))
4.      model.add(Conv2D(filters=64, kernel_size=4, strides = 2, activation='rel
    u'))
5.      model.add(Conv2D(filters=64, kernel_size=3, strides = 1, activation='rel
    u'))
6.      model.add(Flatten())
7.      model.add(Dense(units=512, activation='relu'))
8.      model.add(Dense(units=ACTION_SIZE))
9.
10.     model.compile(loss="mse", optimizer=Adam(lr=0.00025))
```

```
11.
12.     model.summary()
13.     return model
```

Para compilar el modelo se debe especificar la función de pérdida y el factor de aprendizaje que se utilizarán para entrenar la red.

La red neuronal es muy similar a la utilizada en el método de entropía cruzada. Lo único que cambia, es que la ultima capa en vez de una función de activación *Softmax* se utiliza una función *Relu*. Ya que se quiere obtener una salida lineal y no se busca ninguna distribución de probabilidad de acciones en las salidas.

5.5 A2C

En esta sección se explicará el algoritmo A2C. A continuación, se muestra un esquema de su funcionamiento.

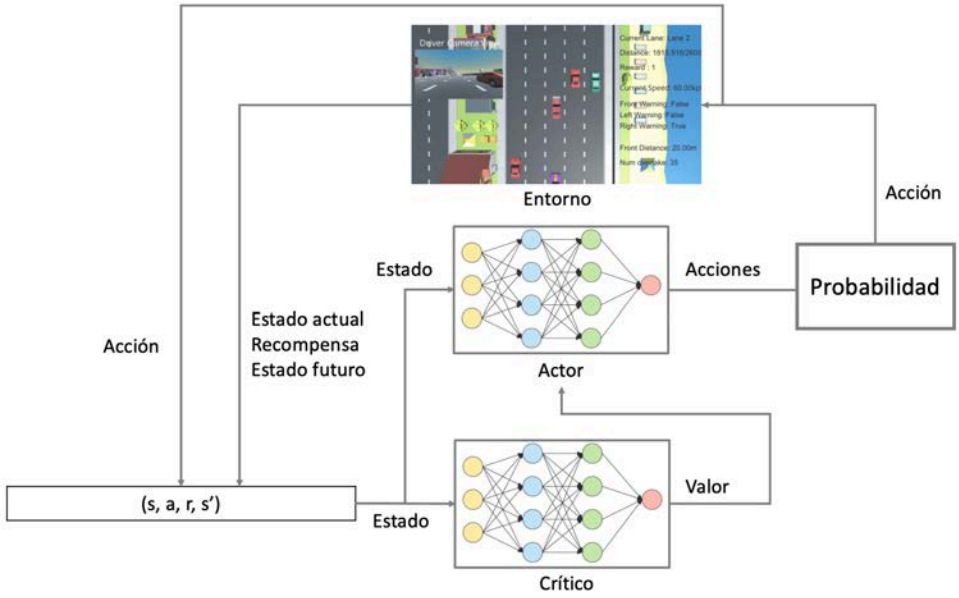


Figura 29. Esquema A2C.

Las variables para este algoritmo son las siguientes.

```
1.  #Training parameters
2.  ACTION_SIZE = 5
3.  self.EPISODES= 400
4.  self.LR = 0.00000025
5.  self.ROWS = 80
6.  self.COLS = 80
7.  self.EPOCHS = 10
8.  self.STATE_SIZE = (4, self.ROWS, self.COLS)
9.
10. #Logging parameters
11. log_dir = "logs_A2C/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
12. tensorboard_callback = keras.callbacks.TensorBoard(log_dir=log_dir)
```

- ACTION_SIZE: Las acciones que se pueden ejecutar en el simulador (5 acciones).
- EPISODES: EPSIODES: Numero de episodios que jugará el agente.
- LR: Factor de aprendizaje de las redes.
- ROWS: Número de filas de la observación/imagen.
- COLS: Número de columnas de la observación/imagen.
- EPOCH: Épocas de entrenamiento. Una época es el número de iteraciones sobre el conjunto de datos para entrenar la red neuronal.
- STATE_SIZE: El tamaño total de la observación (4,80,80).

El bucle principal de A2C, recorre los 400 episodios. Para cada episodio, se ejecuta un *reset* en el entrono y se inicializan las variables. Para cada paso en el episodio, se escoge una acción proveniente de la red Actor y se ejecuta en el simulador para obtener: el siguiente estado, la recompensa y se el episodio ha terminado o no. Para cada paso se entrena la red con la función *replay* que recibe: la observación actual, la acción, la recompensa, el nuevo estado tras ejecutar la acción y si el episodio he terminado o no. Cuando se ha acabado el episodio, se guardan los resultados obtenidos para ser mostrados en Tensorboard.

```
def run(self):
        logging()
        step_track = 1
        for episode in tqdm(range(1, self.EPISODES + 1), unit='Episodes'):
            episode_vehicle_speed = []
            episode_reward = 0
            episode_overtakes = 0
            episode_lane_changes = 0
            episode_vehicle_speed = []
            episode_reward = 0
            step = 1

            total_obs = env.reset(train_mode=True)[default_brain]
            obs_stack, obs_set = reset_frame_stack(total_obs)

            done, score = False, 0

            episode_overtakes = []
            episode_lane_changes = []
            episode_vehicle_speed = []

            while not done:
                # Actor picks an action
                action = self.act(obs_stack)

                # Retrieve new state, reward, and whether the state is termi
nal
                total_obs = env.step(action)[default_brain]
                reward = total_obs.rewards[0]
                done = total_obs.local_done[0]

                new_obs_stack, obs_set = frame_stack(total_obs, obs_set)

                self.replay(obs_stack, action, reward, new_obs_stack,done)

                episode_vehicle_speed.append(100 * total_obs.vector_observat
ions[0,-8])
                episode_overtakes.append(total_obs.vector_observations[0,-
7])
                episode_lane_changes.append(total_obs.vector_observations[0,
-6])
```

```
                obs_stack = new_obs_stack
                episode_reward += reward
                step += 1
                step_track +=1

                if done:
                    tf.summary.scalar("Speed", np.mean(episode_vehicle_speed
), step=episode)
                    tf.summary.scalar("Overtake", episode_overtakes[-
2], step=episode)
                    tf.summary.scalar("Lane Changes", episode_lane_changes[-
2], step=episode)
                    tf.summary.scalar("Reward", episode_reward, step=episode
)

            print("Episode: " + str(episode) + ", Reward: " + str(episode_re
ward) + ", Overtakes: " + str(episode_overtakes[-
2]) + ", Lane Changes: " + str(episode_lane_changes[-
2]) + ", Speed: " + str(np.mean(episode_vehicle_speed)) + ", Steps: " + str(
step) + ", Total steps: " + str(step_track))

        # close environemnt when finish training
        self.env.close()
```

La función *replay*, primero de todo, calcula el valor del estado futuro $V(s')$ y el valor del estado actual $V(s)$ utilizando la red Critico. Se calcula la recompensa descontada utilizando $V(s')$ para después calcular la ventaja restando la recompensa descontada menos el valor del estado actual $V(s)$. Por ultimo, se entrena las dos redes. La red Actor se entrena con el estado y la ventaja como los valores de entrada (x) y las acciones como las salidas (y) la función de perdidas se explicará a continuación. Para la red Crítico, se entrenará utilizando el estado como entrada y la recompensa descontada como las salida, la función de perdidas será el error cuadrático medio, igual que en el algoritmo DQN.

```
def replay(self, state, action, reward, state_, done):
        # Compute discounted rewards
        critic_value_ = self.Critic.predict(state_.reshape((-
1,*state_.shape)))[0]
        critic_value = self.Critic.predict(state.reshape((-
1,*state.shape)))[0]

        target = reward + 0.99*critic_value_*(1-int(done))
        advantage =  target - critic_value

        actions = np.zeros([1, 5])
        actions[np.arange(1), action] = 1

        self.Actor.fit([state.reshape((-
1,*state.shape)), advantage], actions, verbose=0)

        self.Critic.fit(state.reshape((-
1,*state.shape)), target, verbose=0)
```

La función de pérdida personalizada que se ha utilizado para la red Actor tiene en cuenta la ventaja. Se multiplica la ventaja con el logaritmo negativo de la probabilidad actual de

seleccionar la acción que se ha seleccionado. Si la ventaja es negativa, la función de pérdida cambiará de signo, por lo que los gradientes se aplicarán en la dirección opuesta.

En una dimensión es más fácil de entender. Póngase como ejemplo que la predicción objetivo es 1 y la predicción real es 0.6. Una función de pérdida simple se definiría como objetivo - predicción, en este caso 0.4 y las predicciones futuras estarán más cercanas de 1. Ahora bien, si la predicción fuera 1.4, entonces la pérdida sería -0.4. Una pérdida negativa significaría predecir un resultado más bajo en el futuro, y un resultado positivo significaría predecir un resultado más alto en el futuro. Si se cambia el signo de la función de pérdida, la predicción siempre se alejará de 1.

Lo mismo sucede cuando se multiplica la ventaja en la función de pérdida. Una ventaja negativa significaría que esta acción es peor que el valor del estado, por lo que se debe evitar, y una ventaja positiva significa que la acción es debe utilizar más.

Para evitar que la salida de la red tome valores de 0 y que el logaritmo no exista en ese punto, se utiliza la función clip para limitar el resultado de 1e-8 a 1-1e-8.

```
1.  def custom_loss(y_true, y_pred):
2.      out = K.clip(y_pred, 1e-8, 1-1e-8)
3.      log_lik = y_true*K.log(out)
4.
5.      return K.sum(-log_lik*advantage)
```

5.6 PPO

Para implementar PPO simplemente hay que cambiar la función de perdidas respecto al algoritmo A2C. Ahora, la función de perdidas es la siguiente. Esta basada en este articulo: https://arxiv.org/abs/1707.06347.

```
1.  def custom_loss(y_true, y_pred):
2.          advantage, prediction_pick, action = y_true[:, :1], y_true[:, 1:1+act
    ion_space], y_true[:, 1+action_space:]
3.          LOSS_CLIPPING = 0.2
4.          ENTROPY_LOSS = 5e-3
5.
6.          prob = y_pred * action
7.          old_prob = action * prediction_pick
8.          r = prob/(old_prob + 1e-10)
9.          p1 = r * advantage
10.         p2 = K.clip(r, min_value=1 - LOSS_CLIPPING, max_value=1 + LOSS_CLIPPI
    NG) * advantage
11.         loss =  -K.mean(K.minimum(p1, p2) + ENTROPY_LOSS * -
    (prob * K.log(prob + 1e-10)))
12.         return loss
```

Se utiliza un truco que consiste en pasar la ventaja, la predicción y la acción unidas con la función *hstack* para poder utilizar la función de perdida personalizada. Además, a diferencia de A2C, se debe coger la predicción (el *array* con probabilidades para cada acción posible) que retorna la

red Actor y no solo la acción seleccionada para calcular el ratio entre la política actual y la vieja dentro de la función de perdida.

```
y_true = np.hstack([advantage, prediction, action])

self.Actor.fit(state, y_true, epochs=self.EPOCHS, verbose=0, shuffle=True)
self.Critic.fit(state, discounted_r, epochs=self.EPOCHS, verbose=0)
```

6 Resultados

En este apartado se mostraran y compararán los resultados obtenidos por los tres algoritmos. La lista de parámetros para cada algoritmo es muy extensa y simular cada combinación de estos seria una tarea imposible. Por esta razón se han seleccionado los parámetros más relevantes para modificarlos y dejar intactos otros a lo largo de los 4 algoritmos.

Como pauta para comparar los resultados obtenidos, se ha utilizado los resultados que el desarrollador del simulador tiene publicados en su pagina de *Github*. Él, ha utilizado un algoritmo QR-DQN (*Distributional Reinforcement Learning with Quantile Regression*) y no se conoce con cuantos coches (obstáculos) lo ha simulado para obtener dichos resultados.

Configuración	Velocidad (km/h)	Núm. de cambios de carril	Núm. de adelantamientos
Camera RGB	71.0776	15	35.2667

Tabla 2. Resultados base.

También, en este capítulo, también se mostrarán capturas de la carga de la tarjeta grafica (GPU), procesador (CPU) y memoria RAM durante los entrenamientos para cada algoritmo.

6.1 Rendimiento del ordenador

En este apartado se mostrará el rendimiento del ordenador para cada algoritmo. Para medir la carga de CPU y GPU se ha utilizado el programa *nvtop* y para medir el uso de la RAM se ha utilizado el comando de Linux: *free -m*.

Se puede observar en la siguiente imagen el uso de la memoria de la GPU en azul y la carga computacional de la GPU en amarillo. Mas abajo en la imagen se puede ver la carga de CPU por cada proceso.

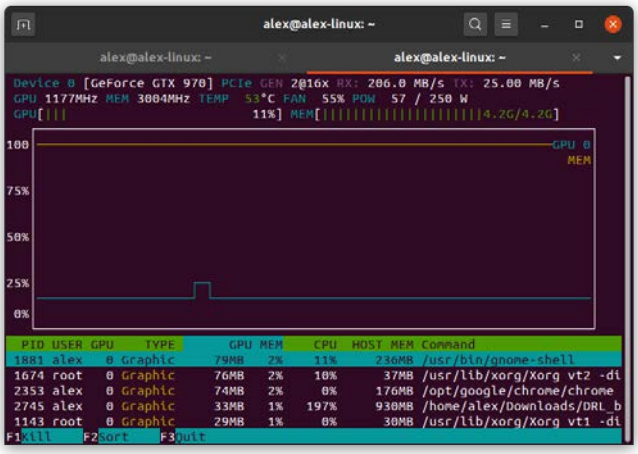


Figura 30. Rendimiento GPU y CPU.

En la siguiente imagen se muestra el uso de la memoria RAM. Cuanta memoria esta siendo utilizada y cuanta memoria está disponible del total de 16Gb.

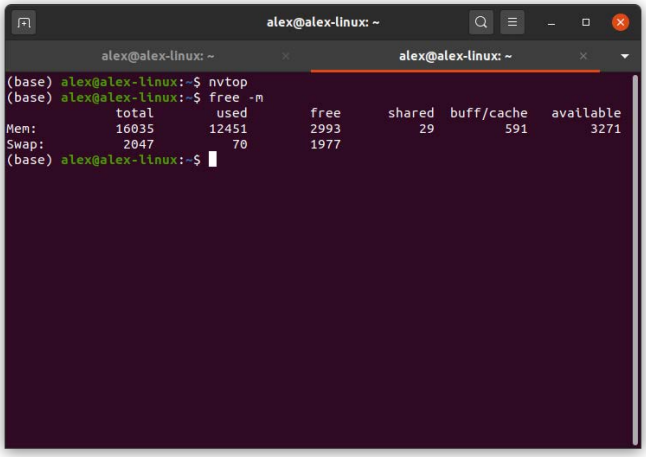


Figura 31. Uso memoria RAM.

La tabla resumen para los 3 algoritmos es la siguiente.

Algoritmo	GPU %	GPU Memoria	CPU	RAM	Tiempo de entrenamiento
Entropía cruzada	11%	100%	90%	13/16Gb	10 horas
DQN	25%	100%	95%	16/16Gb	20 horas
A2C	20%	100%	95%	4.7/16Gb	18 horas
A2C PPO	20%	100%	90%	4.7/16Gb	19 horas

Tabla 3. Resumen rendimiento.

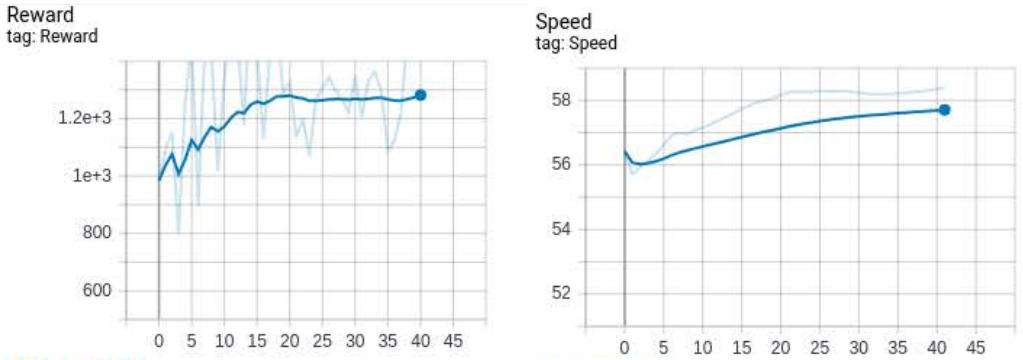
6.2 Entropía Cruzada

A continuación se muestra los resultados del algoritmo de la Entropía Cruzada. Dado a los malos resultados obtenidos y a lo que se explicará a continuación, se ha decido simular este entorno solo una vez con los siguientes parámetros.

Capas Actor NN	Optimizador	Factor de aprendizaje
512	RMS	0.0025

Tabla 4. Parámetros entropía cruzada.

Se utiliza un factor de aprendizaje alto de 0.0025 para reducir el tiempo de entrenamiento, ya que la red se esta entrenando cada 10 episodios.



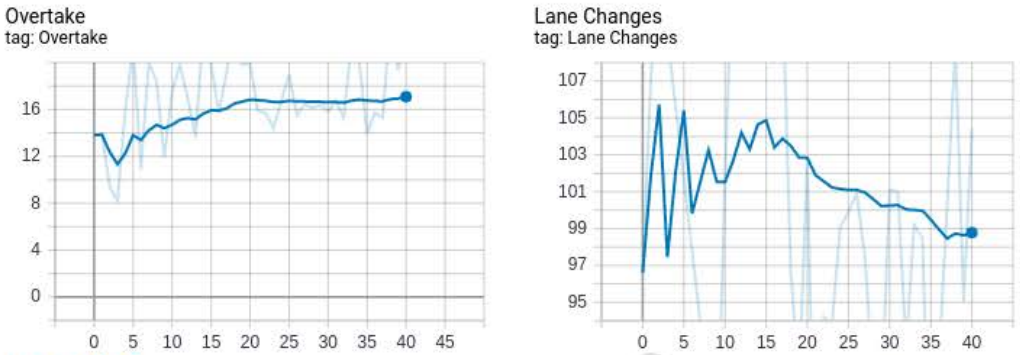


Figura 32. Resultados entropía cruzada.

Cabe destacar, que el gráfico de color más claro son los valores reales de cada episodio, mientras que el gráfico más oscuro es un suavizado de los resultados.

Como se puede observar los resultados no son muy positivos. Los últimos episodios, no tienen muy poca mejora respecto a los primeros episodios donde el actor toma acciones casi aleatorias. Esto se debe a que, a en este tipo de algoritmos se entrena al final de cada episodio. Los episodios en este simulador son excesivamente largos y tomar acciones aleatorias y ejecutar un episodio completo con una recompensa total elevada es prácticamente imposible. Es decir, se esta entrenando la red con los mejores episodios, pero estos, no son lo suficientemente buenos para proporcionar a la red buenos resultados. Este algoritmo podría funcionar mejor en entornos con episodios mas cortos. Al ser un algoritmo que no se adapta correctamente al entorno, no se harán iteraciones cambiando más parámetros ya que no aportarían ninguna mejora sustancial al resultado final.

6.3 DQN

En el algoritmo DQN, los resultados han sido notablemente mejores. Se ha simulado con las siguientes 3 configuraciones.

Variante 1		
Capas red neuronal	Optimizador	Factor de aprendizaje
512	Adam	0.00025
Variante 2		
Capas red neuronal	Optimizador	Factor de aprendizaje
512	RMSProp	0.00025
Variante 3		
Capas red neuronal	Optimizador	Factor de aprendizaje
256	RMSProp	0.0025

Tabla 5. Parámetros DQN.

La variante 1, se utiliza una red de 512 neuronas con el optimizador Adam y un factor de aprendizaje de 0.00025. Los resultados son los siguientes.

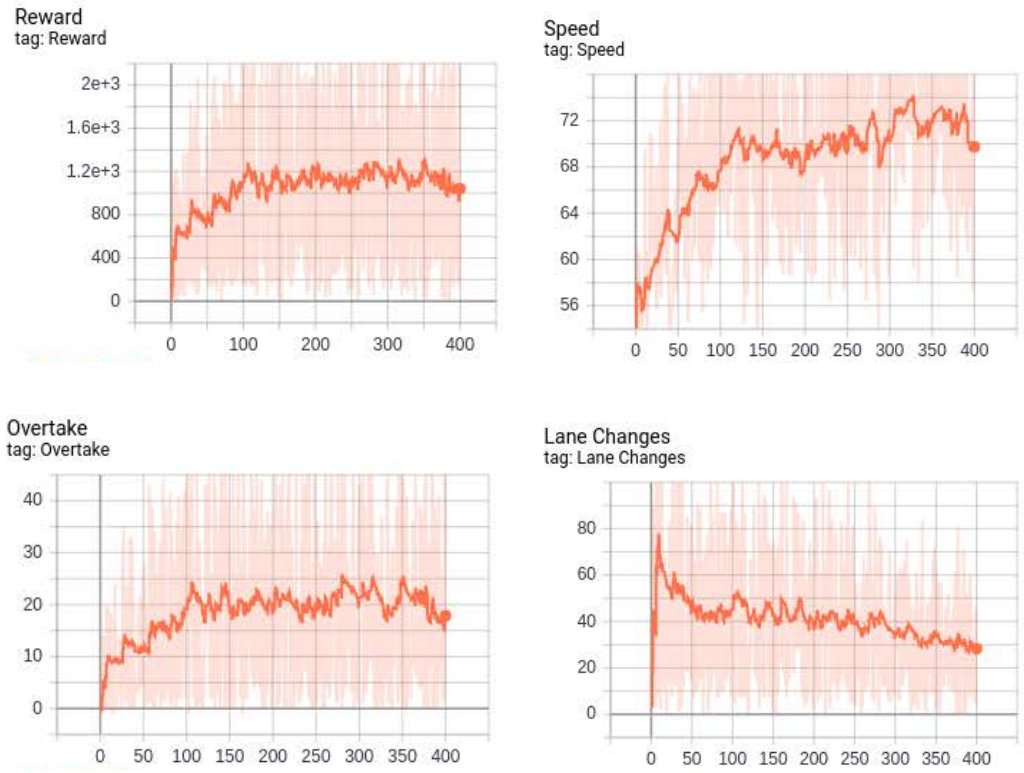


Figura 33. Variante 1 DQN.

Se puede observar unos resultados relativamente buenos, pero no sobresalientes. La recompensa y los adelantamientos son muy bajos, pero por otro lado, la velocidad es muy alta.

Con la variante 2, se ha cambiado el optimizador de Adam a RMSProp.

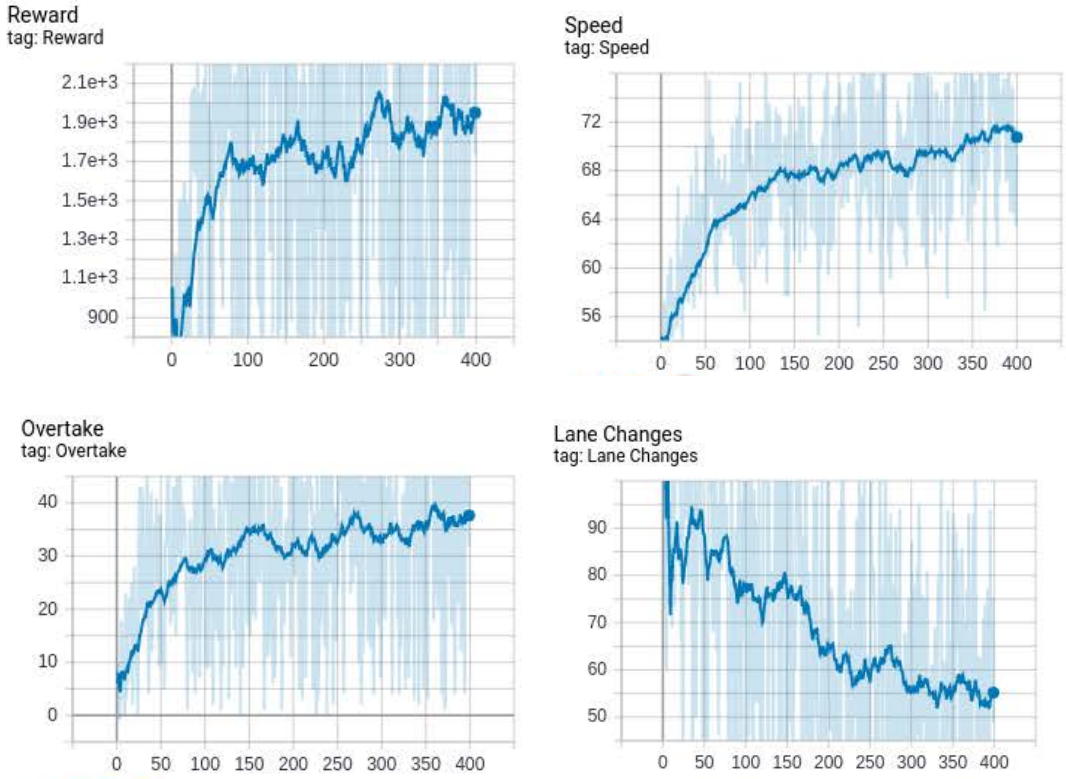


Figura 34. Variante 2 DQN.

Se pueden observar cambios a mejor. Mejorando en los adelantamientos, velocidad y recompensa. En cambio, los cambios de carril, aunque seguían descendiendo a lo largo de los episodios, son bastante elevados, estando en una media final de 60 por episodio.

La variante 3, se ha mantenido el optimizador RMSProp, ya que daba mejores resultados, y se ha variado el factor de aprendizaje y el número de capas de la red.

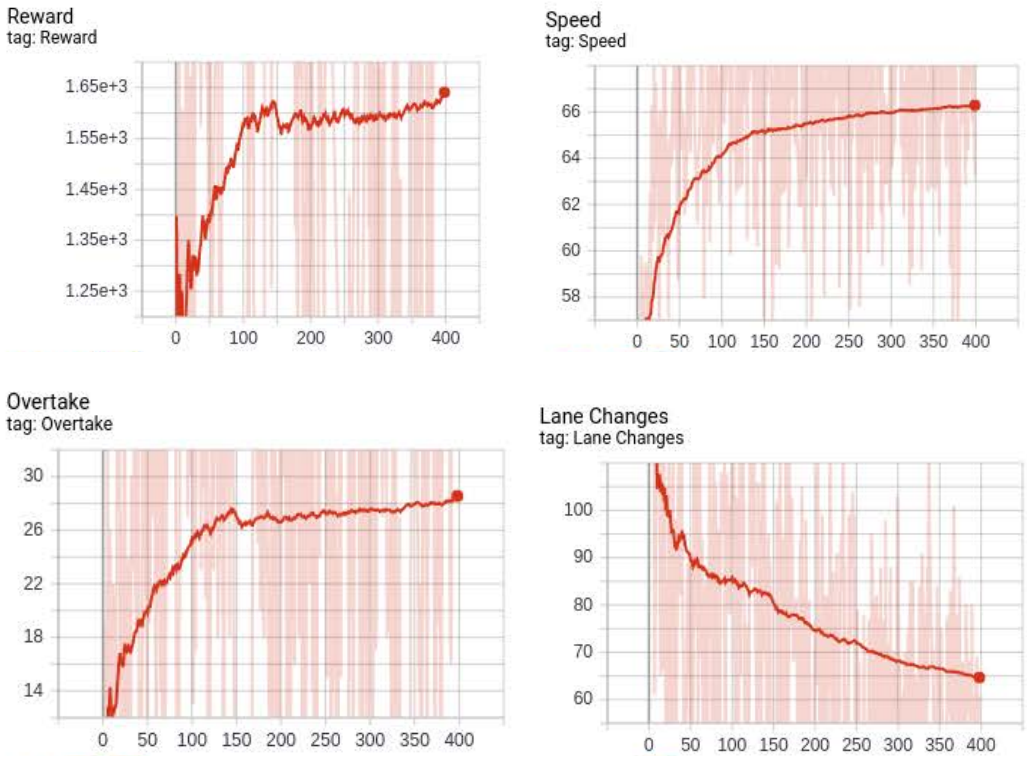
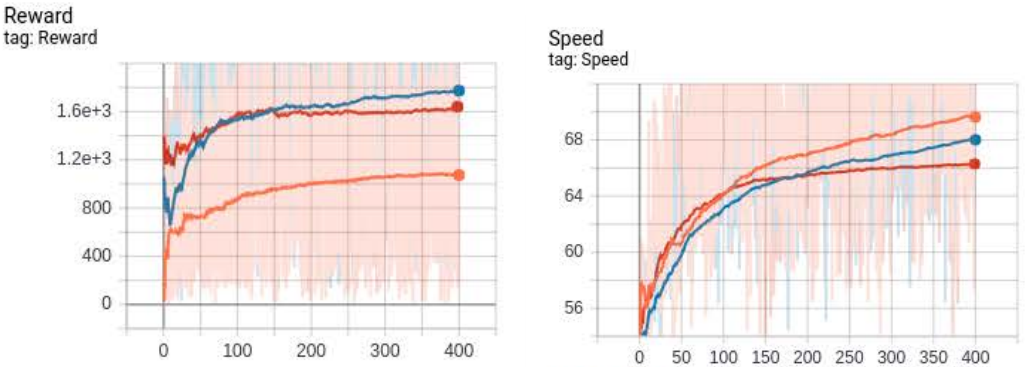


Figura 35. Variante 3 DQN.

Como se puede apreciar, se obtiene una convergencia de la red más rápida, convergiendo en el episodio 100. Pero aun así, los resultados de la variante 2 ofrecen mejores resultados que esta.

Finalmente superponiendo las 3 variantes en una sola imagen se obtiene el siguiente resultado.



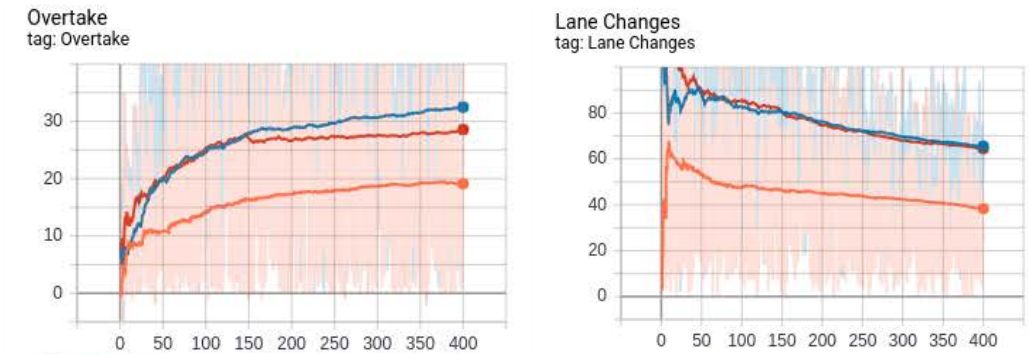


Figura 36. Resultados DQN.

Con las 3 variantes se obtienen resultados muy parecidos. Pero la variante 2, con color azul es la que ofrece los mejores resultados.

6.4 A2C

Con A2C simple no se han obtenido los resultados esperados. Se han variado distintos parámetros sin éxito, todos ellos llevando a conclusiones muy similares. Por esta razón, para simplificar, solo se mostraran 100 episodios con los siguientes parámetros.

Capas Actor	Optimizador Actor	Factor de aprendizaje Actor
512	RMS	0.00000025
Capas Crítico	Optimizador Crítico	Factor de aprendizaje Crítico
512	RMS	0.00000025

Tabla 6. Parámetros A2C.

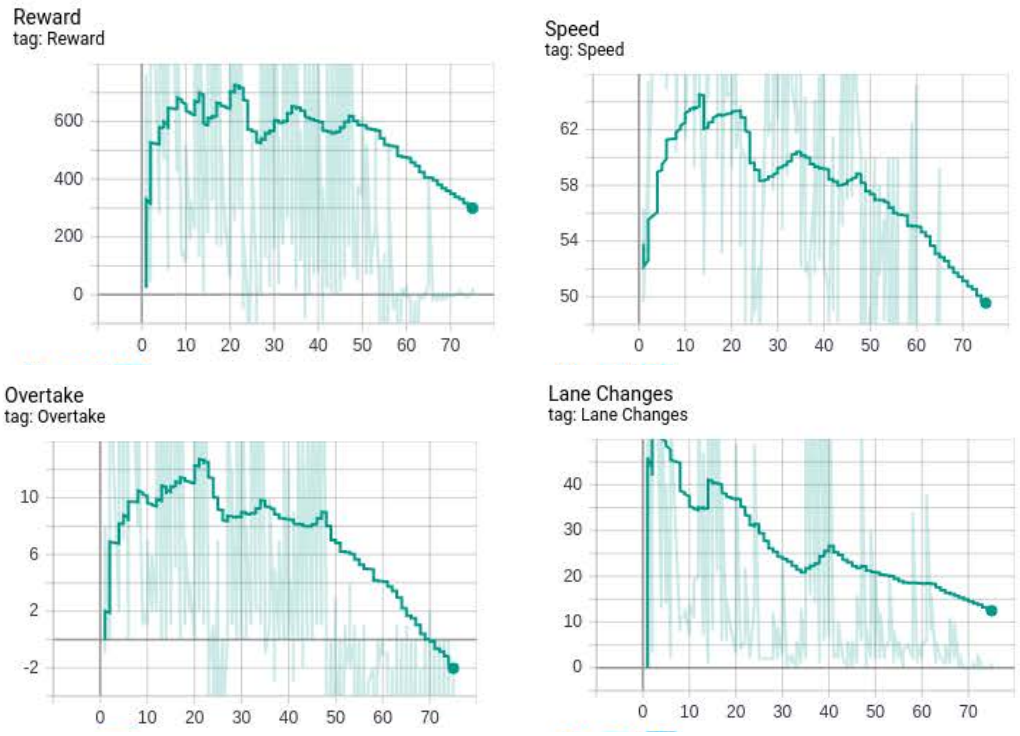


Figura 37. Resultados A2C.

El algoritmo converge siempre políticas similares donde decide que la mejor acción es no realizar ninguna acción. Los cambios de carril y la velocidad tienden a cero, los adelantamientos a números negativos y por consiguiente, la recompensa desciende.

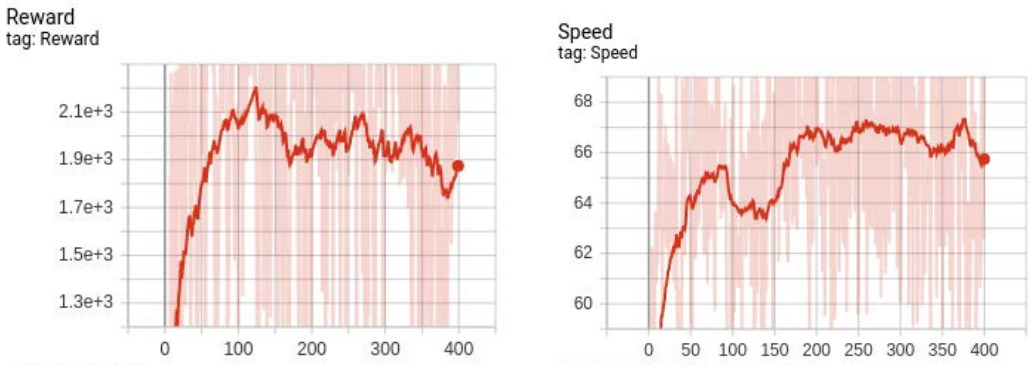
6.5 PPO

Al contrario que el algoritmo A2C, PPO ofrece resultados mucho mejores. Se han simulado 3 variantes con los siguientes parámetros.

Variante 1		
Capas Actor	Optimizador Actor	Factor de aprendizaje Critic
512	RMSProp	0.00000025
Capas Critic	Optimizador Critic	Factor de aprendizaje Critic
512	RMSProp	0.00000025
Variante 2		
Capas Actor	Optimizador Actor	Factor de aprendizaje Actor
512	ADAM	0.00000025
Capas Critic	Optimizador Critic	Factor de aprendizaje Critic
512	ADAM	0.00000025
Variante 3		
Capas Actor	Optimizador Actor	Factor de aprendizaje Actor
256	RMSProp	0.0000025
Capas Critic	Optimizador Critic	Factor de aprendizaje Critic
256	RMSProp	0.0000025

Tabla 7. Parámetros PPO.

La primera variante, con 512 capas por cada red, RMSProp como optimizador y factor de aprendizaje de 0.00000025 ofrece los siguientes resultados.



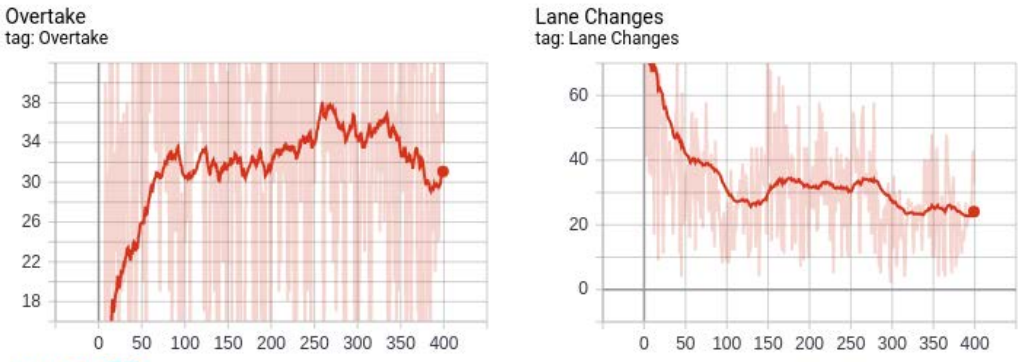


Figura 38. Variante 1 PPO.

Los resultados obtenidos son realmente buenos. Los cambios de carril se reducen a 20, los adelantamientos aumentan a 37 por episodio y la velocidad tiene una media en los últimos episodios de 68 Km/h. El vehículo ha aprendido correctamente una política de conducción y es capaz de esquivar y predecir los movimientos de los otros vehículos de manera eficiente.

En la variante 2 se ha cambiado el optimizador utilizado para entrenar la red manteniendo el resto de parámetros intactos. En este caso se ha utilizado el optimizador ADAM. Los resultados son los siguientes.

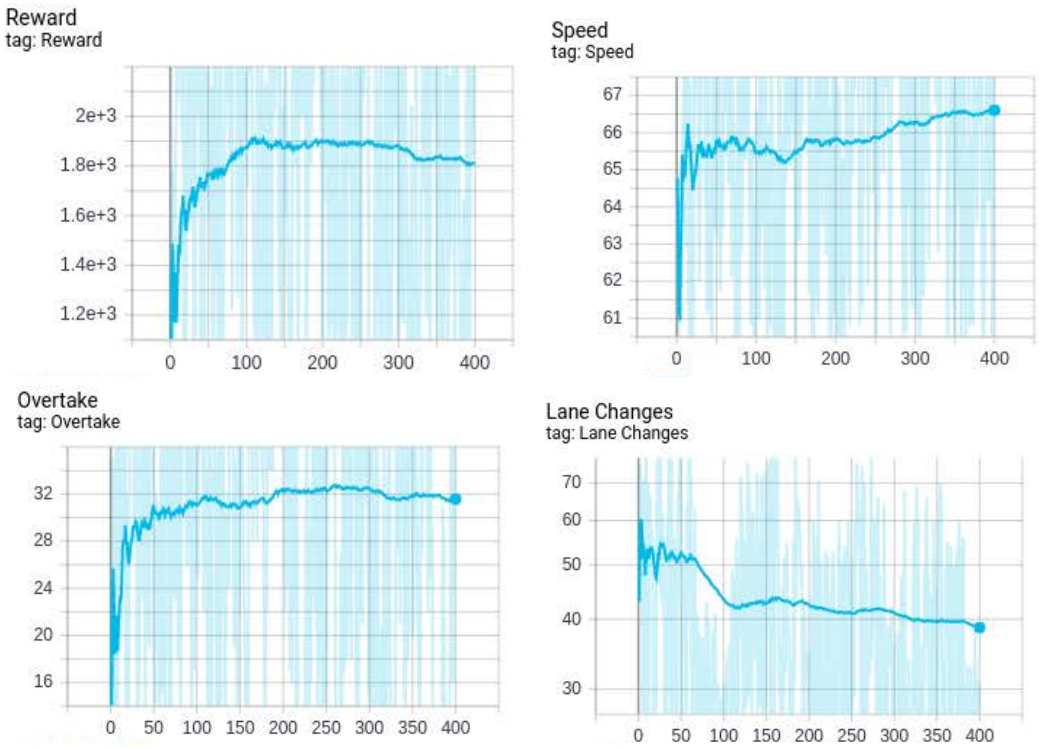


Figura 39. Variante 2 PPO.

Los resultados siguen siendo positivos aunque esta vez, la red converge de forma más rápida. Los adelantamientos, la velocidad y la recompensa son muy similares respecto al la anterior simulación, pero los cambios de carril han aumentado notablemente hasta una media de 38 por episodio.

Por ultimo la variante 3 se ha modificado la capa de la red neuronal reduciendo el numero de neuronas de 512 a 256 y reduciendo también el factor de aprendizaje de 0.00000025 a 0.0000025. Se ha manteniendo el optimizador a RMSProp ya que daba mejores resultados que el optimizador Adam. Los resultados son los siguientes.

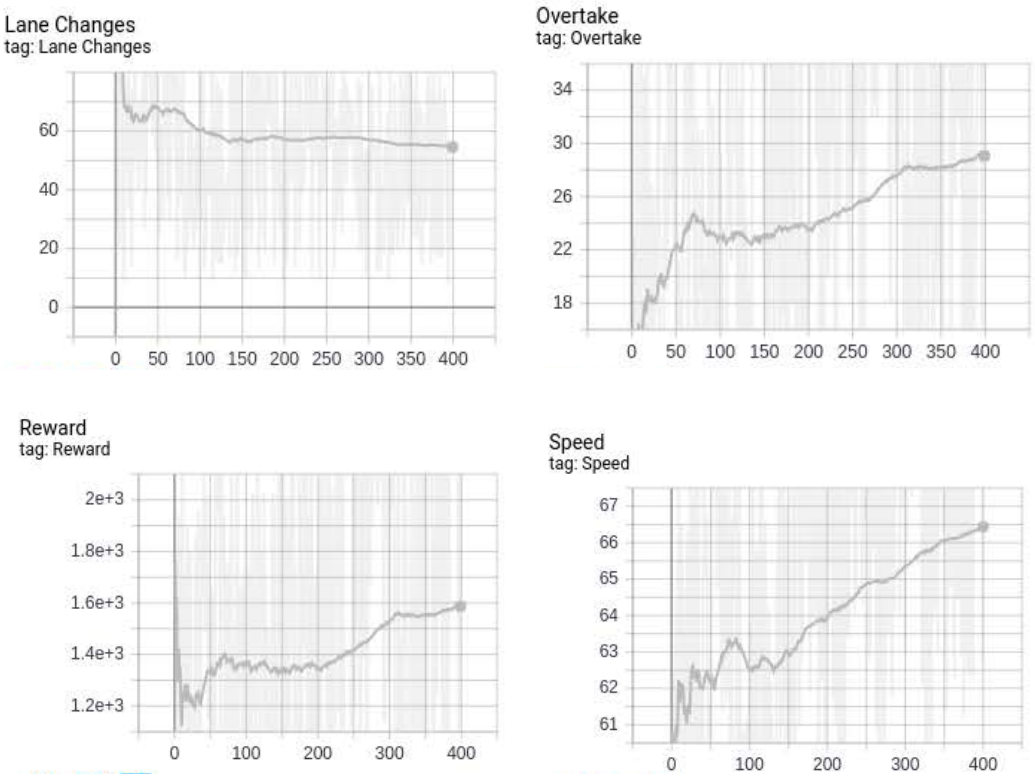
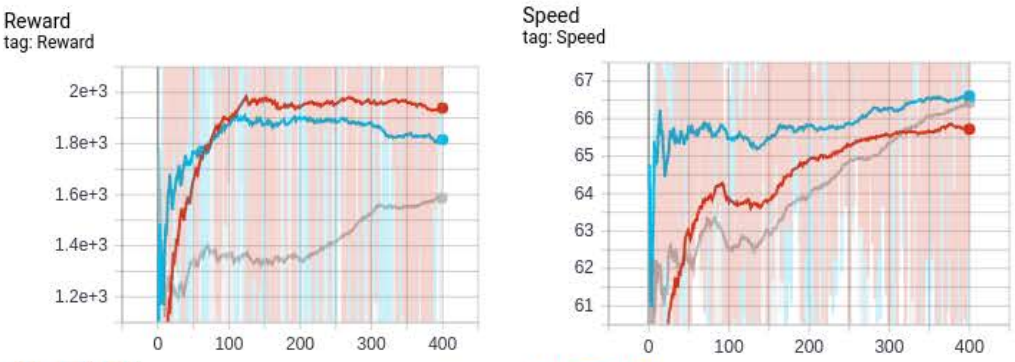


Figura 40. Variante 3 PPO.

Se puede ver que los resultados no son nada buenos, llegando a recompensas muy bajas, pocos adelantamientos y excesivos cambios de carril. La velocidad no es mala y esta a la par con las anteriores variaciones.

Finalmente se muestran las 3 variantes superpuestas para compararlas mejor.



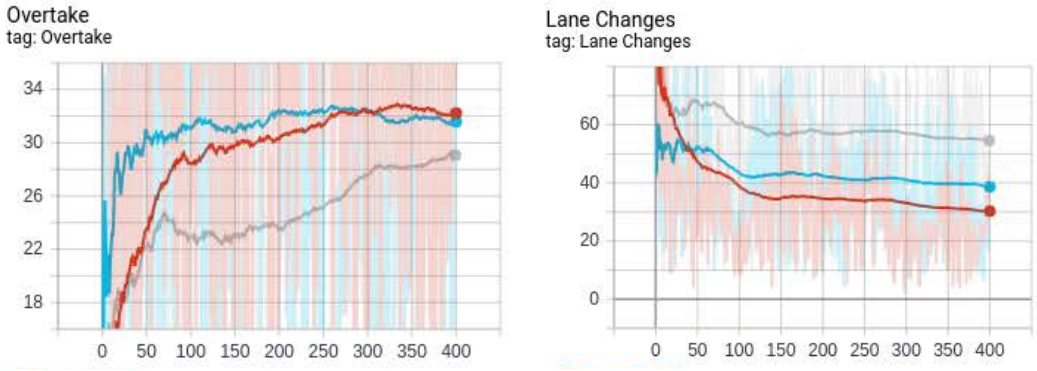


Figura 41. Resultados PPO.

Como se puede observar la mejor variante es la número 1. Está supera a las otras tres en recompensa, adelantamientos y cambios de carril. En velocidad se queda muy cerca

7 Problemas y soluciones

En este apartado se expondrán algunas problemáticas y sus respectivas soluciones encontradas durante el desarrollo del proyecto.

- Problemas con el dimensionamiento en las entradas de la red neuronal: Hay que dimensionar los vectores de la entrada de la red con 4 dimensiones de la siguiente forma: (None, número columnas, número filas, número canales). Al tener diversos tamaño de entradas hay que dimensionar cada vector lo que provocó muchos errores y valores incorrectos en la salida de la red.

- Instalación de las librerías: instalar librerías en un entorno Linux siempre da problemas. Usando el sistema anaconda la librería Tensorflow se debe instalar en un entrono separado al entorno base. Por ello se debe duplicar el entorno base e instalar la librería Tensorflow.

- Memoria RAM insuficiente: Debido al gran numero de muestras que se almacenan para entrenar al algoritmo de DQN se necesito ampliar la RAM del ordenador de 8 Gb a 16Gb. Con este cambio se han obtenido resultados notablemente mejores.

- Tiempo de entrenamiento: Al tener tiempos de entrenamiento tan elevados, al hacer cualquier pequeña modificación del código, se tenia que esperar sobre 12 a 24 horas para poder observar los nuevos resultados. Este problema no tiene solución especifica. Ya que la velocidad depende del simulador, y no del ordenador.

- Pytorch vs Keras: Pytorch y Keras son dos librerías más utilizadas en entornos de aprendizaje profundo. Aunque las dos son muy parecidas y se pueden conseguir mismos resultados, la sintaxis es bastante diferente. Keras es definitivamente más sencillo de utilizar, comprender y poner en marcha rápidamente. No hay que preocuparse por la configuración de la GPU, jugar con código abstracto o, en general, hacer cualquier tarea complicada. Incluso se pueden hacer cosas como implementar capas personalizadas y funciones de pérdida con tan solo una línea de código. Por otro lado, si se quiere a llegar a aspectos más detallados de las redes profundas o se está implementando algo que no es estándar, entonces Pytorch es mejor opción. Aunque el libro que se ha seguido para esta tesis utiliza Pytorch como librería principal para desarrollar los ejemplos, para implementar los algoritmos en esta tesis se ha optado por Keras, priorizando la simplicidad.

- Problemas con previos simuladores complejo: Al principio, el trabajo se enfocó para ser desarrollado utilizando el simulador Carla. La instalación es complicada y gestionar el simulador con el código Python también ya que hay muchos parámetros que tener en cuenta. Además, hay diversos mapas pre construidos de distintas ciudades que son extremadamente grandes y complejos. Lo que supondría entrenar al modelo con un muchas ventanas de simuladores abiertas simultáneamente para llegar a obtener resultados mínimamente decentes. El ordenador que actualmente se ha utilizado no

tiene las especificaciones suficientes para soportar dicha carga. Por eso, se optó por escoger un simulador más sencillo y probar de implementar más algoritmos de RL.

- Convergencia del modelo de A2C: El algoritmo A2C ha dado muchos problemas ya que convergía en una solución demasiado rápido. Al principio se pensó que había un error el código de algún valor de algún *array* que no estaba correctamente dimensionado. Se probó también, de entrenar la red una vez terminado el episodio en vez de en cada paso del episodio, pero los resultados fueron los mismos. Tras muchos intentos y modificaciones de parámetros, se decidió implementar el algoritmo PPO que con tan solo cambiando la función de perdida daba mucho mejores resultados.

- No es posible ejecutar diferentes simuladores al mismo tiempo: Para mejorar la velocidad de aprendizaje, especialmente en algoritmos de Entropía Cruzada, PPO y A2C, se suelen correr diferentes entornos paralelamente. Se ha intentado utilizar la librería *gym-unity* que facilita este proceso adaptando el entorno a uno basado en Open AI Gym que ofrece una comunidad de desarrollo muy amplia. Pero su implementación en este proyecto no ha sido exitosa, ya que el simulador está basado en la versión de ml-agents 0.6 mientras que está librería necesita la versión 0.14 o superior. Actualizar el simulador supondría editar y rehacer muchas partes del código además de investigar el funcionamiento interno de Unity. Por eso, se decidió utilizar solo una agente asumiendo los elevados tiempos de entrenamiento y por consiguiente resultados no tan óptimos.

8 Futuro del aprendizaje por refuerzo

El campo de la investigación de aprendizaje profundo crece a un ritmo acelerado. Hay muchas áreas de investigación potenciales, como las redes neuronales convolucionales (CNN), las redes neuronales recurrentes (RNN), la memoria a largo corto plazo (LSTM), los codificadores automáticos, las redes generativas y mucho más. Pero es el aprendizaje de refuerzo profundo (RL) es lo que parece que está creciendo a un ritmo más acelerado, básicamente las posibilidades de esta tecnología son infinitas. A continuación, se muestran algunos ejemplos de casos prácticos que utilizan RL como algoritmo principal [22].

- Mercados financieros: Se puede hacer uso de algoritmos RL para generar automáticamente modelos de negociación rentables, robustos y sin correlación en cualquier mercado financiero general. Para hacer esto, se presenta un nuevo modelo de proceso de decisión de Markov para capturar los mercados de comercio financiero. Se revisan y proponen varias modificaciones a los enfoques existentes y exploran diferentes técnicas para capturar la dinámica del mercado para modelar los mismos. Luego, se utiliza DRL para permitir que el agente (el algoritmo) aprenda a realizar operaciones rentables en cualquier mercado por sí mismo, mientras sugiere varios cambios en la metodología y aprovecha la representación única del FMDP (MDP financiero). Fuente: https://arxiv.org/pdf/1907.04373.pdf

- Ciberseguridad: La escala de los sistemas conectados a Internet ha aumentado considerablemente, y estos sistemas están expuestos a ataques cibernéticos más que nunca. La complejidad y la dinámica de los ataques cibernéticos requieren mecanismos de protección para ser receptivos, adaptativos y a gran escala. Los métodos de aprendizaje automático, o más específicamente RL, se han propuesto ampliamente para abordar estos problemas.

- Sistemas de energía: El área de gestión de la energía de los edificios ha recibido una gran cantidad de interés en los últimos años. Esta área tiene que ver con la combinación de avances en tecnologías de sensores, comunicaciones y algoritmos de control avanzados para optimizar la utilización de energía. Con el aprendizaje por refuerzo se desarrollan sistemas autónomos de gestión de energía de edificios. Con estos, se producen grandes ahorros de energía.

- Pintura: Con RL se puede enseñar a las máquinas a pintar como pintores humanos. Al combinar el renderizador neural y el DRL, el agente puede descomponer imágenes ricas en texturas en trazos y convertirlas en pinceladas. Para cada trazo, el agente determina directamente la posición y el color del trazo. Se puede lograr un excelente efecto visual utilizando cientos de trazos. Fuente: https://arxiv.org/pdf/1903.04411.pdf

- Fabricación: Muchas empresas de fabricación como Fanuc utilizan el aprendizaje de refuerzo profundo en robots para elegir y coger un objeto de una caja y colocarlo en un contenedor. La maquina memoriza el objeto y adquiere conocimiento y se entrena para hacer este trabajo con alta velocidad y precisión.

- Procesamiento del señal: Debido a sus características inherentes de autoaprendizaje y adaptabilidad, los algoritmos RL se utilizan para diversas tareas en redes ópticas. En estas aplicaciones, las acciones realizadas por los algoritmos RL pueden incluir elegir espectro o formato de modulación, re direccionamiento del tráfico de datos, etc., mientras que la recompensa puede ser la maximización del rendimiento de la red, la minimización de la latencia o la tasa de pérdida de paquetes.

9 Conclusiones

Para concluir, el proyecto tenía como objetivo principal hacer que un vehículo autónomo navegue en una carretera combinando funciones ADAS usando el aprendizaje por refuerzo. Después de la ejecución de este proyecto, se puede ver que el objetivo se ha logrado.

El proyecto puede usarse como base para futuros proyectos más complejos. El desarrollo del proyecto desde cero ha permitido documentar todos los pasos tomados durante la implementación de ambos, el simulador y los algoritmos. Sin embargo, y debido al amplio alcance del proyecto, se han dejado muchas mejoras y pruebas para el futuro. Otros algoritmos basados en valor como Double DQN, Dueling DQN o Categorical DQN podrían haber sido implantados. Así como también, otros algoritmos basados en la política como TRPO, ACKTR o SACK. Variar los parámetros de la red también afecta de manera sustancial los resultados y se podrían haber hecho más pruebas e iteraciones para cada algoritmo.

Como futuros proyectos y teniendo una base solida del funcionamiento de este campo. Se podrían aplicar los conocimientos adquiridos en simuladores más complejos o incluso en vehículos teledirigidos en la vida real. Un ejemplo es *Donkeycar*, un coche teledirigido basado en una *Raspberry Pi* que utiliza una cámara RGB para desplazarse autónomamente [23]. Se podría entrenar utilizando un simulador en el ordenador con cualquier algoritmo RL para después ejecutar el modelo entrenado en la vida real permitiendo al vehículo esquivar obstáculos, mantenerse dentro de un carril o incluso hacer carreras con otros competidores. Adicionalmente, se podría entrenar utilizando algoritmos de aprendizaje supervisado como Behavioral Cloning (BC) donde el vehículo aprende de las acciones mientras una persona lo conduce por un circuito.

Figura 42. Donkeycar.

9.1 Realización de los objetivos

Se repasarán los objetivos del proyecto definidos en los primeros capítulos y se discutirá si se han cumplido con éxito o no.

- Estudiar los simuladores: El primer paso para la implementación del algoritmo RL fue encontrar un simulador que se ajustara a nuestro proyecto. Se ha podido ver que existen varios simuladores dependiendo de su propósito; de simuladores de uso profesional, a otros de nivel más educativo. Después de una comparación entre 9 de estos simuladores, se encontró que Unity ML-Agents Highway Simulator es el mejor para el proyecto. Es un carril de cinco simulador de carretera con más vehículos en la carretera. Además, y para facilitar el futuro implementaciones, se dio una guía y un ejemplo de interacción con el simulador. Aun con los inconvenientes que presentaba el simulador, como el largo tiempo de entrenamiento, ha sido una buena elección, ya que este simulador es bastante intuitivo de utilizar y no se ha perdido mucho tiempo en investigar su funcionamiento. Se ha podido ir directamente a la parte importante de la tesis, los algoritmos.

- Conceptos previos: Se han adquirido los conceptos previos con éxito profundizando aún más con algunos detalles matemáticos que no se describían en el libro que se ha utilizado como guía.

- Reinforcement Learning: Se han estudiado los conceptos de aprendizaje por refuerzo con éxito y ampliado los conocimientos adquiridos en el libro con desarrollos matemáticos extra extraídos de artículos e investigaciones.

- Análisis de los resultados obtenidos: Aunque se ha llegado a resultados conclusivos, no todos los resultados han sido los esperados. Como se ha comentado, el algoritmo de entropía cruzada y A2C proporcionan resultados peores que los otros dos. Aun así, se puede observar como el agente está aprendiendo de una forma u otra a controlar el vehículo por la autopista.

- Futuros proyectos: Se ha hecho una investigación de posibles aplicaciones futuras probando que los algoritmos de aprendizaje reforzado son el futuro de ML.

9.2 Coste

En este apartado se describirán las horas dedicadas para la realización del proyecto. Para llevarlo a cabo se han realizado una serie de tareas o fases, estas se detallan a continuación:

- Planificación: En esta fase se ha estructurado el proyecto y escogido el tema.

- Aprendizaje: En esta fase se ha documentado e informado sobre el tema de la tesis. Además, se han programado algunos algoritmos en otros simuladores más sencillos siguiendo los ejemplos del libro. “Deep Reinforcement Learning Hands-On, Maxim Lapan”.

- Desarrollo: En esta fase se ha instalado todo el entrono de desarrollo y se han programado los 4 algoritmos.

- Depuración de errores: En esta fase se han encontrado y solucionado los errores.

- Realización memoria: Por ultimo, en esta fase se ha realizado toda la documentación del proyecto plasmándolo en esta memoria.

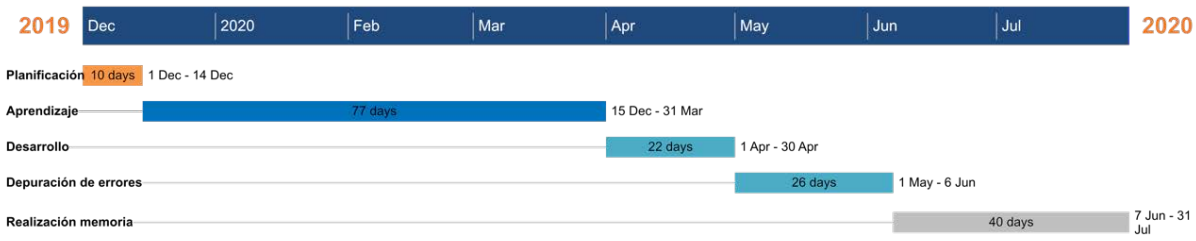


Figura 43. GANTT de las horas invertidas.

Cabe destacar que la imagen anterior muestra una aproximación, los días no representan las horas trabajadas, sino el tiempo transcurrido por cada fase. Como era de esperar las fases de aprendizaje y desarrollo (incluyendo la depuración de errores) han sido las que más tiempo han consumido. En la fase de aprendizaje, se desarrollaron los diferentes algoritmos en entornos más sencillos para poder aprender los conceptos correctamente antes de adentrarse en entornos más complicados. Destacar también, que la fase de depuración de errores se alargó más de lo previsto debido al tiempo de entrenamiento de cada script. Por ultimo, la planificación y la realización de la memoria son lo que han supuesto menor coste tiempo.

10 Referencias

[1] Estos son los 5 niveles de conducción del coche autónomo | Neomotor. Retrieved from https://www.neomotor.com/conduccion/estos-son-los-5-niveles-de-conduccion-del-coche-autonomo.html

[2] Advanced Driver Assistance Systems (ADAS) | First Sensor. Retrieved from https://www.first-sensor.com/en/products/blue-next-cameras/systems-for-adas/

[3] tfm-alex-cabaneros.pdf. Retrieved from https://upcommons.upc.edu/bitstream/handle/2117/133279/tfm-alex-cabaneros.pdf?sequence=1&isAllowed=y

[4] GitHub - MLJejuCamp2017/DRL_based_SelfDrivingCarControl: Deep Reinforcement Learning (DQN) based Self Driving Car Control with Vehicle Simulator. Retrieved from https://github.com/MLJejuCamp2017/DRL_based_SelfDrivingCarControl

[5] Brief History of Neural Networks. Although the study of the human brain | by Kate Strachnyi | Analytics Vidhya | Medium. Retrieved from https://medium.com/analytics-vidhya/brief-history-of-neural-networks-44c2bf72eec

[6] Neural networks and deep learning. Retrieved from http://neuralnetworksanddeeplearning.com/chap1.html

[7] Modern methods of neural network training. Retrieved from https://svitla.com/blog/modern-methods-of-neural-network-training

[8] The History of Neural Networks - Dataconomy. Retrieved from https://dataconomy.com/2017/04/history-neural-networks/

[9] Basics of the Classic CNN. How a classic CNN (Convolutional Neural Network) | by Chandra Churh Chatterjee | Towards Data Science. Retrieved from https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add

[10] Understanding Actor Critic Methods and A2C | by Chris Yoon | Towards Data Science. Retrieved from https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f

[11] Proximal Policy Optimization (PPO) with Sonic the Hedgehog 2 and 3 | by Thomas Simonini | Towards Data Science. Retrieved from https://towardsdatascience.com/proximal-policy-optimization-ppo-with-sonic-the-hedgehog-2-and-3-c9c21dbed5e

[12] Keras: the Python deep learning API. Retrieved from https://keras.io/

[13] NumPy. Retrieved from https://numpy.org/

[14] opencv-python ¬∑ PyPI. Retrieved from https://pypi.org/project/opencv-python/

[15] GitHub - tqdm/tqdm: A Fast, Extensible Progress Bar for Python and CLI. Retrieved from https://github.com/tqdm/tqdm

[16] random - Generate pseudo-random numbers - Python 3.8.6rc1 documentation. Retrieved from https://docs.python.org/3/library/random.html

[17] GitHub - Unity-Technologies/ml-agents: Unity Machine Learning Agents Toolkit. Retrieved from https://github.com/Unity-Technologies/ml-agents

[18] TensorFlow. Retrieved from https://www.tensorflow.org/

[19] 8.3. collections - High-performance container datatypes - Python 2.7.18 documentation. Retrieved from https://docs.python.org/2/library/collections.html

[20] datetime - Tipos básicos de fecha y hora - documentación de Python - 3.8.6rc1. Retrieved from https://docs.python.org/es/3/library/datetime.html
[21] Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games. Retrieved from https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/
[22] 10 Real-Life Applications of Reinforcement Learning - neptune.ai. Retrieved from https://neptune.ai/blog/reinforcement-learning-applications
[23] Donkey Car - Home. Retrieved from https://www.donkeycar.com/