

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y
Automática

“Construir un RC car autónomo con Raspberry Pi, NAVIO2 y TensorFlow/KERAS”

Autores: Adrián Fernández Vázquez y
Noemi Delgado Santa Cruz

Tutor: Manuel Jesús Rodríguez Valido

Departamento de Ingeniería Industrial
Escuela Superior de Ingeniería y Tecnología
Universidad de La Laguna
Tenerife, Septiembre de 2019

AGRADECIMIENTOS

Por la presente, se agradece a todo el profesorado que se ha visto involucrado en el proyecto, no sólo en la aportación de ideas para continuar desarrollando el prototipo, sino también todos los medios necesarios que pudiéramos solicitar.

Agradecer a la Universidad de La Laguna por ofrecernos unas instalaciones en buen estado donde poder fijar la base de operaciones, como ha sido nuestro laboratorio de diseño en la facultad de la Escuela Superior de Ingeniería y Tecnología, así como a la facultad de Biología por contribuir a querer ofrecer un espacio donde poner a correr el vehículo pese a que nunca se llevó a cabo en esa zona.

Además, este proyecto no habría sido capaz de llevarse a cabo sin las contribuciones de Open Source como la del usuario “yconst” en GITHUB, ofreciéndonos el repositorio de “BURRO” a entera disposición de cualquiera que estuviera interesado en trabajarlo.

Agradecer también a la compañía de Udacity por ofrecer al público, de la misma manera anterior, el uso del simulador en Unity para vehículos de conducción autónoma.

Por otro lado, al ser una materia de desconocimientos de los autores, cursos como el impartido por David Fuentes Jiménez, Ingeniero Industrial especializado en Data Science, sobre “Deep Learning e Inteligencia Artificial con KERAS/TensorFlow” bajo la plataforma de Udeemy han sido pieza clave para poder arrancar con el diseño de las redes implementadas.

Finalmente, gracias a la aplicación de nuestro tutor de trabajo de final de grado, Manuel Jesús Rodríguez Valido, no sólo por orientarnos cuando no sabíamos cómo proceder, sino por todo el apoyo tanto en conocimientos como en compromiso con el proyecto, ya que al igual que nosotros se esforzó en analizar y comprender esta materia de inteligencias artificiales.

ÍNDICE DE CONTENIDO

AGRADECIMIENTOS.....	3
RESUMEN	6
ABSTRACT	7
ÍNDICE DE TABLAS	8
ÍNDICE DE FIGURAS	8
ÍNDICE DE CÓDIGOS	9
1. INTRODUCCIÓN	10
1.1. CONDUCCIÓN AUTÓNOMA. ANTECEDENTES	13
1.1.1. FRANCIS P. HOUDINA.	13
1.1.2. JOHN MCCARTHY.	13
1.1.3. DEAN POMERLEAU.	14
1.1.4. DARPA.	15
1.1.4.1. Gran Desafío, 2004.....	16
1.1.4.2. Gran Desafío, 2005.....	16
1.1.4.3. Desafío Urbano, 2007.....	17
1.2. OBJETIVOS	19
1.3. ESTRUCTURA DE LA MEMORIA.....	19
2. METODOLOGÍAS Y HERRAMIENTAS	20
2.1. METODOLOGÍAS.	20
2.2. HERRAMIENTAS.	22
2.2.1. GIT.....	22
2.2.2. Google Drive.	22
2.2.3. Google Colab.	22
2.2.4. IntelliJ/PyCharm.	23
2.2.5. Microsoft Word.	23
2.2.6. Simulador Unity.....	23
2.2.7. Librerías para IA: TensorFlow y KERAS.....	23
3. DISEÑO E IMPLEMENTACIÓN HARDWARE/SOFTWARE DEL RC CAR. 24	
3.1. VEHÍCULO RC BASE.	24
3.1.1. Motor.....	25
3.1.2. Variador electrónico de velocidad (ESC).....	26
3.2. CÁMARA.....	27
3.3. RASBERRY PI 3 B+.....	28
3.4. NAVIO 2.	29
3.5. MANDO LOGITECH F710.....	30
3.6. BATERÍA.....	31
3.7. INTEGRACIÓN DEL HARDWARE.	32

3.8.	FIRMWARE DE CONTROL DEL VEHÍCULO. BURRO SOFTWARE.	35
3.8.1.	Fichero de configuración.	37
3.8.2.	Cambiar entre modos de conducción.	38
3.8.3.	Eliminar pilotos RC.	38
3.8.4.	Cambio controles de velocidad.	39
3.8.5.	Eliminar conversión a la salida de la red neuronal.	40
4.	DISEÑO E IMPLEMENTACIÓN DE LA NEURONA DEL RC CAR	41
4.1.	REDES CLÁSICAS: PERCEPTRONES MULTICAPAS.	41
4.1.1.	Neuronas.	42
4.1.2.	Funciones de activación.	43
4.1.3.	Estructura de la red.	43
4.1.4.	Preprocesado de los datos.	45
4.1.5.	Algoritmo de entrenamiento.	45
4.2.	REDES NEURONALES CONVOLUCIONALES (CNN).	47
4.2.1.	Ventaja frente a las redes clásicas.	47
4.2.2.	Arquitectura de una CNN.	48
4.3.	ELIGIENDO UN MODELO PARA NUESTRA NEURONA: NVIDIA MODEL.	50
4.4.	ADAPTACIÓN DEL MODELO A NUESTRO PROYECTO.	52
4.4.1.	Software de entrenamiento de la red.	55
4.4.2.	Preprocesado de datos: generar los archivos CSV.	56
4.5.	SIMULACIÓN DEL RC CAR EN UNITY.	61
4.5.1.	Entorno del simulador.	61
5.	RESULTADOS Y CONCLUSIONES	66
5.1.	RESULTADOS DE LA FASE DE ADQUISICIÓN DE DATOS.	66
5.2.	RESULTADOS DEL ENTRENAMIENTO NEURONAL.	68
5.3.	CONCLUSIONES.	70
6.	RECOMENDACIONES FUTURAS	71
7.	PRESUPUESTO	73
8.	REFERENCIAS BIBLIOGRÁFICAS	74
9.	ANEXOS	78
9.1.	COMPUTER CONTROLLED CARS.....	78
9.2.	END TO END LEARNING FOR SELF-DRIVING CARS	83

RESUMEN

Hoy en día, la mayoría de sistemas automáticos o de gestión tienden a implementar inteligencias artificiales que favorecen la experiencia con el usuario y realizan un trabajo eficiente y ordenado. Los asistentes inteligentes de los “smartphones” son un claro ejemplo de ello. Por este motivo no es descabellado pensar que entre los sectores en auge, dentro de esta nueva tecnología, todavía reciente y en desarrollo, nos encontremos con vehículos inteligentes.

Actualmente, son muchas las empresas que están implementando este tipo de tecnología en sus coches (Tesla entre las pioneras) y muchas otras que ofrecen herramientas y entornos para que el sector del “*machine learning*” progrese de forma eficaz gracias a la aportación de los usuarios.

Tras profundizar en el tema, hemos encontrado que compañías como NVIDIA ofrecen tanto software como hardware aplicado en conducción autónoma que nos van a ayudar a realizar nuestro objetivo: crear un prototipo de estudio de un coche autónomo a partir de una red neuronal (inteligencia artificial). Para ello, nos involucramos dentro de las librerías desarrolladas por Google en materia específica de redes neuronales: TensorFlow y KERAS. Estas serán cruciales para el desarrollo de nuestro cerebro automovilístico.

El software libre forma, junto con lo anterior, el pilar principal del desarrollo de las tecnologías inteligentes. En nuestro caso, nos valdremos de un sistema “BURRO” que conformará el firmware de nuestro vehículo y, a partir de ahí, trabajar en su implementación y adaptación.

Por tanto, no solo trabajaremos con entornos de programación para establecer una base sólida de funcionamiento del RC Car (coche radio control), sino que tocaremos de primera mano el diseño y puesta en marcha de una inteligencia artificial a partir de “**Deep Learning**”, con redes convolucionales, basadas en las librerías ya mencionadas de TensorFlow y Keras.

Logramos construir el prototipo físicamente sin dificultades e implementamos el firmware de manera exitosa. El diseño neuronal y su entrenamiento también resultaron según lo esperado, aunque un poco por debajo de nuestras expectativas. Aun así, el vehículo es capaz tanto de andar de forma manual como autónoma (aunque cierto porcentaje de error).

ABSTRACT

Nowadays, the mayor of the automated systems is implementing artificial intelligence which improves the user experience and gets a better tidy and efficient work. Smartphones' intelligent assistants are a great example of this. Due to this, it is not crazy to think that between all this industry sectors we found the autonomous vehicles.

There are a lot of companies which are implementing this kind of technology to their cars (Tesla as one of them) and others which offers tools and environments for getting the "machine learning" sector progress in a faster and proper way thanks to the users' contributions.

Once we have been immersed on the topic, we have found companies like NVIDIA industries which offers software and hardware applied to the autonomous conduction, helping us reaching our goal: make a studio prototype of and autonomous car based on a neural network (AI). For this, we are going to analyze in the newly developed libraries about neural science: TensorFlow and KERAS.

The open source system is, besides the previous statements, the primal fount of development of the intelligent technologies. In our case, we are going to serve ourselves from the "BURRO" project, which will be the base for the vehicle's firmware. From this, we are going to work on its implementation and its adaptation to our system.

Therefore, we are not only work on programing environments for setting a solid base of the RC car functional system, but also designing from firsthand and the startup of and artificial intelligent based on convolutional neural networks (CNN), made by the libraries of TensorFlow and KERAS.

Finally, we reach our goal about making a physical prototype without no difficulties and we implement the firmware successfully. The neural design and its training are above the expected results, too, but not as good as we thought. Even so, the vehicle can drive in manual and in autonomous way (with a certain percent of error).

ÍNDICE DE TABLAS

Tabla 1. Especificaciones del motor	25
Tabla 2. Especificaciones del Modelo ESC.....	26
Tabla 3. Especificaciones Cámara.....	27
Tabla 4. Especificaciones Raspberry Pi.....	28
Tabla 5. Características de la Navio 2.....	30
Tabla 6. Características del modelo Logitech	30
Tabla 7. Cálculo simple de la batería requerida	31

ÍNDICE DE FIGURAS

FIGURA 1. Principales sensores de un vehículo autónomo.....	12
FIGURA 2. Sensores de nuestro vehículo.	12
FIGURA 3. Muestra de uno de los circuitos de prueba.....	12
FIGURA 4. Artículo los vehículos de Francis p.Houdina	13
FIGURA 5. Foto de John McCarthy.....	14
FIGURA 6. Dean Pomerleau y Todd Jochem posando para el tour.....	15
FIGURA 7. Mapa de contorno que muestra la ruta del Gran Desafío	16
FIGURA 8. Vehículo de la Universidad de Carnegie Mellon	16
FIGURA 9. Vehículo de la Universidad de Stanford	17
FIGURA 10. Sebastian Thrun	17
FIGURA 11. Coche RC Racing Buggy.....	24
FIGURA 12. Eje transmisor de giro	25
FIGURA 13. Componentes internos Quantum Vandal	25
FIGURA 14. Motor 3652	25
FIGURA 15. WP-10BL50-RTR.....	26
FIGURA 16. Raspberry pi camera	27
FIGURA 17. Raspberry pi 3 B+.....	28
FIGURA 18. Navio 2 junto a Raspberry, Explicando sus módulos	29
FIGURA 19. Logitech F710	30
FIGURA 20. Curva Caída batería [Nimh vs lipo]	31
FIGURA 21. Soporte para los elementos del RC car.....	32
FIGURA 22. Vehículo real solo con la Rspberry y la cámara.....	32
FIGURA 23. Conexión Navio con Raspberry	33
FIGURA 24. Colocación de la cámara	33
FIGURA 25. Conexión de la cámara a la raspberry.....	34
FIGURA 26. Conexionado final completo	34
FIGURA 27. Montaje Final	35
FIGURA 28. Esquema resumen del Firmware “BURRO”.	35
FIGURA 29. Perceptrón simple o neurona	42
FIGURA 30. Ecuación actuación neuronal	42
FIGURA 31. Representaciones relu, sigmoide y tanh respectivamente	43
FIGURA 32. Estructura de una red neuronal	44
FIGURA 33. Representación de la evolución del error (Coste) por sgd	46

FIGURA 34. Aplicación de una capa convolucional.....	48
FIGURA 35. Aplicación de una capa MAX Pooling	49
FIGURA 36. Aplicación de una capa DROPOUT	50
FIGURA 37. Arquitectura de la red del modelo desarrollado por NVIDIA ..	51
FIGURA 38. Entrenamiento de la red neuronal.....	52
FIGURA 39. Logo de Unity	61
FIGURA 40. Noticia sobre el sector automovilístico en su página web	61
FIGURA 41. Interfaz del simulador	62
FIGURA 42. Circuito de la montaña	62
FIGURA 43. Circuito del lago.....	62
FIGURA 44. Interfaz del modo entrenamiento, selección del record path.	63
FIGURA 45. Interfaz del modo autónomo en funcionamiento.	64
FIGURA 46. Circuito de prueba 1 (laboratorio).....	66
FIGURA 47. Collage fotografías toma de entrenamiento 1	67
FIGURA 48. Circuito de prueba 2	67
FIGURA 49. Collage fotografías toma entrenamiento 2	68
FIGURA 50. Checking de versiones	69
FIGURA 51. Resultados del Entrenamiento neuronal	69

ÍNDICE DE CÓDIGOS

CÓDIGO 1. Configuración de parámetros de la cámara	37
CÓDIGO 2. Configuración de parámetros del coche Ackerman	37
CÓDIGO 3. Funcionalidad cambiar modo conducción.	38
CÓDIGO 4. Eliminar pilotos RC.....	39
CÓDIGO 5. Función para valores de ángulo y velocidad con mando.....	39
CÓDIGO 6. Clase para definir un modelo de neurona basado en regresión	40
CÓDIGO 7. Inicializar los datos de entrada para nuestra cámara	53
CÓDIGO 8. Modelo neuronal con los cambios implementados	53
CÓDIGO 9. Checkpoint de entrenamiento neuronal	55
CÓDIGO 10. Función para compilar el modelo	55
CÓDIGO 11. Función fit_generator para el entrenamiento del modelo....	55
CÓDIGO 12. Librerías importadas	56
CÓDIGO 13. Función lista_archivos.....	57
CÓDIGO 14. Función lista_rutas	57
CÓDIGO 15. Función colecting_steering.....	57
CÓDIGO 16. Ejecución principal, escritura del archivo CSV	58
CÓDIGO 17. Librerías importadas	58
CÓDIGO 18. Función lista_archivos.....	58
CÓDIGO 19. Función lista_rutas (local).....	58
CÓDIGO 20. Función de recolección del ángulo	59
CÓDIGO 21. Función principal, escritura del archivo CSV	59
CÓDIGO 22. Módulo args.parse sustituido por easydict	60

1. INTRODUCCIÓN

Actualmente en nuestra sociedad, la inteligencia artificial y el machine learning están revolucionando de forma silenciosa nuestras vidas. Asistentes virtuales, robo-advisors o vehículos autónomos, no serían posibles sin estas tecnologías.

“La inteligencia artificial (AI, en sus siglas en inglés), conjugada con el machine learning -o aprendizaje automático-, va a cambiar el mundo tal y como lo conocemos. Sundar Pichai, CEO de Google, afirmaba hace algunos meses que esta tecnología va a tener un impacto aún mayor que el que en su día tuvieron el fuego o la electricidad, como recogía CNBC.” [1]

Este salto significativo del uso masivo de estas tecnologías inteligentes está liderado por dos empresas importantes Google y NVIDIA.

Por un lado, Google es uno de los referentes en Inteligencia Artificial desde sus orígenes. Pero, con la creación del framework TensorFlow para desarrollar algoritmos inteligentes, no sólo ésta, sino muchas empresas están haciendo uso de dichas librerías para desarrollar sus propios productos.

Por otro lado, la empresa NVIDIA, con su tecnología de computo basada en GPU (Graphic Processing Units) lidera esta revolución en cuanto a soporte hardware para desarrollo de productos basados machine learning e IA.

Esto conforma la base de la revolución, aunque realmente no sería posible sin el apoyo y aportaciones de la comunidad. El uso de las licencias de Open Source (código abierto) facilitan que el desarrollo de este tipo de tecnologías se dispare significativamente. Plataformas como GITHUB ofrecen un amplio abanico de elecciones a los usuarios que quieran compartir sus proyectos y recibir aportaciones de la comunidad.

Como comentamos anteriormente, la conducción autónoma no se está librando de esta revolución, ya que cada vez más empresas del sector están empleando estas tecnologías para desarrollar productos inteligentes que mejoren la movilidad desde el punto de vista sostenible, de seguridad vial y del confort del usuario.

La Sociedad de Ingenieros de la Automoción (SAE) define 5 niveles de automatización para la conducción autónoma de vehículos (6 si contamos con el 0). Empezando por los más bajos, en los que los vehículos solo cuentan con sistemas de asistencia a la conducción para los que se precisa la presencia de un conductor. Hasta el nivel más alto, en el cual no es necesario un conductor ya que el vehículo contaría con sistemas de conducción autónoma totales.

Aunque ya existen vehículos de este último nivel, todavía su conducción está prohibida en la mayoría de los países, ya que existe cierta incertidumbre sobre este campo, como la optimización de los accidentes. Ante una situación de peligro, en la que el vehículo autónomo debe decidir entre la vida del pasajero o la de un viandante, ¿Cuál sería la decisión acertada? ¿Será un vehículo capaz de tomar decisiones que impliquen el uso de la moral humana?

Intentar montar por nuestra cuenta un vehículo autónomo de la suficiente envergadura para aproximar un caso real nos resulta un tanto imposible. Ya no solo hablando del precio de la tecnología necesaria para que éstos alcancen un nivel de fiabilidad adecuado, sino además de no disponer de un espacio de entrenamiento óptimo y la tecnología necesaria.

De esta manera, y como objetivo de nuestro proyecto, hemos desarrollado un prototipo de vehículo de conducción autónoma basado en la actuación de redes neuronales convolucionales a pequeña escala. A partir de entrenamientos, el coche podrá de llevar a cabo la predicción del ángulo de giro del eje de dirección de las ruedas hasta ser capaz de conducirse de forma autónoma.

Resolver un problema de estas características puede ser todo lo complejo que se quiera. Prueba de ello es la inversión en tiempo, dinero y recursos que están invirtiendo las grandes compañías automovilísticas, Tesla (Autopilot), Google (Waymo), General Motors (Cruise) o Mercedes-Benz (Daimler)).

Sin contar la inversión del propio auto, podemos ver en la figura 1 la gran cantidad de sensores (Lidar, GPS, radar, odometría, etc.) que lleva un sistema de estas características y la complejidad que supone la integración de los mismos en el modelo computacional para el coche vaya solo.

En nuestro caso particular y como objetivo de este proyecto, haremos una enorme simplificación. Tanto económica, un coche a escala de hobby (figura 2) como en complejidad, un solo sensor para ver la escena, cámara.

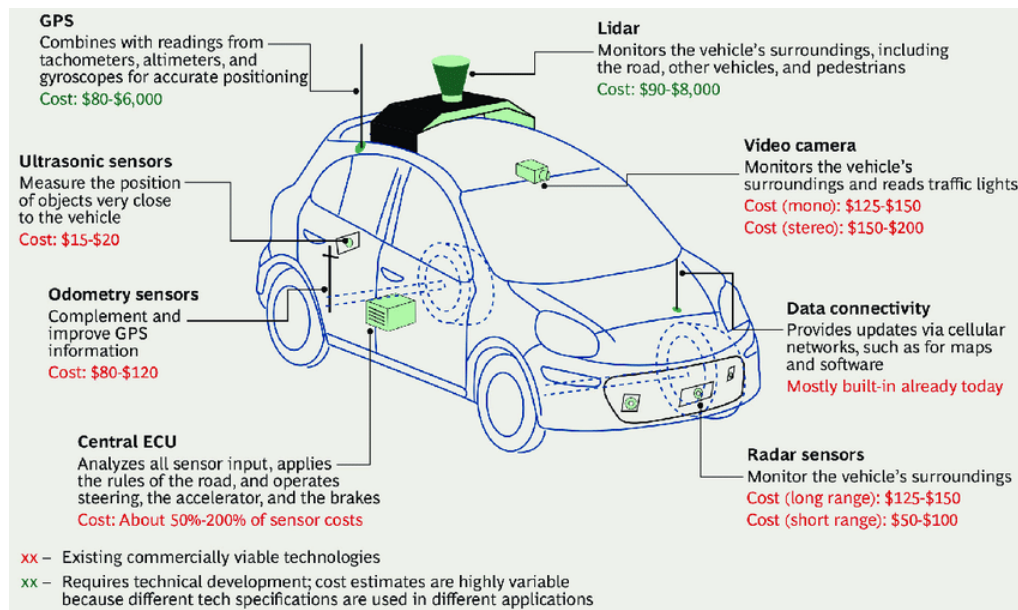


FIGURA 1. PRINCIPALES SENSORES DE UN VEHÍCULO AUTÓNOMO

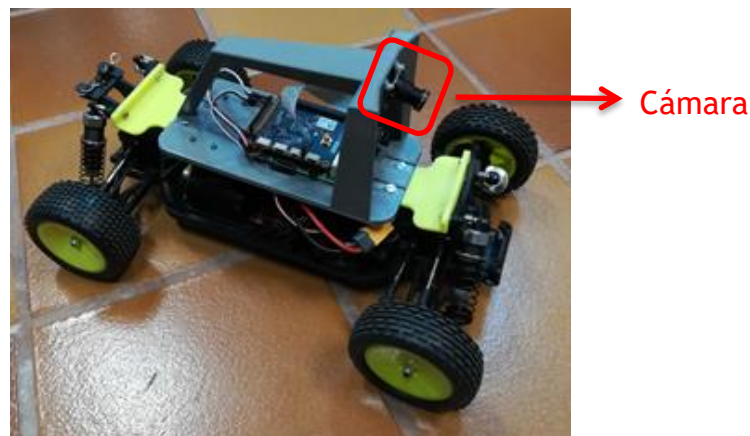


FIGURA 2. SENSORES DE NUESTRO VEHÍCULO.

Como hipótesis, queremos que el vehículo circule por un escenario definido por unas líneas, como el de la figura 3. Este escenario es catado por una única entrada (cámara) y dos salidas: ángulo de la dirección y aceleración. Con esto el vehículo tiene que ser capaz de circular en el circuito acotado por líneas.



FIGURA 3. MUESTRA DE UNO DE LOS CIRCUITOS DE PRUEBA

1.1. CONDUCCIÓN AUTÓNOMA. ANTECEDENTES

1.1.1. FRANCIS P. HOUDINA.

Aunque la conducción autónoma sea un tema todavía en auge hoy en día, no es ni para menos algo nuevo. Podemos remontarnos hasta 1925 en donde se considera extraoficialmente que comenzó la era del coche sin conductor. El inventor Francis P. Houdina quien, ante la atenta mirada de los ciudadanos que se acercaron a su exhibición, mostro un vehículo capaz de arrancar, girar y hacer sonar el claxon a distancia mediante ondas de radio. Para la muestra condujo este vehículo por las calles de Manhattan desde otro que lo precedía a distancia.

Se conoce extraoficialmente debido a que 4 años antes, los militares habían presentado un vehículo con el mismo sistema de conducción, que condujeron por las calles de Dayton (Ohio).



FIGURA 4. ARTÍCULO LOS VEHÍCULOS DE FRANCIS P.HOUDINA

1.1.2. JOHN MCCARTHY.

Los inicios de la IA se establecen en el año 1956 en una conferencia ofrecida por John McCarthy en la universidad de Darmouth en la que planteo

los objetivos de su estudio: *“Este estudio procederá sobre la base de la conjetura de que todos los aspectos del aprendizaje o cualquier otro rasgo de la inteligencia pueden, en principio, ser descritos de una forma tan precisa que se puede crear una máquina que los simule”*. [2]

McCarthy presentó en 1969 un ensayo en el que describe algo similar a un vehículo autónomo moderno. Este ensayo titulado como “Computer Controlled Cars” describe el funcionamiento de lo que denomina un “chofer automático” que era capaz de controlar un vehículo por la vía pública, mediante la captura de imágenes de una cámara de televisión, que correspondería a la visión de un conductor humano.

También se podría establecer mediante un teclado, la posición de destino a la que el vehículo se dirigiría inmediatamente.



FIGURA 5. FOTO DE JOHN MCCARTHY

1.1.3. DEAN POMERLEAU.

Todo esto simplemente era una base teórica estipulada por McCarthy, no fue hasta 1990 en donde Dean Pomerleau investigador de la Universidad Carnegie Mellon de Pensilvania presenta una tesis doctoral en la que describe como las redes neuronales podrían enviar órdenes a los controles de dirección de un vehículo basadas en imágenes tomadas por el mismo.

Pomerleau no era el único investigador trabajando en vehículos autónomos en esta época, pero fue el que demostró que el uso de redes neuronales era mucho más eficiente que la división de las imágenes manualmente por categorías.

La anterior tesis fue la base con la que en conjunto con su colega investigador Todd Jochem, crearon su sistema Navlab de autoconducción. Consistía en una minivan en la que los controles de velocidad dependían del conductor, pero en la que el sistema de giro era un sistema totalmente autónomo.

Con este vehículo recorrieron 2 797 millas desde Pittsburgh, Pensilvania hasta San Diego, California, a este trayecto lo denominaron “Sin manos a través de América”.



FIGURA 6. DEAN POMERLEAU Y TODD JOCHEM POSANDO PARA EL TOUR

1.1.4. DARPA.

En el año 2002, la Agencia de Investigación de Proyectos Avanzados de Defensa (Defense Advanced Research Projects Agency, **DARPA**) de los Estados Unidos, responsable del desarrollo de nuevas tecnologías para uso militar, anuncia su “Gran Desafío” ofreciendo 1 millón de dólares a los investigadores que consigan crear un vehículo autónomo capaz de recorrer las 142 millas del desierto de Mojave que separan Barstow (California) de Primm (Nevada).



FIGURA 7. MAPA DE CONTORNO QUE MUESTRA LA RUTA DEL GRAN DESAFÍO

1.1.4.1. Gran Desafío, 2004.

El 13 de marzo de 2004, día acordado para el que sería el primer Gran Desafío, se presentaron 15 vehículos autónomos para su salida desde Barstow. La rudeza del desierto como la limitada tecnología con la que contaban, demostraron ser demasiado para el intento del desafío. Ninguno de los vehículos consiguió llegar a la meta. El vehículo considerado como ganador perteneciente a la Universidad de Carnegie Mellon, solo pudo completar 7.5 millas del recorrido durante varias horas antes de incendiarse.



FIGURA 8. VEHÍCULO DE LA UNIVERSIDAD DE CARNEGIE MELLON

1.1.4.2. Gran Desafío, 2005.

En el Gran Desafío de 2004, dado a que ningún participante logró el objetivo, el premio no fue reclamado. Aun ante esta situación, la competición no fue vista como una pérdida, sino que ofreció una visión de lo que se podría

conseguir. Tal fue así, que un día después DARPA anunciaba su Segundo Gran Desafío.

En 2005, 18 meses después del primero, se lanzó el Segundo Gran Desafío, en el que participaron en esta ocasión 195 equipos de los cuales 5 consiguieron terminar el recorrido. Siendo el equipo de la Universidad de Stanford codirigido por Sebastian Thrun, el vencedor con un tiempo de 6 horas, 53 minutos y 58 segundos, ganando esta, 2 millones de dólares.



FIGURA 9. VEHÍCULO DE LA UNIVERSIDAD DE STANFORD



FIGURA 10. SEBASTIAN THRUN

1.1.4.3. Desafío Urbano, 2007.

El 3 de noviembre de 2007 se celebró el tercer desafío DARPA, el Urban Challenge en las calles de Victorville, California.

Este reto, como su nombre indica, estipula un objetivo diferente a los anteriores. Debido al éxito del Gran Desafío de 2005, se buscó ir un paso más adelante con el hándicap de que en esta ocasión, los vehículos no solo debían llegar a la meta, sino que además debían hacerlo por las calles de Victorville mientras sorteaban el tráfico y peatones a la vez que cumplían con las normas de circulación vigentes.

Esto parece un paso bastante grande, con respecto al anterior, ya que no solo debía disponer de los sistemas de orientación tanto para alcanzar el destino como para estacionar, sino que debía captar e interpretar multitud de obstáculos en su camino como diferentes tipos de vehículos, señales de tráfico, semáforos, peatones, intersecciones... y tomar una decisión como si de un conductor humano se tratara.

Aun con estos requisitos, a las 13:43 el vehículo del equipo de la Universidad Carnegie Mellon cruzaría la línea de meta en primer lugar con un tiempo de aproximadamente 4 horas, seguido del equipo de la Universidad de Stanford, ganadores del Gran Desafío anterior, acabarían cruzando la meta otros cuatro vehículos que también completarían el desafío.

Este evento fue tan revolucionario que demostró al mundo que la conducción autónoma era posible.

Estos desafíos ayudaron a crear una mentalidad y una comunidad de investigación que, más tarde, no acabarían consiguiendo que los vehículos autónomos circulen hoy en día por las carreteras como algo común, pero si se han construido vehículos que ya cuentan con una gran cantidad de kilómetros en pruebas, como son Tesla (Autopilot), Google (Waymo), General Motors (Cruise) o Mercedes-Benz (Daimler).

1.2. OBJETIVOS

El objetivo principal de este proyecto es el diseño e implementación de un prototipo de vehículo autónomo (RC car) a tamaño reducido (escala 1:10) basado en inteligencia artificial programada con las librerías y API de Google (TensorFlow y Keras). El prototipo será capaz de predecir el ángulo de giro en marcha ante un circuito, para el cual habrá sido entrenado con anterioridad, sólo con una imagen de entrada que le sirva de referencia.

Para conseguir este objetivo general nos valdremos de la consecución de objetivos específicos.

1. Búsqueda de un prototipo de coche abierto y alcanzable a nuestras posibilidades donde podamos implementar el modelo de conducción autónoma.
 - Que sea eléctrico y con cierta autonomía y de fácil accesibilidad para su control.
2. Búsqueda de un sistema empotrado para el control de todo el sistema, así como el diseño de las interfaces humanas para su acceso de monitorización y control.
3. Integración de los sensores y actuadores con sus drivers en el firmware del vehículo.
4. Diseño, implementación y entrenamiento de un modelo de red neuronal capaz de llevar al vehículo por el entorno o escena diseñada para tal fin.
5. Ajuste del sistema mediante pruebas en el circuito de entrenamiento.

1.3. ESTRUCTURA DE LA MEMORIA

En la siguiente memoria, se van a reflejar todos los aspectos teóricos y prácticos que se han tenido que sobrellevar para la realización y diseño del prototipo. Hablaremos también de todas las herramientas usadas durante el proceso, así como las ventajas que nos aportan dichas plataformas digitales u otros recursos.

En un primer apartado se explicarán las metodologías (fases en las que se compone este proyecto) y herramientas empleadas para cumplir con los objetivos. A continuación, en siguiente apartado, se describen las especificaciones de los componentes Hardware que conforman el vehículo, como las modificaciones del Firmware del proyecto open source “Burro” realizadas. En los siguientes y últimos dos apartados se describirá tanto la teoría como el procedimiento de creación de la red neuronal como los resultados y conclusiones obtenidos del desempeño del proyecto.

2. METODOLOGÍAS Y HERRAMIENTAS

En este apartado nos vamos a centrar en los entornos utilizados para desarrollar y llevar a cabo cualquier aspecto del proyecto. No sólo nos vamos a centrar en IDE o entornos de programación, sino en librerías especializadas y plataformas online que nos han ayudado a realizar nuestro trabajo de la manera más eficaz posible.

Respecto a las metodologías, haremos hincapié en el modus operandi que hemos empleado para trabajar en conjunto, puesto que somos un equipo de dos apoyándonos en el compañero continuamente.

2.1. METODOLOGÍAS.

La metodología para alcanzar los objetivos planteados se basará en una metodología secuencial e iterativa por pareja. Esta metodología relacionará los resultados obtenidos y las actividades/tareas de análisis y diseño definidas en trabajos de laboratorio.

Esta la dividiremos en 4 fases de trabajo:

1. **Búsqueda de información, documentación y análisis.**

En esta fase comenzamos con la búsqueda de información de las tecnologías a emplear o existentes, así como de proyectos Open Source en conducción autónoma ya definidos por usuarios en internet de los que podamos adquirir alguna utilidad. Del mismo modo, para fomentar la investigación de los medios que vamos a emplear, realizamos cursos de formación específica para trabajar con redes neuronales (basadas sobre todo en TensorFlow y KERAS).

2. **Planificación y organización de las tareas.**

Con la información a partir del análisis en la fase anterior, en esta fase planificaremos todas las tareas necesarias para llevar al proyecto a buen puerto. Una vez obtenida, tomamos la decisión de utilizar un proyecto Open Source en desarrollo que encaja totalmente con nuestro objetivo, aunque como se encuentra en desarrollo debemos añadir tareas de resolución de errores y modificaciones para que encaje totalmente con el objetivo. Además, una vez conocemos la base, pasamos a planificar y organizar las tareas por el orden que deberemos seguir para hacer lo más eficiente posible el desarrollo.

3. **Diseño e implementación del prototipo.**

Esta es la fase principal que es donde se encuentran las tareas específicas para conseguir los objetivos parciales definidos anteriormente.

- Tarea de construcción del prototipo
- Tarea de diseño e implementación del Firmware del prototipo
- Tarea de diseño e implementación del modelo de Red neuronal

Para ello acordamos en primer lugar comprar los componentes y montar todo el hardware del prototipo, y a continuación, el **estudio del código funcional** que conformará la base del vehículo, no incluyendo en él la neurona. Esta tarea requiere de un **proceso de investigación** en profundidad, entendiendo el código, estructurándolo bien y realizando las modificaciones específicas para que funcione de la misma manera en nuestro sistema. Este código será el firmware del vehículo.

4. Elaboración de la documentación y evaluación.

En último lugar, se realizará la documentación en la que se detalla tanto el esfuerzo realizado para el desarrollo del proyecto y todo lo aprendido en el camino, como la evaluación de lo logrado con el mismo.

2.2. HERRAMIENTAS.

2.2.1. GIT.

Se trata de un software de control de versiones de código en la nube (GitHub) en el que se registran cada uno de los cambios realizados al código del proyecto base y gracias al que tanto si se comete un error como si se decide que el funcionamiento en alguna de las versiones anteriores era más adecuado, se podría volver a esta sin ningún esfuerzo conservando también los cambios hasta el momento.

Y todo esto sin tener que guardar multitud de copias del código para cada uno de los cambios que se quisiera conservar ante algún imprevisto. Actualmente, muchos de los repositorios de los usuarios se encuentran en la nube y son Open Source, es decir, que cualquiera puede aportar mejoras.

2.2.2. Google Drive.

Una de las herramientas más eficientes para guardar archivos y que siempre estén disponibles desde cualquier dispositivo, en otras palabras, toda la información se guarda en la nube. Con Google Drive, hemos conseguido tener acceso a toda la reserva de imágenes de datos del coche para poder entrenar la red neuronal.

2.2.3. Google Colab.

Esta herramienta nos permite establecer un entorno de programación por celdas de forma gratuita. Este tipo de entornos están basados en el formato de Jupyter Notebook que no requieren configuración y se ejecutan completamente en la nube. En Colab, se permite escribir y ejecutar código, compartir archivos fácilmente desde Google Drive e incluso exportar tus Notebooks a GITHUB. Además, Colab ofrece muchas librerías de fácil acceso, incluso fragmentos de código para funciones complejas e interactivas.

En nuestro caso, gracias a Google Colab, hemos podido **desarrollar todo el entrenamiento neuronal en esta plataforma** ya que nos ofrecía un alto grado de libertad, pudiendo elegir el tipo de versión de KERAS y TensorFlow con la que poder trabajar, a diferencia de si lo hubiéramos hecho en Windows de forma local, puesto que TensorFlow no salió para este sistema operativo hasta una versión más avanzada que el prototipo no soporta (preparadas para Python 3).

2.2.4. IntelliJ/PyCharm.

Estas herramientas se les conocen por IDE (del inglés “*Integrated Development Enviroment*”), en otras palabras, son entornos de programación que ofrecen un editor inteligente, navegación entre códigos y archivos en un mismo proyecto e inspecciones y correcciones de posibles errores en él. Incluso cuentan con herramientas de “*debugging*” y testeo, así como implementación de entornos de ejecución y consolas del sistema.

Con esto, conseguimos editar y ejecutar los códigos desarrollados para la parte específica del tratamiento de datos (ficheros CSV que veremos más adelante) y el entrenamiento en local de la neurona para la simulación (en la plataforma Unity, que también veremos en partes posteriores).

2.2.5. Microsoft Word.

Es la herramienta de software ofimático para la redacción de documentos más puntera de Microsoft. Combina la capacidad visual con la técnica de procesamiento de formatos. Principal herramienta de trabajo para desarrollar esta memoria.

2.2.6. Simulador Unity.

A partir del epígrafe 4, veremos que el simulador desarrollado por Udacity a través de la plataforma de entornos 3D Unity nos proporcionará una visión del funcionamiento de las redes neuronales y nos dará las claves para poder entrenar nuestra propia red. En el epígrafe 4.5 se entra en más detalle a cerca de Unity y el simulador.

2.2.7. Librerías para IA: TensorFlow y KERAS

Dentro del mundo de las inteligencias artificiales, existen varias librerías para su desarrollo, pero la más completa y actualizada es TensorFlow. Una librería diseñada en C++ por la gente de Google con una API especializada: KERAS. KERAS nos ofrece todas las ventajas de TensorFlow de forma fácil y eficaz.

La ventaja que nos proporciona usar esta tecnología es que algo tan complejo como son los cálculos computaciones sobre entrenamiento y diseño de redes neuronales quedan reducidos a unas simples líneas de código. En otras palabras, gracias a estas librerías implementar una red neuronal es mucho más fácil para principiantes. Todos nuestros códigos y modelos neuronales están basados en KERAS/TensorFlow.

3. DISEÑO E IMPLEMENTACIÓN HARDWARE/SOFTWARE DEL RC CAR

En este capítulo nos sumergiremos en la selección de componentes del vehículo, así como la estructura de programación utilizada que compone los cimientos para trabajar y desarrollar el proyecto. Se detallarán algunas de las especificaciones más relevantes de cada una de las piezas. Finalmente se introducirán los conceptos que mayor peso tengan para poder entender el funcionamiento de “BURRO”, Software/Firmware del RC Car.

3.1. VEHÍCULO RC BASE.

En el presente proyecto se utiliza un coche a radiocontrol (RC) a escala 1:10 como base. Éste fue adquirido a la empresa Hobby King con modelo QUANUM VANDAL 1:10 4WD ELECTRIC RACING BUGGY (ARR), ya que dispone de los sistemas mínimos para la movilidad del vehículo. A este RC se le añaden otros sistemas que serán mencionados en su correspondiente apartado junto con su propósito.



FIGURA 11. COCHE RC RACING BUGGY

El vehículo cuenta con tracción a las 4 ruedas gracias a su eje de transmisión central (figura 11), que distribuye la potencia del motor de propulsión a ambos ejes (delantero y trasero). También dispone de un **variador electrónico de velocidad (ESC)** que permite controlar la velocidad de giro del motor como su sentido. Además, dispone de un servo motor delantero para el control de la dirección. En la figura 12 y 13 se puede ver el montaje completo con detalles de los componentes.



FIGURA 12. EJE TRANSMISOR DE GIRO

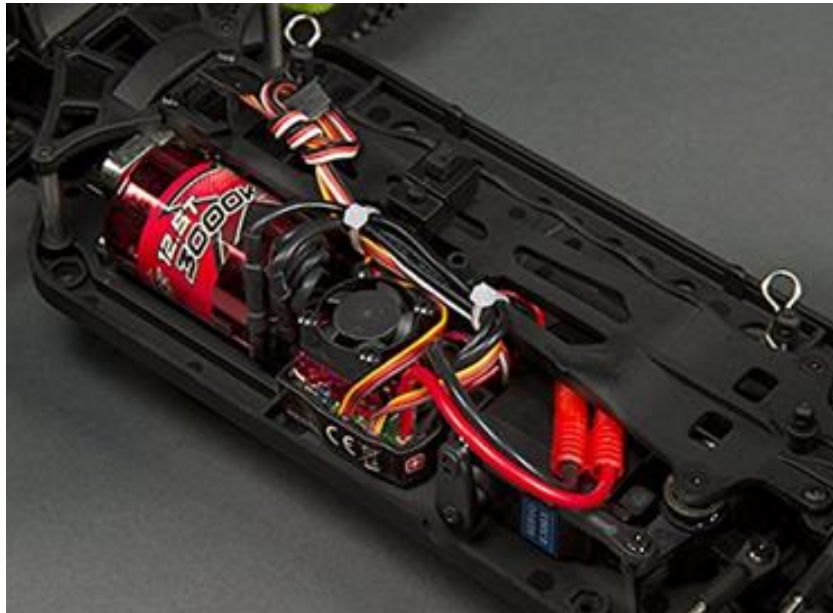


FIGURA 13. COMPONENTES INTERNOS QUANUM VANDAL

3.1.1. Motor.

El primer componente interno del vehículo base corresponde al motor. Se trata de un motor de potencia de tipo sin escobillas (del inglés “*brushless*”), de la empresa Hobbywing, con 4 polos y un **Kv rating**¹ de 3250.

TABLA 1. ESPECIFICACIONES DEL MOTOR



Marca	Hobbywing
Modelo	3652
tipo	Brushless (sin escobillas)
Nº polos	4
Kv rating	3250

FIGURA 14. MOTOR 3652

¹ **Kv rating:** se refiere a la velocidad constante de un motor. Es la relación entre las revoluciones por minuto (rpm) sin carga, con una alimentación de 1 voltio en sus bornes.

3.1.2. Variador electrónico de velocidad (ESC).

El siguiente componente (ESC, figura 15) cumple la función de variar la velocidad de giro del motor, así como su sentido. Las señales para este fin se las proporcionará la Navio 2.

Este componente pertenece a la empresa Hobbywing, siendo el modelo WP-10BL50-RTR que permite una corriente máxima de alimentación al motor de 50A en continuo o de 300A de pico.

Otro aspecto importante de éste es que incorpora un sistema **BEC** que proporciona una alimentación regulable y estable diferente a la de potencia designada para el motor. Este sistema proporciona una salida de 3A/6V utilizada normalmente para la alimentación del receptor RC y el servo. En nuestro caso, solo emplearemos el cable de información (blanco) para la señal PWM que se le envía desde la Navio para controlar la velocidad.



FIGURA 15. WP-10BL50-RTR

TABLA 2. ESPECIFICACIONES DEL MODELO ESC

Marca	Hobbywing
Modelo	WP-10BL50-RTR
Fuente de alimentación	2-3S LiPo o 4-9 células NiMH/NiCd
Protección	IP67
Salida BEC	3A/6V (modo interruptor)
Corriente máxima	50A en continuo/300A de pico
Refrigeración	Ventilador

Queremos resaltar que antes de hacer la puesta en marcha del vehículo la primera vez, hay que tener en cuenta que, cada vez que se modifique el valor de la salida al motor de potencia en el código, hay que calibrar el ESC. Es decir, el código se vale de una señal analógica (0 - 255) para modificar la velocidad del vehículo. Y, por tanto, hay que calibrar el ESC para que reconozca dicho rango de la señal como el rango de potencia que es capaz de suministrar al motor (0 - 100%). El proceso de calibración de este la aporta el fabricante en la documentación adjunta con el vehículo.

3.2. CÁMARA.

Otra parte importante que compone el sistema es la cámara. Se trata de una cámara “Longruner” (figura 16) compatible con Raspberry que se empleará para la obtención de las imágenes que serán procesadas y analizadas por la red neuronal para tomar la decisión sobre el giro (ángulo del servo motor) en base a la situación que identifique en dicha imagen con respecto a lo que conoce hasta el momento.

La cámara es un modelo OV5647 con una resolución de 5 MP, capaz de grabar a 1080p (píxeles) a 30 fps (Frames Per Second) o a 720p a 60fps. Los fps hacen referencia a la cantidad de imágenes por segundo que la cámara puede tomar.

Además de lo anterior, también cuenta con visión nocturna, si se les instalan los correspondientes LEDs a sus laterales (incluidos por el fabricante), un sensor de ¼ de pulgada y un objetivo con apertura de f2.9 (determina la cantidad de luz que puede llegar al sensor a través del diafragma) y una distancia focal de 3.6 mm (distancia entre el centro de la lente y el foco). Estos datos determinan el ángulo de visión (parte de la escena captada por la cámara).



FIGURA 16. RASPBERRY PI CAMERA

TABLA 3. ESPECIFICACIONES CÁMARA

Marca	Longruner
Modelo sensor	OV5647
Resolución	5 MP - 1080p@30fps - 720p@60fps
Tensión alimentación	3.3 V
Visión nocturna	Sí
Sensor CMOS	1/4
Apertura	f2.9
Distancia focal	3.6 mm

3.3. RASBERRY PI 3 B+.

Otro componente fundamental es el que le da soporte computacional al RC car: el sistema empujado. Éste está basado en una Raspberry PI 3 B+ ampliamente extendida hoy en día. Básicamente, sus tareas son: adquisición de las imágenes y usarlas como entrada al modelo neuronal, actuar sobre la dirección y aceleración del vehículo, crear el bus de comunicaciones necesarias entre la interfaz de usuario y el mando y comunicarse con la NAVIO para controlar la potencia del motor.



FIGURA 17. RASBERRY PI 3 B+

Esta cuenta con un procesador Broadcom BCM2837B0 con una arquitectura de 64 bits y 4 núcleos a una velocidad de procesamiento de 1.4 GHz. También cuenta con 1 GB de memoria RAM (Random Acces Memory).

TABLA 4. ESPECIFICACIONES RASBERRY PI

Marca	Raspberry
Modelo	3 B+
Procesador	Broadcom BCM2837B0
Arquitectura	64 bits
Tipo de procesador	Cortex – A53
Núcleos	4
Velocidad	1.4GHz
RAM	1GB SDRAM4 LPDDR2
Alimentación	5V / 2.5A DC

3.4. NAVIO 2.

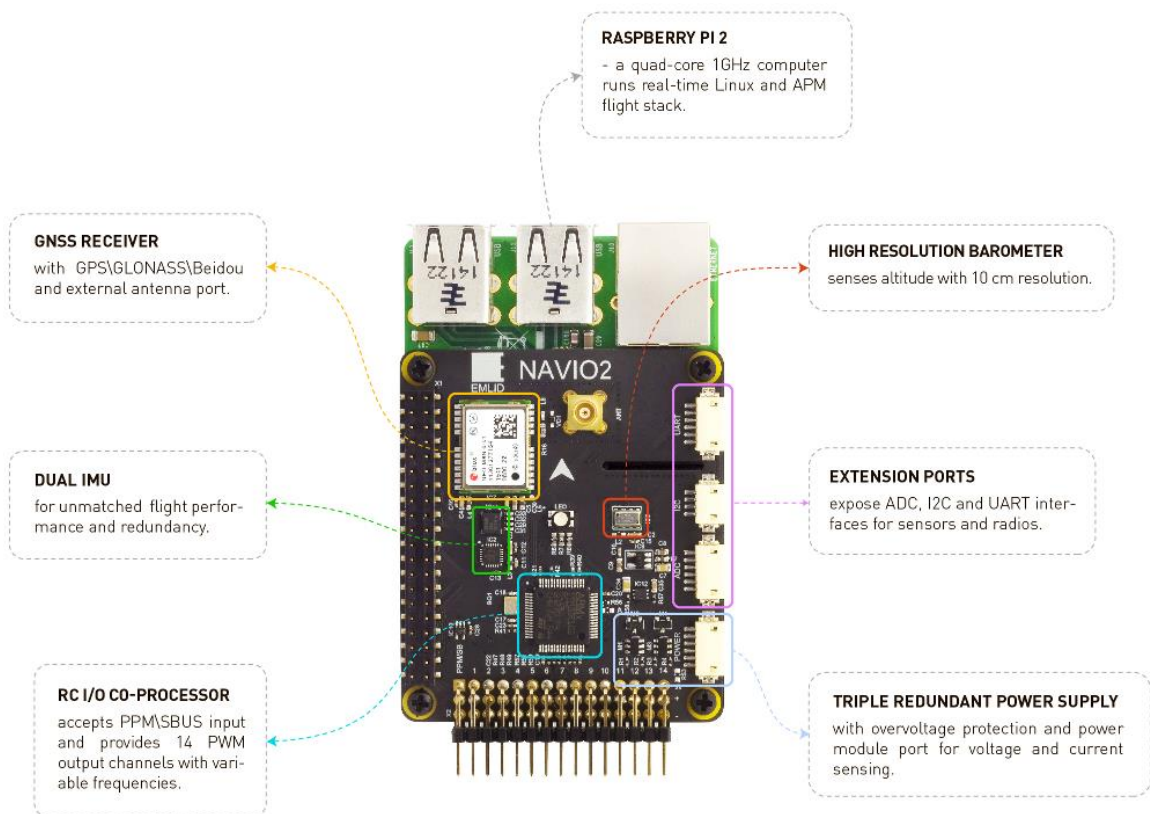


FIGURA 18. NAVIO 2 JUNTO A RASPBERRY, EXPLICANDO SUS MÓDULOS

La placa Navio 2 junto con Raspberry Pi 3 (figura 18) forman la unidad de control de nuestro vehículo. Se trata de un dispositivo que cuenta con una gran cantidad de funcionalidades (sensores), aunque nos valdremos en un principio de ella para controlar las señales PWM dirigidas al servo motor de la dirección y el giro.

Navio 2 cuenta con una gran variedad de sensores incorporados. Entre estos podemos encontrar un receptor GNSS (Global Navigation Satellite System) capaz de rastrear multitud de satélites diferentes (GPS, GLONASS, ...), dos IMU (unidades de medidas inerciales) que encapsula acelerómetros, giroscopios y magnetómetros. Un coprocesador de E/S de RC que proporciona 14 canales de salida PWM para motores y servos. Un barómetro de alta resolución, 10 cm. Puertos de Extensión I2C, UART y ADC para sensores y radios. Y tres fuentes de alimentación redundantes con protección contra sobretensiones.²


² Las especificaciones de la Navio 2 han sido citadas de la página oficial de la misma. 

TABLA 5. CARACTERÍSTICAS DE LA NAVIO 2

Marca	Emlid
Modelo	2
Sensores	MPU9250 9DOF IMU
	LSM9DS1 9DOF IMU
	MS5611 Barometer
	U-blox M8N Glonass/GPS/Beidou
Alimentación	Triple Fuente redundante
Interfaces	UART, I2C, ADC for extensions
	PWM/S.Bus input
	14 PWM servo outputs

3.5. MANDO LOGITECH F710.

Un Logitech F710 se emplea para el control del vehículo en modo manual, así como para el cambio entre manual y autónomo. Dispone de un receptor bluetooth USB a 2.4 GHz para conectar a la Raspberry. Su sistema de alimentación consta de dos pilas AA de 1.5V cada una. También cuenta con 20 canales de información referentes a cada uno de los botones o ejes de los que dispone.



FIGURA 19. LOGITECH F710

TABLA 6. CARACTERÍSTICAS DEL MODELO LOGITECH

Marca	Logitech
Modelo	F710
Alimentación	2 pilas AA
Conexión inalámbrica	2.4GHz

3.6. BATERÍA.

Para la elección de la batería calculamos el consumo base del vehículo, siguiendo las recomendaciones base:

TABLA 7. CÁLCULO SIMPLE DE LA BATERÍA REQUERIDA

Componente	Consumo
Raspberry Pi 3 B+	Min – 230 mA
	Max – 350 mA
Motor + ESC	4.3A (recomendado) (consumo máx. 50A)
TOTAL	4700 mA

La mayor parte de los dispositivos toman la alimentación desde la NAVIO 2, la cual es alimentada por la NAVIO. En base a los consumos facilitados por los fabricantes, calculamos que lo recomendado es seleccionar una batería de unos 5000 mAh, seleccionamos una **batería LiPo 2S (2 celdas), 5200 mAh, 7.4V y 30C**. Al ser 30C significa que en lugar del amperaje anterior es capaz de suministrar 30 veces dicho amperaje, pero su tiempo de descarga se reducirá también.

La selección de la composición se debe a que la densidad de energía es mejor en las LiPo (Litio - Polímero) que en la NiMh (Níquel - Hidruro metálico) al igual que la curva de caída.

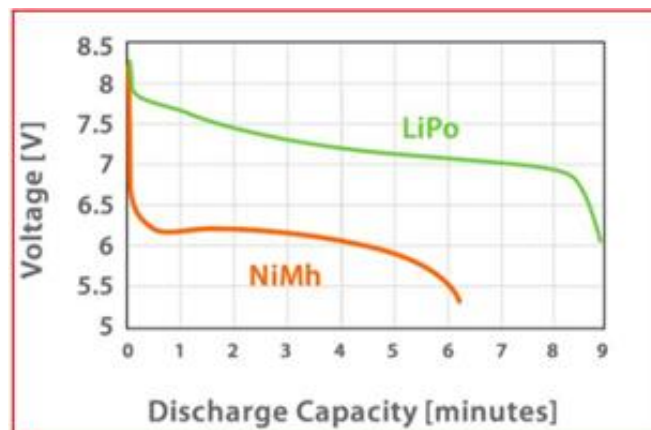


FIGURA 20. CURVA CAÍDA BATERÍA [NiMH VS LIPO]

3.7. INTEGRACIÓN DEL HARDWARE.

Para el ensamblaje, en primer lugar, imprimimos la estructura que soportara el anclaje de las placas de control, como de la cámara. El proyecto DonkeyCar del cual partimos para este proyecto nos facilita un modelo impreso en 3D por 50 dólares, pero hemos conseguido otro significativamente parecido y totalmente libre, impreso en los laboratorios del departamento de la Escuela.



FIGURA 21. SOPORTE PARA LOS ELEMENTOS DEL RC CAR

El inconveniente de este modelo es que no encaja con el chasis de nuestro vehículo, por lo que hemos tenido que crear otras piezas para adaptarlo (piezas amarillas, figura 22). Una vez impresas las piezas necesarias, se quita la protección superior, y se anclan las piezas aprovechando soportes ya existentes.

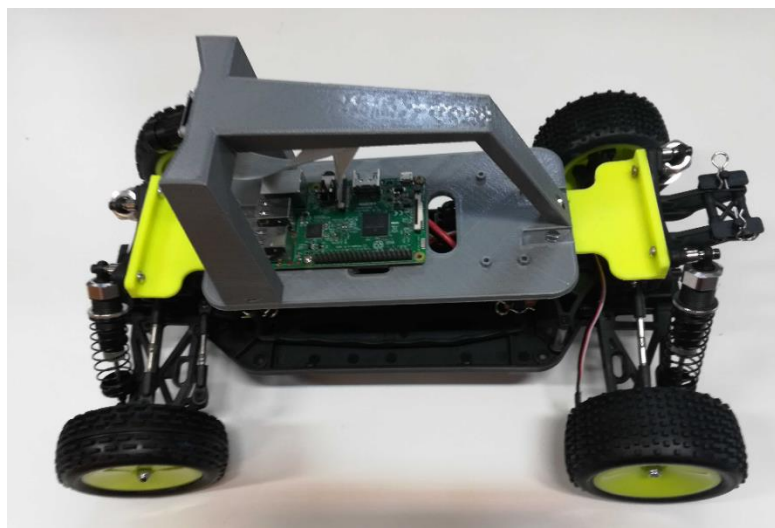


FIGURA 22. VEHÍCULO REAL SOLO CON LA RSPBERRY Y LA CÁMARA

A continuación, se instala la Navio sobre la Raspberry, se disponen los separadores que nos proporciona el fabricante, conectando ambas placas de la siguiente forma (ver figura 23). La unión de ambas placas se coloca en los anclajes destinados para ellas en la base impresa.



FIGURA 23. CONEXIÓN NAVIO CON RASPBERRY

Con la estructura y las placas de control ya montadas, solo queda anclar la cámara en la parte superior y la batería en el soporte del chasis destinado para ello.

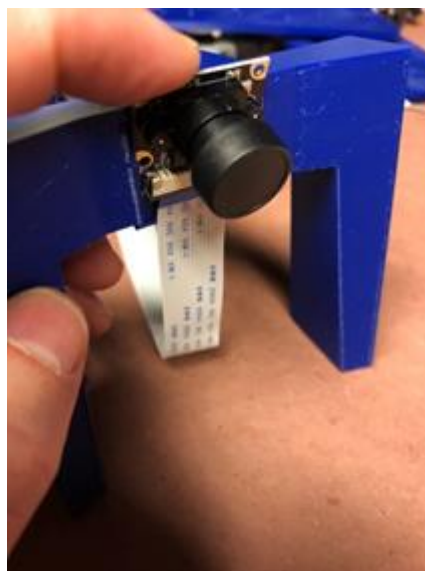


FIGURA 24. COLOCACIÓN DE LA CÁMARA

Sólo falta el conexionado de los diferentes componentes:

- El receptor bluetooth USB se conecta a uno de los puertos de este tipo de los que dispone la Raspberry Pi.
- La Raspberry Pi también dispone de una conexión especial para la cámara.

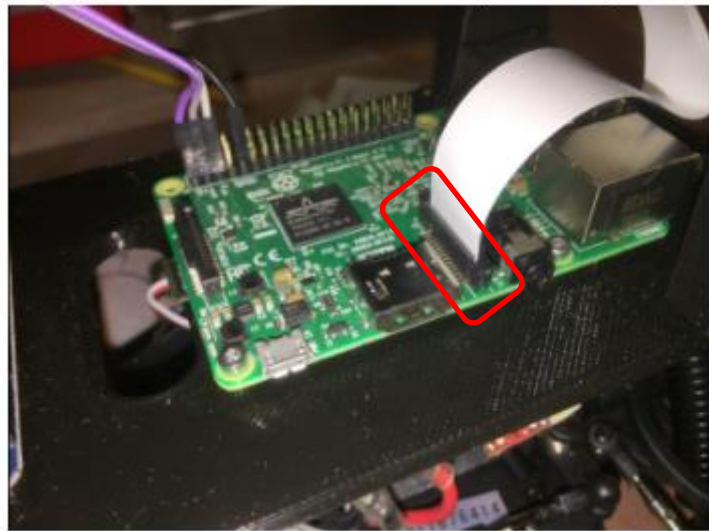


FIGURA 25. CONEXIÓN DE LA CÁMARA A LA RASPBERRY

El resto de las conexiones se resumen en el siguiente esquema:

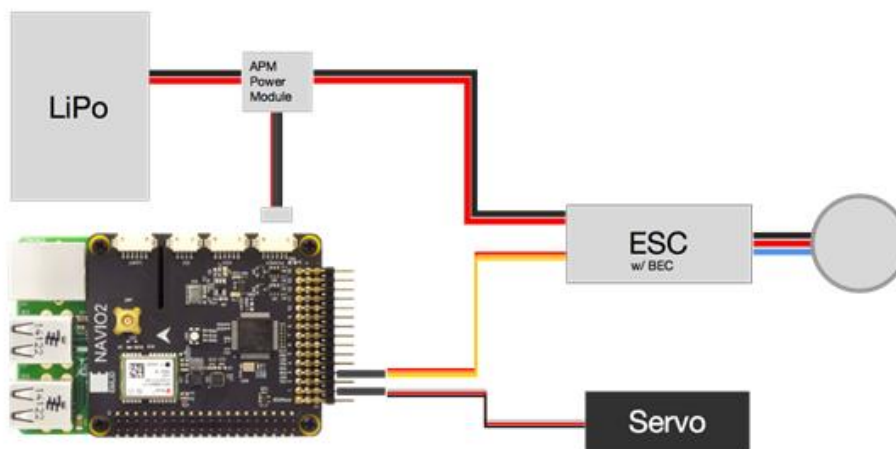


FIGURA 26. CONEXIONADO FINAL COMPLETO

El ESC se conecta con la entrada número 1 y el servo motor con la 2. Esta configuración viene dada por el código base del proyecto “Burro” que seguiremos. Una vez realizado todo el montaje, el prototipo queda como se ve referenciado en la figura 27.

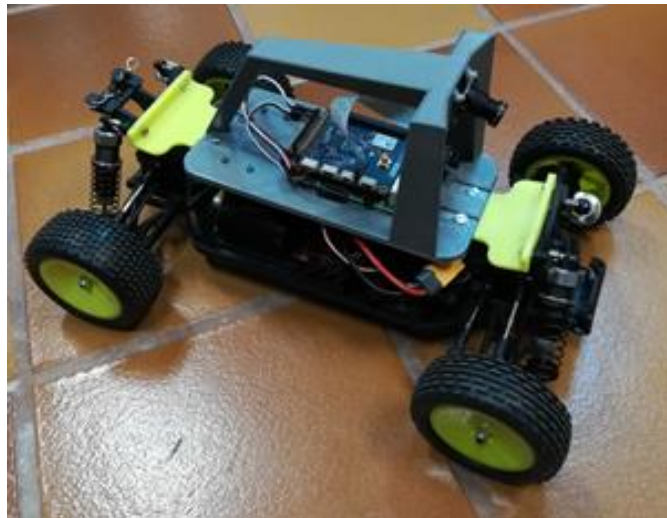


FIGURA 27. MONTAJE FINAL

3.8. FIRMWARE DE CONTROL DEL VEHÍCULO. BURRO SOFTWARE.

El proyecto se basa en uno ya existente Open Source llamado “BURRO” que, a su vez, modifica otro proyecto, “DonkeyCar”, para la utilización de Navio 2 en lugar de un controlador de motores de Arduino. Ambos proyectos son Open Source, basados en programación orientada a objetos (POO) con Python y se encuentran en la plataforma Github.

A continuación, se describe brevemente el funcionamiento principal del Firmware “BURRO”.

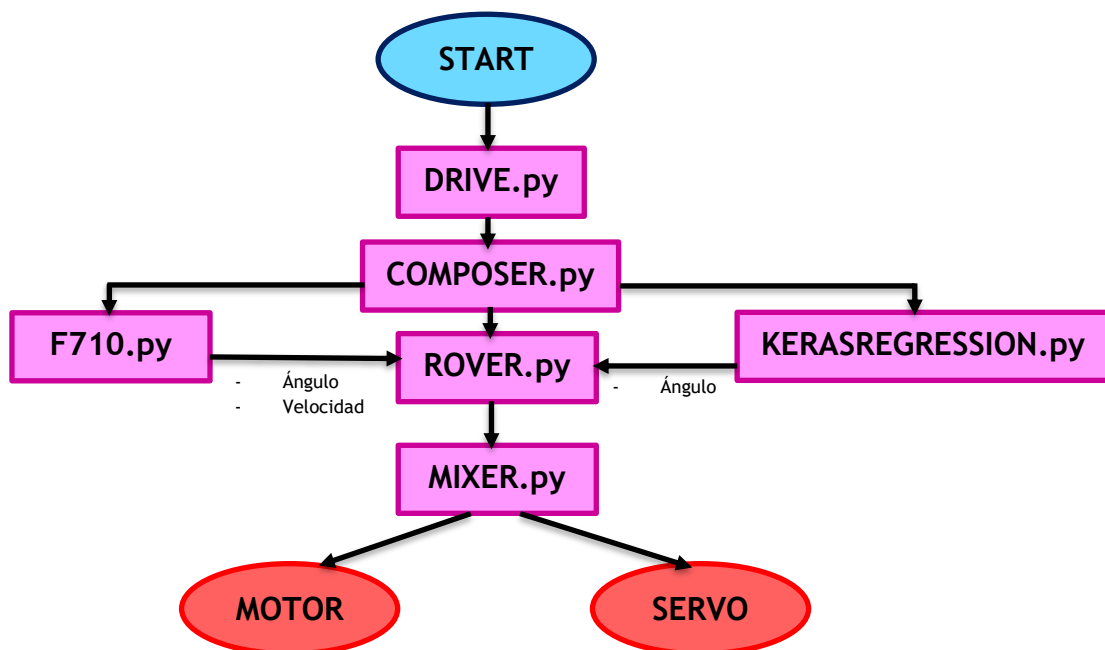


FIGURA 28. ESQUEMA RESUMEN DEL FIRMWARE “BURRO”.

El Firmware “BURRO” se inicia mediante un script (**start.sh**) de terminal encargado de inicializar en modo grabación el fichero **Drive.py**. En este fichero crea un sistema de logs utilizado durante todo el arranque indicando información relevante a este proceso, además de ejecutar el fichero **Composer.py** encargado de detectar los diferentes componentes hardware (el código está preparado para 3 placas de control diferentes: Navio, Navio 2 y Adafruit motor, y dos tipos de control de la dirección: Ackermann y Diferencial) y cargar los drivers correspondientes a estos. También intenta encontrar los diferentes tipos de pilotos manuales (**F710.py**) o autónomo (**cnn.py/KerasRegression**).

Para el primero busca un código específico para este modelo de mando en el bus I2C, para el otro, busca en la ruta específica para este fin, los modelos de redes neuronales (***.h5**) que pudiera haber. Además, inicia tanto la cámara como la Web remota que permite ver en tiempo real la cámara del vehículo siempre que se encuentren dentro de la misma red Wifi.

Con todos los datos anteriores, crea un objeto **Rover.py**. Este objeto es el encargado de detectar en qué modo de conducción nos encontramos (parámetro seleccionable desde el control L1/R1 del mando) y dependiendo de esto obtener el valor del ángulo de giro y la velocidad (ya sea del mando o de la red neuronal) y pasársela al **Mixer.py**. Por último, este pasa dicha información a los correspondientes canales PWM de la Navio 2 para activar los motores. Este procedimiento se realiza de forma cíclica pudiendo alternar entre modos de conducción en tiempo de ejecución.

Al proyecto “BURRO” se le realizaron una serie de modificaciones para ajustarlo a nuestro objetivo y a su vez solucionar algunos problemas de compatibilidad. Estas modificaciones son:

- Añadimos una funcionalidad para poder cambiar entre los modos de conducción (manual o autónomo) en tiempo de ejecución. Inicialmente se cargaban todos los pilotos detectados durante el arranque, pero solo se activaba uno de ellos, si se detectaba el mando USB se iniciaba en manual o, de lo contrario, en autónomo.
- Eliminamos la carga de los pilotos denominados como RC en el Firmware “BURRO”, dado que no disponemos de este tipo de mando (mando especializados para radio control) y creaba conflictos con el F710 (nuestro mando USB).
- Añadimos la opción de poder controlar la velocidad del vehículo no solo desde la cruceta de dirección (velocidad que hemos limitado en este control), sino que también, desde el joystick contrario a la dirección (sin límite).

- Cambiamos uno de los valores de comparación de la lectura de la cruceta de dirección. Esta devuelve valores 1 o -1 dependiendo del sentido, aunque el código esperaba valores de 1 o 2.
- Se eliminó la conversión de **ángulo** (valor del ángulo de giro en rango [-30, 30], yaw por el límite) a **yaw** (mismo valor, pero en rango [-1, 1]). Esta conversión no es necesaria, ya que nuestra red neuronal esta entrenada con el yaw.

A continuación, detallaremos la configuración predefinida en el código y las modificaciones necesarias.

3.8.1. Fichero de configuración.

El fichero de configuración está destinado para aquellas configuraciones que determinan el funcionamiento del software como son:

- Pines de conexión para los motores en la Navio 2.
- Parámetros de la cámara.
- Tipo de dirección del RC (Ackermann o diferencial).
- Configuraciones para la red neuronal.

Algunas de las configuraciones anteriores:

```
[camera]
resolution = (160, 120)
framerate = 30
horizontal_fov = 120.
output_range = (0, 255)
rotation = 0
crop_top = 0
crop_bottom = 0
```

CÓDIGO 1. CONFIGURACIÓN DE PARÁMETROS DE LA CÁMARA

```
[car]
type="ackermann"
reverse_steering = False
max_steering_angle = 30.
L = 0.10
W = 0.06
W_offset = 0
[ackermann_car]
throttle_channel = 2
steering_channel = 0
```

CÓDIGO 2. CONFIGURACIÓN DE PARÁMETROS DEL COCHE ACKERMAN

3.8.2. Cambiar entre modos de conducción.

Para esto nos hemos valido del hilo en segundo plano que se ejecuta para la lectura de los controles manuales del mando. Le añadimos la capacidad de al presionar el control L1/R1 cambiar al modo contrario en el que se encuentre quedando el código de la siguiente forma:

```
def decide(self, img_arr):
    st = self.gamepad._state
    direction = int(st[2])
    if direction == 1: # forward
        self.throttle = -0.095 - st[4] / 255.
    elif direction == -1: # reverse
        self.throttle = 0.095 + st[4] / 255.
    else:
        self.throttle = 0
    self.throttle -= (float(st[8]) - 128.0) / 128.0
    self.yaw = (float(st[10]) - 128.0) / 128.0

    if st[3] == 1
        self.autopilot_index = -1
    elif st[3] == 2:
        self.autopilot_index = 0

    return methods.yaw_to_angle(self.yaw), self.throttle, self.autopilot_index
```

Funcionalidad añadida

CÓDIGO 3. FUNCIONALIDAD CAMBIAR MODO CONDUCCIÓN.

Este fragmento de código fue extraído de la función “*decide*” del fichero **F710.py**. El `st[3]` es el canal del mando correspondiente al control L1. “*autopilot_index*” es la posición de un array y por tanto si su valor es -1, se encontrará fuera del tamaño del mismo y, por tanto, desactivará el modo autónomo. Al contrario, si vale 0, seleccionará el primero de los modelos neuronales que encuentre en el mismo y por tanto activará el modo autónomo.

3.8.3. Eliminar pilotos RC.

Para este fin, simplemente hemos comentado la línea correspondiente a esta funcionalidad que se encuentra junto a la carga de los otros dos tipos de pilotos de conducción.

```
def setup_pilots(self, rover):
    manual_pilots = []
    try:
        f710 = F710()
        manual_pilots.append(f710)
        logging.info("Loaded F710 Gamepad module")
    except Exception as e:
        f710 = None
    if self.board_type is 'navio':
        #Cant get RC for Navio to work yet
        pass
    elif self.board_type is 'navio2':
        #manual_pilots.append(RC())
        logging.info("NOT Loaded RC module -- commented")
    rover.manual_pilots = manual_pilots
```

CÓDIGO 4. ELIMINAR PILOTOS RC.

Este fragmento fue tomado de la función “*setup_pilots*” del fichero *Composer.py*. Se añadió un log para indicar que no se carga el módulo RC.

3.8.4. Cambio controles de velocidad.

Se cambio la función condicional de la dirección de marcha atrás (en este caso es -1 corregida, anteriormente esperaba un valor de 2).

```
def decide(self, img_arr):
    st = self.gamepad._state
    direction = int(st[2])
    if direction == 1: # forward
        self.throttle = -0.095 - st[4] / 255.
    elif direction == -1: # reverse
        self.throttle = 0.095 + st[4] / 255.
    else:
        self.throttle = 0
    self.throttle -= (float(st[8]) - 128.0) / 128.0
    self.yaw = (float(st[10]) - 128.0) / 128.0
    if st[3] == 1:
        self.autopilot_index = -1
    elif st[3] == 2:
        self.autopilot_index = 0
    return methods.yaw_to_angle(self.yaw), self.throttle, self.autopilot_index
```

CÓDIGO 5. FUNCIÓN PARA VALORES DE ÁNGULO Y VELOCIDAD CON MANDO

La variable *st[8]* y *st[10]* representan los canales del mando de los joystick. En el caso de la señal analógica del joystick se tiene en cuenta que el punto central del mismo representa el valor 128, por lo que se le resta y

divide por este número al valor leído de estos canales para que, en el caso de encontrarse en el punto central, el valor enviado sea 0.

3.8.5. Eliminar conversión a la salida de la red neuronal.

Esta conversión se elimina dado que, en este proyecto, la red neuronal ha sido entrenada mediante datos normalizados (con valores entre -1 y 1) que corresponde con el rango de valores de la variable **yaw**. Estos son el rango de valores empleado en el procesamiento del código, pero para el servo motor, debemos enviar este dato en valores de grados que corresponde con los valores del rango anterior con la variable **angle** con valores desde [-30, 30].

```
class KerasRegression(BasePilot):
    """
    A pilot based on a CNN with scalar output
    """
    def __init__(self, model_path, **kwargs):
        import keras

        self.yaw = 0
        self.model = keras.models.load_model(model_path)
        super(KerasRegression, self).__init__(**kwargs)
    def decide(self, img_arr):
        if config.camera.crop_top or config.camera.crop_bottom:
            h, w, _ = img_arr.shape
            t = config.camera.crop_top
            l = h - config.camera.crop_bottom
            img_arr = img_arr[t:l, :]
            img_arr = np.interp(img_arr, config.camera.output_range,
                               config.model.input_range)
            img_arr = np.expand_dims(img_arr, axis=0)
            prediction = self.model.predict(img_arr, 40, 0)
            if len(prediction) == 2:
                yaw = methods.angle_to_yaw(prediction[0][0])
                throttle = prediction[1][0]
            else:
                yaw = prediction[0][0] #methods.angle_to_yaw(prediction[0][0])
                print(yaw)
                throttle = 0.5
            self.yaw = yaw * 1.2
            return methods.yaw_to_angle(yaw), throttle
```

Conversión eliminada

CÓDIGO 6. CLASE PARA DEFINIR UN MODELO DE NEURONA BASADO EN REGRESIÓN

Este método se emplea en caso de que la red neuronal se entrene con valores de **angle**, pero al entrenarla con los valores normalizado no tenemos que realizar una doble conversión, ya que la función anterior devuelve el valor en grados para el servo haciendo la conversión contraria.

4. DISEÑO E IMPLEMENTACIÓN DE LA NEURONA DEL RC CAR

Está quizás es una parte específica de la memoria, puesto que se extenderá en conceptos teóricos/prácticos sobre redes neuronales, tanto las primeras que se desarrollaron, redes neuronales clásicas, basadas en los perceptrones multicapas que conforman la base de todos los principios hasta llegar con las que vamos a profundizar: las redes neuronales convolucionales, CNN (del inglés: “Convolutional Neural Network”).

En nuestro caso, partiremos de un modelo de referencia conocido en la literatura como “*End-to-end Deep learning for Self-Driving Cars NVIDIA model*”. Este es un modelo basado en redes convolucionales, y con el uso de las librerías TensorFlow/Keras nos permite fácilmente definir el modelo, entrenarlo y generarlo para poner en ejecución con el Firmware “BURRO” que controla el vehículo.

Toda nuestra fase de desarrollo la complementaremos con el uso de simulador para conducción autónoma “*Udacity’s Self-Driving Car Simulator*”. Éste será nuestro banco de pruebas para estudiar el comportamiento de la red.

4.1. REDES CLÁSICAS: PERCEPTRONES MULTICAPAS.

Un perceptrón simple o neurona establece relaciones entre patrones comunes para una serie de entrada de datos, lo que permite resolver problemas que tengan una dependencia lineal. Sin embargo, para problemas que se salgan de dicha linealidad, éste se queda muy limitado. Por ese motivo, surgen los llamados perceptrones multicapa, que constan ya no de una única neurona, sino que se conforman con cientos e incluso miles. Esto ofrece un potencial resolutivo mayor: los problemas a resolver pueden manejar relaciones no lineales.

Gracias al trabajo conjunto de las neuronas, los perceptrones multicapa se les conoce como redes neuronales clásicas, que definen un modelo computacional que simula la actuación de las neuronas biológicas del ser humano.

→ **¿Qué tipo de problemas son útiles para resolver con este tipo de modelos?**

Son útiles para resolver problemas de clasificación, ordenación o predicción. Lo bueno de los perceptrones es que son de fácil adaptación al problema a resolver. Para ello, se necesita entrenar la red. Entrenar la red consiste en definir y desarrollar una estructura que sirva y se adecúe a los

objetivos que estamos buscando. Más adelante entraremos en profundidad a cerca de esto.

4.1.1. Neuronas.

Hemos hablado de lo que son los perceptrones multicapa, o en definitiva, las redes neuronales clásicas. Sin embargo, no hemos explicado cómo trabaja la unidad más simple de la que se componen: las neuronas.

Las neuronas son la unidad más simple de cálculo computacional que trabajan a partir de una serie de entradas, multiplicadas por unos pesos. Éstas se hacen pasar por una función de activación, pudiendo ser lineal o no lineal, siendo la segunda prioridad para desarrollarnos en casos reales.

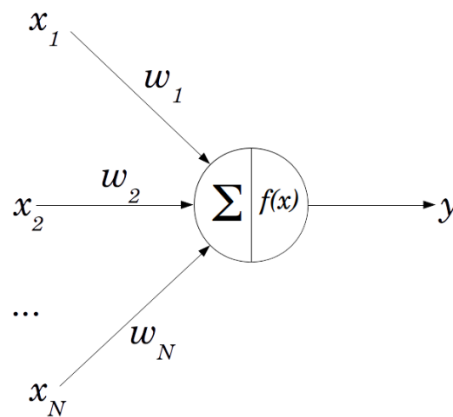


FIGURA 29. PERCEPTRÓN SIMPLE O NEURONA

Algebraicamente hablando, la acción de una neurona consiste en la definición de la siguiente ecuación, introducida por la figura 29. Suponemos que para una serie de N entradas la neurona aporta N pesos diferentes, que son afectados por una función de activación para dar como resultado una salida y:

$$SALIDA = F. ACTIVACIÓN * \left(\sum ENTRADAS * PESOS \right)$$

$$y = f(x) * \left(\sum_{i=1}^N x_i * w_i \right)$$

FIGURA 30. ECUACIÓN ACTUACIÓN NEURONAL

IMPORTANTE: los pesos son coeficientes entrenables, es decir, van a tomar valores en función del entrenamiento al que someteremos la neurona. Establecer valores altos para los pesos de primera instancia en lugar de dejar que la propia neurona los varíe con el entrenamiento puede suponer inestabilidad en el sistema, lo que supone mayor porcentaje de fallos.

4.1.2. Funciones de activación.

La función de activación es una de las claves para el funcionamiento de la red neuronal. Dentro del perceptrón, la función se encarga de aplicar una no linealidad en el sistema y establecer un mapeado de las entradas en las salidas. Esto quiere decir que para trabajos de clasificación, se esperan mejores resultados aplicando no linealidad en ordenaciones que requieran más abstracción de datos (los datos pueden ser de tipo imágenes, números, texto, etc.)

Algunas de las funciones más aplicadas en el campo son la sigmoide (variaciones entre el cero y el uno), la tangente hiperbólica o la RELU (Unidad Rectificada Lineal).

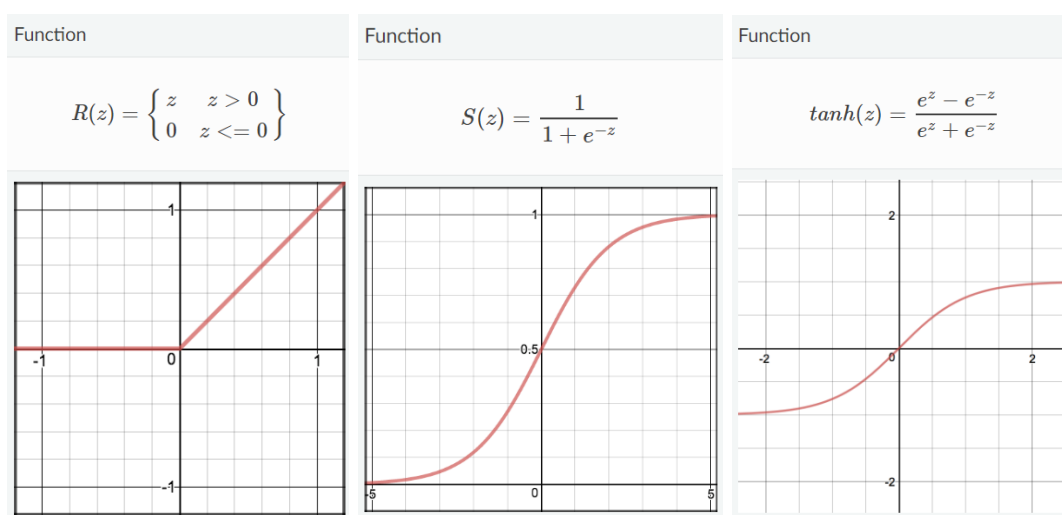


FIGURA 31. REPRESENTACIONES RELU, SIGMOIDE Y TANH RESPECTIVAMENTE

Cada tipo de función tiene sus pros y sus contras. La parte clave de un buen entrenamiento de la red neuronal es saber seleccionar qué tipo de función va a suponer un mayor porcentaje de aciertos en nuestras salidas. A saber, en entrenamiento de redes convolucionales (las que nosotros usaremos), la RELU es de las funciones que mejor respuesta ofrece.

4.1.3. Estructura de la red.

Cuando hablamos de las redes neuronales, nos movemos por una serie de capas interconectadas entre sí que trabajan para ofrecer una salida solución que resuelva el problema en cuestión. Si bien ya hemos visto como son las neuronas simples por dentro, ahora estudiaremos como se conectan entre ellas, en definitiva, la estructura básica de la red neuronal.

A saber, una red neuronal se compone por diferentes capas:

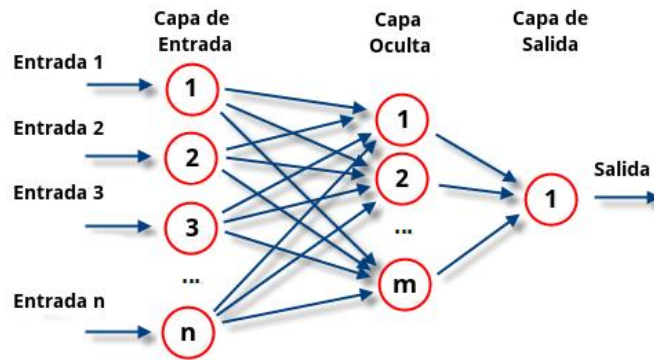


FIGURA 32. ESTRUCTURA DE UNA RED NEURONAL

A) CAPA DE ENTRADA:

Se establece la entrada de datos, aunque no se lleva a cabo ningún procesamiento neuronal. Simplemente, en esta capa se asignan los datos de entrada y su volumen, distribuyéndolas entre la primera capa siguiente que se encargará de procesarlo.

B) CAPA OCULTA:

Aquí es donde se encuentra la clave del funcionamiento de las redes más complejas. En la capa oculta, se llevan a cabo los procesamientos neuronales, es decir, todo el proceso de relación neuronal con funciones de activación y multiplicación por pesos. Se establecen las características del modelo solicitado y se genera una salida correspondiente a ellas.

IMPORTANTE: la capa oculta no se compone con una sola capa de neuronas. Ésta puede comprender muchísimas capas neuronales. A mayor cantidad de capas ocultas, se produce una mayor abstracción de las características a obtener de los datos de entrada. Es aquí donde surge el término clave de nuestro modelo de entrenamiento neuronal: el **Deep Learning**, donde *Deep* hace referencia a la profundidad de dicha capa oculta.

Para tener en cuenta, las primeras capas de neuronas van a captar elementos más simples. En el caso de imágenes, como las de nuestro proyecto, primero se buscarán patrones de formas simples: puntos, manchas, etc. En el resto de capas se produce la abstracción, y se buscarán detalles más precisos: esquinas, colores, etc.

→ **PROBLEMA:** a medida que se añaden capas dentro de la capa oculta, nos permite obtener mejores resultados en nuestros modelos neuronales, a cambio de tiempos de entrenamientos mucho más largos. Además, hay un número límite de capas que se pueden añadir, a partir del cual la neurona deja de entrenar y aprender. Este proceso se conoce como **Desvanecimiento de Gradiente**.

Por todos estos motivos, es importante estudiar cada problema para ver cuántas capas internas realmente necesitamos y cuál es el mejor modo de activación para conseguir el mayor potencial de nuestra red.

C) CAPA DE SALIDA:

Por último, nos encontramos con la capa de salida, encargada de regresar o clasificar los valores respecto a los entrenados. Aquí es donde más influye el tipo de activación que podemos usar, ya que en la capa de salida también se procesan los datos como una capa neuronal más. En la salida, el número de neuronas debe coincidir con la cantidad de valores o datos a esperar. Por ejemplo, en el caso de que queramos que se devuelva una clasificación (gato, perro, pato, etc.) sólo habrá 1 salida.

Por lo tanto, el tipo de activación a la salida va a definir los grados de libertad de ésta. A saber, las más comunes son activación por categorías (SOFTMAX), binaria (Sigmoidal) y por regresiones (Lineales), siendo esta última la que ofrece una plena libertad de decisiones.

4.1.4. Preprocesado de los datos.

Pese a que este paso parezca obvio, todos los modelos neuronales requieren de una normalización y escalado de los datos de entrada. De esta manera, vamos a querer que nuestros datos se encuentren entre cero y uno (típico valor de normalización). Si estos datos no son preprocesados correctamente, las activaciones se pueden volver inestables y generar salidas y comportamientos no deseados. De esta forma, la neurona le va a resultar más fácil aprender evadiendo comportamientos negativos e incluso conseguir aumentar la velocidad de entrenamiento (mayor velocidad para procesar la información).

4.1.5. Algoritmo de entrenamiento.

A) DESCENSO POR GRADIENTE ESTOCÁSTICO (SGD):

Este algoritmo compara las salidas de nuestra base de datos con la salida generada por nuestra red. Con esto, compara los resultados obtenidos y calcula el error que se ha producido respecto a la salida que se esperaba. Una vez realizado este tipo de cálculo, vamos a propagar este error hacia atrás, es decir, a las capas ocultas. Este método se conoce como **Backpropagation**.

Al usar este método, conseguimos que los pesos de la neurona cambien, de forma que en las capas más próximas a la salida los pesos variarán de forma más notable. Gracias a los nuevos pesos, siguiendo el cálculo del error se verá si ha disminuido o a aumentado (buscamos siempre ir a la zona con menos

error). Esto se traduce en que las capas más influyentes a la hora de encaminar tomar una decisión u otra son las finales dentro de las ocultas.

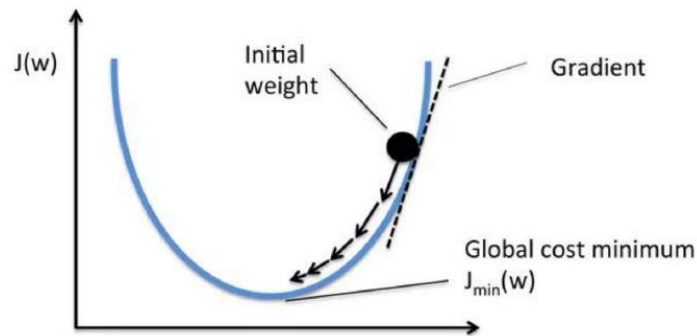


FIGURA 33. REPRESENTACIÓN DE LA EVOLUCIÓN DEL ERROR (COSTE) POR SGD

En la figura 33, vemos que se estudia el error o coste (J) a medida que se comparan las salidas obtenidas por las entradas. Al principio, el gradiente es mayor, pero a medida que nos vamos acercando al coste mínimo (que es el que buscamos con la validación de nuestros datos de entrenamiento), la variación de este gradiente es menor. Al ratio de cambio del gradiente en el momento del aprendizaje o entrenamiento lo conocemos como **Learning Rate**, y puede afectar significativamente a nuestro resultado final del modelo si no se trabaja con el correctamente. En el ejemplo de la figura, si suponemos que el ratio de aprendizaje es constante todo el rato, el coste puede que nunca llegue al mínimo e incluso que sea mucho más dispar.

Este proceso se lleva a cabo con todo el bloque de datos hasta que se terminan. A nuestro bloque de datos que le suministramos a la neurona para entrenar lo vamos a denominar **batch**. En él, se encuentran un determinado número de muestras que el usuario habrá definido. En el caso de la conducción, nuestro **batch** vendrá conformado por las imágenes de muestra y los ángulos de salida. Más adelante explicaremos el enfoque de nuestro proyecto con más detalle.

B) OPTIMIZADOR: ADAM

ADAM es un optimizador de entrenamiento, un algoritmo basado en la optimización de primer orden de los gradientes de las funciones de activación de nuestro modelo. Además, ADAM devuelve unos parámetros alternativos para modificar él solo el ratio de entrenamiento a lo largo de las iteraciones de entrenamiento. Esto permite que de forma dinámica se cambia el **Learning Rate** para que se adapte de forma correcta. Hay que tener en cuenta que ADAM sólo va a funcionar de manera óptima con una buena cantidad de datos (**batch**), ya que hay que ofrecerle tiempo para que el **learning rate** se adecue correctamente.

Esto convierte al optimizador ADAM el más potente y popular dentro del campo del *Deep Learning* ¿A qué se debe este éxito? A diferencia de otros optimizador más complejos de segundo orden, a medida que vamos añadiendo datos, estos se van procesando correctamente y se ajustan a lo largo del tiempo sin incorporar muchas demoras en el entrenamiento. En cambio, uno de segundo orden añadiría demasiado peso computacional como para poder manejar la cantidad de datos.

4.2. REDES NEURONALES CONVOLUCIONALES (CNN).

Hasta ahora, nos hemos centrado en explicar cómo funcionaban las redes clásicas, así como definir sus partes y entrar en detalle de algún que otro algoritmo de entrenamiento. Todo esto no es en vano, ya que muchas de las herramientas que se utilizan en las redes convolucionales también se usaban en las redes clásicas. A continuación, vamos a entrar en profundidad sobre qué son las CNN.

Las CNN son un tipo de redes neuronales que intentan imitar el funcionamiento de la corteza visual del cerebro humano. Esto se debe a que nosotros como personas somos unos eficientes clasificadores, y la inteligencia artificial buscará imitar lo más posible el funcionamiento de nuestro mecanismo.

Como en las redes clásicas, las redes convolucionales están compuestas de múltiples capas de filtros convolucionales de una o más dimensiones que proporcionan esa relación no lineal que aportaban las funciones de activación. Dentro del comportamiento de las redes, podemos delimitar dos:

- A) **EXTRACCIÓN DE CARACTERÍSTICAS:** se define una fase inicial compuesta por neuronas convolucionales, que simulan la visión humana. En esta fase, a medida que se avanza, menos se reacciona a los valores de entrada. De este modo, al principio se buscan bordes o características simples y, al final, ya se identifica el objeto en sí que queremos buscar (estamos suponiendo caso de imágenes 2D).
- B) **CLASIFICACIÓN:** se utilizan las capas densas, formadas por las neuronas convencionales. Esto va a permitir que se generen las regresiones que estamos buscando a la salida (una respuesta).

4.2.1. Ventaja frente a las redes clásicas.

Echando la vista hacia atrás, las neuronas clásicas no establecen ningún tipo de relación entre los datos de entrada (suponemos que si hablamos de imágenes, nos referimos al contenido que hay en ellas). Ahora lo que nosotros

manejamos es la característica en sí del objeto y no un pixel del objeto. Esto influye a una mejora de los parámetros y la velocidad de procesamiento.

4.2.2. Arquitectura de una CNN.

A) CAPAS CONVOLUCIONALES

La principal característica de las redes convolucionales son las capas convolucionales, que trabajan sobre los datos de entrada aplicando convoluciones discretas a partir de diferentes filtros finitos, como podemos observar en la siguiente figura.

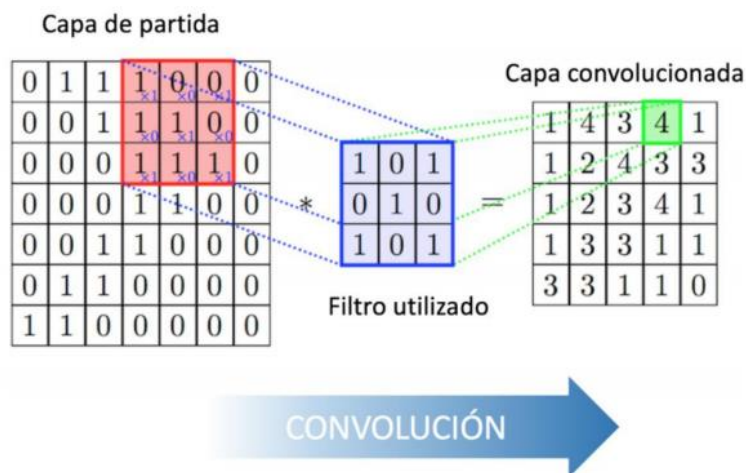


FIGURA 34. APLICACIÓN DE UNA CAPA CONVOLUCIONAL

Para entender este proceso mejor, vemos en la figura 34 un ejemplo de cómo actúa el cálculo y establece así una relación entre el valor de los datos colindantes. Los valores del filtro utilizado van a tener que ir cambiando en función de los resultados obtenidos. Este proceso se realiza durante el entrenamiento y de forma automática, hasta alcanzar la salida óptima.

Como las neuronas normales, cada capa que vamos añadiendo va a adquirir mayor abstracción de características. Al principio se identificarán cambios de contraste, formas simples y a medida que avanzamos se buscarán formas más complejas como esquinas, bordes incluso hasta llegar a identificar círculos, formas geométricas, etc.

B) CAPAS DENSAS (Fully Connected Layer):

Estas capas se definen del mismo modo del que trabajan las neuronas clásicas (perceptrones). Componen una de las capas más importantes, puesto que se establecen al final de nuestro modelo neuronal con el objetivo de relacionar todas las características y datos procesados anteriormente y generar una regresión (salida). Normalmente vienen agrupadas en forma de capas densas, que vienen agrupadas para realizar dicha tarea.

C) CAPA “MAX POOLING”.

Otra de las capas más importantes es la de **Max Pooling**, encargada de obtener una reducción del flujo de datos tras el efecto de las capas convolucionales, pero no de cualquier manera. Su acción va a reducir la información a partir de un “pool” o tamaño de datos de forma que los datos que resulten ofrecerán la información más importante o de mayor valor que realice la máxima activación dentro de ese “pool”.

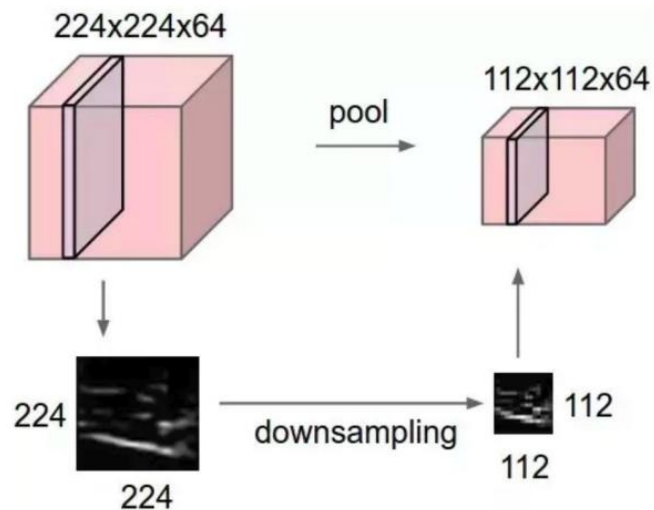


FIGURA 35. APLICACIÓN DE UNA CAPA MAX POOLING

IMPORTANTE: al utilizar esta técnica reducimos notablemente el flujo de datos a procesar, ya que las zonas de máxima activación se representan completamente, pero también puede suponer una pérdida de datos sustancial para nuestra red. Hay que tener cuidado con que no se abuse de su uso.

D) CAPA “DROPOUT”.

Cuando un modelo es entrenado recursivamente, puede darse la ocasión de que las neuronas establezcan relaciones entre sí y en lugar de buscar los parámetros a elegir en función de las activaciones, se aprenda el banco de datos de memoria y sólo trabaje bien con esos datos. En definitiva, el “dropout” se encarga de eliminar la relación de ciertas neuronas, suspendiendo la actividad de cierto número de ellas elegidas de forma aleatoria, con el fin de evitar este efecto, conocido como **Overfitting** o **Sobreentrenamiento** de las CNN.

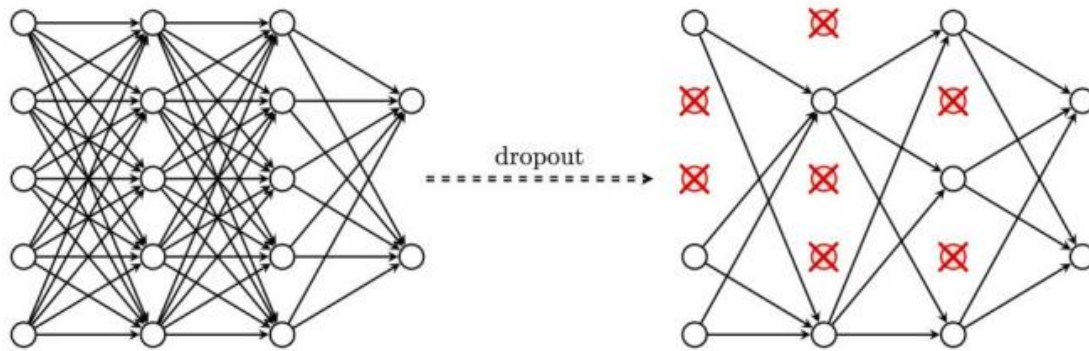


FIGURA 36. APLICACIÓN DE UNA CAPA DROPOUT

4.3. ELIGIENDO UN MODELO PARA NUESTRA NEURONA: NVIDIA MODEL.

En este apartado, nos vamos a centrar en explicar el modelo que hemos implementado para nuestro vehículo, de tal modo que lo entenderemos y adaptaremos a las capacidades de nuestro sistema RC car. Hemos elegido el modelo secuencial “*End-to-end Deep learning for Self-Driving Cars NVIDIA model*”, ya que nos ofrece un alto rendimiento en cuanto a capacidad de procesamiento de los datos que le podamos proporcionar y, además, el código del modelo está pensado sobre todo para conducción autónoma.

A) DESCRIPCIÓN DEL MODELO DE NVIDIA.

Este modelo está construido a partir de KERAS y TensorFlow, librerías que como hemos explicado en anteriores epígrafes son clave para el desarrollo de los modelos neuronales para inteligencias artificiales.

Este modelo ha sido desarrollado exclusivamente para trabajar con sistemas de vehículos autónomos. El propósito de NVIDIA es hacer desaparecer la necesidad de implementar características diseñadas para humanos y que el coche conduzca perfectamente solo, evitando obstáculos, conviviendo con otros vehículos en carretera, etc. Para ello, no sólo se trabaja con el tipo de datos recolectados, sino con una red neuronal convolucional diseñada para trabajar a partir del aprendizaje por propagación del error (“Back propagation”).

El número de capas convolucionales ha sido elegido empíricamente por NVIDIA a partir de estudios realizados con diferentes cantidades de éstas y comparando los resultados obtenidos. En la figura 37, podemos apreciar la estructura base de este modelo neuronal.

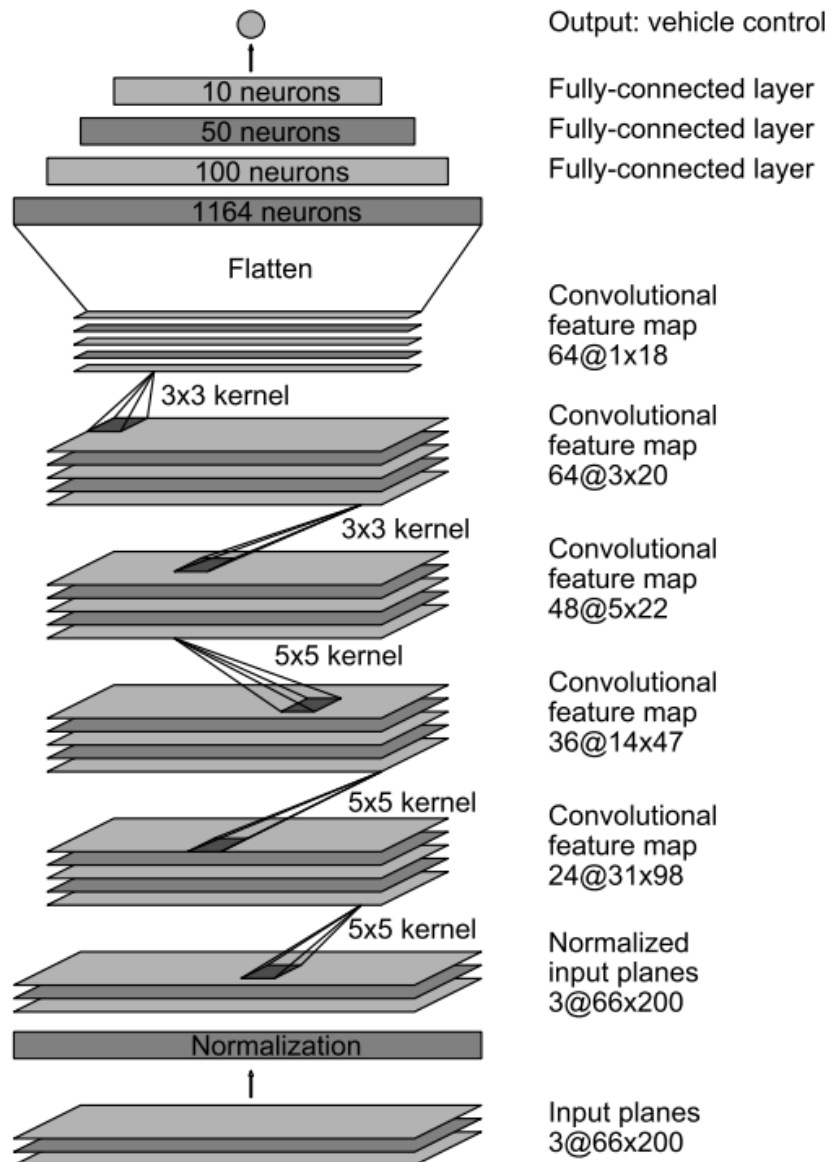


FIGURA 37. ARQUITECTURA DE LA RED DEL MODELO DESARROLLADO POR NVIDIA

Este modelo, consta de las siguientes características:

- 3 entradas en formato imagen de 66x200 tamaño píxel.
- 1 capa de normalización de la capa de entrada.
- 5 capas convolucionales de diferentes tamaños de kernel y diferentes filtros aplicables.
- 1 capa Flatten para aplanar las entradas en un solo array de datos.
- 3 capas densas para estructurar una salida neuronal.
- 1 capa densa final de salida para establecer la regresión.

El entrenamiento del modelo (del orden de 800.000 parámetros entrenables) va a requerir la información de las entradas (3 cámaras), un tratamiento de datos para mejorar la cantidad y calidad de las muestras y el cálculo de la propia red. La salida se comparará con los resultados que se

esperan y se aplicará un ajuste del error de forma automática a modo de realimentación (como vimos en el “Backpropagation”). En la figura 38, podemos ver un diagrama de bloques que simplifica este proceso:

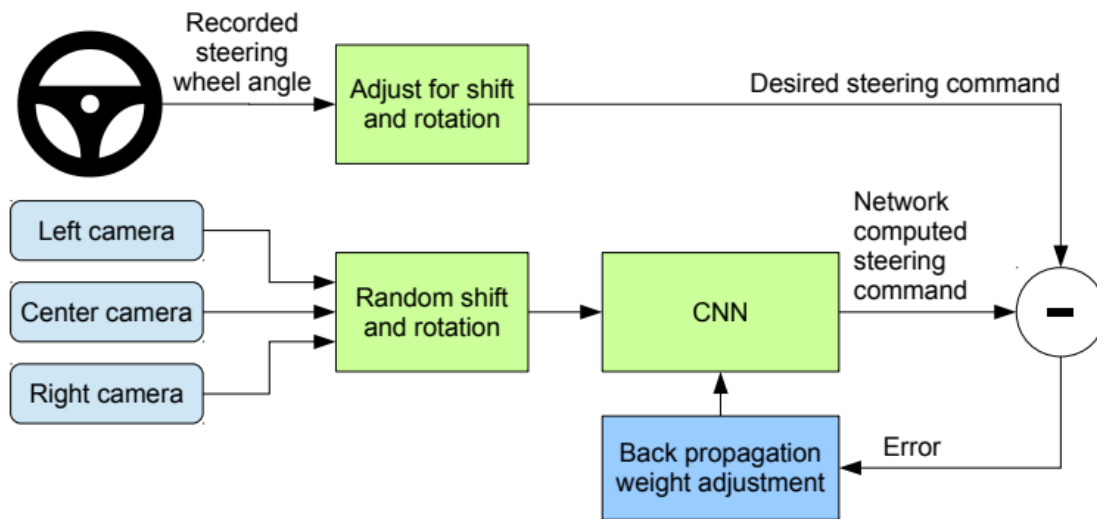


FIGURA 38. ENTRENAMIENTO DE LA RED NEURONAL

4.4. ADAPTACIÓN DEL MODELO A NUESTRO PROYECTO.

A continuación, vamos a describir los cambios que hemos tenido que cambiar en el código para adaptarlo a las características de nuestro problema.

Para empezar, el modelo de NVIDIA está diseñado para un sistema de estereovisión más una cámara central de base, es decir, 3 fuentes de imágenes de datos de referencia para tomar decisiones. En nuestro caso, simplemente somos capaces de obtener **una fuente de información de nuestra cámara delantera**. En el código, hemos de modificar los parámetros al cargar los datos de trabajo y luego cambiar las dimensiones del “INPUT_SHAPE” para adaptarlas a nuestras imágenes. En resumen, con los datos debemos:

- Cambiar el número de fuentes de datos (de 3 a 1)
- Cambiar las columnas de importación del archivo CSV a sólo “center”
- Redimensionar la variable de entrada “x” como un array de 1 dimensión.
- Cambiar el valor de “INPUT_SHAPE” en el archivo de origen “utils.py”
- Añadir una capa de “Droupout” al modelo base, para evitar overfitting.

```
data_df = pd.read_csv(os.path.join(os.getcwd(), args.data_dir,
                                  'driving_log.csv'), names=['center', 'steering'])

#yay dataframes, we can select rows and columns by their names
#we'll store the camera images as our input data
x = data_df['center'].values
#and our steering commands as our output data
y = data_df['steering'].values
```

CÓDIGO 7. INICIALIZAR LOS DATOS DE ENTRADA PARA NUESTRA CÁMARA

```
model = Sequential()
model.add(Lambda(lambda x: x/127.5-1.0, input_shape=INPUT_SHAPE))
model.add(Conv2D(24, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(36, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(48, 5, 5, activation='elu', subsample=(2, 2)))
model.add(Conv2D(64, 3, 3, activation='elu'))
model.add(Conv2D(64, 3, 3, activation='elu'))
model.add(Dropout(args.keep_prob))
model.add(Flatten())
model.add(Dense(100, activation='elu'))
model.add(Dense(50, activation='elu'))
model.add(Dense(10, activation='elu'))
model.add(Dense(1))
```

CÓDIGO 8. MODELO NEURONAL CON LOS CAMBIOS IMPLEMENTADOS

Básicamente las capas de las que se componen son las siguientes, como se puede ver en el bloque superior de código:

1) Capa de normalización Lambda.

Ésta es fundamental para normalizar el valor de los píxeles de las imágenes, evitando la saturación y haciendo que la técnica por gradientes funcione mejor. Se espera de X un valor entre $[0 - 255]$ por cada píxel, siendo reescalado entre -1 y 1 . El valor de entrada “*input_shape*” ha de ser el tamaño de píxeles de las imágenes. Este valor se define en el fichero “*utils.py*”.

2) Capas de Convoluciones 2D.

Se define un “*kernel_size*”, es decir, el tamaño de la ventana de actuación del filtro convolucional en cada capa. Además, se especifican los filtros que se van a usar y en algunas ocasiones se le puede pasar el parámetro “*subsample*” que nos va a establecer un “*stride*”, *en otras palabras, distancia a la cual se llevará a cabo otra convolución dentro de una misma muestra.*

- 3 capas de Conv2D con un *kernel_size* de 5x5, 24, 36 y 48 filtros con strides de 2x2.
- 2 capas de Conv2D con un *kernel_size* de 3x3, 64 filtros.

Vemos que en cualquier caso, la función de activación siempre es de tipo “**ELU**” - *Exponential Linear Unit*. Para identificación de líneas o espacios delimitantes este tipo de activación responde bastante bien, al igual que otras por el estilo como “RELU”, ya mencionadas en apartados anteriores.

3) Capa “Dropout”.

Realiza la función de prevención de “*overffiting*” como vimos en la explicación de dicha capa. En este caso tiene un ratio de efecto de un 0,5. Este valor está definido como un argumento de entrada: “*args.keep_prob*”.

4) Capa “Flatten”.

Esta capa se encarga de aplanar la entrada. Como vimos en la capa de normalización, las convoluciones tienen una entrada en **tensores**³ de 3 dimensiones. Lo que se encarga “flatten” es aplanar este tensor para convertirlo en un array de una sola dimensión: no afecta al contenido del tensor, reorganiza los resultados.

5) Capas Densas (“Dense”).

Se aplican 3 capas densas de 100, 50, 10 y finalmente 1 de salida, ofreciendo al final el mismo número de “outputs” esperadas por el modelo.

Por otro lado, el proyecto software para definir el modelo está formado por 2 archivos Python que se encargarán de trabajar exclusivamente con el modelo neuronal (CNN) y su entrenamiento:

- A) “**model.py**” - Aquí básicamente se define el modelo neuronal, la carga de datos para su entrenamiento, el entrenamiento en sí y la puesta en marcha de todo el proceso. El resultado será un archivo “***.h5**” en el que se contiene toda la información del modelo entrenado (información de los pesos, funciones de activación, etc.)
- B) “**utils.py**” - Más adelante veremos que el modelo tiene un propio generador de imágenes adicionales que se crean a partir de nuestro “*batch*” de datos jugando con el brillo, el contraste, recortes de las imágenes, sombras aleatorias, etc. Esto sirve para añadir un número mayor de muestras con el que la neurona pueda entrenar y evitar un posible “*Overfitting*”. Así pues en este archivo se almacenan todas esas funciones en Python que se encargan de llevar a cabo todo el proceso.

³ **Tensores:** los tensores son la unidad fundamental de trabajo para los módulos que se basan en TensorFlow, de ahí su nombre. Son contenedores de información que pueden tener múltiples dimensiones de trabajo.

4.4.1. Software de entrenamiento de la red.

Sin entrar en detalle en la parte del código, el entrenamiento necesita generar un data set de training y validación. El set de training es generado a partir de un conjunto de imágenes que viene a representar un 75% del mismo. El resto, 25%, es usado para validar el modelo. Para ello usaremos una función “fit_generator”, que veremos en el código 9.

```
checkpoint = ModelCheckpoint( 'model_burro_ULL.h5',
                             monitor='val_loss',
                             verbose=0,
                             save_best_only=args.save_best_only,
                             mode='auto')
```

CÓDIGO 9. CHECKPOINT DE ENTRENAMIENTO NEURONAL

En el código superior, se establece un “*checkpoint*” o punto de guardado. Esto va a guardar el modelo entrando por cada época que pase. Una época se define como el periodo de tiempo que tarda el modelo en entrenarse con todo un “batch” de datos. En este caso, no se van a guardar todos los modelos, sino que sólo se guardará el que menos valor de pérdidas tenga [“save_best_only - “val_loss”].

Como paso previo a entrenar el modelo, debemos compilar el modelo, diciéndole qué tipo de valor de error vamos a usar para el aprendizaje por descenso de gradiente y sobre todo, el tipo de optimizador, que será crucial para que el “*learning_rate*” se vaya ajustando en cada iteración. En nuestro caso trabajaremos con una referencia del error cuadrático medio y usaremos el optimizador ADAM explicado en epígrafes anteriores.

```
model.compile(loss='mean_squared_error',
              optimizer=Adam(lr=args.learning_rate))
```

CÓDIGO 10. FUNCIÓN PARA COMPILAR EL MODELO

```
model.fit_generator(batch_generator(args.data_dir, x_train, y_train,
                                   args.batch_size, True),
                   args.samples_per_epoch,
                   1, #args.nb_epoch
                   max_q_size=1,
                   validation_data=batch_generator(args.data_dir,
                                                    x_valid, y_valid, args.batch_size, False),
                   nb_val_samples=len(x_valid),
                   callbacks=[checkpoint],
                   verbose=1)
```

CÓDIGO 11. FUNCIÓN FIT_GENERATOR PARA EL ENTRENAMIENTO DEL MODELO

Como comentamos, la función “fit_generator” mediante la función “batch_generator”, se encarga de generar un data set lo suficientemente grande para poder entrenar el modelo, a partir de un set de imágenes iniciales adquiridas por la cámara. En otras palabras, si se desea tener un

tamaño mayor de “batch” [simples_per_epoch], se añadirán más fotos modificadas sobre las originales (brillo cambiado, recortes, sombras aleatorias) para favorecer aún más el entrenamiento neuronal. Finalmente se cambia el número de épocas y se deja en una, ya que poner más aumenta mucho el tiempo de entrenamiento, puesto que trabajamos en un entorno de Google Colab que no permite tiempos largos de entrenamiento.

4.4.2. Preprocesado de datos: generar los archivos CSV.

Como hemos mencionado anteriormente, uno de los parámetros a modificar son las columnas de importación del archivo CSV pero ¿qué es un archivo CSV?

- Un **archivo CSV** (Comma Separated Values) es un tipo de archivo de texto sin formato que utiliza una estructura tipo tabla para tabular y organizar los datos simplemente separándolos por comas. Como se trata de un archivo de texto, sólo puede contener datos de texto reales (caracteres ASCII o Unicode).

Los archivos CSV trabajan de forma similar a como lo hacen las tablas en formato Office Excel. Además, se pueden definir cabeceras para las columnas e incluso para las filas. En un archivo estilo modelo de NVIDIA, se almacenan tres columnas de datos de entrada y una para el resultado del ángulo de giro. Sabiendo esto, nosotros vamos a pasar de tener: [center, left, right] a sólo **[center]** en la columna de entradas. Para el valor de ángulo, se queda como está: **[steering]**

IMPORTANTE: estos scripts han sido diseñados desde cero, tomando las referencias correspondientes especificadas en cada script.

1. GENERAR ARCHIVO CSV (GOOGLE COLAB):

Estos pasos a seguir están específicamente diseñados para trabajar con Google Colab como herramienta de entrenamiento. Si se va a usar una herramienta de desarrollo local, pase al siguiente apartado. Aunque, de cualquier modo, es preferible generar el archivo CSV en un entorno local y luego usarlo en el Google Colab.

- 1) Importamos las librerías con las que vamos a trabajar.

```
import sys
import string
import csv
import glob
from os import scandir, getcwd, listdir
from os.path import isfile, join, abspath
```

CÓDIGO 12. LIBRERÍAS IMPORTADAS

- 2) Definimos una función que devuelva un array con el nombre de los elementos que se encuentran dentro de un determinado directorio.

```
def lista_archivos(ruta = getcwd()):  
    return [arch.name for arch in scandir(ruta) if arch.is_file()]
```

CÓDIGO 13. FUNCIÓN LISTA_ARCHIVOS

- 3) Definimos una función que devuelva un array con el nombre del directorio de Google Colab en el que trabajaremos, para ello hemos de ir mirando que es lo que nos devuelve nuestro script de entrenamiento para ir modificando a gusto el texto del directorio manual.

```
def lista_rutas(archivos = getcwd()):  
    nombre_final = []  
    for i in range(0, len(archivos) - 1):  
        nombre_final.append('Fotos_modelo/2019_08_02__05_07_07_PM/'  
                            + archivos[i])  
  
    return nombre_final
```

CÓDIGO 14. FUNCIÓN LISTA_RUTAS

- 4) Definimos una función que extraiga el valor del “steering” del nombre del archivo y lo almacene en un número punto flotante. Debemos tener en cuenta que los valores de “steering” hemos de normalizarlos antes de escribirlos en el CSV.

```
def colecting_steering(name_file):  
    position_agl = name_file.index('agl')  
    position_mil = name_file.index('mil')  
    crop = slice(position_agl+4, position_mil-1)  
  
    steering = float(name_file[crop])  
    steering = steering / 300.  
  
    return steering
```

CÓDIGO 15. FUNCIÓN COLECTING_STEERING

- 5) Establecemos lo que va a hacer el “*main*” en el siguiente orden:
- Generar la lista de archivos de un directorio.
 - Generar un array con el nombre de los directorios.
 - Abrir un nuevo archivo CSV, definiendo sus cabeceras como ['center', 'steering'].
 - Calcular el número de archivos a partir de la lista de archivos.
 - Escribir una fila por cada archivo, almacenando su directorio y su “steering”.

```
def main():
listado_archivos = lista_archivos(sys.argv[1])
listado_rutas = lista_rutas(listado_archivos)
with open('driving_log.csv', mode='w', newline='') as drive_log:
    cabeceras = ['center', 'steering']
    writer = csv.DictWriter(drive_log, cabeceras)

    archivos = len(listado_archivos)
    for i in range(0, archivos - 1):
        steering = colecting_steering(listado_archivos[i])
        writer.writerow({'center': listado_rutas[i], 'steering':
                        steering})
if __name__ == '__main__':
    main()
```

CÓDIGO 16. EJECUCIÓN PRINCIPAL, ESCRITURA DEL ARCHIVO CSV

NOTA: esto es el código base que se necesita. En los anexos puede que aparezca este mismo código con elementos añadidos para una mayor facilidad de comprensión con el usuario.

2. GENERAR ARCHIVO CSV (ENTRENAMIENTO LOCAL):

A continuación, explicaremos el script que se ha utilizado para pruebas en local. En cierta medida, se parecerá mucho al anterior, pero cambiando ciertos detalles importantes:

1) Importamos las librerías con las que vamos a trabajar.

```
import sys
import string
import csv
import glob
from os import scandir, getcwd, listdir
from os.path import isfile, join, abspath
```

CÓDIGO 17. LIBRERÍAS IMPORTADAS

2) Definimos una función que devuelva un array con el nombre de los elementos que se encuentran dentro de un determinado directorio.

```
def lista_archivos(ruta = getcwd()):
    return [arch.name for arch in scandir(ruta) if arch.is_file()]
```

CÓDIGO 18. FUNCIÓN LISTA_ARCHIVOS

3) Definimos una función que devuelva un array con el nombre del directorio completo de cada uno de los elementos de un determinado directorio

```
def lista_rutas(ruta = getcwd(), archivos = getcwd()):
    return [abspath(arch.path) for arch in scandir(ruta) if
            arch.is_file()]
```

CÓDIGO 19. FUNCIÓN LISTA_RUTAS (LOCAL)

- 4) Definimos una función que extraiga el valor del “steering” del nombre del archivo y lo almacene en un número punto flotante. Debemos tener en cuenta que los valores de “steering” hemos de normalizarlos antes de escribirlos en el CSV.

```
def colecting_steering(name_file):
    position_agl = name_file.index('agl')
    position_mil = name_file.index('mil')
    crop = slice(position_agl+4, position_mil-1)

    steering = float(name_file[crop])
    steering = steering / 300.

    return steering
```

CÓDIGO 20. FUNCIÓN DE RECOLECCIÓN DEL ÁNGULO

- 6) Establecemos lo que va a hacer el “*main*” en el siguiente orden:
- Generar la lista de archivos de un directorio.
 - Generar un array con el nombre de los directorios.
 - Abrir un nuevo archivo CSV, definiendo sus cabeceras como ['center', 'steering'].
 - Calcular el número de archivos a partir de la lista de archivos.
 - Escribir una fila por cada archivo, almacenando su directorio y su “steering”.

```
def main():
    listado_archivos = lista_archivos(sys.argv[1])
    listado_rutas = lista_rutas(sys.argv[1])
    with open('driving_log.csv', mode='w', newline='') as drive_log:
        cabeceras = ['center', 'steering']
        writer = csv.DictWriter(drive_log, cabeceras)

        archivos = len(listado_archivos)
        for i in range(0, archivos - 1):
            steering = colecting_steering(listado_archivos[i])
            writer.writerow({'center': listado_rutas[i], 'steering':
                            steering})
    if __name__ == '__main__':
        main()
```

CÓDIGO 21. FUNCIÓN PRINCIPAL, ESCRITURA DEL ARCHIVO CSV

3. **EXTRA:** Sustitución módulo “argparse” (Google Colab):

Otro de los problemas que nos podemos encontrar es que el módulo “argparse” no funcione correctamente dentro de la plataforma de Google Colab, dependiendo de la versión en la que nos encontremos. Para solucionar este problema, se ha decidido modificar este cacho de código mediante “*easydict*”, aportando el mismo contenido de una manera un poco más casera:

```
def main():  
  
    args = easydict.EasyDict(  
        {  
            "data_dir": '/content/drive/My Drive/TFG  
                        2019/cnn/cnn_propia/',  
            "test_size": 0.2,  
            "keep_prob": 0.5,  
            "nb_epoch": 1,  
            "samples_per_epoch": 15000,  
            "batch_size": 40,  
            "save_best_only": 'true',  
            "learning_rate": 1.0e-4  
        })
```

CÓDIGO 22. MÓDULO ARGS.PARSE SUSTITUIDO POR EASYDICT

4.5. SIMULACIÓN DEL RC CAR EN UNITY.

A modo de entender cómo funciona la red neuronal y prueba de la misma, hemos usado un banco de pruebas que está basado en un simulador bajo el nombre de “*Udacity’s Self-Driving Car Simulator*”.

El simulador y todo su contenido son de acceso libre y gratuito para todo aquel que quiera aventurarse en el mundo de las inteligencias artificiales. En el repositorio de GIT se encuentran los **modelos de NVIDIA preparados**, unos modelos “*.h5” ya entrenados de prueba y el **simulador en formato ejecutable de Windows (“.exe”)**



FIGURA 39. LOGO DE UNITY

El ejecutable es fruto del desarrollo de un entorno de programación en 3D especializado para videojuegos conocido como **Unity**. Este programa ofrece muchas ventajas a la hora de preparar entornos gráficos, ya que ofrece herramientas de física como aceleración de motores, fricción de las ruedas, tipos de terrenos, etc. Además, proporciona todos los modelos gráficos (coches, árboles) para definir diferentes tipos de terrenos.



The real-time revolution in auto

Una nueva era para la industria automotriz; únete a la revolución en tiempo real con Unity

Diseñar y crear un vehículo en realidad virtual, permitir a los clientes experimentar un automóvil antes de que este exista, capacitación sin las restricciones de los límites físicos, automóviles que se conducen a sí mismos y la experiencia de vehículo del mañana; cosas que una vez parecían ser un sueño futurista, actualmente se convierten en realidad con Unity, la mayor plataforma de desarrollo en tiempo real del mundo.

FIGURA 40. NOTICIA SOBRE EL SECTOR AUTOMOVILÍSTICO EN SU PÁGINA WEB

4.5.1. Entorno del simulador.

Al abrir el simulador, nos aparece la siguiente interfaz de usuario. Nosotros nos hemos descargado la versión más simple de este simulador (Term 1 Beta) en razón a nuestro propósito meramente lúdico. Como vemos, podemos seleccionar dos tipos de modos: entrenamiento (“Training”) y autónomo (“Autonomous”).

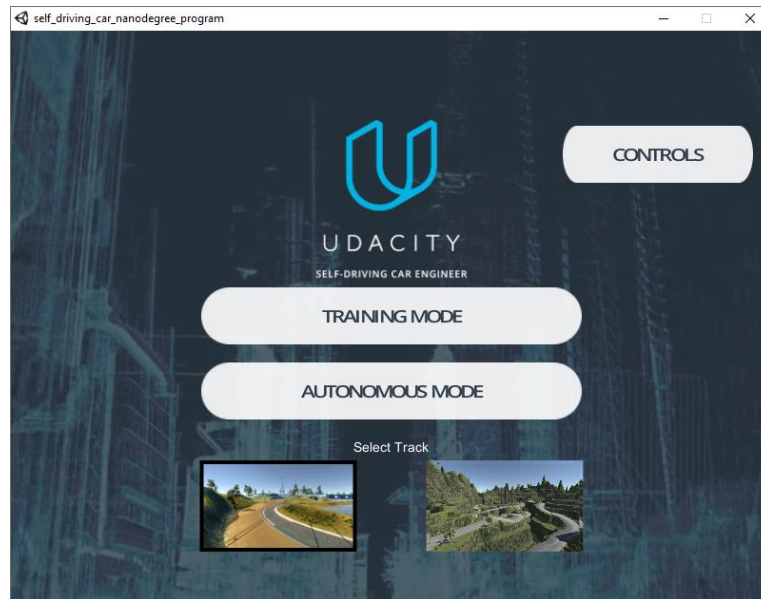


FIGURA 41. INTERFAZ DEL SIMULADOR

1) MODO ENTRENAMIENTO:

En este modo, seremos capaces de generar nuestro propio “batch” de datos para que la neurona sea capaz de entrenar a partir del modelo de NVIDIA. Lo primero que tenemos que seleccionar en la parte de abajo es el tipo de circuito en el que vamos a entrar. Es importante saber que los resultados de la neurona no serán los mismos si trabajamos con un circuito para el que no ha sido entrenado.



FIGURA 42. CIRCUITO DE LA MONTAÑA

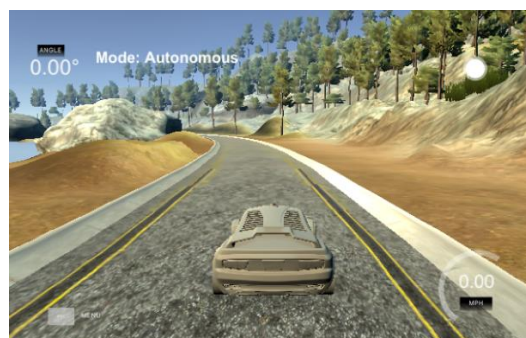


FIGURA 43. CIRCUITO DEL LAGO

Como el vehículo tiene tres cámaras para la adquisición de datos, se generará una carpeta “IMG” donde se almacenarán dichas imágenes. Cada una tendrá un nombre identificativo que las diferenciará de “left”, “right” o “center”. Además, en el directorio indicado también se generará el fichero CSV de forma automática.

Pero, antes que nada hemos de especificar dónde se van a almacenar toda esta información. Para ello, simplemente pulsamos la tecla “R” o pulsamos el botón de grabar y nos saldrá un menú para seleccionar el directorio deseado.

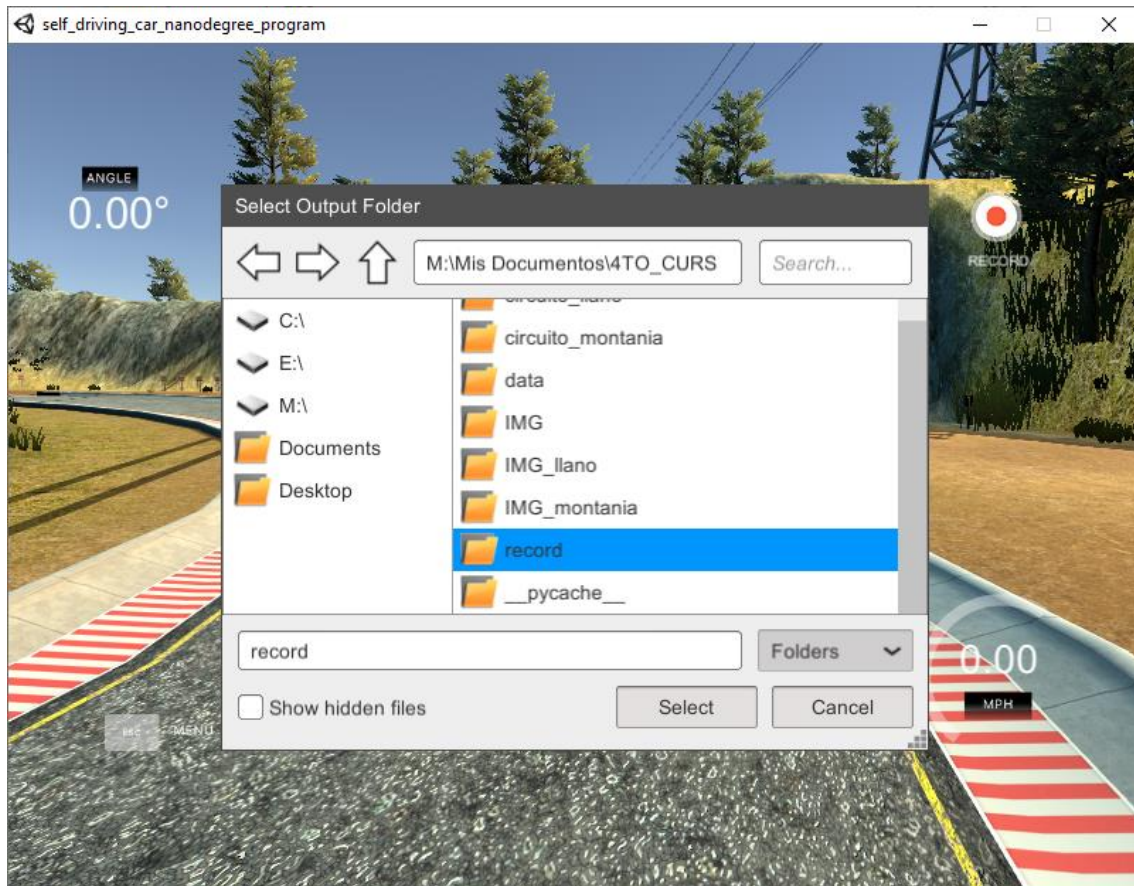


FIGURA 44. INTERFAZ DEL MODO ENTRENAMIENTO, SELECCIÓN DEL RECORD PATH.

Una vez hemos realizado los pasos, simplemente comenzamos a grabar todo lo que hace el vehículo. Nuestro objetivo es tomar muestras que sean lo más perfectas posibles, ya que nuestra CNN va a tomar de referencia el comportamiento de éste. Se sobreentiende que el coche se conduce de forma manual en todo este proceso de adquisición de datos.

2) MODO AUTÓNOMO:

Aquí también hemos de tener precaución en cuál de los circuitos vamos a abrir el modo, para que coincida con los datos con los que ha sido entrenada la red neuronal. El modo autónomo es bastante simple: adquiere el control manual si detecta que el usuario maneja el coche y se pasa al automático cuando no lo detecta.

IMPORTANTE: para poner en marcha el modo autónomo no basta sólo con entrar en dicho modo. Hemos de preparar el entorno con un script de conducción que se nos ofrece también como recurso en el repositorio de Udacity.

- ➔ “drive.py” - este es el nombre del script que vamos a emplear para establecer una comunicación con la red neuronal, que previamente hemos entrenado, y la interfaz del simulador.

Sin entrar en detalle de cómo funciona el script, es probable que tengamos que instalar algunos paquetes de recursos que el propio archivo utiliza para la comunicación del vehículo. Esto se especificará en los posibles errores de consola que nos devuelva al ejecutarlo.

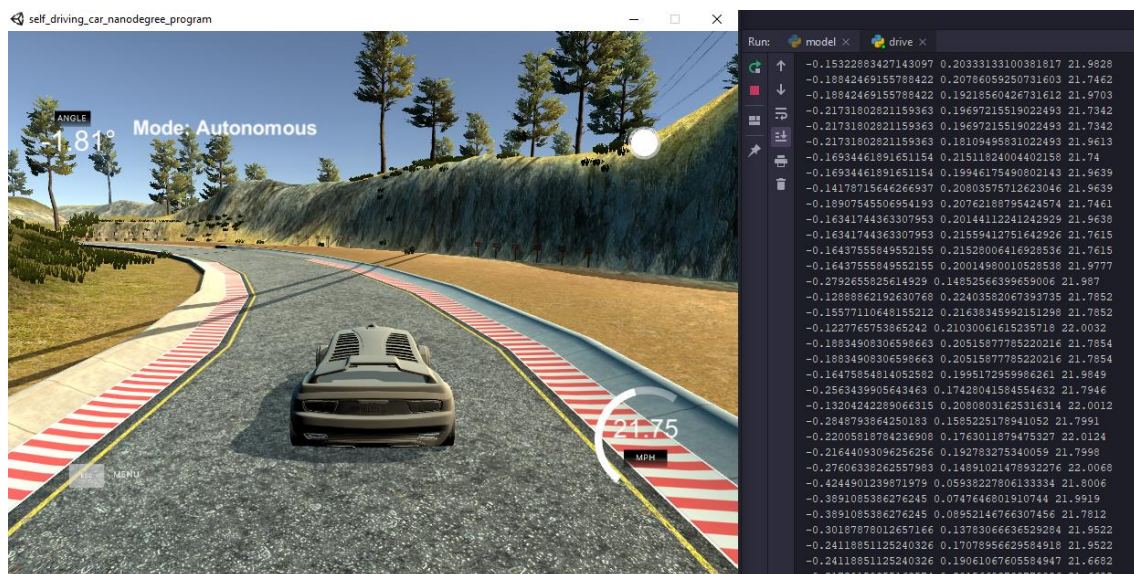


FIGURA 45. INTERFAZ DEL MODO AUTÓNOMO EN FUNCIONAMIENTO.

Como podemos observar en la figura superior, el script de conducción nos devuelve ciertos valores por consola a cerca de la aceleración, la velocidad y el giro que está aplicando en cada momento. Por defecto, la velocidad tiene un límite máximo de 25 km/h.

→ ¿Para qué es útil aprender el funcionamiento de este simulador?

El uso de este simulador nos ofrece muchas ventajas antes de ponernos manos a la obra con el diseño de nuestra propia neurona:

1. Entender la fase de adquisición de datos.
2. Entender qué es un CSV y cómo se estructura.
3. Entender cómo funciona el script de entrenamiento de NVIDIA.
4. Qué partes de dicho entrenamiento conforman el modelo, la carga de datos y el entrenamiento.
5. Cómo se comporta una neurona bajo un entorno conocido.
6. Cómo se comporta una neurona bajo un entorno NO conocido.

Sabiendo todo esto y como resultado de nuestro aprendizaje de haber usado el simulador “*Udacity’s Self-Driving Car Simulator*”, ahora estamos en disposición de:

1. **Buscar la manera más eficiente de adquirir los datos:** hemos visto que en el modo entrenamiento, se les pone un nombre a las imágenes con información importante (a qué cámara pertenecen, fecha y hora de la muestra). Es útil pensar que las imágenes que grabemos deberán tener un etiquetado similar a como nos la devuelve el simulador.
2. **Crear nuestro propio fichero CSV:** en el caso de simulador, se genera de forma automática, pero podemos observar de qué manera se guardan los datos. Con esta información, podemos generar nuestro propio fichero CSV con nuestro “batch” de datos (ver epígrafe 4.4.1) específico para entrenar el modelo de NVIDIA.
3. **Modificar las partes adecuadas en función del tipo de sistema que tengamos:** con esto, nos referimos sobre todo a la parte de carga de datos para el entrenamiento. Como vimos, el simulador trabajara con tres cámaras a diferencia de nuestro coche que sólo tiene 1.
4. **Qué tipo de resultados esperar una vez pongamos nuestra neurona en marcha:** hemos observado que en entornos similares a los de entrenamiento, el simulador obtiene buenos resultados, pero para circuitos diferentes, la red neuronal pierde todo su potencial. Extrapolado a nuestro proyecto, es importante mantener un mismo entorno de entrenamiento para que los resultados sean fiables.

5. RESULTADOS Y CONCLUSIONES

Aquí presentaremos los resultados que hemos obtenido, fruto de todo el trabajo realizado en la investigación de un campo desconocido y con la ayuda de muchos elementos que nos simplificaban el trabajo (como software libre o librerías que simplifican la programación).

Se verán los **resultados de los entrenamientos** neuronales, así como las pequeñas pruebas que hemos realizado en los diferentes entornos físicos (tipos de suelos, distintos contrastes, etc.) y extraeremos conclusiones de todo eso.

Como resultado clave, hemos conseguido crear un vehículo de conducción autónoma a pequeña escala basado en una inteligencia artificial, más concretamente, en una red neuronal convolucional, con nada más que 1 sensor de entrada: una cámara delantera.

5.1. RESULTADOS DE LA FASE DE ADQUISICIÓN DE DATOS.

Hemos realizado varias pruebas en diferentes circuitos diseñados a partir de cinta aislante negra en el suelo. A la hora de diseñar los circuitos, tenemos en cuenta la curvatura máxima que es capaz de realizar el coche. En nuestro caso, el ángulo de giro máximo corresponde al límite establecido por el servo, que es del orden de ± 30 grados.

También es importante que, para obtener posteriormente buenos resultados, exista un contraste alto entre las líneas que delimitan el circuito y el tipo de suelo. En la figura 46, tenemos de los primeros circuitos que usamos correspondientes al suelo del pasillo enfrente del laboratorio de diseño en la facultad ESIT de la ULL.



FIGURA 46. CIRCUITO DE PRUEBA 1 (LABORATORIO)

Ahora se muestra unas cuantas fotografías tomadas en el circuito de prueba 1 con el vehículo en modo manual (entrenamiento), que graba con un framerate (ratio de captura) de 30 imágenes por segundo.

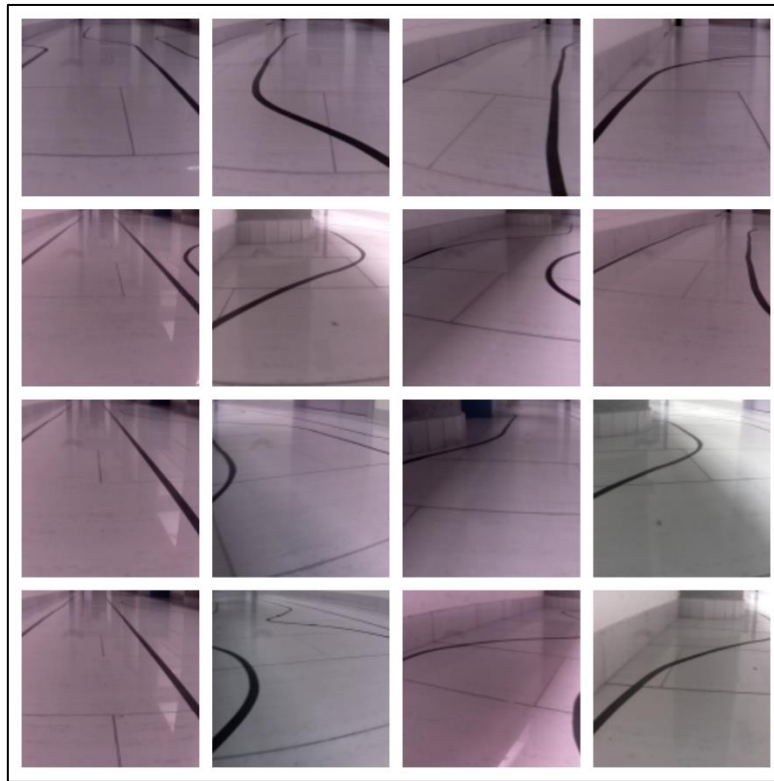


FIGURA 47. COLLAGE FOTOGRAFÍAS TOMA DE ENTRENAMIENTO 1

Otro de los circuitos con los que trabajamos fue en una zona de hormigón que sirve como garaje residencial. El circuito construido se puede ver en la figura 48, seguido de otro collage de muestra de las imágenes capturadas.

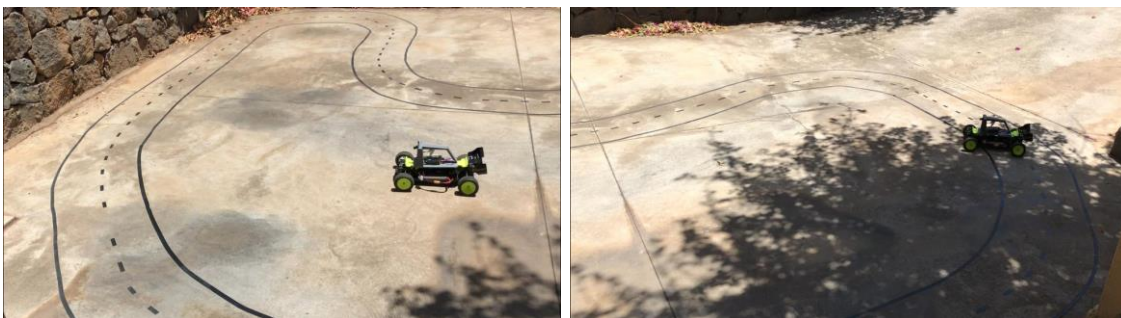


FIGURA 48. CIRCUITO DE PRUEBA 2



FIGURA 49. COLLAGE FOTOGRAFÍAS TOMA ENTRENAMIENTO 2

Aproximadamente, para un buen entrenamiento neuronal se darán entre 2 y 3 vueltas a nuestro circuito. Esto puede generar una cantidad de imágenes en torno a los 2 mil elementos. Cada vez que se quiera entrenar la neurona por incorporar algún cambio en el circuito o en las condiciones de ambiente, es necesario realizar este paso de adquisición.

Aunque aquí sólo se reflejen 2 entrenamientos clave, realmente para entrenar una neurona hay que realizar decenas de ellos, a modo de ensayo error. Por un lado, esto sirve para cambiar e implementar cambios en el modelo de la red, cambios en el circuito, cambios en el hardware (como modificar la posición de la cámara) y cambios en el script de entrenamiento.

5.2. RESULTADOS DEL ENTRENAMIENTO NEURONAL.

Para el entrenamiento de la red debimos tener en cuenta una serie de aspectos importantes antes de seleccionar la herramienta para ello. Tras varias pruebas por ensayo error, las versiones de software con la que trabajaba el coche y con las que entrenábamos y generábamos nuestro modelo neuronal debían ser las mismas, puesto que si eran dispares, luego el modelo no podía ser leído correctamente.

Por este motivo, no somos capaces de entrenar la neurona en entorno local, puesto que la mayoría de dispositivos Windows GPU sólo permiten versiones de TensorFlow para Python 3, mientras que todo el árbol de archivos de la base “BURRO” está programado en Python 2.

Entonces, con nuestra herramienta de Google Colab, nos aseguramos de que las versiones de programación sean las mismas que están instaladas en el vehículo. En nuestro caso:

- ➔ KERAS versión: 2.0.8
- ➔ TensorFlow versión: 1.1.0

```
[ ] import keras
import tensorflow as tf
print(keras.__version__)
print(tf.__version__)

[ ] Using TensorFlow backend.
2.0.8
1.1.0
```

FIGURA 50. CHECKING DE VERSIONES

De acuerdo con esto, para el entrenamiento de la red debemos tener disponibles el “batch” con nuestros datos de entrada (las imágenes que previamente hemos de grabar manejando el vehículo de forma manual en un circuito de entrenamiento), así como el fichero CSV que nos indique su dirección y las medidas del ángulo de giro para la validación.

Layer (type)	Output Shape	Param #
lambda_2 (Lambda)	(None, 120, 160, 3)	0
conv2d_6 (Conv2D)	(None, 58, 78, 24)	1824
conv2d_7 (Conv2D)	(None, 27, 37, 36)	21636
conv2d_8 (Conv2D)	(None, 12, 17, 48)	43248
conv2d_9 (Conv2D)	(None, 10, 15, 64)	27712
conv2d_10 (Conv2D)	(None, 8, 13, 64)	36928
dropout_2 (Dropout)	(None, 8, 13, 64)	0
flatten_2 (Flatten)	(None, 6656)	0
dense_5 (Dense)	(None, 100)	665700
dense_6 (Dense)	(None, 50)	5050
dense_7 (Dense)	(None, 10)	510
dense_8 (Dense)	(None, 1)	11
=====		
Total params: 802,619		
Trainable params: 802,619		
Non-trainable params: 0		
=====		
Epoch 1/1		
14999/15000 [=====>.] - ETA: 0s - loss: 0.1124Epoch 00000: val_loss: 0.0637		
15000/15000 [=====>.] - 14814s - loss: 0.1123 - val_loss: 0.0637		

FIGURA 51. RESULTADOS DEL ENTRENAMIENTO NEURONAL

En la figura 51 podemos observar una tabla resumen con los parámetros entrenables (802.619) de nuestra red, así como las características de las capas internas. Como se refleja, sólo hemos entrenado bajo una época, puesto que el tiempo requerido por **cada entrenamiento que queremos realizar es de 14814 segundos** para este caso en concreto, lo que equivale a entre unas 4 y 6 horas de entrenamiento.

Esto se debe a que por motivos de compatibilidad de versiones, no pudimos aprovecharnos de la ventaja que ofrece utilizar tecnología de aceleración por GPU. Además, el entorno de Google se desconecta a las 12 horas de realizar cualquier trabajo (para evitar minería de criptomonedas), por lo que las épocas de entrenamiento están limitadas en función de las muestras a entrenar.

Sin embargo, aunque la neurona se ha entrenado correctamente y el valor de pérdidas no es muy alto, estudiando la componente de precisión (“**accuracy**”) vemos que no llega siquiera a una precisión del 20%. Esto nos hace sospechar varias cosas: o que los datos de entrada no se están procesando bien, ya sea en la función “**fit_generator**” o dentro de las funciones retoque (edición de las fotos), o que el modelo no está perfectamente ajustado a nuestro prototipo.

5.3. CONCLUSIONES.

Hemos integrado y puesto en marcha en un vehículo de Radio control todo un sistema empotrado con sensores y actuadores capaz de adquirir imágenes y realizar acciones para la conducción autónoma del mismo en un circuito.

Dentro del sistema “**BURRO**”, una parte fundamental del proyecto ha sido su **análisis y comprensión**. Esto se debe a que consta del pilar base que conforma el firmware de nuestro equipo, encargado de manejar los sensores y actuadores y todo el procesamiento de datos. Para ello, hemos aprendido a desenvolvemos en entornos de programación orientada a objetos por Python y modificado los parámetros clave del código previamente estudiado.

Este sistema “**BURRO**” implementado está desarrollado hasta un punto en el que en los próximos TFG se puedan desarrollar partir de un nivel donde solo nos podamos centrar en el desarrollo de IA inteligencia artificial y así optimizar la misma en otros escenarios más complejos donde el vehículo pueda navegar.

En modo manual, el comportamiento es el adecuado, respondiendo de manera eficiente a las señales enviadas por el mando de comunicación bluetooth. En modo automático, también se esperan las respuestas deseadas en cuanto a firmware. Sin embargo, el comportamiento de la neurona no es

de la eficiencia esperada. Mientras que en periodos de rectas se mantiene dentro del circuito, cuando alcanza las curvas hay veces que sí actúa de manera correcta y otras que simplemente no reacciona o lo hace de forma equívoca.

Quizás, el modelo implementado necesita de algún que otro parámetro que ajuste mejor a nuestras condiciones del sistema, puesto que sólo tenemos una entrada de datos. De cualquier modo, por cuestiones de tiempo, la mayor parte de la investigación se ha centrado en entender el hardware y el firmware a utilizar. Sin ello, proceder con algo de jerarquía mayor sería inútil si no se entiende la estructura base. A partir de aquí, se debería profundizar en el modelo y maneras de realizar mejoras.

6. RECOMENDACIONES FUTURAS

En base a los resultados finales y a todo el proceso de desarrollo e investigación del prototipo, realizaremos una serie de sugerencias de mejora tanto de implementación software como de hardware que, por cuestiones de plazos, no se han podido implementar. Siempre estas mejoras están basadas en la experiencia que hemos tenido como usuarios del prototipo, y que sean mejoras de alcance accesible (que no sean imposibles de conseguir).

1) ENTRENAMIENTO DE LA IA CON ACELERADOR GPU.

Sería muy interesante que los entrenamientos futuros para las redes neuronales se realicen bajo un **entorno acelerado por GPU**. Esto nos lleva a utilizar versiones más recientes de TensorFlow y KERAS. Aquí nos vale utilizar cualquier entorno de programación, ya sea Google Colab de nuevo (para trabajar en la nube) o probar con uno local. Sin embargo, para poder usar las versiones más recientes de estas dos librerías tendríamos que ocuparnos de la siguiente recomendación:

2) MIGRAR EL CÓDIGO BURRO A PYTHON 3.

Como hemos mencionado en anteriores apartados, la aceleración por GPU sólo está disponible a partir de versiones que utilizan el motor de Python 3 en lugar del pronto obsoleto Python 2. A causa de esto, sólo será posible entrenar el modelo neuronal y llevarlo a la práctica si la base del software de **“BURRO” se migra a Python 3**, es decir, adaptar el código para instalarlo en versión Python 3 y sus correspondientes versiones actualizadas de TensorFlow y KERAS.

3) IMPLEMENTAR ESTEREO VISIÓN.

Cómo hemos podido corroborar gracias a la práctica del simulador, una conducción con estereo visión mejoraría notablemente la actuación del modelo neuronal, puesto que la precisión aumenta considerablemente bajo un mismo entorno de trabajo. No debemos olvidar que el modelo *“End-to-end Deep learning for Self-Driving Cars NVIDIA model”* estaba diseñado en una primera instancia para esta cantidad de entrada de datos.

4) MEJORAR LA ÓPTICA DE LA CÁMARA.

En algunas ocasiones, en base a las muestras de imágenes de entrenamiento, podemos pensar que la cámara no está lo suficientemente ajustada para captar bien el trazado de la pista en el suelo. Sería bueno estudiar una mejor manera de acoplar la cámara (si el ángulo de visión hacia el suelo puede ser más agudo, es decir, más mirando hacia abajo) o si es necesario hacer un post-procesado de las imágenes para recortar cierta banda de visión.

5) APROVECHAR MEJOR LAS FUNCIONES DE LA NAVIO2.

Realmente, nuestra tarjeta de acople NAVIO2 nos sirve para gestionar las señales generadas por la Raspberry calculadas a través del firmware de “BURRO” destinadas a los actuadores (motores). Como hemos visto en las especificaciones del fabricante, la NAVIO2 posee unos cuantos sensores extra que pueden resultar útiles para algunas funciones, como es el caso de las IMU.

6) AÑADIR UN RADAR FRONTAL.

Inicialmente estuvimos barajando la hipótesis de implementar un **radar de posicionamiento de objetos** en la parte frontal del vehículo, que le sirviera a modo de, junto con la cámara, un sistema más para mejorar el rendimiento en la conducción y la tasa de posibles choques, si en un futuro también se desea implementar la velocidad controlada de forma autónoma.

7) IMPLEMENTAR UNA NEURONA DE TIPO “INCEPTION”.

“INCEPTION”, o la llamada “Google .eNET”, es una de las redes más precisas y que consiguen mejores resultados ante procesos de categorización y regresión. Es una red predefinida que funciona a partir de un filtrado multiescala, definiendo una arquitectura por bloques. En estos bloques se realizan convoluciones en diferentes niveles y escalas, acompañado de un MaxPooling. De esta manera, se consigue extraer características de diferentes niveles en vez de limitarse en formas convolucionales limitadas de 3x3 o 5x5.

Estaría interesante estudiar la posibilidad de implementar este tipo de neurona en nuestro sistema, ya que nos ofrece una gran precisión. Eso sí, hay que ver si es compatible con las versiones, además de si se puede implementar sin que suponga un retardo en el tiempo de cálculo computacional del sistema.

7. PRESUPUESTO

Para concluir, esbozaremos unos presupuestos contrastados con los medios materiales que hemos utilizado, así como el tiempo empleado y su estimación a primera escala.

Componente	Modelo	Precio (€)
Coche RC	QUANUM VANDAL	135.45
Placa Navio	2	226.08
Raspberry Pi	3 B+	36.95
Batería	5200 mAh / 7.4V / 30C / 2S	21.54
Cargador batería	TURNIGY Accucell 6	28.57
Tarjeta SD	SAMSUNG EVO 16GB	15.99
Mando radio control	Logitech F710	36.99
Cámara	Longruner camera module	25.99
Estructura impresión 3D	--	10.00
Trabajo personal	120 días (6 horas de trabajo)	4800.00
TOTAL		5337.56 €

8. REFERENCIAS BIBLIOGRÁFICAS

REFERENCIAS:

- [1] “Inteligencia artificial y machine learning: llega la revolución”. Septiembre 2018. Disponible en: <https://www.silicon.es/fondo-inteligencia-artificial-machine-learning-revolucion-2372422>
- [2] “Computer Controlled Cars”. 1969. Disponible en: <http://www-formal.stanford.edu/jmc/progress/cars/cars.html>
- [3] “TensorFlow, o cómo será el futuro de la Inteligencia Artificial según Google” por Manuel Zaforas, 2017. Disponible en: <https://www.paradigmadigital.com/dev/tensorflow-sera-futuro-la-inteligencia-artificial-segun-google/>
- [4] “How to Simulate a Self-Driving Car” 2017. Disponible en: <https://www.youtube.com/watch?v=EaY5QiZwSP4&t=742s>
- [5] “A self-driving car simulator built with Unity” 2017. Disponible en: <https://github.com/udacity/self-driving-car-sim>
- [6] Curso Udemy: “Deep Learning e Inteligencia artificial con KERAS/ TensorFlow” 2019. Disponible en: <https://www.udemy.com/course/curso-de-deep-learning-con-kerastensorflow-en-python/>
- [7] “Lane Following Autopilot with Keras & Tensorflow” 2017. Disponible en: <https://wroscoc.github.io/keras-lane-following-autopilot.html>
- [8] Modelo de NVIDIA, repositorio “How_to_simulate_a_self_driving_car”, 2017. Disponible en: https://github.com/llSourcell/How_to_simulate_a_self_driving_car/blob/master/model.py
- [9] “Self-Driving Car Steering Angle Prediction Based on Image Recognition”, por Shuyang Du, Haoli Guo y Andrew Simpson, 2017. Disponible en: <http://cs231n.stanford.edu/reports/2017/pdfs/626.pdf>
- [10] “Navio2 Docs”. Disponible en: <https://docs.emlid.com/navio2/>
- [11] “Teaching Cars To Drive Using Deep Learning – Steering Angle Prediction”, Eddie Forson, 2017. Disponible en: <https://towardsdatascience.com/teaching-cars-to-drive-using-deep-learning-steering-angle-prediction-5773154608f2>
- [12] Google Colaboratory - Introducción. Disponible en: https://colab.research.google.com/notebooks/welcome.ipynb#scrollTo=n_bCKVLZ4vBfC
- [13] PyCharm - Features. Disponible en: <https://www.jetbrains.com/pycharm/features/>
- [14] Conocimientos de CSV, Disponibles en: <https://realpython.com/python-csv/>
- [15] “NVIDIA model developer blog”, Disponible en: <https://devblogs.nvidia.com/deep-learning-self-driving-cars/>

- [16] “End to End learning for Self-Driving Cars”, 2016. Disponible en: <https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf>
- [17] Información de las capas de KERAS, Disponible en: <https://keras.io/layers/core>
- [18] Niveles de conducción autónoma, 2016, Disponible en: <https://www.motorpasion.com/tecnologia/la-conduccion-autonoma-paso-a-paso-del-conductor-humano-al-coche-que-se-conduce-el-solo>
- [19] “John McCarthy, el arranque de la inteligencia artificial”, 2011, Disponible en: https://elpais.com/diario/2011/10/27/necrologicas/1319666402_850215.html
- [20] “La historia del Pontiac autónomo que cruzó Estados Unidos "sin manos" en 1995”, 2015, Disponible en: <https://www.diariomotor.com/breve/coche-autonomo-1995-no-hands-across-america/>
- [21] “The Grand Challenge”, Disponible en: <https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles>
- [22] “DARPA Urban Challenge”, Disponible en: <https://www.darpa.mil/about-us/timeline/darpa-urban-challenge>
- [23] “The DARPA Grand Challenge: Ten Years Later”, 2014, Disponible en: <https://www.darpa.mil/news-events/2014-03-13>
- [24] “Autonomous Vehicles Complete DARPA Urban Challenge”, 2007, Disponible en: <https://spectrum.ieee.org/transportation/advanced-cars/autonomous-vehicles-complete-darpa-urban-challenge>
- [25] “DARPA Autonomous Vehicle Race Proves What's Possible”, 2005, Disponible en: <https://archive.defense.gov/news/newsarticle.aspx?id=18095>
- [26] “La historia de los carros autónomos contada en unos pocos hitos”, 2017, Disponible en: <https://es.digitaltrends.com/autos/historia-carros-autonomos/>
- [27] “Las tres empresas que ganan (por ahora) la carrera de los coches autónomos”, 2018, Disponible en: <https://www.lainformacion.com/management/las-tres-empresas-que-ganan-por-ahora-la-carrera-de-los-coches-autonomos/6347765/>
- [28] “Hobbywing 1/10 Waterproof Brushless ESC & 3250KV 4-pole 540 Motor Combo”, Disponible en: https://www.ebay.co.uk/itm/Hobbywing-1-10-Waterproof-Brushless-ESC-3250KV-4-pole-540-Motor-Combo-/264222415751?_trksid=p2349526.m4383.l4275.c1
- [29] “Navio 2 Documentación”, Disponible en: <https://docs.emlid.com/navio2/>
- [30] “Navio 2 Especificaciones”, Disponible en: <http://ardupilot.org/rover/docs/common-navio2-overview.html>

- [31] “Longrunner Camera Module for Raspberry Pi 5MP 1080p OV5647”, Disponible en: <http://longrunnerpro.com/a/Products/20171014/79.html#targetText=Longrunner%20Camera%20Module%20for%20Raspberry%20Pi%205MP%201080p%20V5647&targetText=Raspberry%20Pi%20Night%20Vision%20Camera,Adjustable%20focal%20length.>
- [32] “Raspberry Pi 3 Model B+”, Disponible en: <https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>
- [33] “Esta es la energía que consume cada modelo de Raspberry Pi, incluido el nuevo Raspberry Pi Zero W”, 2017, Disponible en: <https://www.redeszone.net/2017/03/02/energia-consumida-raspberry-pi/>

IMÁGENES Y FIGURAS:

- [34] https://www.researchgate.net/profile/Anas_Alhashimi/publication/328749277/figure/download/fig1/AS:689758384705539@1541462622588/A-simplified-drawing-of-the-various-sensors-present-in-Googles-self-driving-cars-The.png -- Figura 1.
- [35] <https://tr3.cbsistatic.com/hub/i/r/2016/10/25/fe1bb470-856b-42f7-b9e0-219ab041c78f/resize/770x/a250414eff225598524098d8fdad/5.jpg> -- Figura 4.
- [36] https://ep01.epimg.net/diario/imagenes/2011/10/27/necrologicas/1319666402_850215_0000000000_sumario_normal.jpg -- Figura 5.
- [37] https://s.hdnux.com/photos/62/40/24/13239562/3/gallery_medium.jpg -- Figura 6.
- [38] https://archive.darpa.mil/grandchallenge04/Web_Photos/sat/Full/DARPA_GCSa_32.jpg -- Figura 7.
- [39] <https://spectrum.ieee.org/image/Mjk20Tk1NQ.jpeg> -- Figura 8.
- [40] https://insider.si.edu/wp-content/uploads/2009/11/2005_stanley-driving.JPG -- Figura 9.
- [41] https://assets.pando.com/_versions/2013/04/seb_thrun_google_featured.jpg -- Figura 10.
- [42] <https://cdn-global-hk.hobbyking.com/media/catalog/product/cache/1/image/660x415/17f82f742ffe127f42dca9de82fb58b1/1/7/177491.jpg> -- Figura 11.
- [43] https://static.wixstatic.com/media/cc7476_fcc2da0584ee4694a21bd4332ac420d3~mv2.jpg/v1/fill/w_792,h_268,al_c,lg_1,q_85/cc7476_fcc2da0584ee4694a21bd4332ac420d3~mv2.jpg -- Figura 12.
- [44] <https://images-na.ssl-images-amazon.com/images/I/51yPcJhesXL.jpg> -- Figura 13.
- [45] <https://i.ebayimg.com/images/g/wLoAAOSw0Tpce725/s-l640.jpg> -- Figura 14.

- [46] <https://i.ebayimg.com/images/g/w~AAAOSwe-lce73g/s-l640.jpg> -- Figura 15.
- [47] https://images-na.ssl-images-amazon.com/images/I/61RA8iyEa6L._SL1000_.jpg -- Figura 16.
- [48] https://i.blogs.es/7d4fa6/rpi3b-1/450_1000.jpg -- Figura 17.
- [49] <http://emlid.com/wp-content/uploads/2015/12/Navio2-features.jpg> -- Figura 18.
- [50] https://cdn.coordiutil.com/imagen-gamepad_logitech_f710_inalambrico-1301954-800-600-1-75.jpg -- Figura 19.
- [51] http://docs.donkeycar.com/assets/build_hardware/traxxas.PNG -- Figura 20.
- [52] https://cdn.thingiverse.com/renders/c3/bd/62/61/1e/4a098b03023889d2ce751ea4ec993f73_preview_featured.jpg -- Figura 21.
- [53] <https://docs.emlid.com/navio2/ardupilot/img/navio2-mount.png> -- Figura 23.
- [54] http://docs.donkeycar.com/assets/build_hardware/assemble_camera.jpg -- Figura 24.
- [55] http://docs.donkeycar.com/assets/build_hardware/6b.PNG -- Figura 25.
- [56] https://raw.githubusercontent.com/wiki/yconst/burro/images/navio_setup.png -- Figura 26.
- [57] <https://www.lucidarme.me/simplest-perceptron-update-rules-demonstration/> -- Figura 29.
- [58] https://es.wikipedia.org/wiki/Perceptr%C3%B3n_multicapa -- Figura 32.
- [59] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html -- Figura 31.
- [60] Referencia [6] -- Figuras 30, 33, 34, 35 y 36.
- [61] Referencia [16] -- Figuras 37 y 38.
- [62] <https://unity.com/es/solutions/automotive-transportation-manufacturing> -- Figuras 39 y 40.

9. ANEXOS

9.1. COMPUTER CONTROLLED CARS

John McCarthy

Computer Science Department

Stanford University

Stanford, CA 94305

jmc@cs.stanford.edu

<http://www-formal.stanford.edu/jmc/>

Abstract:

This article is originally from approximately 1968, but was probably revised in the 1970s.

There have been a number of proposals for automatic control of cars. Mostly, they have involved simple servo-mechanisms that sense a cable buried in the roadway and some other mechanism for sensing the distance of the car ahead. Such a scheme was studied at RCA at the instigation of Zworykin, but the work was eventually abandoned.

In science fiction, systems in which a single computer controls all the cars in a wide area have been depicted but without telling how the system would actually work.

We are also proposing the computer control of cars. Our system requires a computer in the car equipped with television camera input that uses the same visual input available to the human driver. Essentially, we are proposing an automatic chauffeur. Our goal is a system with the following qualities:

The user enters the destination with a keyboard, and the car drives him there. Other commands include: change destination, stop at that rest room or restaurant, go slow, go at emergency speed.

The user need not be a driver and need not even accompany the car. This would permit children, old people, and the blind greater personal freedom. It also permits a husband to be driven to work, then send the car home for his wife's use, and permits her to send it back for him at the end of the day. The car can be sent for servicing or to a store where a telephone ordered purchase will be put in it. If there is a suitable telephone system, the car can deliver a user to a place where there is no parking, go away and park, and return when summoned. Thus, the system is to have almost all of the capabilities of a chauffeur.

In contrast to a system based on a central computer, the proposed system will be of advantage to the first person who buys one, whether anyone else has it or not. It will require no change in existing roads, but will be able to take orders from traffic control computers when they are installed. When freeway lanes can be dedicated to computer controlled cars they will multiply the capacity of existing freeways by permitting 80 mile per hour bumper-to-bumper traffic with greater safety than we have at present. Since the system is a product and not a public utility, competition among suppliers will be possible.

A key goal is to achieve greater safety than we have at present. A fivefold reduction in fatalities is probably required to make the system acceptable. Much better is possible since humans really are rather bad drivers, but complete safety cannot be guaranteed.

Now we shall consider the problems that have to be solved in order to realize the system.

Performance of the computer, cameras, and associated electronics. Present computers seem to be fast enough and to have enough memory for the job. However, commercial computers of the required performance are too big. We envisage that a computer of about the power of the Digital Equipment PDP-10 will be required. Military versions of similar computers have volumes of one or two cubic feet, but the requirements for memory and secondary storage would be difficult to meet in a reasonable volume at present. However, the development of more compact computers and other electronic circuitry is proceeding at a rate that makes achieving the required compactness not the pacing item. Some improvements in the performance and compactness of television cameras is also required, but it is not yet clear what these requirements are.

Cost of the computer and other electronics. At present prices, a computer capable of controlling a car but containable only in a large van would cost \$400,000 to \$800,000. A few thousand dollars worth of other electronics

would be required. Ten years should bring the cost down by a factor of ten. Mass production would give another factor of three. This would permit the system to be available as a luxury item. Another five to ten years might be required before computer control would only double the price of the car. These estimates must be regarded as guesses.

Reliability of the computer and other electronics. We can attempt to compute the required reliability by demanding that present traffic fatalities be reduced to a fifth the present number, i.e. to 10,000 per year, and by allocating only half of these fatalities to unreliability of the electronics. This further depends on the fraction of failures that lead to fatality which can be kept quite low by having the computer check its health and that of the electronics every tenth of a second, giving it programs for dealing with partial failures, and providing a "dead man switch" for stopping the car if the computer fails to reassure it every tenth second. There are many possibilities in this direction and the expenditure of much cleverness is called for. The reader is advised against using his unaided intuition to estimate the results. Nevertheless, present computer failure rates would not be acceptable even if they never led to accident simply because of the inconvenience. We estimate that an improvement of 1000 in mean-time-between-failures is required. Rapid progress is being made in this field, and we expect that ten to fifteen years normal progress of the computer field will give the required result.

Performance of the driving programs. Developing the required computer programs is the most difficult of the required tasks; it will probably take the longest time; and the amount of time required is very difficult to predict. Work on computer control of vehicles has started at the Stanford University Artificial Intelligence Project. An experimental vehicle has been equipped with a television camera and connected to the computer with a two-way TV and radio link. A simple program to guide the vehicle to follow a white line like that in a road has been successfully checked out, and programs for determining the course of the road and detecting cars and other obstacles are being developed. However, before computer controlled cars become a reality a much larger scale effort will have to be made.

The nature and extent of this effort are not easy to foresee yet. We are far from having exhausted the possibilities of our present equipment, but eventually the radio link to the computer will have to be replaced by a computer in the vehicle, and television equipment capable of seeing better into shadows in the presence of bright areas will be required. We need to be able to identify many different types of objects on the road such as: persons, vehicles, animals, traffic police, shadows, pieces of paper, cardboard boxes, objects that have dropped from vehicles, traffic signs and other signals, intersections, house numbers, and other information required for navigation.

It will have to be equipped with programs to recognize and deal with a variety of emergency conditions. It will surely be possible to make it better at this than humans since its attention won't lapse, it can sense the mechanical condition of the car continuously, and it can look to the side, underneath the car, and behind every second.

The most intricate single problem is the visual pattern recognition.

Testing. After the required performance is demonstrated and before the system can be trusted without a human driver an extensive testing program is required. To demonstrate that the system is five times safer than a human driver approximately 25,000 automobile years will be required. This might be reduced somewhat by concentrating testing on situations in which humans make most of their fatal mistakes, but we would still need to be sure that situations in which the program made fatal blunders peculiar to the computer system were rare enough. Developments in the mathematical theory of computation may permit getting rid of ordinary programming errors and proving that they are absent, but possible inadequacies in the algorithms themselves can only be obviated by testing.

Public acceptance. Automobiles without qualified human drivers will require changes in the law. Fortunately, testing such systems with a driver present to take over if necessary does not. Moreover, computer driven cars will not be able to obey oral instructions from policemen, so a digital system will have to be developed. A general resistance to technological innovation on the part of the literary culture will have to be overcome, but it seems to me that after the test phase the advantages will be clear enough so that this will not be difficult.

Support for research and development. The development of computer controlled cars will cost hundreds of millions of dollars. A computer program capable of reliably taking care of all the contingencies that can arise in driving a car will have to be more complex than any ever written, and adequate testing will require a complex organization. Fortunately, The commitment of large amounts of money will be required only after spectacular though unreliable performance will have been demonstrated. So far as I know, the Stanford Artificial Intelligence Project is the only organization now working on computer control of a vehicle using vision. This work is part of a basic research project on artificial intelligence supported by the Department of Defense. Even at the present stage of the work, other projects are needed to secure an adequate diversity of approach. While

considerable additional progress will certainly be made with the present support, even a prototype will require more money than is now available.

Fortunately, this problem is within the jurisdiction of the Department of Transportation. The automobile companies and the computer companies also might be expected to help, but their past record of seeing beyond the ends of their noses is not encouraging. Because the programming is the pacing item, more support at this time will hasten the day when computer control of cars is achieved, but the possibilities will be much more obvious in five years with the advances in hardware and programming that are already taking place for other reasons.

Finally, we would like to deal with some arguments that might be raised against supporting research aimed at computer controlled cars:

Cars must be done away with because they produce smog, require too much space, and use up too much natural resources. We believe the smog devices will eventually be made to work well, or if not, another form of propulsion can be found. Computer controlled cars will require less space than equivalent present cars because they can go faster and closer together on streets, roads and freeways, because they can park at a distance from a place where they discharge passengers, and because a computer driven car can be shared more easily than a conventional car. If hydrocarbon fuel runs out and is still required for cars, then with nuclear energy, the burning reaction can be driven backwards and fuel synthesized from carbon dioxide and water.

A simpler scheme of automatic control is preferable. The buried cable and other simple schemes do not increase human freedom and convenience. They only permit us to use the freeways a bit more efficiently. Because of their inability to detect dogs, children, potholes, and objects that have fallen from trucks they may require unrealizable control of access to the highway in order to achieve safety.

Some form of automated mass transportation is obviously better. The automobile can go point to point in areas of both low and high density. We believe that these advantages should not and will not voluntarily be given up. We favor the development of improved mass transportation, but predict that the automobile will be given up only for something that works better in all ways such as an individual computer controlled flying machine capable of point to point transportation.

9.2. END TO END LEARNING FOR SELF-DRIVING CARS

REFERENCIA [16]

End to End Learning for Self-Driving Cars

Mariusz Bojarski NVIDIA Corporation Holmdel, NJ 07735	Davide Del Testa NVIDIA Corporation Holmdel, NJ 07735	Daniel Dworakowski NVIDIA Corporation Holmdel, NJ 07735	Bernhard Firner NVIDIA Corporation Holmdel, NJ 07735
Beat Flepp NVIDIA Corporation Holmdel, NJ 07735	Prasoon Goyal NVIDIA Corporation Holmdel, NJ 07735	Lawrence D. Jackel NVIDIA Corporation Holmdel, NJ 07735	Mathew Monfort NVIDIA Corporation Holmdel, NJ 07735
Urs Muller NVIDIA Corporation Holmdel, NJ 07735	Jiakai Zhang NVIDIA Corporation Holmdel, NJ 07735	Xin Zhang NVIDIA Corporation Holmdel, NJ 07735	Jake Zhao NVIDIA Corporation Holmdel, NJ 07735
Karol Zieba NVIDIA Corporation Holmdel, NJ 07735			

Abstract

We trained a convolutional neural network (CNN) to map raw pixels from a single front-facing camera directly to steering commands. This end-to-end approach proved surprisingly powerful. With minimum training data from humans the system learns to drive in traffic on local roads with or without lane markings and on highways. It also operates in areas with unclear visual guidance such as in parking lots and on unpaved roads.

The system automatically learns internal representations of the necessary processing steps such as detecting useful road features with only the human steering angle as the training signal. We never explicitly trained it to detect, for example, the outline of roads.

Compared to explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously. We argue that this will eventually lead to better performance and smaller systems. Better performance will result because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e. g., lane detection. Such criteria understandably are selected for ease of human interpretation which doesn't automatically guarantee maximum system performance. Smaller networks are possible because the system learns to solve the problem with the minimal number of processing steps.

We used an NVIDIA DevBox and Torch 7 for training and an NVIDIA DRIVE™ PX self-driving car computer also running Torch 7 for determining where to drive. The system operates at 30 frames per second (FPS).

1 Introduction

CNNs [1] have revolutionized pattern recognition [2]. Prior to the widespread adoption of CNNs, most pattern recognition tasks were performed using an initial stage of hand-crafted feature extraction followed by a classifier. The breakthrough of CNNs is that features are learned automatically from training examples. The CNN approach is especially powerful in image recognition tasks because the convolution operation captures the 2D nature of images. Also, by using the convolution kernels to scan an entire image, relatively few parameters need to be learned compared to the total number of operations.

While CNNs with learned features have been in commercial use for over twenty years [3], their adoption has exploded in the last few years because of two recent developments. First, large, labeled data sets such as the Large Scale Visual Recognition Challenge (ILSVRC) [4] have become available for training and validation. Second, CNN learning algorithms have been implemented on the massively parallel graphics processing units (GPUs) which tremendously accelerate learning and inference.

In this paper, we describe a CNN that goes beyond pattern recognition. It learns the entire processing pipeline needed to steer an automobile. The groundwork for this project was done over 10 years ago in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE) [5] in which a sub-scale radio control (RC) car drove through a junk-filled alley way. DAVE was trained on hours of human driving in similar, but not identical environments. The training data included video from two cameras coupled with left and right steering commands from a human operator.

In many ways, DAVE-2 was inspired by the pioneering work of Pomerleau [6] who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. It demonstrated that an end-to-end trained neural network can indeed steer a car on public roads. Our work differs in that 25 years of advances let us apply far more data and computational power to the task. In addition, our experience with CNNs lets us make use of this powerful technology. (ALVINN used a fully-connected network which is tiny by today's standard.)

While DAVE demonstrated the potential of end-to-end learning, and indeed was used to justify starting the DARPA Learning Applied to Ground Robots (LAGR) program [7], DAVE's performance was not sufficiently reliable to provide a full alternative to more modular approaches to off-road driving. DAVE's mean distance between crashes was about 20 meters in complex environments.

Nine months ago, a new effort was started at NVIDIA that sought to build on DAVE and create a robust system for driving on public roads. The primary motivation for this work is to avoid the need to recognize specific human-designated features, such as lane markings, guard rails, or other cars, and to avoid having to create a collection of "if, then, else" rules, based on observation of these features. This paper describes preliminary results of this new effort.

2 Overview of the DAVE-2 System

Figure 1 shows a simplified block diagram of the collection system for training data for DAVE-2. Three cameras are mounted behind the windshield of the data-acquisition car. Time-stamped video from the cameras is captured simultaneously with the steering angle applied by the human driver. This steering command is obtained by tapping into the vehicle's Controller Area Network (CAN) bus. In order to make our system independent of the car geometry, we represent the steering command as $1/r$, where r is the turning radius in meters. We use $1/r$ instead of r to prevent a singularity when driving straight (the turning radius for driving straight is infinity). $1/r$ smoothly transitions through zero from left turns (negative values) to right turns (positive values).

Training data contains single images sampled from the video, paired with the corresponding steering command ($1/r$). Training with data from only the human driver is not sufficient. The network must learn how to recover from mistakes. Otherwise the car will slowly drift off the road. The training data is therefore augmented with additional images that show the car in different shifts from the center of the lane and rotations from the direction of the road.

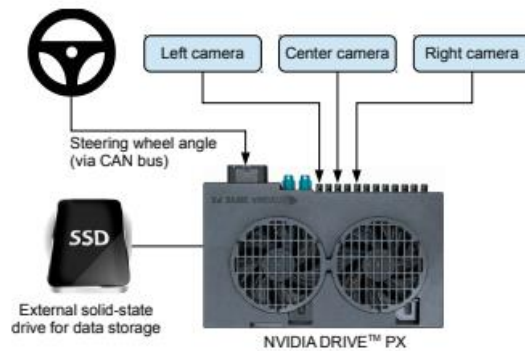


Figure 1: High-level view of the data collection system.

Images for two specific off-center shifts can be obtained from the left and the right camera. Additional shifts between the cameras and all rotations are simulated by viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don't have. We therefore approximate the transformation by assuming all points below the horizon are on flat ground and all points above the horizon are infinitely far away. This works fine for flat terrain but it introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately these distortions don't pose a big problem for network training. The steering label for transformed images is adjusted to one that would steer the vehicle back to the desired location and orientation in two seconds.

A block diagram of our training system is shown in Figure 2. Images are fed into a CNN which then computes a proposed steering command. The proposed command is compared to the desired command for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation as implemented in the Torch 7 machine learning package.

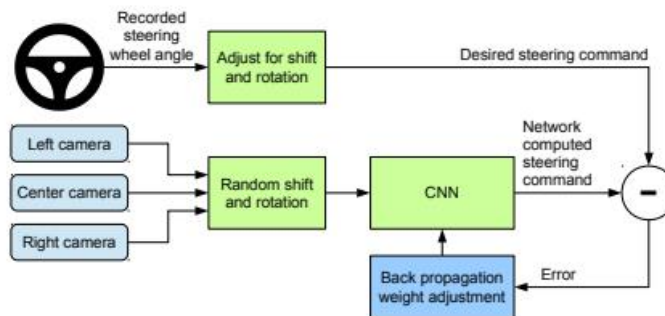


Figure 2: Training the neural network.

Once trained, the network can generate steering from the video images of a single center camera. This configuration is shown in Figure 3.



Figure 3: The trained network is used to generate steering commands from a single front-facing center camera.

3 Data Collection

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Most road data was collected in central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York. Other road types include two-lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Data was collected in clear, cloudy, foggy, snowy, and rainy weather, both day and night. In some instances, the sun was low in the sky, resulting in glare reflecting from the road surface and scattering from the windshield.

Data was acquired using either our drive-by-wire test vehicle, which is a 2016 Lincoln MKZ, or using a 2013 Ford Focus with cameras placed in similar positions to those in the Lincoln. The system has no dependencies on any particular vehicle make or model. Drivers were encouraged to maintain full attentiveness, but otherwise drive as they usually do. As of March 28, 2016, about 72 hours of driving data was collected.

4 Network Architecture

We train the weights of our network to minimize the mean squared error between the steering command output by the network and the command of either the human driver, or the adjusted steering command for off-center and rotated images (see Section 5.2). Our network architecture is shown in Figure 4. The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers. The input image is split into YUV planes and passed to the network.

The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. Performing normalization in the network allows the normalization scheme to be altered with the network architecture and to be accelerated via GPU processing.

The convolutional layers were designed to perform feature extraction and were chosen empirically through a series of experiments that varied layer configurations. We use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel and a non-strided convolution with a 3×3 kernel size in the last two convolutional layers.

We follow the five convolutional layers with three fully connected layers leading to an output control value which is the inverse turning radius. The fully connected layers are designed to function as a controller for steering, but we note that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor and which serve as controller.

5 Training Details

5.1 Data Selection

The first step to training a neural network is selecting the frames to use. Our collected data is labeled with road type, weather condition, and the driver's activity (staying in a lane, switching lanes, turning, and so forth). To train a CNN to do lane following we only select data where the driver was staying in a lane and discard the rest. We then sample that video at 10 FPS. A higher sampling rate would result in including images that are highly similar and thus not provide much useful information.

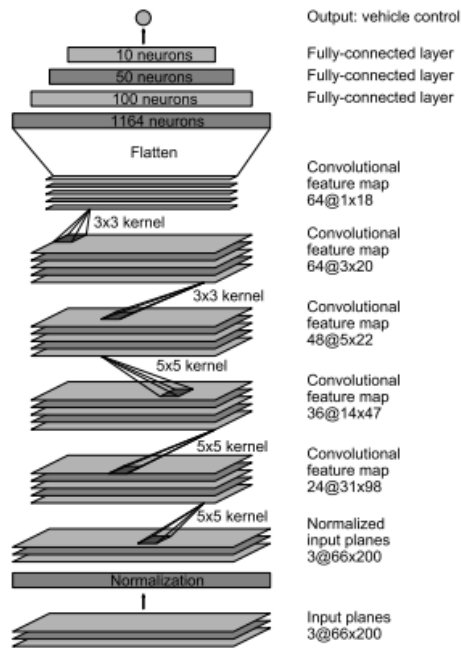


Figure 4: CNN architecture. The network has about 27 million connections and 250 thousand parameters.

To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves.

5.2 Augmentation

After selecting the final set of frames we augment the data by adding artificial shifts and rotations to teach the network how to recover from a poor position or orientation. The magnitude of these perturbations is chosen randomly from a normal distribution. The distribution has zero mean, and the standard deviation is twice the standard deviation that we measured with human drivers. Artificially augmenting the data does add undesirable artifacts as the magnitude increases (see Section 2).

6 Simulation

Before road-testing a trained CNN, we first evaluate the networks performance in simulation. A simplified block diagram of the simulation system is shown in Figure 5.

The simulator takes pre-recorded videos from a forward-facing on-board camera on a human-driven data-collection vehicle and generates images that approximate what would appear if the CNN were, instead, steering the vehicle. These test videos are time-synchronized with recorded steering commands generated by the human driver.

Since human drivers might not be driving in the center of the lane all the time, we manually calibrate the lane center associated with each frame in the video used by the simulator. We call this position the “ground truth”.

The simulator transforms the original images to account for departures from the ground truth. Note that this transformation also includes any discrepancy between the human driven path and the ground truth. The transformation is accomplished by the same methods described in Section 2.

The simulator accesses the recorded test video along with the synchronized steering commands that occurred when the video was captured. The simulator sends the first frame of the chosen test video, adjusted for any departures from the ground truth, to the input of the trained CNN. The CNN then returns a steering command for that frame. The CNN steering commands as well as the recorded human-driver commands are fed into the dynamic model [8] of the vehicle to update the position and orientation of the simulated vehicle.

The simulator then modifies the next frame in the test video so that the image appears as if the vehicle were at the position that resulted by following steering commands from the CNN. This new image is then fed to the CNN and the process repeats.

The simulator records the off-center distance (distance from the car to the lane center), the yaw, and the distance traveled by the virtual car. When the off-center distance exceeds one meter, a virtual human intervention is triggered, and the virtual vehicle position and orientation is reset to match the ground truth of the corresponding frame of the original test video.

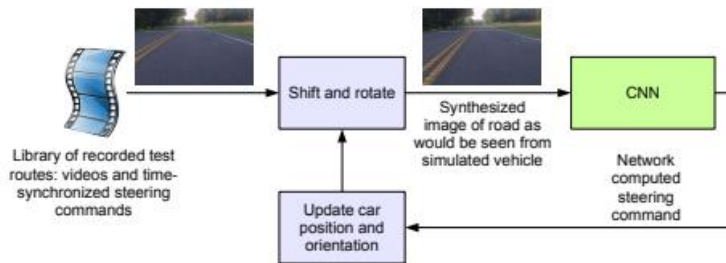


Figure 5: Block-diagram of the drive simulator.

7 Evaluation

Evaluating our networks is done in two steps, first in simulation, and then in on-road tests.

In simulation we have the networks provide steering commands in our simulator to an ensemble of prerecorded test routes that correspond to about a total of three hours and 100 miles of driving in Monmouth County, NJ. The test data was taken in diverse lighting and weather conditions and includes highways, local roads, and residential streets.

7.1 Simulation Tests

We estimate what percentage of the time the network could drive the car (autonomy). The metric is determined by counting simulated human interventions (see Section 6). These interventions occur when the simulated vehicle departs from the center line by more than one meter. We assume that in real life an actual intervention would require a total of six seconds: this is the time required for a human to retake control of the vehicle, re-center it, and then restart the self-steering mode. We calculate the percentage autonomy by counting the number of interventions, multiplying by 6 seconds, dividing by the elapsed time of the simulated test, and then subtracting the result from 1:

$$\text{autonomy} = \left(1 - \frac{(\text{number of interventions}) \cdot 6 \text{ seconds}}{\text{elapsed time [seconds]}}\right) \cdot 100 \quad (1)$$



Figure 6: Screen shot of the simulator in interactive mode. See Section 7.1 for explanation of the performance metrics. The green area on the left is unknown because of the viewpoint transformation. The highlighted wide rectangle below the horizon is the area which is sent to the CNN.

Thus, if we had 10 interventions in 600 seconds, we would have an autonomy value of

$$\left(1 - \frac{10 \cdot 6}{600}\right) \cdot 100 = 90\%$$

7.2 On-road Tests

After a trained network has demonstrated good performance in the simulator, the network is loaded on the DRIVETM PX in our test car and taken out for a road test. For these tests we measure performance as the fraction of time during which the car performs autonomous steering. This time excludes lane changes and turns from one road to another. For a typical drive in Monmouth County NJ from our office in Holmdel to Atlantic Highlands, we are autonomous approximately 98% of the time. We also drove 10 miles on the Garden State Parkway (a multi-lane divided highway with on and off ramps) with zero intercepts.

A video of our test car driving in diverse conditions can be seen in [9].

7.3 Visualization of Internal CNN State

Figures 7 and 8 show the activations of the first two feature map layers for two different example inputs, an unpaved road and a forest. In case of the unpaved road, the feature map activations clearly show the outline of the road while in case of the forest the feature maps contain mostly noise, i. e., the CNN finds no useful information in this image.

This demonstrates that the CNN learned to detect useful road features on its own, i. e., with only the human steering angle as training signal. We never explicitly trained it to detect the outlines of roads, for example.

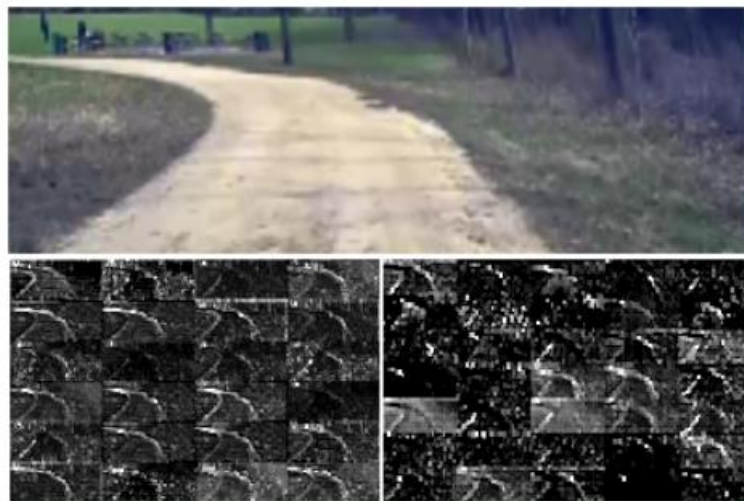


Figure 7: How the CNN “sees” an unpaved road. Top: subset of the camera image sent to the CNN. Bottom left: Activation of the first layer feature maps. Bottom right: Activation of the second layer feature maps. This demonstrates that the CNN learned to detect useful road features on its own, i. e., with only the human steering angle as training signal. We never explicitly trained it to detect the outlines of roads.

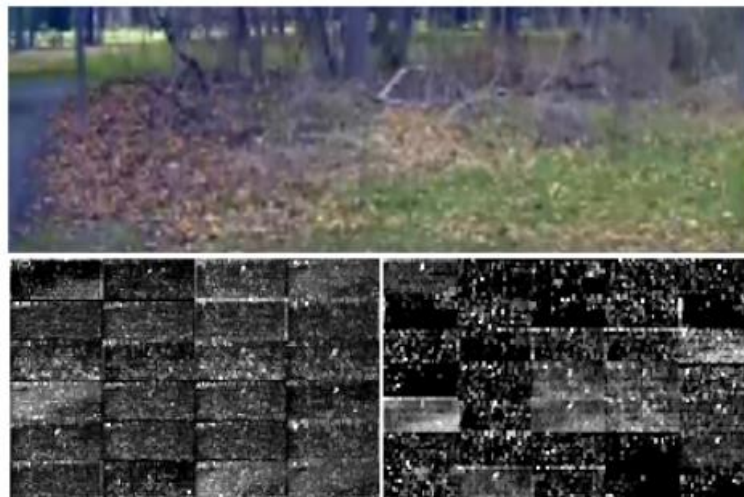


Figure 8: Example image with no road. The activations of the first two feature maps appear to contain mostly noise, i. e., the CNN doesn't recognize any useful features in this image.

8 Conclusions

We have empirically demonstrated that CNNs are able to learn the entire task of lane and road following without manual decomposition into road or lane marking detection, semantic abstraction, path planning, and control. A small amount of training data from less than a hundred hours of driving was sufficient to train the car to operate in diverse conditions, on highways, local and residential roads in sunny, cloudy, and rainy conditions. The CNN is able to learn meaningful road features from a very sparse training signal (steering alone).

The system learns for example to detect the outline of a road without the need of explicit labels during training.

More work is needed to improve the robustness of the network, to find methods to verify the robustness, and to improve visualization of the network-internal processing steps.

References

- [1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Winter 1989. URL: <http://yann.lecun.org/exdb/publis/pdf/lecun-89e.pdf>.
- [2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL: <http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf>.
- [3] L. D. Jackel, D. Sharman, Stenard C. E., Strom B. L., and D Zuckert. Optical character recognition for self-service banking. *AT&T Technical Journal*, 74(1):16–24, 1995.
- [4] Large scale visual recognition challenge (ILSVRC). URL: <http://www.image-net.org/challenges/LSVRC/>.
- [5] Net-Scale Technologies, Inc. Autonomous off-road vehicle control using end-to-end learning, July 2004. Final technical report. URL: <http://net-scale.com/doc/net-scale-dave-report.pdf>.
- [6] Dean A. Pomerleau. ALVINN, an autonomous land vehicle in a neural network. Technical report, Carnegie Mellon University, 1989. URL: <http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci>.
- [7] Wikipedia.org. DARPA LAGR program. http://en.wikipedia.org/wiki/DARPA_LAGR_Program.
- [8] Danwei Wang and Feng Qi. Trajectory planning for a four-wheel-steering vehicle. In *Proceedings of the 2001 IEEE International Conference on Robotics & Automation*, May 21–26 2001. URL: <http://www.ntu.edu.sg/home/edwwang/confpapers/wdwicar01.pdf>.
- [9] DAVE 2 driving a lincoln. URL: <https://drive.google.com/open?id=0B9raQzOpizn1tKrIa241znBEcjQ>.