UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Automatización de la conducción de tranvías

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Emilio Carrión Peñalba

Tutor: Carlos David Martínez Hinarejos

Curso 2018-2019

Resum

En aquest treball hem plantejat un sistema per a l'automatització d'una xarxa de tramvies. La prova de concepte compta amb una simulació feta amb el motor gràfic Unity que consta d'un tramvia circulant en un circuit tancat. El tramvia té simulat en el seu frontal un sensor Sick 2D LIDAR amb una obertura de 190º, que registra deu vegades per segon una lectura de 190 punts de col·lisió. Amb aquestes dades hem creat un tensor de 190 elements compostos per la distància relativa al vehicle i l'angle del punt d'incidència respecte al circuit del tramvia. A cadascuna d'aquestes lectures li hem assignat una classe, 0 o 1, sent 0 parar i 1 accelerar. Hem recopilat mostres de forma orgànica, simulant ser un conductor humà i recorrent el circuit mentre es van registrant lectures i asignant-li la classe segons estiguem accelerant o frenant. Un cop hem recopilat una quantitat significant de mostres hem entrenat una xarxa profunda utilitzant la implementació en Python per part de TensorFlow de Keras, una API d'alt nivell per entrenar models d'aprenentatge profund. Finalment, amb el model entrenat hem aconseguit predir si hi ha una possible col·lisió i la seva probabilitat i actuar en conseqüència.

Paraules clau: Conducció automàtica, Deep learning, LIDAR, Tensorflow, Unity

Resumen

En este trabajo hemos planteado un sistema para la automatización de una red de tranvías. La prueba de concepto cuenta con una simulación hecha con el motor gráfico Unity que consta de un tranvía circulando en un circuito cerrado. El tranvía tiene simulado en su frontal un sensor Sick 2D LIDAR con una apertura de 190º, que registra diez veces por segundo una lectura de 190 puntos de colisión. Con estos datos hemos creado un tensor de 190 elementos compuestos por la distancia relativa al vehículo y el ángulo del punto de incidencia respecto al circuito del tranvía. A cada una de estas lecturas le hemos asignado una clase, 0 ó 1, siendo 0 parar y 1 acelerar. Hemos recopilado muestras de forma orgánica, simulando ser un conductor humano y recorriendo el circuito mientras se van registrando lecturas y asignándole la clase según estemos acelerando o frenando. Una vez hemos recopilado una cantidad significtiva de muestras hemos entrenado una red neuronal profunda utilizando la implementación en Python por parte de TensorFlow de Keras, una API de alto nivel para entrenar modelos de aprendizaje profundo. Finalmente, con el modelo entrenado hemos conseguido predecir si hay una posible colisión y su probabilidad y actuar en consecuencia.

Palabras clave: Conducción automática, Deep learning, LIDAR, Tensorflow, Unity

Abstract

In this work we have proposed a system for the automation of a tram network. The proof of concept has a simulation made with the Unity graphic engine that consists of a tram traveling in a closed circuit. The streetcar has a simulated Sick 2D LIDAR sensor at the front with an opening of 190º, which records a reading of 190 collision points ten times per second. With this data we have created a tensor of 190 elements composed of the relative distance to the vehicle and the angle of the point of incidence with respect to the tram circuit. To each of these readings we have assigned a class, 0 or 1, being 0 stop and 1 accelerate. We have collected samples organically, pretending to be a human driver and traveling the circuit while recording readings and assigning the class as we accelerate or slow down. Once we have collected a significant amount of samples we have trained a deep neural network using the implementation in Python by Keras TensorFlow, a high

level API to train deep learning models. Finally, with the trained model we have been able to predict if there is a possible collision and its probability and act accordingly.

Key words: Automatic driving, Deep learning, LIDAR, Tensorflow, Unity

Índice general

Apéndice

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

En la época en la que vivimos, la informática y la automatización de procesos industriales están cambiando el mundo. Estamos viviendo una cuarta revolución industrial en la que la robótica y los sistemas inteligentes están avanzando el progreso tecnológico a pasos agigantados. Los sistemas capaces de predecir y replicar comportamientos humanos están cambiando las reglas del juego. Hoy en día, las posibilidades que nos ofrece la inteligencia artificial aplicada a procesos cotidianos son ilimitadas. Estas posibilidades incluyen desde sugerir canciones teniendo en cuenta tus gustos personales hasta la conducción autónoma de alto nivel.

1.1 Motivación

Durante los casi 4 años de duración de este grado nos hemos enfrentado innumerables veces a pausas en el servicio de tranvías que nos acerca a la universidad. El tranvía es un medio de transporte que sigue un recorrido cerrado con la única problemática general de enfrentarse a obstáculos inesperados. Con las posibilidades del aprendizaje automático en mente y la oportunidad de mejorar este sistema, nació la idea para este trabajo: plantear la automatización total o parcial del sistema de tranvías de Valencia para agilizar el transporte de estudiantes y trabajadores a sus respectivos puestos. Este trabajo también viene influenciado por el futuro incierto que nos espera. Según datos de la ONU, para 2030 se prevé un crecimiento del número de habitantes en núcleos urbanos a 5000 millones. Es decir, un aumento del 66 % sobre el estado actual, presentando una situación insostenible que nos ofrece retos a los que nos hemos de enfrentar para poder asegurar un futuro y una calidad de vida mejor. Si no aseguramos medios de transporte sostenibles y escalables que permitan a personas de todos los estratos sociales moverse libremente por la ciudad que habitan, nos hallaremos ante un problema clave que impedirá la interacción e incluso el trabajo de millones de personas por no poder llegar a tiempo al lugar que necesitan alcanzar.

1.2 Objetivos

Con estas ideas, los objetivos propuestos para este trabajo son:

- Analizar las posibles soluciones que puedan abordar la conducción automática de un sistema de tranvías.
- Crear un entorno virtual para emular y obtener datos precisos de un sistema de tranvías funcional.

- Analizar y obtener los datos que nos pueda proporcionar un tranvía para aplicar técnicas de aprendizaje automático para definir su conducción.
- Explorar las diferentes técnicas disponibles para resolver el problema de automatizar la conducción del tranvía.
- Implementar la aproximación elegida y hacer un análisis de los resultados empíricos obtenidos.

1.3 Impacto esperado

Con este trabajo se pretende mostrar un entorno controlado en el que se demuestre la viabilidad de un sistema de tranvías sin conductor en un núcleo urbano. La hipotética implementación de las soluciones a las problemáticas derivadas de la automatización de la red permitirían reducir el personal necesario para controlar los tranvías, disminuir el tiempo de demora (aumentando así la eficiencia y fluidez del sistema) e incluso disminuir el número de accidentes derivados de errores humanos. Este trabajo está estrechamente relacionado con el objetivo de desarrollo de la ONU de ciudades y comunidades sostenibles, en concreto con la meta 11.2

11.2 De aquí a 2030, proporcionar acceso a sistemas de transporte seguros, asequibles, accesibles y sostenibles para todos y mejorar la seguridad vial, en particular mediante la ampliación del transporte público, prestando especial atención a las necesidades de las personas en situación de vulnerabilidad, las mujeres, los niños, las personas con discapacidad y las personas de edad.

1.4 Metodología

Para realizar este proyecto, la primera tarea ha sido investigar cómo empresas e instituciones de todo el mundo están implementando actualmente sus sistemas de conducción autónoma. Hemos analizado qué herramientas y componentes están al nuestro alcance para poder implementar nuestro propio sistema y qué técnicas son las más adecuadas para enfrentarse al problema.

Una vez conocido todo lo que tenemos disponible para empezar a trabajar, nos hemos planteado una serie de soluciones alcanzables y hemos analizado sus diferentes problemáticas.

Finalmente, con una solución elegida, hemos continuado con el desarrollo que se detalla en este trabajo.

1.5 Estructura de la memoria

En este trabajo hemos seguido una aproximación clásica en su estructuración.

Primero hemos analizado el problema, su contexto en la época actual y hemos explorado las posibles soluciones que puedan abordarlo (cap. 2).

Seguidamente, hemos planteado el diseño de nuestra solución elegida, hemos explicado cómo hemos estructurado el sistema, sus partes y sus funciones. De forma general, sin entrar en mucho detalle, hemos comentado como funcionan cada uno de los componentes y cómo interactuan entre sí (cap. 3).

A continuación, hemos detallado las tecnologías usadas para el desarrollo del proyecto, como funciona y su base teórica (cap. 4).

Como nudo del trabajo, hemos explicado el desarrollo de la solución propuesta, cómo hemos abordado los diferentes aspectos técnicos de la misma y cómo hemos desarrollado cada uno de los componentes del sistema (cap. 5).

Finalmente, como desenlace, hemos comentado los resultados obtenidos, los hemos analizado y hemos sacado conclusiones. En este último capítulo también hemos hablado de trabajos futuros y de la relación de este proyecto con el grado (cap. 6).

Como anexo sumplementario a este trabajo también hemos añadido una breve historia sobre las redes neuronales y sus pioneros, con referencias a los trabajos clave que nos han traído hasta el punto en el que nos encontramos actualmente (anexo A).

CAPÍTULO 2
Problema

2.1 Análisis del problema

La conducción de tranvías no cuenta con una gran problemática. Circulan por un circuito cerrado con paradas estáticas y programadas. El único factor de riesgo a tomar en cuenta es los posibles obstáculos que puede encontrarse en el camino, como peatones u otros vehículos.

Este problema parece ser trivial a simple vista. Se puede pensar que con un sensor de distancia podemos hacer que el tranvía frene si tiene delante un obstáculo. Sin embargo, esta situación solo se presenta en trayectos rectos. Cuando añadimos al escenario curvas, la distancia frontal al tranvía se vuelve arbitrariamente no decisiva. Podemos tener un obstáculo delante nuestra a escasos metros, pero al ser una curva y nuestro próximo movimiento sea girar, puede que ese obstáculo no esté en nuestro camino.

2.2 Estado del arte

En la fecha de publicación de este trabajo, nos encontramos en un momento de inflexión en lo que se refiere a conducción autónoma. Una gran cantidad de empresas están trabajando sin descanso para llegar a ser los primeros en ofrecer, de manera comercial, un vehículo con autonomía de nivel 5 (en una escala usualmente definida entre 0 y 5). Esto implica coches los cuales pueden prescindir de cualquier interfaz de conducción, ya que están diseñados para manejarse solos en cualquier situación, por compleja que sea.

Aunque es verdad que la conducción autónoma de coches no es directamente extrapolable a la de tranvías, la situación actual nos ofrece un gran transfondo y un número más o menos extenso de referencias que explorar (aunque muchas empresas no publican sus resultados ya que son confidenciales).

Un buen punto de partida es analizar la ChauffeurNet [1], la solución presentada por Waymo, empresa subsidiaria de Google que se encarga de su departamento de conducción autónoma. La ChauffeurNet es una red neuronal bastante compleja, ya que tiene una gran cantidad de parámetros a contemplar; es comprensible decir que la automatización de la conducción de un coche en un entorno urbano es extremadamente compleja. Sin embargo, puede darnos algunas ideas para empezar el diseño de nuestro sistema. Según el artículo que publicaron, los investigadores de Waymo descubrieron que imitar un comportamiento correcto no es suficiente para poder enfrentarse a cualquier tipo de situación. Su solución fue enfrentar el sistema a perturbaciones en la conducción de un conductor experto para desviar las muestras a situaciones más interesantes, como colisiones o desviaciones en el rumbo. Esto les ofreció una mayor gama de situaciones para

entrenar a la red neuronal y así obtener mejores resultados. También usan una función de pérdida que penaliza eventos indeseados y premia el progreso, idea que hemos aprovechado en el desarrollo de este trabajo.

Por otro lado, el campo de la automatización de tranvías no está del todo inexplorado. La empresa Siemens mostró en diciembre de 2018 el primer tranvía autónomo del mundo [2]. El vehículo depende de sensores LIDAR para analizar su entorno y actuar en consecuencia. No dejan claro qué sistema usan para tomar las decisiones, pero podrían estar usando también redes neuronales, como hemos optado en nuestra solución.

2.3 Posibles soluciones

Hay muchas formas de predecir una posible colisión: mediante sensores de distancia, predicción de colisiones, visión por computador, escáneres 3D, etc.

A continuación exploraremos algunas de las opciones que vamos a plantear con sus puntos fuertes y problemáticas.

2.3.1. Frenado por distancia

Una de las formas más sencillas de conseguir que el tranvía evite choques es implantar en la parte delantera un sensor de distancia. El sensor registra cada pocos milisegundos la distancia entre el vehículo y el objeto más cercano. En caso de detectar una entidad inicia los sistemas de frenado, evitando así la colisión.

Problemática

Es muy sencillo detectar obstáculos en un trayecto recto, pero falla en curvas donde la distancia frontal no es decisiva en un posible accidente.

En una línea recta podemos tener un coche delante que, en efecto, obstaculice nuestro camino, pero si el coche está aparcado, por ejemplo, en una curva donde el tranvía va a realizar un desvío y no entra en conflicto con la trayectoria final del vehículo estaríamos ante un falso positivo.

2.3.2. Detección de objectos

Otra forma viable de frenar en caso de tener un obstáculo en el camino es utilizar visión por computador para identificar entidades como peatones o coches y frenar si están en una posición peligrosa. Se puede instalar una cámara en el frontal del tranvía que envíe continuamente imágenes a un *script* que use bibliotecas de visión, como OpenCV [3], las cuales realicen una detección de objetos sobre las imágenes. En caso de etiquetar un objeto como peligroso se debe frenar como medida cautelar.

Problemática

Esta aproximación es más compleja e imprecisa, además de funcionar peor sin luz o en un entorno con lluvia u otros fenómenos ambientales.

Podemos usar un modelo ya entrenado de los muchos que hay disponibles en la red. Esto nos permitiría detectar objetos reconocibles como vehículos y peatones. El problema de esta solución se basa en que si te enfrentas a un objeto que el modelo no ha visto en su

fase de entrenamiento no sabrá que es un potencial obstáculo, llevándonos a la colisión. También tenemos el problema de la lluvia y de la conducción nocturna.

Empresas como Tesla se basan completamente en visión por computador para sus sistemas de conducción autónoma, pero la gran mayoría de sus competidores, como Waymo, han descartado esta aproximación [4].

2.3.3. Predicción de colisiones

Una tercera manera de conseguir evitar accidentes es predecir posibles colisiones. Para ello, se puede implantar un sensor en la parte delantera del tranvía que recoja información del entorno que rodea al vehículo y que calcule la probabilidad de colisión en el futuro inmediato. De esta forma, el tranvía puede frenar o disminuir la velocidad si cree que existe la posibilidad de choque y así impedir el accidente.

Gran parte de las empresas que están desarrollando sistemas de conducción autónoma (principalmente para coches) se basan en esta aproximación, usando escáneres 3D que mapean el entorno y usando un sistema de decisión (bastante más complejo que el desarrollado para este trabajo) para definir las acciones del vehículo.

Problemática

Implementar un sistema de predicción fiable que detecte posibles colisiones poco antes de que se produzcan.

2.4 Solución propuesta

En este caso, hemos planteado un sistema de predicción de colisiones que, teniendo en cuenta dos lecturas del entorno en el que se encuentra el tranvía, separadas por un tiempo determinado muy corto, prediga la probabilidad de choque.

Hemos elegido esta solución ya que implementar un sistema de frenado por distancia sería trivial y no serviría para un entorno realista, ya que solo funcionaría en trayectos rectos. Tampoco hemos contemplado la posible solución de implementar un sistema de detección de objetos, que tampoco serviría para un entorno realista ya que no sería funcional en situaciones sin luz o con lluvia.

Para conseguir implementar el motor de predicción de colisiones hemos utilizado las lecturas de un sensor LIDAR situado en el frontal del tranvía y un sistema entrenado para calcular la probabilidad de choque a partir de dos lecturas consecutivas de dicho sensor.

CAPÍTULO 3
Diseño de la solución

3.1 Arquitectura

Como hemos comentado, hemos implementado un sistema de predicción de colisiones que nos permite anticipar accidentes. Para ello hemos necesitado un sensor en el frontal del vehículo capaz de muestrear el entorno y entrenar un sistema que calcule de manera fiable la probabilidad de impacto. Construir y probar un sistema así es complejo y costoso; por ello hemos utilizado un simulador en su lugar.

Hemos hecho uso del motor gráfico Unity para simular un circuito y un tranvía funcional. También hemos implementado un sensor LIDAR que es capaz de registrar su entorno de forma similar a como lo hace en la realidad y lo hemos situado en la parte delantera del tranvía. Con este simulador montado ya tenemos el entorno de pruebas perfecto para probar y entrenar nuestro sistema de predición de colisiones.

Para el motor de predicción en este caso hemos utilizado el lenguaje de programación Python y la librería de *Deep Learning* Tensorflow [18], con la que hemos entrenado un modelo capaz de calcular la probabilidad buscada.

Podemos ver un esquema de la arquitectura en la figura 3.1.

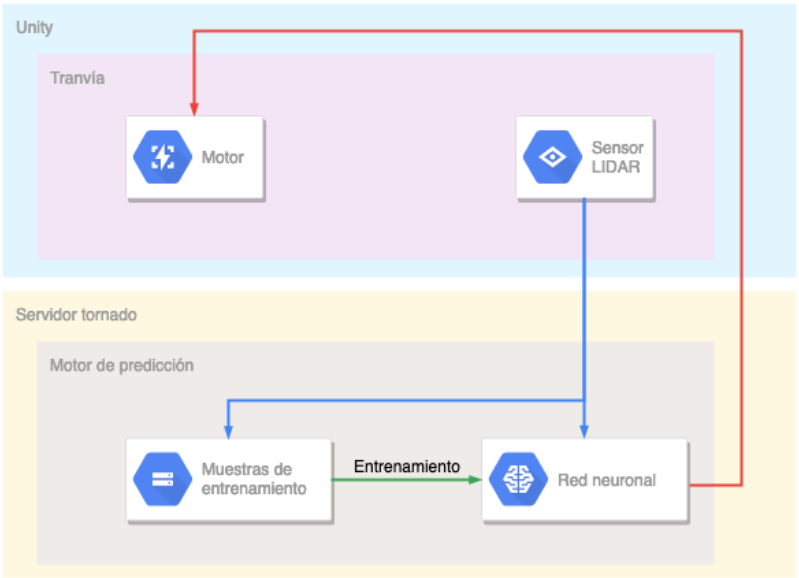


Figura 3.1: Arquitectura

3.2 Simulador

Para poder desarrollar el sistema de conducción autónoma necesitamos un entorno donde poder hacer pruebas y entrenarlo. El uso de un tranvía real o de un modelo a escala es costoso y poco útil. El desarrollo de un simulador donde ejecutar diferentes entornos y situaciones resulta más versátil a la hora de conseguir muestras con las que entrenar el sistema.

3.2.1. Entorno

Para la creación de este simulador hemos utilizado el motor gráfico Unity, que es un motor que permite crear entornos 3D, y *scripts* en C# con los que interactuar con las entidades que lo forman.

3.2.2. Circuito

Para el circuito de pruebas hemos utilizado varios modelos de uso libre para crear una ciudad en 3D y la herramienta BansheeGz [6], con la que podemos crear un camino fijo en bucle por el que dirigir nuestro tranvía. Con la herramienta de creación de rutas podemos obtener la posición del tranvía en todo momento y tener una referencia de su posición relativa en el circuito. Esto nos permite calcular los ángulos entre el tranvía y el camino que va a seguir. Esta información nos ayudará a entrenar el sistema de detección de colisiones.

3.2.3. Tranvía

Para el tranvía hemos usado un modelo de licencia libre [5] al que hemos modificado las texturas para darle la entidad de la Etsinf.

Controles

Hemos dotado al tranvía de unos controles básicos que podemos definir como una variable binaria: o acelerar o frenar. De esa forma simplificamos el sistema de toma de decisiones a un clasificador de dos clases. Para poder recoger muestras de entrenamiento y probar el circuito también hemos añadido controles manuales, que simulan los controles básicos de aceleración de un tranvía. Hay un control que acelera el tranvía y que, al soltarlo, desacelera el tranvía con frenada.

Sensor LIDAR

Para la interacción del tranvía con el entorno hemos simulado el comportamiento de un sensor LIDAR siguiendo las especificaciones de uno de los mayores fabricantes a nivel europeo, para así acercarnos lo máximo posible a una situación real.

CAPÍTULO 4
Tecnologías usadas

4.1 Unity

Unity es uno de los motores gráficos más populares actualmente. Tiene una gran variedad de herramientas que dan una gran flexibilidad para desarrollar entornos 3D y poder crear cualquier tipo de juego que se imagine.

Tiene una API de *scripting* escrita en C# y una integración con Visual Studio para acelerar el desarrollo. También ofrece poder implementar los *scripts* en JavaScript para quien se sienta más cómodo con ese lenguaje.

Además, cuenta con una tienda de *assets* [7] donde la comunidad aporta una gran variedad de material con el que poder trabajar directamente en el motor, y gran parte de ello está disponible de forma gratuíta.

Cabe destacar que Unity se ha utilizado en una gran variedad de proyectos científicos que aprovechan las posibilidades que ofrece [8] [9] [10].

4.2 LIDAR

Para el desarrollo de este trabajo vamos utilizar la tecnología LIDAR. Los sensores LIDAR permiten calcular la distancia entre dos puntos iluminando una superficie con un láser pulsado y calculando la distancia de los pulsos reflejados con un receptor. Las diferencias entre los tiempos de retorno y las longitudes de onda se pueden utilizar para hacer representaciones en tres dimensiones del objeto escaneado.

La tecnología LIDAR tiene un gran rango de aplicaciones, entre las que se encuentra la conducción de vehículos autónomos.

Una implementación de estos sensores son los sensores Flash LIDAR, que permiten crear una imagen 3D del entorno gracias a la habilidad de la cámara de emitir un gran flash y calcular las relaciones espaciales y las dimensiones del área de interés a partir de la energía devuelta. Esto nos permite un mapeo del entorno más preciso gracias a que las imágenes procesadas no han de ser superpuestas y, por lo tanto, el sistema no es sensible al movimiento, consiguiendo así menor distorsión.

El principio básico de la tecnología LIDAR es muy simple. El cálculo de los puntos de colisión se obtiene del cálculo de la distancia desde la que vuelve el fotón de luz rebotado y se simplifica a:

$$D = ct/2$$

Donde c es igual a la velocidad de la luz y t es el tiempo de retorno.

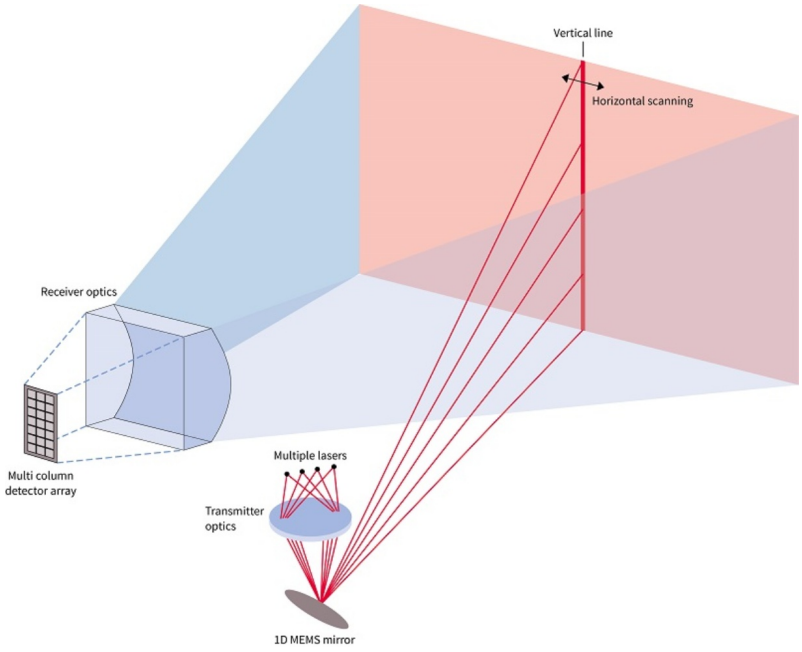


Figura 4.1: Sensor LIDAR

El sensor, detallado en la figura 4.1, envía pulsos láser a alta velocidad hacia una superficie. Algunos de ellos emiten hasta 150.000 pulsos por segundo. Un receptor en el sensor entonces calcula el tiempo que tarda cada pulso en rebotar y ser devuelto. La luz se mueve a una velocidad constante conocida, así que el sensor puede calcular la distancia entre el punto de colisión y sí mismo con gran precisión. Repitiendo rápidamente este proceso, el sensor es capaz de construir un mapa en tres dimensiones de la superficie que está midiendo. En sensores aéreos también se necesita registrar otros datos para asegurar la precisión. Necesitamos conocer la altura, localización y orientación del instrumento para determinar la posición del láser en el momento de la emisión y de la recepción. Esta información es crucial para asegurar la fiabilidad del sensor. Con sensores terrestres [11], una señal simple GPS puede ser utilizada para calcular cada localización en la que el láser emite y recibe los pulsos.

Generalmente se emplean dos métodos de detección: detección directa de energía (también conocida como incoherente) y detección coherente. Los sistemas coherentes son mejores para medidas Doppler [12] o sensibles a las fases, y generalmente usan detección óptica mediante heterodino [13]. Esto permite que operen con mucha menos energía pero con el coste extra de un sistema de transmisión más complejo. En ambos tipos de sensor hay dos modelos de pulso principales: micropulsos [14] y sistemas de alta energía [15]. Los sistemas de micropulsos se han desarrollado para aprovechar ordenadores más potentes y con mayor potencia de cálculo. Estos láseres usan menos energía y son seguros a los ojos, permitiendo ser usados con menores medidas de seguridad. Los sistemas de alta energía son comúnmente utilizados para estudios atmosféricos, donde son usualmente empleados para medir una serie de parámetros atmosféricos como la altura, las capas, la densidad de las nubes, etc.

4.3 Redes neuronales

Antes de entrar en detalles sobre cómo funcionan las redes neuronales, se puede consultar un poco de su historia en el anexo A.

Las redes neuronales de las que hablamos en el campo de la computación son estrictamente redes neuronales artificiales, ya que son un modelo matemático sobre lo que sabemos, o creemos saber, de la estructura y funcionamiento del cerebro humano.

Las redes neuronales son un modelo compuesto por un número de nodos sencillos, inteconectados entre sí, llamados neuronas. Estas neuronas se distribuyen en capas, las cuales procesan información de entrada y generan respuestas a partir de una serie de funciones de activación.

El uso de las redes neuronales es muy útil en el reconocimiento de patrones, especialmente en patrones demasiado complejos para ser analizados por un ser humano o para ser configurados en un sistema tradicional.

Las capas de una red se componen por la capa de entrada y de salida y las capas ocultas. A estas últimas se les llama ocultas por el mero hecho de que no interactuan con el exterior del modelo, y es donde gran parte del cálculo se lleva a cabo.

Entre estas capas tenemos las interconexiones que unen las neuronas, las cuales están ponderadas mediante unos pesos y unos *bias*. Cuando se recibe un dato de entrada, la neurona calcula una suma ponderada añadiendo el *bias*, y según el resultado y la función de activación de esa capa, se decide si la neurona debería activarse o no. En ese momento la neurona transmite la información procesada a la siguiente capa a través de las interconexiones ponderadas. Al final de este proceso de activación en las diferentes capas, la última de ellas se conecta a la capa de salida, donde tenemos una neurona por cada posible dato de salida.

Con la estructura de las redes neuronales un poco más clara podemos explicar cómo funciona. Para ello primero analizaremos qué tipos de neuronas podemos usar y qué funciones tienen.

El primer tipo y más clásico es el perceptrón [29]. Cada vez se usa menos pero fue uno de los primeros éxitos en el campo del aprendizaje automático y nos lleva en buen camino para entender cómo funcionan las neuronas modernas.

El perceptrón utiliza una función, mostrada en la figura 4.2, para construir un clasificador binario transformando un vector de datos de entrada a una salida binaria. También puede ser usado para entrenamiento supervisado usando muestras ya etiquetadas.

Su funcionamiento es muy sencillo: multiplica los datos de entrada por una serie de pesos que indican cómo es de importante cada dato, hace la suma ponderada de los mismos más un *bias* y aplica un umbral para obtener un resultado binario (0 si está por debajo del umbral, 1 si está por encima).

$$f(x_1, x_2, ..., x_m) = \begin{cases} 0 & \text{if } b + \sum w_i \cdot x_i < 0 \\ 1 & \text{if } b + \sum w_i \cdot x_i \geq 0 \end{cases}$$

Perceptron Formula

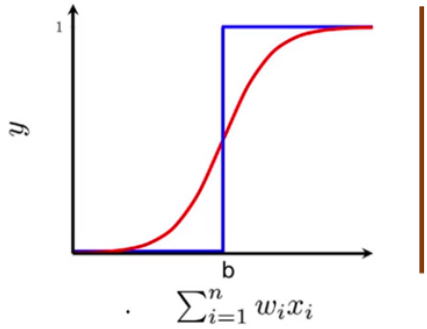
Figura 4.2: Fórmula del perceptrón

Una de las grandes características de este algoritmo es que, modificando los valores de los pesos y de los *bias*, podemos obtener diferentes clasificadores. Podemos aumentar

los pesos de aquellas entradas que pensemos que influyen más en la salida deseada y así aumentar su precisión. También podemos modificar el valor del *bias*, el cual define la dificultad de obtener un 1 en la salida. Podemos observar que un valor grande en el *bias* hará muy sencillo obtener un 1 en la salida. Al contrario, un valor grande pero en negativo hará muy dificil obtener el 1.

Como hemos podido observar, un perceptrón puede analizar unos datos de entrada y hacer decisiones a partir de ellos siguiendo unos parámetros preestablecidos. Tambien podemos crear redes más complejas con más de una capa de perceptrones donde cada capa acepte como entrada los datos de salida de la anterior, y así realizar decisiones más complejas.

Sin embargo, los perceptrones tienen algunos fallos. Uno de ellos es que con pequeñas variaciones en los pesos o los *bias*, aunque sea en solo una de las neuronas, puede cambiar drásticamente el resultado. Lo que necesitamos es poder ir introduciendo variaciones poco a poco en la red y así modificar su comportamiento gradualmente. Aquí es donde entran en escena las neuronas modernas, las cuales usan la función de activación sigmoide (últimamente reemplazada por otros tipos como Tanh y ReLu [17]), mostrada en la figura 4.3. La principal diferencia entre un sigmoide y el perceptrón es que la entrada y la salida dejan de ser binarias y pueden comprender un valor entre 0 y 1. La salida se obtiene aplicando la función sigmoide a las entradas ponderándolas con los pesos y los *bias*.



$$y = \frac{1}{1+e^{-(w^T x+b)}}$$

Figura 4.3: Fórmula sigmoide

Lo interesante de esta función es que con cambios pequeños en los pesos y los *bias* genera cambios pequeños en la salida, que es el comportamiento que no nos ofrecía el perceptrón. Sin embargo, también cuenta con ciertos problemas que nos han hecho sustituirla por otra función de activación, como veremos un poco más adelante en este capítulo.

La característica importante de las redes neuronales es su capacidad de aprender a predecir la salida esperada. Esto se reduce a ajustar los pesos que definen las conexiones entre neuronas y hacerlas más precisas, con el objetivo de que la salida computada se aproxime al valor real para todos los datos de entrenamiento.

Para entrenar la red neuronal tenemos que saber cómo de equivocada está la predicción obtenida. Para ello necesitamos conocer el error. El error podemos calcularlo definiendo una función de coste que obtendrá la diferencia entre lo que esperamos y la salida obtenida. En el campo de las redes neuronales la función de coste más extendida es la función de coste cuadrática o error cuadrático medio, la cual podemos ver en la figura 4.4.

Este coste es más usado que el error lineal, ya que en las redes neuronales pequeños cambios en los pesos y en los *bias* no producen ningún cambio en el número de salidas

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Figura 4.4: Error cuadrático medio

correctas. Por ello se usa la función cuadrática, donde grandes diferencias tienen más efecto relativo en el resultado de la función de coste que las pequeñas.

El objetivo de entrenar la red es que el valor de la función de coste sea lo más pequeño posible. Para ello necesitamos encontrar los valores para los pesos y los *bias* que minimicen el coste lo máximo posible. La herramienta más utilizada para ello es el algoritmo de descenso por gradiente.

Lo que buscamos obtener son los valores que hacen la función de coste mínima. Sabemos que una función tiene un valor mínimo y un valor máximo y calcularlos es sencillo, derivándola y igualando a 0. Sin embargo esta técnica deja de ser trivial para funciones con más de dos variables. En el caso de la función cuadrática esto no es posible y necesitamos otra aproximación.

El algoritmo de descenso por gradiente se basa, de forma muy simplificada, en elegir un punto al azar en la función e ir variando su valor x hacia valores de y más pequeños, como podemos ver en la figura 4.5.

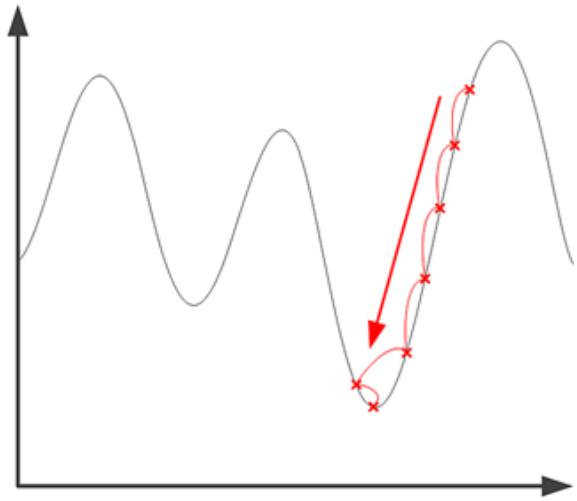


Figura 4.5: Descenso por gradiente

Formalizando el algoritmo, la variación en la dirección x es Δx y la de y es Δy y calculamos el cambio del valor en nuestra función como ΔC.

Como el ratio de variación en la dirección es la derivada de la función, podemos expresar ΔC como:

$$\Delta C = \frac{\partial C}{\partial x}\Delta x + \frac{\partial C}{\partial y}\Delta y$$

Ahora usamos la definición del cálculo del gradiente de una función:

$$\nabla C = \left(\frac{\partial C}{\partial x}, \frac{\partial C}{\partial y}\right)$$

Y reescribiremos el cálculo de la variación en nuestra función (ΔC) como:

$$\Delta C = \nabla C \Delta X$$

La cantidad de variación que aplicamos al valor x es el ratio de aprendizaje y define lo rápido que aprendemos. Un valor muy pequeño puede hacer que necesitemos muchas iteraciones para llegar al mínimo; un cambio muy grande puede hacer que nos lo pasemos. Una vez seleccionado un ratio de aprendizaje adecuado podemos seguir con las iteraciones hasta obtener un error aceptable.

Sin embargo, a veces calcular el gradiente puede ser una tarea compleja. Hay maneras de reducir el coste de cómputo necesario, y una de ellas es usar una versión del algoritmo llamado descenso por gradiente estocástico. Este algoritmo funciona estimando el valor del gradiente ∇C, computándolo para un subconjunto aleatorio de las entradas de entrenamiento. Eso nos ofrece una buena estimación del verdadero gradiente y nos permite acelerar el proceso de aprendizaje.

Para el cálculo del gradiente usamos un algoritmo llamado *backpropagation*. Este algoritmo computa las derivadas parciales de la función de coste respecto a un peso y un *bias*. Esto significa que calcula el error de los vectores empezando por la última capa y lo propaga a capas anteriores para actualizar sus pesos y sus *bias*. Podemos observar las fórmulas que utiliza en la figura 4.6.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Figura 4.6: *Backpropagation*

Siendo δ^L el error en la capa de salida, δ^l el error en la capa l y δ^l_j el error de la neurona j en la capa l. La teoría detrás de cada fórmula no entra en el alcance de este proyecto pero puede consultarse en la referencia bibliográfica [16].

Sin embargo, el algoritmo de *backpropagation* calcula el gradiente de la función de coste para una sola muestra de entrenamiento, por lo que necesitamos combinarlo con un algoritmo de aprendizaje para computar el gradiente para todo el conjunto de entrenamiento.

Como hemos comentado, para usar el descenso por gradiente con el algorítmo *backpropagation* para entrenar redes neuronales necesitamos una función de activación. Como también hemos explicado, el perceptrón tiene sus fallos y la función sigmoide está siendo sustituida por otras más modernas. En este caso hemos hecho uso de la función de activación lineal rectificada o ReL [17]. Una unidad o neurona que usa esta función se la denomina unidad de activación lineal rectificada o ReLU, para hacernos la lectura más amena. A las redes neuronales que implementan esta función usualmente también se les conoce como redes rectificadas.

La adopción de ReLU puede ser considerada uno de los hitos de la revolución del aprendizaje profundo, por una serie de motivos que detallaremos un poco más adelante en este capítulo.

La función de activación lineal rectificada es un cálculo muy sencillo, que devuelve directamente el valor proporcionado como entrada o el valor 0 si la entrada es 0 o menor.

Podemos expresar esta función en forma de código como se muestra en el *listing* 4.1.

```
if input > 0:
  return input
else:
  return 0
```

Listing 4.1: ReLU

También podemos describir esta función matemáticamente usando la función max() sobre el valor de entrada z y el valor 0 como se muestra a continuación:

$$g(z) = \text{máx}\{0, z\}$$

Esto nos ofrece una función lineal para valores mayores que cero, lo que tiene muchas de las propiedades interesantes que una función de activación lineal ofrece al entrenamiento de una red neuronal (fáciles de optimizar en métodos como el descenso por gradiente y generalizar bien en modelos entrenados).

Algunas de las ventajas que ha hecho que ReLU se convierta en la función estándar en la mayoría de las redes neuronales son las que detallamos a continuación:

- Su sencillez computacional, ya que es trivial de implementar, necesitando solo una función max(). Otras funciones como la función sigmoide y la función Tanh hacen uso de un cálculo exponencial.
- Su representación dispersa, ya que puede devolver un 0 absoluto, lo que quiere decir que entradas negativas pueden devolver el valor 0, permitiendo a las capas ocultas de la red contener uno o más ceros. Esto se llama representación dispersa y es una propiedad que puede acelerar el aprendizaje y simplificar el modelo. Funciones como la sigmoide o la Tanh pueden aproximarse al valor 0 pero no devolver un 0 absoluto.
- Parece y se comporta como una función lineal, hecho que, como hemos comentado anteriormente, hace que sea sencilla de optimizar.
- Las redes que la implementan casi evitan el problema del desvanecimiento de los gradientes, ya que los gradientes se mantienen proporcionales a las activaciones de los nodos, hecho que no ocurre con la no linealidad de las funciones sigmoide y Tanh.

Explicada la teoría y conocidos los diferentes tipos de neuronas, ya disponemos de los conocimientos necesarios sobre el funcionamiento básico de las redes neuronales, que es el que se aplica por parte de las bibliotecas usadas en este trabajo. Hay muchas formas de mejorar el rendimiento y la precisión de la red y es un campo en constante desarrollo. Algunas de estas mejoras las veremos en la siguiente sección.

4.4 TensorFlow

El campo de las redes neuronales no es una disciplina sencilla. TensorFlow [18] es una biblioteca de código abierto, creada por el equipo de Google Brain, diseñada para manejar la computación a gran escala necesaria para este tipo de aprendizaje automático.

TensorFlow recopila una gran variedad de modelos y algoritmos de aprendizaje automático o profundo que nos ayudan a implementar nuestras redes neuronales de una forma más sencilla.

Nos ofrece una interfaz creada en Python para hacer su uso más sencillo y amigable, mientras que la ejecución de los cálculos los realiza en C++ para obtener un mayor rendimiento.

Se ha usado TensorFlow en una gran cantidad de áreas de estudio como: reconocimiento de dígitos manuscritos, reconocimiento de imágenes, redes neuronales recurrentes, procesamiento de lenguage natural, etc.

Una de las características más potentes que nos aporta TensorFlow es que tiene implementadas una gran cantidad de herramientas que nos ayudan a mejorar la precisión de nuestras redes neuronales. Vamos a explicar brevemente aquellas que hemos aprovechado en este proyecto.

Una de ellas es la capa de *dropout* [20]. Cuando entrenamos una red neuronal podemos sobreentrenarla accidentalmente. Esto significa que la red será muy precisa para los datos de entrenamiento utilizados pero no generalizará a un mayor número de muestras. Para solucionar este problema podemos usar la técnica de *dropout*, que se basa en desconectar aleatoriamente un número de neuronas de la red. Esto consigue que no se ajuste tanto al entrenamiento y generalice a un número mayor de muestras.

Otra herramienta que nos ofrece TensorFlow son los optimizadores. Uno de ellos es el que hemos comentado en la sección anterior, el descenso por gradiente estocástico. TensorFlow nos ofrece una gran cantidad de ellos. En este trabajo hemos usado el optimizador *Adam* [21]. Este optimizador es una extensión del ya comentado descenso por gradiente estocástico y está siendo utilizado para aplicaciones de visión por computador y lenguaje natural. No vamos a entrar en detalles de cómo funciona pero podemos hacernos una idea sabiendo que combina otras dos extensiones del descenso por gradiente estocástico, el algoritmo de gradiente adaptativo (AdaGrad [22]) que mejora el rendimiento en gradientes dispersos y el algoritmo de propagación del valor cuadrático medio (RMSProp [23]) que mejora el rendimiento en problemas *online* y no estacionarios (con ruido, por ejemplo).

También comentar que TensorFlow nos ofrece la implementación de una gran variedad de funciones de pérdida (similares a las funciones de coste comentadas anteriormente). En este proyecto hemos usado la entropía cruzada categórica dispersa, un nombre largo para un cálculo sencillo que podemos observar en la figura 4.7. El hecho de que sea dispersa es porque no usa todas las clases posibles para el cálculo.

$$-\frac{1}{N}\sum_{i=1}^{N}\log p_{model}\left[y_i \in C_{y_i}\right]$$

Figura 4.7: *Sparse categorical cross-entropy*

4.5 Tornado

Tornado es una biblioteca de *networking* asíncrona. Nos permite crear y ejecutar servidores con los que implementaremos la comunicación entre el simulador desarrollado en C# y el sistema de decisión desarrollado en Python.

Tornado nos ofrece la posibilidad de establecer una comunicación usando el protocolo TCP, el cual nos permite transmitir información mediante el uso de *sockets*. Al poder enviar información en diferentes formatos, como por ejemplo texto plano, somos capaces de intercambiar paquetes de datos entre diferentes aplicaciones que funcionan usando lenguajes y plataformas diferentes.

CAPÍTULO 5
Desarrollo

5.1 Simulador

Para poder hacer las pruebas y entrenar el sistema de detección de colisiones necesitamos tener un entorno realista que nos permita emular el funcionamiento y recorrido de un tranvía y obtener datos sobre sus interacciones con el entorno que lo rodea.

Para conseguir estos requisitos hemos implementado un simulador de conducción de tranvías en el motor gráfico Unity, una herramienta de creación 3D que nos permite crear mapas, entidades y *scripts* que nos ayudan a simular mecánicas y recopilar datos.

El primer objetivo es recrear un circuito urbano por donde circule el vehículo. El circuito se compone de un tramo paralelo a una avenida, curvas con obstáculos frontales y un tramo con varios cruces por los que circulan otros coches y obstáculos potenciales. Podemos observar una vista aerea del circuito en la figura 5.1.

Para el desarrollo de la ciudad hemos utilizado modelos de código libre que se pueden obtener directamente a través de la tienda de recursos del motor gráfico [7]. También hemos añadido vegetación, elementos arquitectónicos decorativos y elementos de iluminación urbana, como farolas. Esto nos ayudará a tener un entorno más completo y realista en el que realizar las pruebas.

Figura 5.1: Circuito

Para el tranvía hemos utilizado otro modelo de dominio público [5], pero en este caso lo hemos modificado para incluir en su estructura el logo de la Etsinf. Para ello simplemente hemos editado una de las texturas que usa, donde hemos añadido del el logo de la escuela. El resultado es visible en las figuras 5.2 y 5.3.

Figura 5.2: Tranvía

Figura 5.3: Logo del tranvía

Con el entorno 3D definido y un circuito viable ya solo necesitamos emular el movimiento del vehículo.

Para ello hemos hecho uso de una herramienta llamada BGCurve [6] creada por la comunidad de Unity para crear caminos y aplicar fórmulas de movimiento a entidades. Con esta herramienta hemos creado un camino compuesto por marcadores que siguen el recorrido definido para el tranvía.

Hemos creado un *script* para el vehículo, mostrado en el *listing* 5.1, que simula la aceleración y desaceleración del vehículo teniendo en cuenta una fricción definida. También lo hemos dotado de unos controles que nos permiten controlarlo desde el teclado y de forma remota, característica que usaremos para automatizar su conducción mediante el sistema de detección de colisiones.

```
1  public class TramController : MonoBehaviour
2  {
3      public Transform StreetCar;
4      private float distance;
5      private float speed = 0;
6      BGCcMath math;
7
8      void Update() {
9          if (SickSensor.run || Input.GetKey(KeyCode.I)) {
10             speed = Math.Min(speed + 5 * Time.deltaTime, 20);
11         }
12         else {
13             speed = Math.Max(speed - 20 * Time.deltaTime, 0);
14         }
15
16         distance = (distance + speed * Time.deltaTime) % math.GetDistance();
17         Vector3 tangent;
18         StreetCar.position = math.CalcPositionAndTangentByDistance(distance,
               out tangent);
```

```
19            StreetCar.rotation = Quaternion.LookRotation(tangent) * Quaternion.
                  Euler(new Vector3(0, 180, 0));
20        }
21    }
```

Listing 5.1: TramController

Con este controlador ya podemos simular la conducción del tranvía por el circuito designado. Para obtener muestras de la conducción del vehículo tambien necesitamos obstáculos estáticos y móviles que interactúen con las físicas de nuestro tranvía y simulen accidentes y colisiones. Para simular otros vehículos hemos creado una entidad llamada *Car* a la cual hemos programado un movimiento muy simple de vaivén para representar cruces transitados de coches. Podemos ver la estrategia que sigue en el *listing* 5.2

```
1  public class CarController : MonoBehaviour {
2      int direction = 1;
3      float speed = 0.1F;
4      float distance = 50;
5
6    void Update () {
7          var motion = (Vector3.forward + new Vector3(90, 0, 0)) * Time.deltaTime
                * direction * speed;
8          var oldPos = transform.position;
9          transform.position += motion;
10         distance -= Vector3.Distance(oldPos, transform.position);
11         if (distance < 0) {
12             direction = -direction;
13             distance = 50;
14         }
15     }
16 }
```

Listing 5.2: CarController

Las entidades *Car* recorren una distancia de cincuenta unidades y al acabar invierten su dirección para hacer el camino inverso. Como su movimiento no va sincronizado con la velocidad del tranvía ni su recorrido obtenemos cierta aleatoriedad cada vez que el tranvía realiza el recorrido, mejorando la diversidad de muestras que podemos registrar.

5.2 Sensor LIDAR

Para la implementación del sensor LIDAR en el simulador hemos investigado los diferentes sensores disponibles comercialmente. Hay muchas configuraciones de sensores disponibles en el mercado actualmente. Hay escáneres multicapa (fig. 5.4) con diferentes rangos de amplitud, frecuencias y tolerancias.

Para este proyecto, y para simplificar el concepto, hemos utilizado un sensor de una sola capa, es decir, un escáner que mapea una línea de puntos a su alrededor indicando las colisiones que encuentra.

De los diferentes modelos disponibles hemos elegido el modelo LMS5xx de la empresa Sick Sensors, ya que nos ofrece las siguientes características:

- Es uno de los sensores LIDAR 2D de gran precisión más pequeños.
- Detección rápida y fiable de objetos bajo cualquier condición ambiental.
- Bajo consumo energético.
- Rangos de detección de hasta 80 metros.

- Compacto y con certificación IP 67 contra elementos externos.

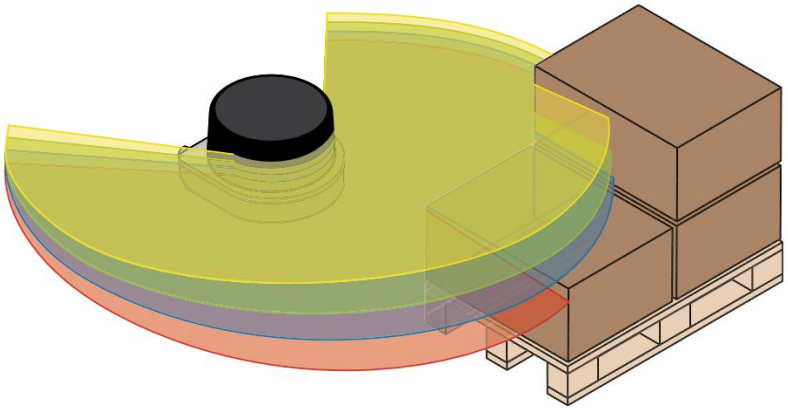


Figura 5.4: Sensor LIDAR multicapa

En concreto, el escáner simulado cuenta con una apertura de detección de 190º y un alcance limitado de 30 metros en el que obtiene la distancia de colisión con un objeto en cada uno de los grados de apertura, consiguiendo así 190 puntos de colisión con los que trabajar.

Para la implementación en el motor Unity hemos utilizado una función de las físicas del motor llamada Raycast, que nos permite emitir un rayo desde un punto de origen en una dirección y devuelve el punto de colisión obtenido. Con ese punto y el origen obtenemos la distancia, que es un resultado similar al devuelto por un sensor LIDAR real. Podemos ver un fragmento del código en el *listing* 5.3.

```
void Check () {
    RaycastHit hit;
    Vector3 origin = transform.position;

    float[,] data = new float[190];

    // Creamos un arco de 190 rayos en el frontal
    for (int i = -95; i < 95; i++) {
        Vector3 vector = Quaternion.Euler(0, i, 0) * transform.forward;

        if (Physics.Raycast(origin, vector, out hit, 30)){
            var distance = Vector3.Distance(hit.point, origin);
            data[90 + i] = distance;
        } else {
            data[90 + i] = -1;
        }
    }

    process(data);
}
```

Listing 5.3: Implementación del sensor

Como hemos comentado en capitulos anteriores, la distancia frontal de colisión con los posibles obstáculos no es suficiente para determinar si hay riesgo de accidente o no, ya que depende de si el obstáculo se encuentra o no en nuestra trayectoria. Un buen ejemplo de este hecho es un coche que permanezca aparcado en una de las curvas del circuito; aunque la distancia frontal pueda indicar que es un posible obstáculo, al no estar el camino que sigue el tranvía no hay peligro de colisión.

Para enseñar al sistema a tener en cuenta estos casos hemos añadido a cada una de las distancias de las muestras de entrenamiento el ángulo relativo del punto de colisión con

el punto más cercano del circuito a éste. En un caso real esta información sería sencilla de obtener, ya que un tranvía realiza un circuito cerrado y podemos saber su posición exacta ya sea por la distancia recorrida, por marcadores en el circuito o por paradas y semáforos.

5.3 Red neuronal

Con el sistema de conducción planteado tenemos un solo control para dirigir el tranvía: podemos acelerar o frenar. Esto traduce el problema de decisión a una variable binaria, es decir, un sistema de clasificación de dos clases que se simplifica a decidir si estamos a punto de chocar o no. Como las muestras no siguen una distribución lineal, hemos optado por utilizar una red neuronal para calcular la probabilidad de que en un momento concreto vayamos a colisionar o no.

5.3.1. Diseño

Para el diseño de la red neuronal hemos probado diferentes estructuras y distribuciones de las capas ocultas. Es un problema multiparámetro en el que hemos ido variando el número de neuronas y de capas, así como el *dropout* y las funciones de activación. Los resultados se muestran en el cap. 6.

Al final de las pruebas hemos observado que, con una estructura similar a la utilizada para reconocimiento de texto manuscrito [19], obtenemos los mejores resultados. Esto tiene sentido desde el punto de vista de las muestras utilizadas, ya que son un mapeo del entorno que rodea el tranvía, por lo que en realidad es una silueta de dos dimensiones representada en un vector de una dimensión, como el trazo de un carácter. La topología que hemos acabado implementando podemos verla en la figura 5.5.

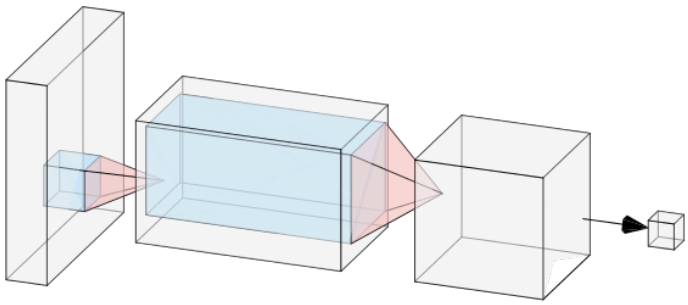


Figura 5.5: Topología

Se compone por una capa de entrada de 760 valores (correspondientes a dos muestras consecutivas del sensor), una capa oculta densa de 512 nodos con la función de activación ReLU, una capa de *dropout* con una probabilidad del 20 % y, finalmente, una capa de salida de 2 valores (la probabilidad de colisionar y no colisionar) con la función de activación Softmax.

5.3.2. Aprendizaje

Para entrenar la red neuronal vamos a alimentarla con un conjunto de muestras ya etiquetadas de un comportamiento correcto del tranvía. Cada muestra estará formada por dos lecturas del entorno separadas por 100 milésimas de segundo. Así entrenaremos la red neuronal para que calcule la probabilidad de colisión según el contexto en el que

se encuentra el tranvía, teniendo en cuenta el movimiento del mismo y de los objetos que lo rodean. Podemos ver un fragmento del *script* de entrenamiento en el *listing* 5.4.

```
def train_nn(records, labels):
    x = []
    y = []

    # Cada una de las muestras es la concatenación de 2 registros consecutivos
    for i in range(1, len(records)):
        x.append(records[i - 1] + records[i])
        y.append(labels[i])

    x = np.array(x, np.float32)
    y = np.array(y, np.int32)

    # Evitamos el 'gradient vanishing problem'
    ii = [all(e) for e in isfinite(x)]
    y = y[ii]
    x = x[ii]

    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(512, activation=tf.nn.relu),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(2, activation=tf.nn.softmax)
    ])

    model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(x, y, epochs=5, batch_size=32)

    return model
```

Listing 5.4: Implementación de la red neuronal

Entrenamiento orgánico

Una forma sencilla de recopilar los datos necesarios para entrenar la red neuronal es realizar un recorrido de prueba del tranvía conduciéndolo de forma manual e ir registrando muestras en el proceso.

Al proporcionar al sistema un recorrido real, realizado por un conductor humano, conseguimos tener registros de situaciones que son correctas, para así ayudar al sistema a imitar las decisiones que ha hecho el conductor real.

Lo más imporante a tener en cuenta es que se deben forzar situaciones en el recorrido que permitan al sistema tener ejemplos tanto de conducción normal como de accidentes y cómo han sido evitados.

Hay que valorar la recopilación de muestras y el circuito elegido, ya que la repetición del mismo circuito un número elevado de veces podría llevar a un sobreentrenamiento, y el testeo de la red podría darnos falsos resultados favorables.

Las muestras que hemos acabado usando han sido recopiladas con una frecuencia de 10Hz (10 pulsos del láser por segundo). Hemos sometido al conductor a una batería de accidentes forzados distribuidos a lo largo de un circuito con tramos de conducción sin obstáculos. Al frenar frente a los obstáculos obtenemos muestras de accidentes evitados que ayudan al sistema a frenar frente a posibles colisiones. Con los tramos de conducción normal conseguimos que el sistema sea menos vulnerable a falsos positivos.

Para las pruebas que podemos ver en el capítulo 6, hemos registrado un minuto de muestras, 600 en total, que hemos separado en 6 entrenamientos aumentando en 100 el número de muestras usadas para cada iteración.

Entrenamiento por refuerzo

Para el entrenamiento de la red neuronal del sistema de conducción hemos automatizado la generación de muestras y hemos aplicado una técnica de aprendizaje llamada aprendizaje por refuerzo [24]. El aprendizaje por refuerzo es una técnica de aprendizaje que se basa en castigar las acciones que lleven a un estado indeseado. Básicamente el funcionamiento es reafirmar el conocimiento de la red neuronal mientras lo esté haciendo bien y corregirle cada vez que lo haga mal.

En nuestro caso, hemos ejecutado la simulación y hemos hecho que el tranvía acelere de forma predeterminada, utilizando para ello una versión trivial de la red neuronal. Usando un sensor de colisiones hacemos que cada vez que el tranvía choque contra un obstáculo aísle las muestras recogidas en los últimos 2 segundos (el tiempo medio de frenada) y las etiquete como frenado. Usando las muestras modificadas actualizamos los pesos de la red neuronal, consiguiendo así que aprenda que las decisiones que han llevado a esa colisión son erróneas y han de ser de frenado.

Las muestras recopiladas con este tipo de entrenamiento son similares a las obtenidas con el entrenamiento orgánico. Hemos sometido al tranvía a un minuto de recorrido en el que ha tenido accidentes, corrigiendo así el etiquetado de aquellas muestras que han llevado a la colisión. También cabe destacar que, para evitar falsos positivos, hemos implementado un sistema auxiliar que corrige las muestras en el caso de que el tranvía haya estado 3 segundos parado sin tener un obstáculo potencial en su camino.

5.3.3. Predicción

Con la red neuronal ya entrenada podemos calcular la probabilidad de colisión del tranvía partiendo de dos muestras consecutivas del sensor. Usamos las dos muestras concatenadas como entrada y obtenemos las dos probabilidades (de que colisione y de que no). A partir de estas probabilidades decidimos si hemos de frenar el tranvía o continuar avanzando. Esta decisión es trivial, ya que si la probabilidad de colisión supera a la de no colisión tenemos que frenar el tranvía para evitar accidentes.

Con más datos podemos calcular la probabilidad mínima de colisión en la que se producen accidentes y realizar el frenado usando ese dato como umbral en vez de la comparación de probabilidades anterior.

CAPÍTULO 6
Pruebas y conclusiones

6.1 Resultados

Hemos sometido el sistema de conducción autónoma a diferentes pruebas tanto usando el entrenamiento orgánico de la red neuronal como el entrenamiento por refuerzo que se realiza de forma automática.

Al ser un problema multiparámetro, la batería de pruebas que hemos realizado se ha basado en modificar las variables que influyen en la fase de entrenamiento de la red neuronal. Los diferentes parámetros que podemos modificar son:

- El tiempo entre cada muestra registrada, o frecuencia del sensor.
- El número total de muestras utilizado para el entrenamiento.
- El número de nodos en la capa oculta de la red neuronal.
- El valor de la capa de *dropout*.

Aparte de estos valores también podemos modificar la topología de la red, pero el número posible de pruebas a realizar sería demasiado alto y el tiempo necesario para entrenar la red en cada caso lo haría inviable, teniendo en cuenta el alcance del proyecto.

Para simplificar las pruebas, y poder sacar mejores conclusiones, hemos optado por buscar unos valores para el número de neuronas de la capa oculta y para el porcentaje de la capa de *dropout* que nos den buenos resultados. También hemos hecho constante la frecuencia del sensor LIDAR ya que un aumento o disminución en el que habíamos propuesto hace que la precisión de la red disminuya notablemente. Los valores usados en estas pruebas son:

- Frecuencia del sensor: 10Hz (10 pulsos por segundo).
- Número de nodos en la capa oculta: 512.
- Valor de la capa de *dropout*: 20 %.

Con estos valores ya definidos, hemos sometido la red a una serie de entrenamientos variando el número de muestras usadas. Para cada iteración hemos registrado el número de muestras utilizadas para el entrenamiento de la red (10 muestras por segundo) por cada tipo de muestreo, el porcentaje de precisión alcanzado por el entrenamiento de la red neuronal y el número de accidentes reales detectados en un recorrido de prueba de 5 minutos. También hemos calculado una aproximación del número de falsos positivos

y falsos negativos obtenidos, teniendo en cuenta que un accidente ha sido causado por 2 segundos de falsos negativos (unas 20 muestras) y que cada acción de frenado en un tramo libre de obstáculos es un falso positivo. Se muestran estos resultados en las tablas 6.1 y 6.2.

Tabla 6.1: Resultados utilizando entrenamiento orgánico

Número de muestras	Precisión	Accidentes reales	FN	Tasa FN	FP	Tasa FP
100	83.4 %	16	320	10.67 %	13	0.43 %
200	80.3 %	16	320	10.67 %	32	1.06 %
300	78.3 %	14	280	9.33 %	35	1.17 %
400	78.1 %	10	200	6.67 %	30	1.00 %
500	76.4 %	8	160	5.33 %	25	0.83 %
600	79.3 %	7	140	4.67 %	23	0.76 %

Tabla 6.2: Resultados utilizando entrenamiento por refuerzo

Número de muestras	Precisión	Accidentes reales	FN	Tasa FN	FP	Tasa FP
100	81.9 %	16	320	10.67 %	20	0.67 %
200	80.4 %	15	300	10.00 %	26	0.87 %
300	80.1 %	11	220	7.33 %	24	0.80 %
400	79.3 %	7	140	4.67 %	20	0.67 %
500	76.7 %	4	80	2.67 %	21	0.70 %
600	72.5 %	3	60	2.00 %	14	0.47 %

De los resultados podemos obtener mucha información. Lo primero que se observa es que para el mismo número de muestras, el entrenamiento orgánico genera más accidentes que el entrenamiento por refuerzo. Esto es comprensible, ya que el entrenamiento orgánico se basa en que un conductor humano se enfrente a obstáculos y los evite, y el entrenamiento por refuerzo se somete al accidente y luego corrige el comportamiento que ha llevado a ese accidente. Obteniendo y clasificando las muestras de esta forma obtiene más información sobre cada situación y cómo evitarla.

También podemos observar otro suceso obvio, y es que con un número pequeño de muestras el número de accidentes reales es muy alto, ya que el sistema no tiene la información necesaria para evitar la mayoría de colisiones.

Si tenemos en cuenta las tasas de falsos negativos y falsos positivos podemos observar que el número de falsos positivos es mucho inferior al de falsos negativos. Este hecho se debe a la forma en que hemos entrenado el sistema de colisiones. Como el entrenamiento reafirma la conducción normal y castiga los accidentes, el número de positivos en un tramo libre de obstáculos es muy bajo, ya que el factor de tener un obstáculo delante tiene una gran influencia en el resultado positivo de la predicción, por lo tanto, solo se producen falsos positivos (y bastante escasos) en tramos con curvas y obstáculos que compliquen la predicción.

Si analizamos el descenso de los accidentes en relación con el número de muestras utilizadas en el entrenamiento orgánico, como podemos ver en la figura 6.1, podemos comprobar que el descenso no es muy pronunciado, ya que con el aumento del número de muestras puede que no haya aumentado su conocimiento sobre el entorno. Como ya hemos comentado, el entrenamiento orgánico se basa en gran parte en forzar situaciones de colisión para que, al evitarlas, el sistema aprenda cómo hacerlo por sí mismo. El

problema de esta forma de entrenamiento es que no se puede generar toda la casuística necesaria para hacer el sistema altamente fiable.

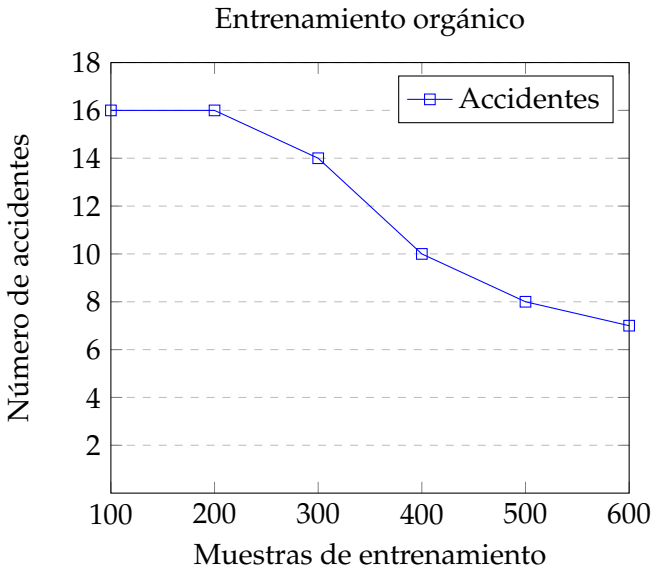


Figura 6.1: Resultados utilizando entrenamiento orgánico

Si analizamos el descenso de accidentes por muestra obtenido por el entrenamiento por refuerzo, como podemos ver en la figura 6.2, podemos comprobar que, efectivamente, este tipo de muestreo obtiene mejores resultados. Esto es debido que las muestras que son clasificadas como posible colisión son aquellas que de verdad llevaron a una colisión y después fueron etiquetadas por esa razón, al contrario que con el entrenamiento orgánico, en el cual el etiquetado de las muestras queda sujeto al comportamiento del conductor humano y de su tiempo de reacción.

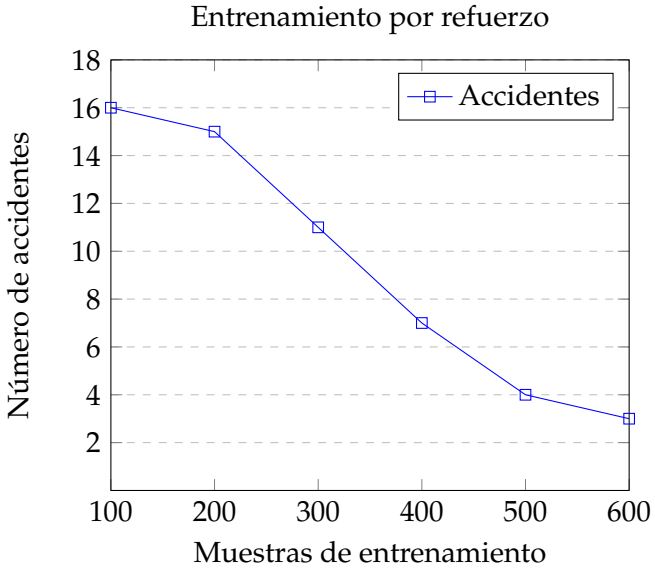


Figura 6.2: Resultados utilizando entrenamiento por refuerzo

Observando los porcentajes de precisión alcanzados con el entrenamiento de la red neuronal podemos comprobar que no son directamente proporcionales al número de accidentes reales obtenidos. Este hecho de debe a que la precisión calculada hace uso de un

subconjunto de prueba dentro de las muestras de entrenamiento; sin embargo, el número de accidentes se ve más influenciado por la cantidad de datos que usamos para entrenar la red, que se traduce en la cantidad de situaciones a las que hemos sometido al tranvía en su fase de entrenamiento. Este hecho también se hace evidente al observar el número de falsos negativos, con mayor cantidad de muestras, mayor es la información de la que dispone el sistema, por lo que la cantidad de falsos negativos disminuye considerablemente.

Tambíen se puede observar que, al aumentar el número de muestras, no tiene por qué aumentar o disminuir la precisión. Como hemos utilizado conjuntos de entrenamiento separados para cada una de las pruebas y este índice de precisión se ve fuertemente influenciado por la calidad de las muestras usadas, no se ha obtenido una variación proporcional al aumento del número de muestras.

6.2 Trabajos futuros

Como podemos comprobar mirando los resultados obtenidos, el sistema está lejos de ser perfecto. La detección simple de obstáculos usando un predictor de colisiones de un solo sensor no es suficiente para evitar todos los accidentes con los que puede encontrarse el tranvía. Sin embargo, puede ser un buen punto de paritda. Podríamos combinar el sistema de predicción de colisiones con un sistema de identificación de objetos basado en visión por computador (como hacen en sus vehículos empresas como Tesla). Podemos ver un ejemplo de la información que podríamos obtener en la figura 6.3.

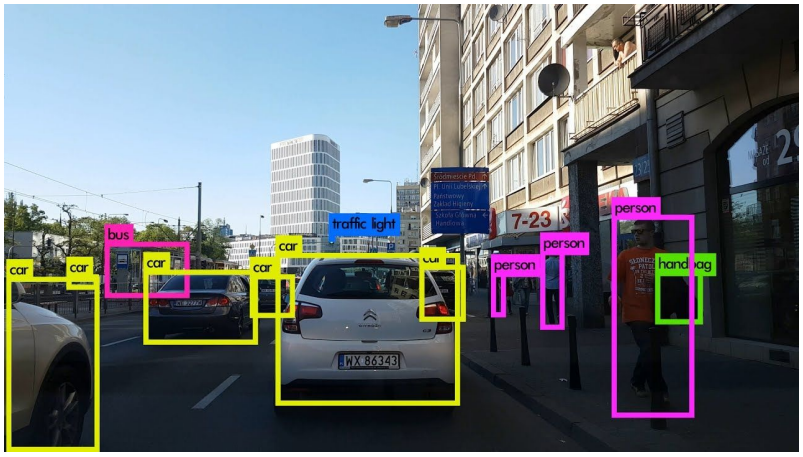


Figura 6.3: Detección de objetos

También podríamos aumentar el número de sensores y variar su distribución en el frontal del tranvía, consiguiendo así menos puntos ciegos. También podríamos implementar un sistema de reconstrucción de entornos 3D [25] a partir de una entrada de vídeo, como podemos ver en la figura 6.4.

Aplicando cualquiera de estos cambios nos daría más información con la que poder trabajar para prever y evitar los accidentes.

Aunque es verdad que este trabajo ha sido planteado de forma realista, pensando en una posible implementación física, el proyecto en sí es un ejemplo académico que muestra las posibilidades que nos ofrecen las redes neuronales en la detección de patrones complejos. No se ha entrado en mucho detalle en todas las técnicas posibles que podrían haberse aplicado para ayudar a mejorar el sistema y que algún día fuera viable en la vida real. Este precisamente sería el principal objetivo de futuras extensiones de este proyec-

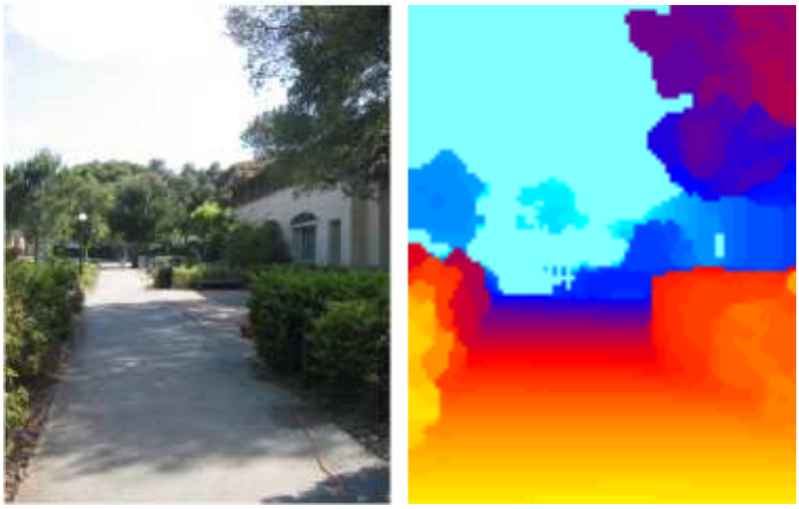


Figura 6.4: Reconstrucción de profundidad

to, explorar formas de mejorar la detección de obstáculos, tanto basadas en visión por computador como basadas en otro tipo de sensores.

6.3 Relación con el grado

Este proyecto ha sido principalmente inspirado por la rama de computación impartida en el grado. El trabajo está fuertemente relacionado con las asignaturas de Percepción y de Aprendizaje automático, asignaturas donde se aprenden formas de recopilar y almacenar datos para luego poder procesarlos analíticamente y técnicas para construir clasificadores. Sobretodo está muy ligado al tema del perceptrón multicapa, impartido en la clase de Aprendizaje automático, ya que el perceptrón multicapa es un ejemplo de red neuronal y es donde se conocen el descenso por gradiente y el algoritmo *backpropagation*.

Este trabajo también está relacionado con algunas de las competencias transversales que se evaluan en el grado. Como, por ejemplo, el conocimiento de problemas contemporáneos, el análisis y resolución de problemas y la aplicación y pensamiento práctico.

Bibliografía

[1] Bansal, Mayank, Alex Krizhevsky, and Abhijit Ogale. *Chauffeurnet: Learning to drive by imitating the best and synthesizing the worst*. arXiv preprint arXiv:1812.03079 (2018).

[2] RailEngineer. *Autonomous trams demonstrated in public*. https://www.railengineer.co.uk/2018/12/10/autonomous-trams-demonstrated-in-public/

[3] Bradski, Gary, and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.*. O'Reilly Media, Inc. 2008.

[4] Waymo *Waymo Safety Report: On the Road to Fully Self-Driving*. https://storage.googleapis.com/sdc-prod/v1/safety-report/waymo-safety-report-2017-10.pdf

[5] Mojo-Structure *Tram 1*. https://assetstore.unity.com/packages/3d/vehicles/land/tram-1-20864

[6] BansheeGz *BGCurve*. https://www.bansheegz.com/BGCurve/

[7] Unity *Asset store*. https://assetstore.unity.com/

[8] Craighead, J., Burke, J., & Murphy, R. (2008, September). *Using the unity game engine to develop sarge: a case study*. Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008).

[9] Jafri, R., Campos, R. L., Ali, S. A., & Arabnia, H. R. (2017). *Visual and infrared sensor data-based obstacle detection for the visually impaired using the Google project tango tablet development kit and the unity engine*. IEEE Access, 6, 443-454.

[10] Bartneck, C., Soucy, M., Fleuret, K., & Sandoval, E. B. (2015, August). *The robot engine—Making the unity 3D game engine work for HRI*. 2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN) (pp. 431-437). IEEE.

[11] Liu, X. (2008). *Airborne LiDAR for DEM generation: some critical issues*. Progress in Physical Geography, 32(1), 31-49.

[12] Chanin, M. L., Garnier, A., Hauchecorne, A., & Porteneuve, J. (1989). *A Doppler lidar for measuring winds in the middle atmosphere*. Geophysical research letters, 16(11), 1273-1276.

[13] Clifford, S. F., & Wandzura, S. (1981). *Monostatic heterodyne lidar performance: the effect of the turbulent atmosphere*. Applied optics, 20(3), 514-516.

[14] Spinhirne, J. D. (1993). *Micro pulse lidar*. IEEE Transactions on Geoscience and Remote Sensing, 31(1), 48-55.

[15] Koch, Grady J., et al. *High-energy 2 Doppler lidar for wind measurements*. Optical Engineering 46.11 (2007): 116201.

[16] Michael Nielsen *How the backpropagation algorithm works*. http://neuralnetworksanddeeplearning.com/chap2.html

[17] Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. *Rectifier nonlinearities improve neural network acoustic models*. Proc. icml. Vol. 30. No. 1. 2013.

[18] Abadi, Martín, et al. *Tensorflow: A system for large-scale machine learning*. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 2016.

[19] Graves, Alex, and Jürgen Schmidhuber. *Offline handwriting recognition with multidimensional recurrent neural networks*. Advances in neural information processing systems. 2009. Pages 545-552.

[20] Srivastava, Nitish, et al. *Dropout: a simple way to prevent neural networks from overfitting*. The Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[21] Kingma, Diederik P., and Jimmy Ba. *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980 (2014).

[22] Duchi, John, Elad Hazan, and Yoram Singer. *Adaptive subgradient methods for online learning and stochastic optimization*. Journal of Machine Learning Research 12.Jul (2011): 2121-2159.

[23] Tieleman, Tijmen, and Geoffrey Hinton. *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural networks for machine learning 4.2 (2012): 26-31.

[24] Sutton, Richard S., and Andrew G. Barto. *Introduction to reinforcement learning*. Vol. 135. Cambridge: MIT press, 1998.

[25] Saxena, Ashutosh, Sung H. Chung, and Andrew Y. Ng. *3-d depth reconstruction from a single still image*. International journal of computer vision 76.1 (2008): 53-69.

[26] C. Eberhart, Russell & Shi, Yuhui. (2007). *Computational Intelligence: Concepts to Implementations*. 10.1016/B978-155860759-0/50009-3. Elsevier.

[27] McCulloch, W.S. & Pitts, W. *Bulletin of Mathematical Biophysics (1943) 5: 115.* Kluwer Academic Publishers

[28] Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory*. New York: Wiley. ISBN: 0-8058-4300-0

[29] Rosenblatt, F. (1958). *The perceptron: A probabilistic model for information storage and organization in the brain.* Psychological Review, 65(6), 386-408.

[30] B. Widrow and M.E. Hoff, Jr. *Adaptive Switching Circuits* IRE WESCON Convention Record, 4:96-104, August 1960.

[31] Widrow and M.E. Hoff. *Associative Storage and Retrieval of Digital Information in Networks of Adaptive Neurons* Biological Prototypes and Synthetic Systems, 1:160, 1962.

[32] Minsky, M., Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA, USA: MIT Press.

[33] Teuvo Kohonen *Correlation Matrix Memories* IEEE Transactions on Computers 1972.

[34] James A. Anderson *A simple neural network generating an interactive memory* Volume 14, Issues 3–4, 1972, Pages 197-220, ISSN 0025-5564.

[35] Paul John Werbos *New Tools for Prediction and Analysis in the Behavioral Sciences* Beyond Regression, Harvard University, 1975.

[36] J J Hopfield *Neural networks and physical systems with emergent collective computational abilities* Proceedings of the National Academy of Sciences Apr 1982, 79 (8) 2554-2558.

[37] Reilly, D.L., Cooper, L.N. & Elbaum *A neural model for category learning* C. Biol. Cybern. (1982) 45: 35.

APÉNDICE A
La breve historia de las redes neuronales

El campo de las redes neuronales, pese empezar a ser de gran relevancia a nivel comercial en la última década, surgió en los años 40. En 1943, el neurofisiólogo Warren McCulloch y el matemático Walter Pitts escribieron un artículo [27] explicando como podrían funcionar las neuronas en el cerebro humano. Años más tarde, en 1949, Donald Hebb en *Organization of Behavior* [28] sugirió que las neuronas y sus conexiones se refuerzan cada vez que se usan. Argumentó que si dos nervios se activan en el mismo momento, la conexión entre ellos mejora.

Con el crecimiento de la potencia de cálculo de los ordenadores, en 1950 se desarrolló la primera implementación de una red neuronal, llevada a cabo por Nathanial Rochester en los laboratorios de investigación de IBM sobre un IBM 704. Sin embargo, ese primer intento no fue muy exitoso.

En 1958, Frank Rosenblatt, basandose en el modelo de la neurona de McCulloch y Pitts, inventó el algoritmo perceptrón [29], un modelo matemático que simula cómo operan las neuronas de nuestro cerebro. Este algoritmo acepta una serie de datos de entrada binarios, los multiplica por una serie de pesos y aplica un umbral a su suma para obtener una salida binaria de 0 o 1. Este algoritmo es uno de los primeros utilizados como neurona.

En 1959, Bernard Widrow y Marcian Hoff desarrollaron en Stanford el conocido modelo ADALINE [30], desarrollado para reconocer patrones binarios en líneas telefónicas y así predecir el siguiente bit de información transmitido. Así surgío MADALINE, la primera red neuronal aplicada a un problema del mundo real y que, pese a tener 60 años, aún se utiliza a nivel comercial.

Tres años más tarde, en 1962, Widrow y Hoff desarrollaron un procedimiento de aprendizaje [31] que examina el valor antes de que el peso se ajuste usando la regla: Cambio del peso = (Peso anterior) * (Error / (Número de entradas)). Esto se basa en la idea de que si un perceptrón activo tiene un gran valor de error, podemos ajustar los pesos para distribuirlo a través de la red.

A pesar de los resultados en el campo de las redes neuronales, la arquitectura tradicional von Neumann eclipsó su relevancia y su investigación fue dejada de lado. Irónicamente, el mismo John von Neumann sugirió en 1957 la imitación de funciones neuronales usando relés de telégrafos y tubos de vacio.

En el mismo periodo de tiempo, un artículo [32] sugirió que las redes neuronales de una sola capa no podían ser extendidas a redes neuronales multicapa. Ese hecho se sumó a que los primeros éxitos de algunas redes provocaron una exageración del potencial

de las redes neuronales, especialmente considerando la tecnología de esa época. No se cumplieron las expectativas y se realizaron aproximaciones y preguntas filosóficas que provocaron miedo en la comunidad. Algunos escritores evaluaron los posibles efectos de las llamadas "máquinas pensantes" sobre el ser humano, ideas que aún circulan hoy en día. Como resultado, la investigación y la inversión descendieron drásticamente.

En 1972, Kohonen [33] y Anderson [34] desarrollaron, cada uno por su lado en países diferentes, dos redes muy parecidas, las cuales usaban cálculos matriciales para describir sus ideas. Pero no se dieron cuenta de que lo que estaban haciendo era crear un sistema análogo a los circuitos ADALINE. Se suponía que las neuronas activaran una serie de salidas en vez de una sola.

En 1975, Paul Werbos inventó el algoritmo *backpropagation* [35] y apareció la primera red multicapa sin supervisión.

En 1982 el interés en el campo aumentó; John Hopfield de Caltech presentó un artículo [36] a la *National Academy of Sciences* donde mostraba una aproximación para crear máquinas más útiles usando caminos bidireccionales. Previamente, las conexiones entre neuronas habían sido de un solo sentido.

Ese mismo año, Reilly y Cooper [37] usaron una red híbrida multicapa, con cada capa usando una estrategia de resolución de problemas diferente.

También en 1982 se hizo una conferencia conjunta entre Estados Unidos y Japón sobre redes neuronales. Japón anunció una quinta generación de esfuerzos en el área de redes neuronales y artículos estadounidenses mostraron miedo a que Estados Unidos quedara detrás en el campo. La quinta generación involucra inteligencia artificial. La primera usaba interruptores y cables, la segunda usó el transistor, el tercer estado usó circuitos integrados y lenguajes de programación de alto nivel y la cuarta generación se basaba en generadores de código. Como resultado, surgieron fondos para la investigación de las redes neuronales.

En 1986, con noticias sobre el desarrollo redes multicapa, el problema era cómo extender la regla de Widrow-Hoff a múltiples capas. Tres grupos independientes de investigadores, uno de ellos incluyendo a David Rumelhart, un miembro original del departamento de psicología de Standford, llegaron a ideas similares, ahora llamadas redes con *backpropagation* porque distribuyen errores en la detección de patrones a través de la red. Las redes híbridas solo usan dos capas, las redes con *backpropagation* usan varias. El resultado es que estas redes aprenden de forma más lenta, necesitando posiblemente miles de iteraciones para aprender.

Hoy en día las redes neuronales se usan en una gran variedad de aplicaciones y deben de ser capaces de ejecutarse en ordenadores de consumo. El futuro de las redes neuronales está definido principalmente en el desarrollo de hardware. Sistemas de aprendizaje profundo como el jugador de ajedrez de Deep Blue necesitan hardware específico, rápido y eficiente para su uso. Empresas como Nvidia ya están invirtiendo fuertemente en tarjetas gráficas con tensores dedicados para el cómputo de redes neuronales. Aunque es verdad que redes simples pueden ser ejecutadas en sistemas tradicionales, los requisitos de las nuevas estrategias de aprendizaje profundo necesitarán cada vez más sistemas a medida para obtener resultados óptimos.