Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención Ingeniería del Software

Sistema de detección de señales y vehículos mediante Redes Neuronales Convolucionales

Autor:
D. Óscar Fernández Angulo
Tutor:
Dr. Benjamín Sahelices Fernández
D. Fernando Rodríguez Vicente

Agradecimientos

A Arturo y Diego, del Grupo Trasgo, pues sin su generosidad este trabajo no habría sido posible.

A mis tutores, por su apoyo y consejo durante todo el trabajo.

A mis amigos, que tantas sonrisas me han sacado durante estos años. Vosotros sois, sin duda, lo mejor que me llevo de esta carrera.

A mi gran familia, de quienes, cada domingo, aprendo más de lo que ninguna escuela podrá enseñarme. En especial a mi hermana, quien lleva siendo mi referente toda mi vida.

Este proyecto ha sido parcialmente financiado por el MICINN y el programa ERDF de la Uni on Europea: proyecto HomProg-HetSys (TIN2014-58876-P), la red CAPAP-H5 (TIN2014-53522-REDT) y el COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Resumen

Con el aumento de la capacidad de cómputo de los ordenadores y el avance de los algoritmos de *Machine Learning*, surge la oportunidad de abordar problemas clásicos desde nuevas perspectivas. Por esta razón, se propone la construcción de un sistema capaz de detectar señales de velocidad y vehículos, con algoritmos de Visión Artificial. Además, se pretende que estos modelos se puedan ejecutar tanto desde móviles como desde sistemas embebidos, por lo que es fundamental que sea ligero y fácil de utilizar por diferentes clientes. Para ello, se ha realizado un estudio del estado del arte de los algoritmos de detección de objetos, se ha recogido y clasificado un set de datos, y se han entrenado y modificado los algoritmos hasta obtener buenos resultados. Por último, se ha implementado un componente, capaz de ejecutar Redes Neuronales Convolucionales complejas, que englobe el algoritmo y permita su uso desde cualquier dispositivo.

Abstract

With the increase of computing capacity and the improvement of *Machine Learning* algorithms, the opportunity to approach classical problems from new perspectives arises. For this reason, we propose the implementation of a system capable of detecting signals and vehicles with Computer Vision algorithms. In addition, it is a requirement that these models can be run from mobile phones and embedded systems, so it is essential that it is light and easy to use by different customers. To that end, a study of the state of the art of object detection algorithms has been carried out, a dataset has been collected and classified, and the algorithms have been trained and modified until obtaining good results. Finally, a component has been implemented, capable of executing complex Convolutional Neural Networks, which wraps the algorithm and allows its use from any device.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

El avance de los últimos años en los algoritmos de *Machine Learning*, y en la capacidad de cómputo de los ordenadores, permite abordar desde nuevas perspectivas problemas que, hasta el momento, se veían relegados a soluciones más inflexibles. Por esta razón, se propone realizar un reconocedor de señales y vehículos por medio de Visión Artificial, algo que se venía afrontando desde la perspectiva del GPS o de los sensores de proximidad.

1.1. Objetivos

El objetivo de este trabajo es la realización de un sistema capaz de reconocer tanto vehículos como señales de velocidad. Informando de la localización de los vehículos y de la velocidad máxima de la vía. Este sistema deberá ser desplegable tanto en teléfonos móviles como en sistemas embebidos en vehículos, por lo que es fundamental que sea ligera y veloz.

Este solo es el punto de partida de un sistema de asistencia completo, por lo que el desarrollo de un sistema escalable es uno de los requisitos principales. El desarrollo de un código legible y bien documentado toma también una especial relevancia.

1.2. Estructura del trabajo

La memoria se ha estructurado de la siguiente manera:

En el capítulo 1: Introducción, se introducen los objetivos del proyecto. Además, se muestran las convenciones matemáticas y lingüísticas que se seguirán a lo largo de la memoria. Por último, se presenta, brevemente, la historia del *Machine Learning* para mostrar la evolución de la misma y poder, observando el pasado, aventurar la importancia que tomará en el futuro.

En el capítulo 2: Plan de Proyecto, se argumenta el por qué de haber elegido mapas mentales frente a RUP y SCRUM como metodología. Además, se explica cómo se ha utilizado para planificar

este proyecto.

En el capítulo 3: Fundamentos teóricos, se presentan todos los conocimientos teóricos necesarios para la comprensión de este trabajo. Están enfocados a un Ingeniero Informático no especializado en Inteligencia Artificial. Estos conocimientos van desde una pequeña introducción al *Machine Learning* hasta las arquitecturas de Redes Neuronales Convolucionales punteras.

En el capítulo 4: Análisis, se realiza un análisis completo del problema. Incluyendo en este la especificación de los requisitos, los casos de uso y el modelo del dominio.

En el capítulo 5: Diseño, se presenta el diseño de la solución. Se comienza con el diseño de la Red Neuronal, donde se decide entorno a que arquitectura se va a trabajar y se estudian los diferentes *frameworks* y bibliotecas de las que nos provee el mercado. Después, se analiza la arquitectura del componente, haciendo una mención especial a la interfaz externa del mismo, pues es la parte más importante. Por último, se presentan los diagramas de secuencia que se corresponden con los diferentes casos de uso expuestos en el capítulo anterior.

En el capítulo 6: Implementación y Pruebas, se comienza con una breve descripción de como se ha implementado el componente, los problemas que han surgido y como se han solucionado. Después, se detalla el proceso por el cual se ha desarrollado el algoritmo. Se muestran todas las pruebas e iteraciones que se ha realizado sobre el mismo, así como el análisis de los errores que iban revelando en que dirección debían ir las modificaciones. Por último se muestran algunos ejemplos sobre el conjunto de prueba para analizar los resultados obtenidos por esta Red Neuronal mostrando especial énfasis en sus debilidades.

En el capítulo 7: Conclusiones, se hace una reflexión acerca de la respuesta propuesta al problema planteado en el capítulo de Análisis. Así como una exposición de algunas de las conclusiones que se extraen del desarrollo de la misma. Por último, se comentan los objetivos que se deberían priorizar en las siguientes iteraciones de este trabajo. También se presentan un conjunto de vías alternativas en las que podría desembocar en el futuro.

1.3. Convenciones

En cuanto a la notación matemática, se utilizan los siguientes símbolos: L representa el número de capas (sin contar la capa de entrada, por lo que el mínimo número de capas será 1), $n^{[l]}$ el número de neuronas de la capa l y m el número de ejemplos de entrenamiento. El símbolo $a_n^{[l](i)}$ representa la activación de la capa l en la neurona n para el ejemplo i, $a^{[l](i)}$ representa la activación de la capa completa en el ejemplo i. Por lo tanto, $a^{[0](0)}$ representa la entrada a la Red Neuronal en el primer ejemplo y $a^{[L](0)}$ la salida. $W^{[l]}_{n_1 n_2}$ representa el peso entre la neurona m de la capa l y la neurona n de la capa $l+1$. Por consiguiente, $W^{[l]}$ representa la matriz de pesos entre l y $l+1$. La función de activación se designará con $\sigma()$.

En cuanto al idioma empleado, la memoria se presenta completa en castellano (excepto algún anglicismo, que se presentarán en cursiva) puesto que está dirigida a un tribunal español. Sin embargo,

la implementación y la documentación interna del sistema se realizarán en inglés para internacionalizar el uso y mantenimiento de la aplicación.

1.4. Historia

Se presenta, por último, la historia del *Machine Learning*, así como una reflexión acera del futuro del mismo, para poder comprender la importancia de comenzar a enfocar las soluciones a problemas reales desde esta perspectiva.

Se puede situar el inicio del *Machine Learning* en el año 1943, cuando Warren McCulloch, neurólogo, y Walter Pitts, matemático, modelaron una Red Neuronal Artificial con circuitos eléctricos. Siete años después, cuando la tecnología fue lo suficientemente avanzada, comenzaron los primeros intentos de recrear una Red Neuronal —por medio de una computadora— a manos de Nathananial Rochester. Para su desgracia, estos intentos fueron fallidos. No fue hasta 1952 cuando Arthur Samuel, pionero en las investigaciones sobre Inteligencia Artificial, desarrolló un programa capaz de jugar a las Damas y mejorar después de cada partida. A partir de este momento los avances no han dejado de sucederse. En 1959, Bernard Widrow y Marcian Hof desarrollan la primera Red Neuronal aplicada a un problema real. MADALINE (Multiple ADAptive LINear Elements), capaz de eliminar el eco de una llamada telefónica [15]. En 1985, Terry Sejinowski implementa una Red Neuronal de 300 neuronas y una sola capa oculta capaz de aprender en una semana como pronunciar 20.000 palabras [1].

Es entonces, en 1997, cuando llega uno de los hitos más conocidos del *Machine Learning*. Deep Blue gana a Garri Kaspárov, un *Gran Maestro* de ajedrez. Mostrando así, al mundo entero, los avances que se estaban realizado en este campo [24]. Tras 30 años de evolución en estas técnicas, una máquina consigue una hazaña mucho mayor, ganar al mejor jugador de Go, el que es considerado el juego de mesa más complicado (el número de partidas posibles excede por mucho al número de átomos en el universo observable) [18]. Esta tecnología sigue avanzando de tal manera que, en tan solo tres años más, un algoritmo es capaz de aprender a jugar al Go de manera autónoma, sin necesidad de la intervención humana durante este proceso, simplemente jugando contra ella misma. Este algoritmo, denominado AlphaGo Zero, alcanza al comentado anteriormente en tan solo tres días. A los 21 días alcanza el nivel de AlphaGo Master, un algoritmo que derrotó a 60 jugadores expertos y derrotó al campeón mundial en las tres partidas que jugaron. A los 40 días, AlphaGo Zero había superado todos los algoritmos predecesores, convirtiéndose (probablemente) en el mejor jugador de la historia [8]. Este algoritmo es especialmente relevante para la evolución del *Machine Learning* porque la intervención humana fue muy limitada. A diferencia de otros algoritmos que necesitan miles o incluso millones de datos clasificados por humanos expertos para aprender, AlphaGo Zero se valió de las reglas del juego para entrenarse de manera autónoma. Un gran avance en estas técnicas.

Otro punto a destacar dentro del aprendizaje autónomo son los algoritmos que se han ido desarrollando durante el ultimo año, capaces de construir Redes Neuronales de forma autónoma y dando lugar a arquitecturas de lo más pintorescas, obteniendo estas resultados similares e incluso mejores que Redes Neuronales desarrolladas por expertos en la materia [4].

Por ultimo, se expondrá el nombre de dos robots, cuyas historias son más anecdóticas que técnicas, pero que podrían anticipar el futuro al que nos dirigimos. No se entrará en juicios morales a cerca de estos acontecimientos, pero sin duda merecen un capítulo en la historia del *Machine Learning*. Sophia, diseñado por Hanson Robotics. Su fin es dialogar con los humanos de la manera más empática posible, con ayuda de su rostro, expresando emociones e incluso contando algún chiste. El 27 de octubre de 2017, este robot recibía la nacionalidad de Arabia Saudita, convirtiéndose en el primer robot ciudadano de la historia [7]. Michihito Matsuda, desarrollado por Norio Murakami (exempleado de Google) y Tetsuzo Matsuda (vicepresidente de Softbank). El 15 de marzo de 2018, este robot se presentó a la alcaldía de Tama, un distrito de Tokio. Los puntos fuertes de su candidatura fueron: acabar con la corrupción, ofrecer políticas imparciales y estudiar las peticiones de los ciudadanos una por una, analizando estadísticamente los aspectos positivos y negativos de cada una. Este programa le colocó en tercera posición con más de cuatro mil votos [2].

Como se puede observar, estos algoritmos existen desde hace años, pero es ahora cuando están comenzando a obtener buenos resultados. Esto se debe al aumento en la capacidad de computo de los ordenadores, que permiten trabajar con modelos de gran complejidad. Además, por la gran cantidad de datos de los que se dispone actualmente (y de los que vendrán acompañados del tan comentado *Big Data*) que son necesarios para entrenar estos modelos. Y es de esperar que se convierta en uno de los pilares fundamentales de la revolución que está comenzando. Por esta razón, es fundamental anteponerse a los cambios y comenzar a apostar por estas tecnologías.

Capítulo 2

Plan de Proyecto

2.1. Metodología empleada

A la hora de planificar el proyecto se analizaron diferentes metodologías hasta encontrar la que mejor encajaba con el tipo de proyecto. Se presentaran a continuación las principales opciones valoradas y los pros y contras de cada una de ellas.

2.1.1. *Rational Unified Process*

La primera que se analizó fue RUP, pues es una de las metodologías más utilizadas para el desarrollo de software y más común en este tipo de proyectos.

Como puntos fuertes se podría destacar que es una metodología centrada en la Ingeniería del Software y está completamente orientada a las fases de Requisitos, Análisis, Diseño, Implementación y Pruebas; lo que facilita un correcto desarrollo de sistemas orientado a objetos. Además, es un proceso iterativo e incremental, por lo que minimiza el número de cambios a realizar a lo largo del proyecto, disminuyendo enormemente el coste y la duración del mismo. Además, al final de cada iteración se obtiene un prototipo, por lo que es posible presentarselo al cliente y obtener correcciones en un punto del proyecto en el que es asequible corregirlos.

Sin embargo, RUP tiene una serie de características que hace que no encaje con este proyecto. Para empezar, es una metodología ideada para equipos grandes, y en este proyecto solo trabaja una persona. Además, los objetivos, fecha de entrega y recursos están predefinidos a la hora de comenzar el proyecto, por lo que carece de sentido centrarse en los posibles cambios que estos podrían sufrir, así como analizar los tiempos que nos llevaran las diferentes tareas. Como añadido, prácticas como los diagramas gantt —que suelen acompañar a esta metodología— no tienen sentido, pues es una única persona la que desarrolla el proyecto, por lo que no es necesario repartir las tareas y le confiere una rigidez al proyecto que no aporta ningún beneficio, sino todo lo contrario. Asimismo, no se puede planificar las tareas, ya que, de primeras, no se conocen las que van a ser necesarias. Por esta última razón, parece que podría encajar mejor una metodología ágil, como SCRUM.

2.1.2. SCRUM

En los últimos años, esta metodología ágil está cogiendo mucha fuerza en el mundo de la informática. Tiene un gran número de ventajas, como la elección de las tareas a realizar en cada Sprint (entre dos y tres semanas), algo que encaja muy bien con este proyecto ya que, como se ha comentado con anterioridad, éstas no se conocen al comenzar el desarrollo. Igualmente, es un gran beneficio poder presentar un prototipo al cliente después de cada una de estas fases, pues así nos aseguramos de que, al finalizar el proyecto, el cliente estará conforme con el producto desarrollado. Otro de los puntos fuertes es que, de esta manera, no es necesario fijar el alcance al comenzar el proyecto y se puede ir avanzando hasta que el cliente quede conforme, sin restricciones de tiempo ni de recursos; uno de los principales problemas de las metodologías clásicas, pues estimar el tiempo que va a llevar desarrollar un sistema es una tarea muy compleja y en gran parte de los proyectos quedan sin finalizar o con sobre-costes debido a una mala planificación.

Sin embargo, en este proyecto no es posible trabajar de esta manera, pues los recursos, tiempos y objetivos están fijados desde el inicio del desarrollo. En cuanto al desarrollo basado en prototipos, no es posible debido a que este proyecto es muy difícil de prototipar, ya que gran parte de la carga de trabajo reside en la investigación de los algoritmos de Visión Artificial. Otro punto en contra de esta metodología es la necesidad de un equipo de trabajo, pues sin estos, muchos de los pilares de esta metodología carecen de sentido. Algunas de estas caracteristicas serían las reuniones diarias que realiza el equipo; los roles bien definidos en los que se divide el trabajo, como el *SCRUM Master*, los *QA* (aseguramiento de calidad) o el *Product Owner*; las reuniones retrospectivas y todo el conjunto de estrategias enfocadas en como repartir las tareas entre los miembros del equipo de desarrollo. Por estas razones, en particular en la de la importancia de la investigación, se decidió analizar una opción mucho más innovadora: los mapas mentales.

2.1.3. Mapas Mentales

Los mapas mentales son una herramienta de representación de conceptos de forma visual. Su objetivo principal es el de sintetizar una unidad de información a la mínima expresión posible, evitando la redundancia y manteniendo las ideas claves. Esto se consigue mediante un diagrama en el que se coloca el tema principal en el centro de la hoja y luego se van realizando conexiones por medio de líneas, hacia conceptos secundarias, los cuales a su vez se vuelven a ramificar, permitiendo una vista global del tema.

La principal ventaja de esta forma de organizar proyectos es que permiten organizar tanto las tareas como el conocimiento. Esto es muy útil en un proyecto con mucha carga de investigación, pues los artículos analizados y las opciones a valorar son muy numerosas, y estos esquemas te permiten organizar toda la información recabada durante meses de investigación de una forma sintetizada y muy visual. Además te permite tener localizados todos los recursos, sin necesidad de tener que organizarlos en los tediosos arboles de directorios. Pero esta no es la única ventaja, estos diagramas fortalecen el pensamiento no-lineal, aumentando así la creatividad y el desarrollo de los temas a tratar de una forma mucho más paralela, teniendo siempre en mente el proyecto como un todo.

Asimismo, aumentan la productividad porque no fuerzan a trabajar en una tarea después de otra, sino que permiten saltar entre ellas, disminuyendo así las interrupciones por falta de concentración o motivación (permitiendo trabajar al desarrollador, en cada momento, en lo que más le llame la atención). Son muy útiles también en problemas como el que se aborda en este proyecto, en el que se desconoce la solución, pues estos diagramas evolucionan continuamente permitiéndote así comparar de forma más visual las diferentes opciones y organizar las tareas de la opción elegida. Por estas razones, se decidió apostar por esta nueva metodología y probar sus beneficios en este proyecto.

2.2. Plan de acción

A pesar de que las tareas y los conocimientos se organizaron mediante un mapa mental, esto no implicó dejar de lado todas las buenas ideas del resto de metodologías. Por supuesto, el proceso se basó en las etapas del desarrollo de software de Análisis del problema, Diseño de la solución, Implementación y Pruebas. Ejecutando las etapas en este orden, pero permitiendo iterar sobre las anteriores de una manera mucho más flexible. Además, de las metodologías ágiles se heredó el desarrollo de prototipos —en las últimas fases del proyecto—, permitiendo así presentárselas al cliente (Fernando Rodríguez, tutor de HP) y analizando posibles cambios tanto en los algoritmos seleccionados como en la estructura del sistema. De esta manera, se aprovecharon todas las ventajas de la Ingeniería del Software, de una manera muy productiva.

Para minimizar el riesgo de perdida de información, todo el proyecto se fue almacenando periódicamente en un repositorio de GitHub, evitando que un fallo en el hardware eliminase toda la información y almacenando todas las versiones, para poder volver a una anterior, si es que fuera necesario.

En la figura 2.1 se puede ver el mapa mental final, con cuatro niveles de profundidad. Se puede ver cómo se van anotando las diferentes tareas, así como las ideas que van surgiendo, por ejemplo, en la implementación del algoritmo. Este mapa, por supuesto, es mucho más profundo (llegando a ocho niveles de profundidad), pero es demasiado extenso para presentarlo en su totalidad.

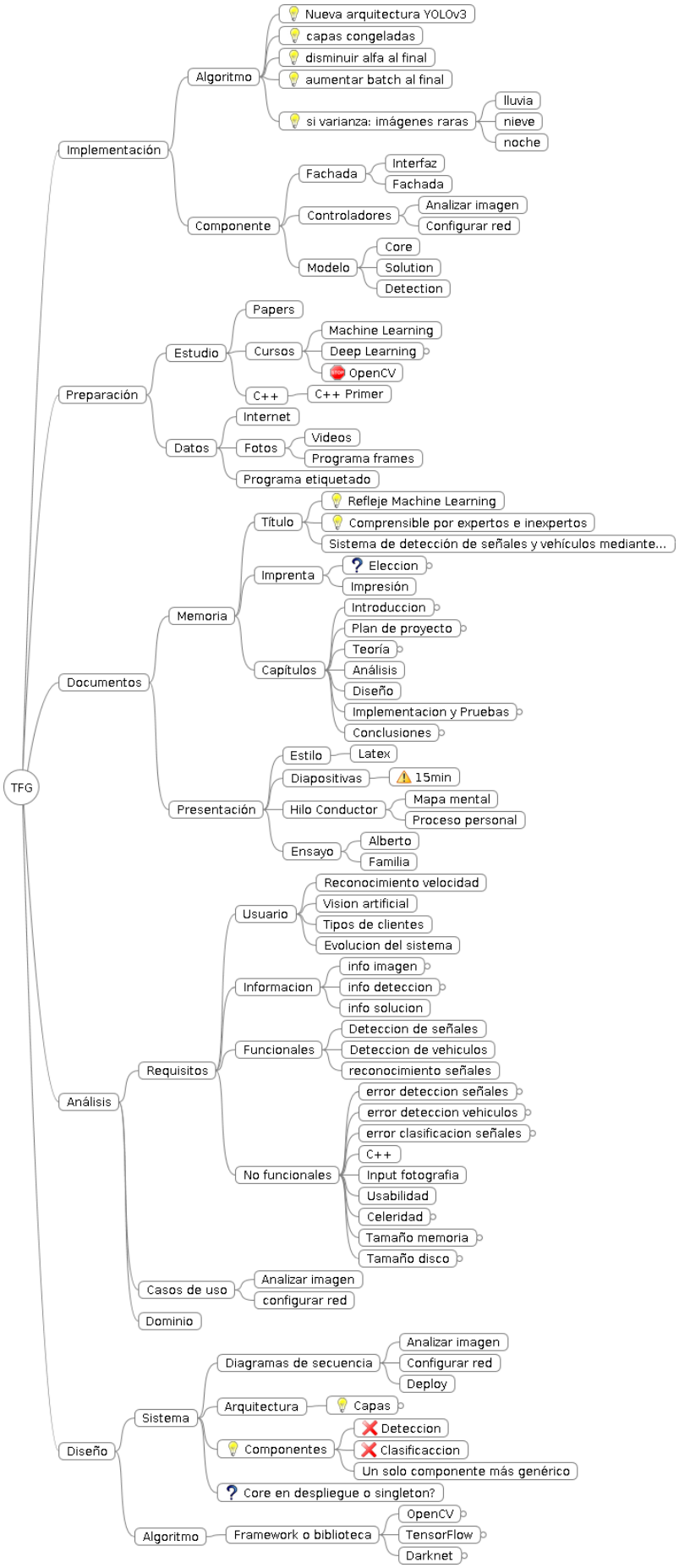


Figura 2.1: Mapa mental (cuatro niveles de profundidad).

Capítulo 3

Fundamentos teóricos

Se presenta a continuación los fundamentos teóricos necesarios para la completa comprensión del trabajo. Están enfocados a una persona con los conocimientos obtenidos al finalizar esta titulación: Amplios conocimientos de informática, no especializados en *Machine Learning* y básicos de matemáticas y estadística. Útiles también para un perfil general de informático, no especializado en la Inteligencia Artificial.

3.1. Introducción

El campo de estudio que da a las máquinas la capacidad de aprender sin ser explícitamente programadas (o *Machine Learning*) cuenta con un gran número de diferentes técnicas y algoritmos. Se pueden clasificar de dos maneras, en función de qué pretenden aprender y de cómo pretenden hacerlo.

Desde el punto de vista del tipo de problema que quieren aprender, podemos catalogarlos en: algoritmos de regresión y algoritmos de clasificación. La regresión tiene el objetivo de predecir valores continuos, como por ejemplo, fijar el precio de una casa en función del número de habitaciones, metros cuadrados de la vivienda, etc. La clasificación tiene la tarea de asignar una clase, es decir, predecir a que clase pertenece un conjunto de datos. Aquí es importante entender que, en los problemas de clasificación, los valores son discretos. Un ejemplo podría ser decidir si el animal de una fotografía es un perro o un gato.

En función de su forma de aprender, se pueden dividir en supervisados y no supervisados. El aprendizaje supervisado es el más común de los dos. Reciben ese nombre porque —de la misma manera que un niño aprende aritmética de su profesor— estos algoritmos necesitan un experto para que les diga lo que esta bien y lo que no lo esta. Por lo tanto, todos los datos suministrados en la fase de aprendizaje deben estar correctamente etiquetados. Por el contrario, los problemas enfocados sin supervisión son aquellos de los que se desconoce como va a ser la salida. Como ejemplo práctico, este tipo de algoritmo es útil para la segmentación de clientes, porque devolverá grupos basados en parámetros que un humano puede no considerar debido a prejuicios preexistentes. En otras palabras,

mientras un algoritmos de clasificación supervisado aprende a relacionar entradas clasificadas con imágenes de animales, su homólogo no supervisado mira las similitudes entre las imágenes para separarlas en grupos a su propia elección [6].

Según los tipos de distinciones presentados anteriormente podemos agrupar los algoritmos en cuatro grupos:

- Clasificación supervisada: Al ser el tipo de problema más común, contamos con un vasto catálogo de algoritmos donde elegir. Redes Neuronales, SVM (Support Vector Machine), Regresión Logística, Arboles de decisión, etc.

  Entre los problemas comúnmente enfocados desde esta perspectiva se encuentran: Clasificación de imágenes, reconocimiento de escritura, reconocimiento del habla, descubrimiento de drogas, así como cualquier otro problema que pretenda separar datos en clases previamente definidas.

- Clasificación no supervisada: Para solucionar un problema de estas características podríamos seleccionar entre, por ejemplo, K-means, Apriori o FP-Grouwth.

  En este grupo entran los problemas que tratan de separar conjuntos de manera automática basándose en sus características.

- Regresión supervisada: Estos problemas también tiene un gran conjunto de algoritmos donde elegir. Redes Neuronales, Regresión Logística, SVR (Support Vector Regressor), GPR (Gaussian Processes Regression), etc.

  Este tipo de problemas incluye la detección de objetos, así como la predicción de cualquier valor lineal.

- Regresión no supervisada: Es una clasificación en la que es difícil encajar un problema, por lo que se encuentran pocas soluciones de este tipo. Algún algoritmo para resolver estos problemas podría ser SVD (Singular Value Decomposition) o PCA (Principal Component Analysis).

  Con ellos podríamos implementar un detector de anomalías o resolver el famoso "Cocktail party effect", que consiste en separar un sonido concreto, como podría ser la voz de una persona, acompañado de mucho ruido, como podría ser una fiesta.

Se continuará exponiendo los detalles de las Redes Neuronales ya que, como se mostrará más adelante, es el algoritmo seleccionado para lograr los objetivos de este trabajo. A parte de esto, y a pesar de que a veces se encuentren Redes Neuronales con aprendizajes no supervisados, a partir de aquí será tratado como un algoritmo exclusivamente supervisado [28].

3.1.1. Redes Neuronales Artificiales

Una Red Neuronal Artificial es un modelo computacional inspirado en la forma en la que las redes neuronales biológicas procesan la información. Las Redes Neuronales Artificiales han generado un gran número de seguidores dentro del campo del *Machine Learning* debido a sus grandes resultados en el reconocimiento del habla, Visión Artificial y reconocimiento de escritura, entre otros.

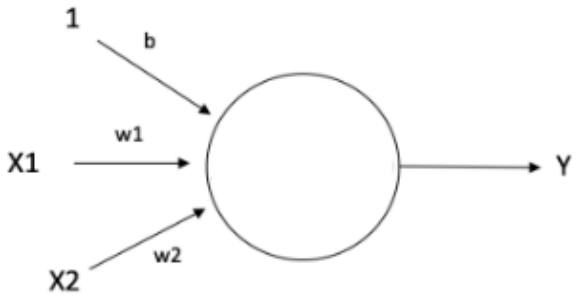


Figura 3.1: Modelo de una neurona.

Neurona

La unidad básica de computación en una Red Neuronal (como no podía ser de otra manera) es la neurona. Como se muestra en la figura 3.1, recibe la entrada (X_i) del exterior o de otras neuronas de la red y computa la salida (Z). Cada entrada está asociada a un peso (W_i), los cuales asignan la importancia que toma para esta neurona cada una de las entradas. La neurona aplica una suma ponderada de las entradas y de un termino independiente. La formula general quedaría de la siguiente manera[1]:

$$Z = \sum_{n=1}^{i} W_i X_i + b$$

Después, le aplica a esta suma una formula llamada función de activación (σ), de la que hablaremos más tarde.

$$Y = \sigma(Z)$$

Hay tres tipos de neuronas:

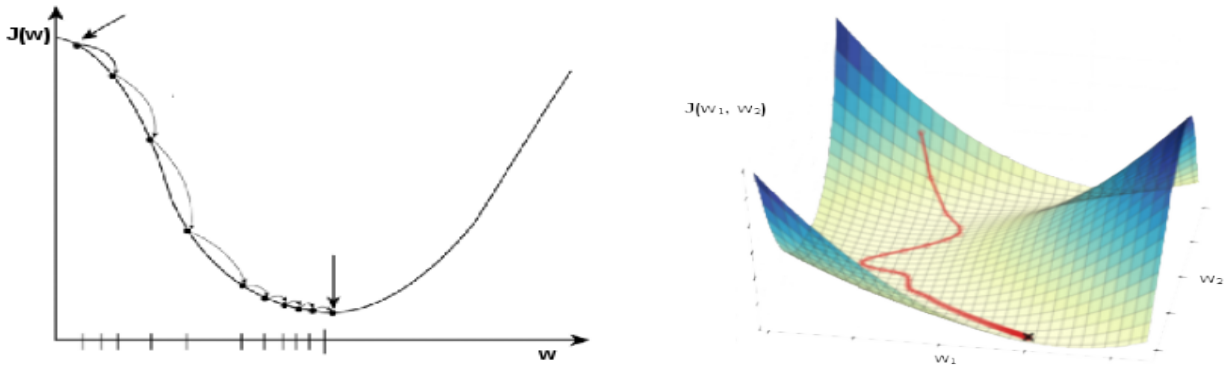
- Neuronas de entrada: Son las encargadas de proporcionar información del exterior de la red. No realizan ningún cálculo, suministran la información directamente a las neuronas ocultas. Todas juntas forman la capa de entrada.
- Neuronas ocultas: No tienen conexión directa con el exterior. Realizan los cálculos y son las encargadas de enviar la información desde la entrada a la salida. Un conjunto de neuronas ocultas forman una capa oculta.
- Neuronas de salida: Son las encargadas de proporcionar información al exterior de la red. A diferencia de las neuronas de entrada, estas si que realizan cálculos sobre sus valores de entrada. Todas juntas forman la capa de salida.

[36]

Fase de entrenamiento

Se usa un conjunto de datos o patrones de entrenamiento para determinar los pesos que definen el modelo de Red Neuronal. Se calculan de manera iterativa, de acuerdo con los valores de entrena-

[1] En nuestro caso de ejemplo, esta formula correspondería a: $Z = W_1X_1 + W_2X_2 + b$.



(a) (b)

Figura 3.2: Representación de la función de coste con un parámetro de entrada (a) y con dos (b).

miento, con el objeto de minimizar el error cometido entre la salida obtenida y la deseada. Es decir, se aprende de los errores cometidos.

Para aprender de los errores, primero tenemos que definir es que es el error. A la función que nos calcula el error en función de la salida deseada y la obtenida, la llamaremos función de perdida ($L(h(x), y)$). Modificando esta función, podemos dar más importancia a falsos positivos o a falsos negativos. Existen muchos ejemplos como $-(y \log(h(x)) + (1-y)\log(1-h(x))$ o como $\frac{1}{2}(h(x)-y)^2$.

A partir de esta, podemos definir la "función de coste" como la media de las funciones de perdida de todo el conjunto de datos de entrenamiento.

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} L(h(x^{(i)}), y^{(i)})$$

Una vez que la función esta definida, hay que definir de que manera se modificaran los pesos (se aprenderán los errores). Para esto, lo más usual es utilizar el *Gradient Descent*. Imaginemos ahora que estamos entrenando una red con un solo parámetro de entrada. Si representamos su función de coste, nos quedaría algo como la figura 3.2a. El objetivo del entrenamiento sería llegar al punto más bajo de la curva, donde el error es mínimo. Para lograr esto, seguiremos la dirección de la derivada, lo que nos llevará siempre "cuesta abajo". El tamaño del salto que demos en cada iteración dependerá de un parámetro denominado *Learning Rate* (α). Si el valor de α es muy grande, puede que en lugar de llegar al mínimo nos alejemos, y si es muy pequeño puede que el aprendizaje sea demasiado lento. Normalmente es recomendable probar varios valores[2]. Como se ve en la figura 3.2b, con más dimensiones la idea es la misma, ir siempre en dirección de la pendiente hasta llegar al mínimo. Matemáticamente, la actualización de los pesos quedaría de esta manera:

$$W_i = W_i - \alpha \frac{\partial J(W_i)}{\partial W_i}$$

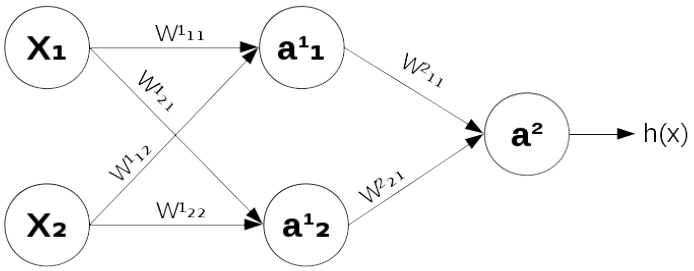


Figura 3.3: Representación de un sencillo Perceptrón Multicapa.

3.2. Función de activación

La función de activación es el último paso del computo que realiza cada neurona dentro de una red. La gente a menudo piensa que carecen de importancia, pero son fundamentales para romper la linealidad de la red.

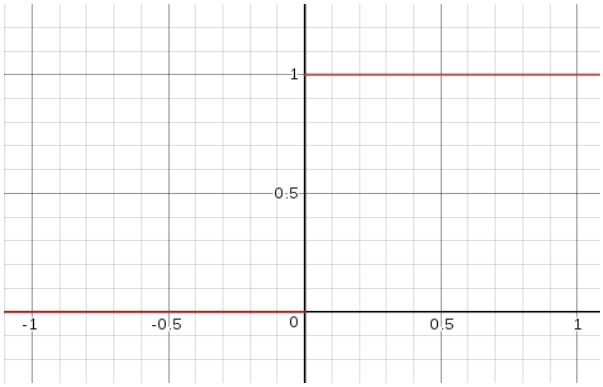
La figura 3.3 dispone de tres neuronas encargadas de realizar cálculos. Supongamos eliminamos las funciones de activación de todas ellas. En ese caso, la salida de las neuronas de la capa oculta sería: $a_1^{[1]} = W_{11}^{[1]}X_1 + W_{12}^{[1]}X_2 + b_1^{[1]}$ y $a_2^{[1]} = W_{21}^{[1]}X_1 + W_{22}^{[1]}X_2 + b_2^{[1]}$. Si seguimos propagando los cálculos obtendríamos que $a^{[2]} = W_{11}^{[2]}a_1^{[1]} + W_{21}^{[2]}a_2^{[1]} + b^{[2]}$. Si realizamos las substituciones y cálculos pertinentes, obtenemos que la salida de la red es: $(W_{11}^{[2]}W_{11}^{[1]} + W_{21}^{[2]}W_{21}^{[1]})X_1 + (W_{11}^{[2]}W_{12}^{[1]} + W_{22}^{[2]}W_{21}^{[1]})X_2 + (W_{11}^{[2]}b_1^{[1]} + W_{21}^{[2]}b_2^{[1]} + b^{[2]})$ que vuelve a ser una función del tipo: $aX_1 + bX_2 + c$, es decir, una función lineal (igual que si hubiéramos conectado las entradas directamente con la salida).

Función de salto

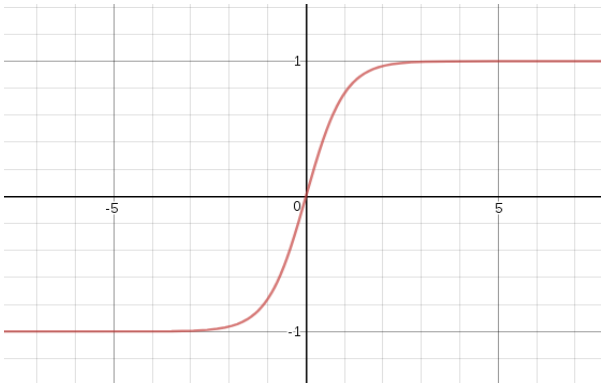
Otra faceta importante —retomando la analogía con el cerebro humano— es decidir si la neurona debería comunicar el impulso o no. Es decir, si debería activarse o no. Con esta intuición surge una de las primeras funciones de activación utilizadas, la función de salto (figura 3.4a). La idea es la siguiente: definir un *threshold*, y darle el valor uno a partir de este punto y cero antes de este punto. Es muy sencilla y no da lugar a confusiones. Pero tiene un problema, se intentará escenificar con el siguiente ejemplo:

Supón que estas creando un clasificador binario, en el que tienes que responder "si" o "no" (1 o 0). Esta función de activación podría resolver el problema, pero supón que tienes varias clases y la función devuelve 1 para más de una clase. ¿Con cual de todas nos quedamos? Hubiera sido mejor si la neurona nos devolviese un porcentaje de activación. Con esa idea surgen el resto de funciones.

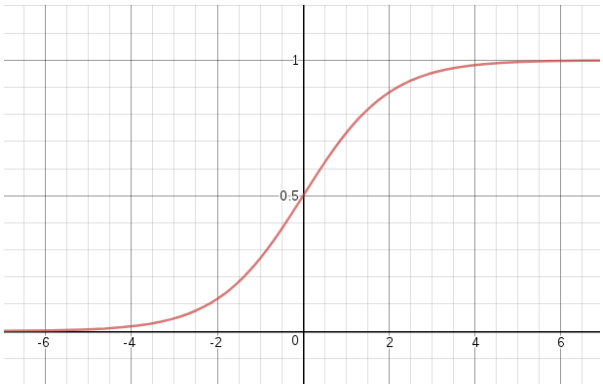
[2]Se suele recomendar probar: 3, 1, 0.3 ... 0.003, 0.001.



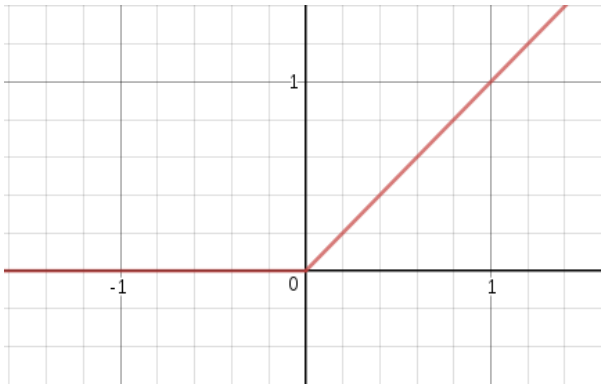
(a) Función de salto (*threshold* = 0).



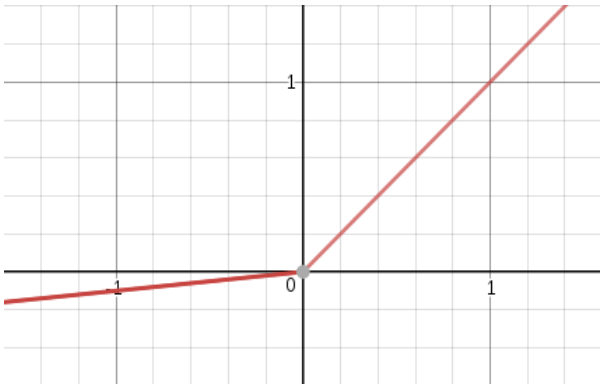
(b) Función tangente hiperbólica.



(c) Función sigmoidea.



(d) Función ReLU.



(e) Función Leaky ReLU.

Figura 3.4: Funciones de activación

Función sigmoidea

Una de las funciones más utilizadas es la función sigmoidea (figura 3.4c). Su expresión matemática es de la siguiente manera:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Su apariencia es como la función de salto, solo que suavizando la discontinuidad. Esta función es ideal para las neuronas de salida en un problema de clasificación, ya que los valores siempre están entre 0 y 1. Además, resuelve todos los problemas planteados hasta el momento, pero aun tiene algún aspecto que se puede mejorar. En cuanto el valor de Z se aleja del 0, la función se vuelve muy plana y el gradiente muy bajo, por lo que le resulta muy difícil a una neurona aprender algo cuando se encuentra en esa zona.

Función tangente hiperbólica

Otra de las funciones mas utilizadas es la tangente hiperbólica (figura 3.4b). Su expresión matemática es la siguiente:

$$\sigma(x) = \frac{2}{1 + e^{-2x}} - 1$$

Su forma es muy similar a la de la función sigmoidea, de hecho, $tanh(x) = 2sigmoid(2x) - 1$, por lo tanto tiene características similares a esta. En este caso, el recorrido es (-1, 1) y la derivada se aproxima más rápido a cero, por lo que el problema de la función sigmoidea se amplifica un poco en la tangente hiperbólica.

ReLU

Más tarde se comenzo a utilizar tambien la funcion ReLU (figura 3.4d) y actualmente esta empezando a coger mucha fuerza.

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

Como las otras funciones, rompe la linealidad, por lo que puede clasificar cualquier función. Además, hay un fenómeno denominado *sparsity of the activation*, que discutiremos mas tarde, que se produce de forma espontánea por la naturaleza de esta función. Pero lo que hace que sea realmente competitiva con respecto a las otras es su bajo coste computacional, algo que es realmente importante a la hora de entrenar Redes Neuronales profundas. A pesar de todas estas ventajas, ReLU también tiene un defecto. Debido a la parte horizontal de la función, el gradiente puede valer 0, por lo que una vez que una neurona entra en esa zona, no puede salir de ahí. Dicho con otras palabras, dejará de aprender. Tras ciertas iteraciones, el proceso de aprendizaje podría llevar a varias neuronas a ese estado, teniendo gran parte de la red en un estado pasivo que deja de responder a cualquier estimulo.

Para solucionar el problema de las neuronas existen algunas funciones que modifican la parte izquierda de la función, reduciendo mucho su pendiente pero sin llegar a hacerla nula. Un ejemplo de estas funciones es la denominada Leaky ReLU (figura 3.4e).

3.2.1. ¿Cuál elegir?

Esta es una pregunta complicada, pues cada una de estas funciones tiene sus ventajas y sus desventajas. En rasgos generales, ReLU suele aproximar bien la mayoría de las funciones y de forma rápida. La función sigmoidea es ideal para las neuronas de salida —incluso de las ocultas— de una red dedicada a problemas de clasificación. La tangente hiperbólica también vine bien para ciertos problemas dado que permite a la red “perjudicar” ciertos caminos.

Conclusión, no hay una única respuesta a esta pregunta, dependerá siempre del problema al que nos enfrentemos. La mejor solución es probar con las diferentes funciones o basarse en soluciones empleadas en problemas similares [37].

3.3. Conexiones entre neuronas

Una vez descrito el completo funcionamiento de las neuronas, es el momento de describir las diferentes maneras de las que se pueden relacionar. Como se ha comentado previamente, las neuronas se suelen agrupar en capas, y son estas capas las que se relacionan unas con otras.

Totalmente conectadas

Es la manera más sencilla de conectar dos capas entre si. Consiste, simplemente, en relacionar cada neurona con todas las neuronas de la siguiente capa. El número de conexiones sería el producto entre el número de neuronas de la primera capa y las de la segunda. Este es, en ocasiones, el problema de este tipo de conexiones, que requieren un coste computacional muy alto.

Convolucionales

Este tipo de conexiones surgió con el estudio de la Visión Artificial, donde el problema de las capas totalmente conectadas se vuelve crítico. Imaginemos una imagen de 450 x 450 píxeles (resolución baja). Una Red Neuronal que recibiera imágenes de estas características necesitaría 450 x 450 x 3 (intensidad de cada color RGB) neuronas de entrada (un total de 607.500). Si conectamos esta capa con otra de tamaño similar, el número de conexiones (y por lo tanto de operaciones) ascendería exponencialmente.

De nuevo, la inspiración para este tipo de conexiones vino de nuestras neuronas. Concretamente del ojo humano. La corteza visual tiene pequeñas regiones de células que son sensibles a regiones específicas del campo visual. Esta idea fue ampliada por un fascinante experimento de Hubel y Wiesel en 1962 donde mostraron que algunas neuronas respondían solo en presencia de bordes de cierta inclinación. Hubel y Wiesel descubrieron que todas estas neuronas estaban organizadas en una arquitectura columnar y que, juntas, podían producir percepción visual. Esta idea es la base de las Redes Neuronales Convolucionales.

Lo primero que necesitamos es entender que es un filtro. Este filtro es una matriz llena de pesos que se coloca en la esquina superior izquierda de la imagen, computa el valor de la siguiente neurona y se desliza a la siguiente posición de la imagen, así hasta cubrir la imagen entera. La idea detrás de esto es que cada filtro aprende un atributo concreto y que lo busca a través de toda la imagen. Ilustraremos esta idea con un ejemplo. En la figura 3.5 se puede ver el proceso de activación de un filtro. La matriz de la izquierda es la zona de la imagen que se pretende comparar con el filtro. La matriz de la derecha es un filtro que ha aprendido a detectar una forma curva concreta. Cuando este filtro se compara con una forma como la que estamos buscando, el número obtenido es alto (la neurona se activa) y cuando no, el número es más bajo. En el ejemplo de la imagen, la primera imagen obtendría un resultado de 6600 y la segunda de 0. Aunque en el ejemplo se muestran como matrices de dos dimensiones, las matrices suelen tener tres dimensiones. En esos casos, el tamaño de la tercera dimensión, tanto de la imagen como de el filtro, debe coincidir. La combinación de una

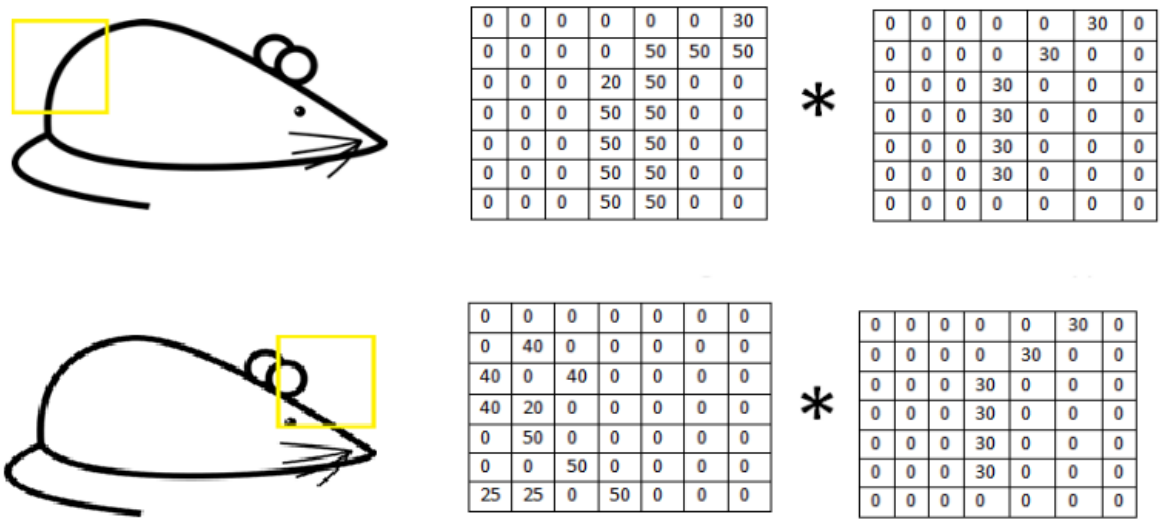
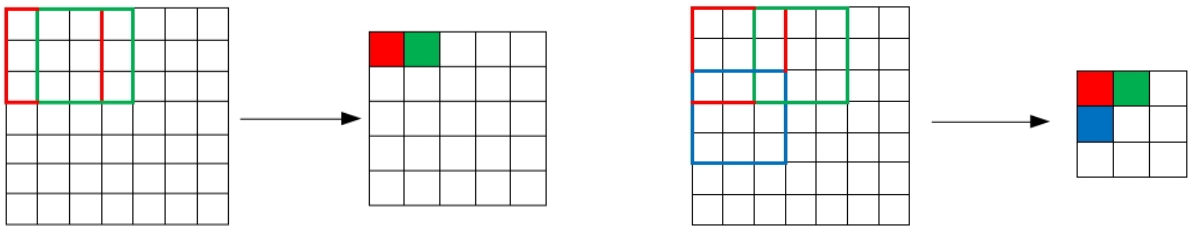


Figura 3.5: Ejemplo gráfico de un filtro de una Red Neuronal Convolucional.



(a) *stride* = 1.

(b) *stride* = 2.

Figura 3.6: Representación gráfica del *stride*.

imagen con un filtro siempre da como resultado una matriz de dos dimensiones. La razón por la que las matrices son tridimensionales es que se emplean diversos filtros —de las mismas características, pero con pesos diferentes— y el resultado de cada uno de estos filtros se apilan uno detrás de otro, dando lugar a un volumen tridimensional.

Una vez comprendido el funcionamiento básico de un filtro, se presentaran dos parámetros que permiten modificar su comportamiento. El *stride* y el *padding*.

El *stride* controla la forma en la que el filtro se mueve a través de la entrada. Como vemos en la figura 3.6, el *stride* define cuantas posiciones se salta cada filtro en el desplazamiento. También vemos como afecta la modificación del *stride* en el volumen de salida. En ambos casos el volumen de entrada es de 7 x 7 y el filtro de 3 x 3, la única diferencia es el stride. En el ejemplo a es de 1 y la capa de salida es de 5 x 5, en el ejemplo b es 2 y la salida es de 3 x 3. De aquí podemos inferir que la relación entre la entrada y la salida es: $n^{[l]} = \frac{n^{[l-1]} - f^{[l]}}{s^{[l]}} + 1$, lo que quiere decir que el volumen siempre decrece, lo que puede ser un problema en el caso de que necesitemos una Red Neuronal muy profunda (con muchas capas). Otro problema es que los laterales de la imagen quedan muy poco “observados”, mientras que el centro entra en numerosas iteraciones del filtro. Además, solo hay unos pocos *stride* que encajan perfectamente en la entrada (imagine se el lector un *stride* igual a 3 en el ejemplo de la imagen) y por lo tanto, solo hay unos pocos valores de salida disponibles. Para solventar estos tres problemas surge el *padding*.

El *padding* no es más que un contorno de ceros al rededor de la matriz. De esta manera aumentamos el tamaño de la entrada, por lo que la salida no tiene porque disminuir de tamaño. Además, centramos la información relevante, por lo que los laterales de la entrada caen dentro de la mirada de muchos más filtros. Por otra parte, la relación entre entrada y salida se vuelve mucho más flexible, por lo que podemos jugar con un mayor numero de tamaños de filtro y de *stride*. Añadido este nuevo concepto, la formula general que relaciona tamaño de entrada con tamaño de salida quedaría de la siguiente forma:

$$n^{[l]} = \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1$$

Esto permite utilizar filtros de diferentes tamaños y características, ajustando el *padding* para que las dimensiones de salida sean todas iguales. No es una técnica muy utilizada, pero en algunas redes que estudiaremos más adelante son fundamentales [10].

Pooling

Se puede considerar un tipo especial de capa convolucional, pero se ha preferido poner por separado porque sus operaciones y su objetivo son diferentes. Se aplica algo similar al filtro. Normalmente el *stride* es igual al tamaño del filtro y el *padding* es igual a 0. La diferencia con los filtros convolucionales es que en lugar de realizar una suma ponderada, estos filtros realizan una operación diferente. La operación más común es devolver el número mayor dentro del filtro, la idea detrás de esto es que un número grande es haber encontrado una característica, y eso es lo que realmente importa. Otra operación que se utiliza en alguna ocasión es la media de todos los elementos del filtro. Su objetivo es reducir el tamaño de la capa de la red sin perder ninguna información relevante. Normalmente las capas convolucionales ajustan el *padding* para que la capa de entrada sea igual a la de salida y son estas capas las que se encargan de ir reduciendo el tamaño de las capas a lo largo de la red.

3.4. Arquitecturas

Una vez presentado de que manera se relacionan las capas unas con otras, es hora de conocer algún ejemplo de arquitecturas complejas que utilizan estas conexiones para resolver problemas reales. Centrando la atención en las redes utilizadas en el campo de la Visión Artificial, que presentaron mayores avances. Hasta llegar a las arquitecturas que se suelen utilizar en la actualidad [9].

3.4.1. LeNet-5

Esta red es la pionera en el novedoso mundo de las Redes Convolucionales (Cuadro 3.1). Su objetivo era clasificar dígitos escritos a mano. Transformaron las imágenes de entrada a escala de grises, por eso la capa de entrada tiene profundidad uno. A pesar de ser la primera Red Neuronal Convolucional, hay ciertas características que se mantienen en la actualidad.

Tipo de capa	Tamaño salida	Tamaño filtro	stride	padding
Entrada	32x32x1	-	-	-
Convolucional (sigmoidea)	28x28x6	5x5	1	0
Pooling (media)	14x14x6	2x2	2	0
Convolucional (sigmoidea)	10x10x16	5x5	1	0
Pooling (media)	5x5x16	2x2	2	0
Totalmente conectada (sigmoidea)	120	-	-	-
Totalmente conectada (sigmoidea)	84	-	-	-
Salida	10	-	-	-

Cuadro 3.1: Arquitectura LeNet-5.

Una de las cosas que siguen funcionando de manera similar es que las capas suelen ir estrechándose en cuanto a altura y anchura, pero aumentando en profundidad. Otra idea heredada de esta arquitectura es que la primera parte de la red consta de capas convolucionales y pooling intercaladas, y la última parte son unas pocas capas totalmente conectadas.

Esta arquitectura contiene algunos errores que es importante comentar. La función de activación utilizada es la sigmoidea, cuando la función ReLU es más rápida, cosa muy importante cuando se trata de Redes Neuronales de gran envergadura. Otra mejora que podríamos proponer es cambiar la operación del *pooling*, sustituyendo la media por el máximo, que mantiene mejor la información de los atributos encontrados [20].

3.4.2. AlexNet

El *paper* en el que se presenta esta arquitectura es uno de los más influyentes en este área de conocimiento. Consiguió convencer a la comunidad de que las Redes Convolucionales profundas son una gran solución para problemas de Visión Artificial. La arquitectura de esta red es muy similar a LeNet-5, pero mucho más grande. Esta contaba con unos seis mil parámetros y AlexNet cuenta con aproximadamente seis millones. Mejoró mucho la eficiencia sustituyendo las funciones de activación por ReLUs y entrenando la red a través de dos GPUs, de esta manera fue capaz de entrenar la red con más de quince millones de imágenes a color [19].

3.4.3. VGG-16

La aportación de esta Red Neuronal fue la simplicidad de su arquitectura a pesar de su profundidad. Todas las capas convolucionales utilizan filtros 3x3, con *stride* $= 1$ y un *padding* tal, que la salida de cada capa siempre es igual a la entrada. Las capas de *pooling* siempre utilizan filtros 2x2, *stride* $= 2$ y la operación que selecciona el máximo. Además, este *paper* es muy interesante porque nos permite seleccionar configuraciones cada vez más profundas en función de nuestras necesidades, yendo los parámetros desde 133 millones para la menos profunda hasta 144 la más profunda. Todas estas configuraciones acaban con tres capas completamente conectadas. En el cuadro 3.3 se

Tipo de capa	Tamaño salida	Tamaño filtro	stride	padding
Entrada	227x227x3	-	-	-
Convolucional (ReLU)	55x55x96	11x11	4	0
Pooling (max)	27x27x96	3x3	2	0
Convolucional (ReLU)	27x27x256	5x5	1	same
Pooling (max)	13x13x256	3x3	2	0
Convolucional (ReLU)	13x13x384	3x3	1	same
Convolucional (ReLU)	13x13x384	3x3	1	same
Convolucional (ReLU)	13x13x256	3x3	1	same
Pooling (max)	6x6x256	3x3	2	0
Desenrollar	9219	-	-	-
Totalmente conectada (sigmoidea)	4096	-	-	-
Totalmente conectada (sigmoidea)	4096	-	-	-
Salida	1000	-	-	-

Cuadro 3.2: Arquitectura AlexNet.

puede ver la configuración D, una de las más profundas. Cabe destacar la forma tan sistemática en la que decrece n_H y aumenta n_C [33].

3.4.4. ResNet

Este *paper* comienza mostrando que, a pesar de lo que se creía, hacer una Red Neuronal más profunda no siempre garantiza mejores resultados. De hecho, cuando superamos cierto número de capas, esta comienza a funcionar cada vez peor. Sin embargo, con las ideas propuestas con esta arquitectura, podemos realizar Redes Neuronales tan profundas como queramos, mejorando siempre su funcionamiento.

La innovación de esta red son los denominados bloques residuales. Consisten en dos capas en las que se une la entrada de la primera con la salida de la segunda (antes de realizar la función de activación). De esta manera, la salida del bloque residual se podría definir de la siguiente manera:

$$a^{[l]} = \sigma(z^{[l]} + a^{[l-2]})$$

Estos bloques son muy útiles en redes muy profundas porque son capaces de aprender la matriz identidad de forma muy sencilla, y por lo tanto, ignorar todas las capas que no sean necesarias. Para ver esto más fácilmente desarrollaremos la función presentada antes: $a^{[l]} = \sigma(w^{[l]}a^{[l-1]}+b^{[l]}+a^{[l-2]})$. Bastaría con tener valores muy bajos en la matriz de pesos y en el *bias* para que $z^{[l]}$ tuviera un valor nulo, y al utilizar la función de activación ReLU, $a^{[l]}$ seria equivalente a $a^{[l-2]}$. Y por lo tanto, es sencillo también añadir pequeños cambios, que no modifiquen demasiado $a^{[l-2]}$ pero que añadan cierta mejora [16].

Tipo de capa	Tamaño salida	Tamaño filtro	stride	padding
Entrada	224x224x3	-	-	-
Convolucional (ReLU) x2	224x224x64	3x3	1	same
Pooling (max)	112x112x64	2x2	2	0
Convolucional (ReLU) x2	112x112x128	3x3	1	same
Pooling (max)	56x56x128	2x2	2	0
Convolucional (ReLU) x3	56x56x256	3x3	1	same
Pooling (max)	28x28x256	2x2	2	0
Convolucional (ReLU) x3	28x28x512	3x3	1	same
Pooling (max)	14x14x512	2x2	2	0
Convolucional (ReLU) x3	14x14x512	3x3	1	same
Pooling (max)	7x7x512	2x2	2	0
Desenrollar	4096	-	-	-
Totalmente conectada (ReLU)	4096	-	-	-
Totalmente conectada (ReLU)	4096	-	-	-
Salida	1000	-	-	-

Cuadro 3.3: Arquitectura VGG-16.

3.4.5. Inception Network

Cuando se diseña la arquitectura de una Red Convolucional, hay que decidir constantemente el tamaño de los filtros que se quieren utilizar. La idea detrás de esta red es: "¿Por qué no utilizarlos todos?".

La forma de conseguir realizar convoluciones con filtros de diferentes tamaños es gracias a ajustar el *padding* para que la salida de todos los filtros sea del mismo tamaño. Después, solo tendríamos que apilar las salidas unas detrás de otras. En este ejemplo concreto, la GoogLeNet, se utilizaros 64 filtros de tamaño 1x1, 128 de tamaño 3x3, 32 de tamaño 5x5 y 32 de una clase especial de *pooling* que reduce la dimensión n_C en lugar de n_H. Esta idea tiene un problema, el coste computacional se vuelve muy alto. Como solución se añadió un filtrado de tamaño 1x1, que reducía la profundidad de la capa anterior, delante de los filtros de tamaño 3x3 y 5x5. Lo que, a pesar de dividir el número de operaciones entre diez, no hizo que la red clasificase peor las imágenes.

El avance que proporcionó esta arquitectura es que hizo ver que las capas convolucionales no siempre tienen que ir una detrás de otra, y que una arquitectura más creativa puede proporcionar buenos resultados [34][21].

3.4.6. R-CNN

Las arquitecturas mostradas hasta el momento tenían como objetivo la clasificación de imágenes, es decir, dada una imagen, decir que es lo que se encuentra en ella. Cuando se abordaba un problema de detección de objetos, es decir, detectar en que zona exacta de la imagen se encuentra y decir de

que objeto se trata, lo que se hacía era deslizar ventanas de diferentes tamaños por la imagen y suministrar a la red los bits que se encontraban dentro de la ventana. De esta manera, cuando la red reconocía —con un alto grado de certeza— que ese cacho de imagen era un objeto conocido, se asumía que ahí se encontraba ese objeto.

La gran innovación que propuso este *paper*[14] fue añadir una primera fase en la que se proponían las regiones en las que era más probable que se encontrase un objeto. De esta manera, en lugar de buscar con regiones de diferentes tamaños a lo largo de toda la imagen, solo se tenían que probar ciertas regiones más propensas. Estas regiones se proponían mediante un proceso de vectorización que no comentaremos en este trabajo.

Estos resultados eran muy prometedores, por lo que gozaron de gran popularidad durante varios años. A pesar de esto, eran demasiado lentos, por lo que surgieron un par de mejoras que merece la pena comentar. Un año más tarde, se presento *Fast R-CNN*[13]. Su innovación fue clasificar todas las regiones a la vez por medio de una Red Convolucional, lo que mejoro mucho la velocidad de ejecución. Tan solo unos meses más tarde, se publico *Faster R-CNN*[32], la cual, no solo clasificaba las regiones por medio de Redes Convolucionales, sino que también utilizaba una de estas redes para proponer las regiones.

La gran evolución de esta red fue comenzar a tratar los problemas de clasificación de imágenes y de detección de objetos como problemas diferentes. Dejando de lado la "fuerza bruta" utilizada hasta el momento y comenzando a buscar los objetos de una manera más "inteligente".

3.4.7. YOLO

El problema de deslizar una ventana a través de la fotografía es que muchas veces no encaja ni tamaño ni la posición exactamente. También puede ser que la forma del objeto no sea un cuadrado, sino que sea más alargada o más achatada. Por esta razón, es muy difícil conseguir un recuadro exacto alrededor del objeto. El algoritmo que se presenta a continuación resuelve este problema utilizando exclusivamente Redes Convolucionales. Como su propio nombre indica, "You only look once", lo hace de una sola pasada, lo que la convierte en una solución realmente eficiente.

La gran aportación de este algoritmo es la forma en la que se etiquetan las imágenes. Una vez hecho esto, solo habría que entrenar una Red Convolucional de la forma tradicional para obtener los resultados esperados. Para realizar el etiquetado, se divide la entrada en varios cuadrantes (normalmente 19x19). La red devolverá un vector solución para cada uno de estos cuadrantes. Este vector contiene los siguientes valores: El primero (P_C) indica la probabilidad de que se encuentre un objeto en ese cuadrante. Después, dos valores (b_X y b_Y) definen la posición relativa del objeto. Los dos siguiente (b_H y b_W) definen su altura y anchura relativas. Por ultimo, un valor para cada clase estudiada (C_n), iniciando a cual pertenece esta solución. En el caso de que P_C sea menor que un *threshold* previamente definido (normalmente 0.5) —es decir, que no se encuentre ningún objeto en dicho recuadro— los valores del resto del vector no serán analizados. A la hora de decidir en que recuadro incluir cada objeto, hay que mirar el centro del objeto. Será en este recuadro en el que se incluirá el objeto, y será la posición de este centro la que se almacenará en b_X y b_Y. En cuanto

al tamaño, es probable que muchos de estos objetos sobresalgan de su cuadrante, por lo que está permitido que b_H y b_Y tengan valores mayores de 1. Estos vectores estarán todos en un mismo bloque. En el caso de definir una cuadricula de 19x19 y tener tres clases diferentes que clasificar, este bloque tendria de tamaño 19x19x8. El número de cuadrantes por el tamaño del vector solución. Todo este cuadrante se calcularía de una sola iteración, como nos vienen acostumbrando las Redes Convolucionales.

Intersection over union

Para este tipo de soluciones, diseñadas exclusivamente para resolver problemas de detección de objetos, se necesita una nueva medida para comparar dos *bounding boxes*. Y por lo tanto, para calcular el error de una solución. Esta nueva medida se llamó *Intersection over union* (IoU) y mide la superficie que tienen en común los *bounding boxes* que se deseen comparar. Esto se calcula dividiendo, como su propio nombre indica, la intersección entre la unión de la superficie de ambos *bounding boxes*.

$$IoU = \frac{b_1 \cap b_2}{b_1 \cup b_2}$$

Una detección perfecta devolvería un valor de 1. Normalmente se toma correcta una solución mayor que 0.5, pero este *threshold* es modificable.

Non-max supresion

Cuando ponemos una cuadricula con muchos cuadrantes, es probable que varios clasifiquen un mismo objeto como suyo. Para solucionar este problema, se utiliza la técnica denominada *non-max supresion*. Esta técnica consiste en eliminar los *bounding boxes* que tengan menor IoU. El algoritmo procedería de la siguiente manera: Primero se eliminan todos los *bounding boxes* con un valor de P_C menor que el *threshold* que se haya definido. Después, por cada clase que estemos clasificando, seleccionamos la solución con mayor probabilidad y eliminamos el resto de soluciones de la misma clase con un IoU mayor que cierto *threshold*. Este último paso se repite hasta que todas las soluciones hayan sido analizadas.

Anchor boxes

En ocasiones, puede suceder que dentro del mismo recuadro entre el centro de varios objetos. ¿Cual de todos ellos se debería clasificar?. La respuesta es clara: Todos. Para lograr esto, se amplía el número de vectores solución por cada recuadro. Esta ampliación del vector se haría añadiendo un vector de las mismas características detrás del actual. Se añadirían tantos vectores como soluciones se desee suministrar por recuadro.

Con esta solución surge otro problema. Cuando queremos añadir la etiqueta de un objeto en un recuadro, ¿en cual de los vectores disponibles tenemos que hacerlo?. Esto se resuelve definiendo un

anchor box por cada vector solución dentro de un mismo recuadro. De esta manera, incluiríamos la solución en el vector perteneciente al *anchor box* con un mayor IoU con respecto al objeto a etiquetar.

Esta solución, además, nos permite especializar más cada uno de los *anchor boxes*. Por ejemplo, si estamos clasificando coches y personas, y disponemos de un anchor vertical y otro horizontal. El horizontal se especializará en clasificar coches y el vertical en clasificar humanos.

Para seleccionar estos *anchor boxes*, la técnica más utilizada suele ser definirlos a mano, pero la mejor manera de definirlos es utilizar un algoritmo de clasificacion no supervisada denominado K-means. El cual nos definiria las posiciones y los tamaños que minimizarian el error en el conjunto de entranamiento.

Este algoritmo es el estado del arte en el área de la Visión Artificial, haciendo de la detección de objetos un problema tratable en tiempo real [25].

3.5. Diagnóstico y corrección de errores

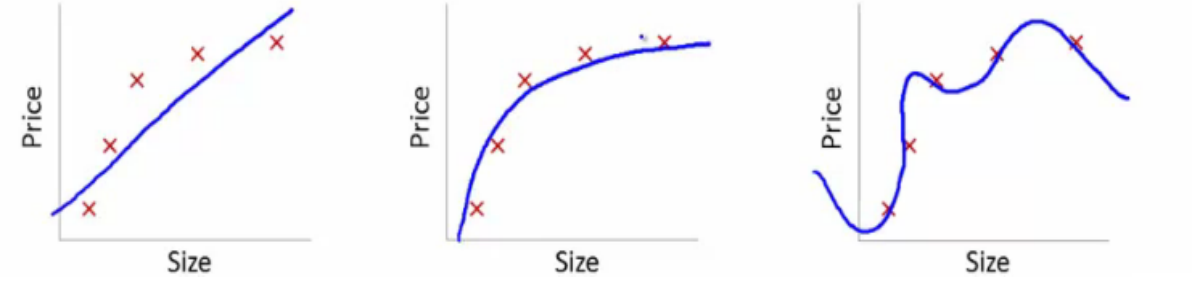
Sin lugar a duda, la parte más importante a la hora de conseguir que una Red Neuronal funcione adecuadamente es saber cómo detectar los errores que comete el modelo. Si, por ejemplo, para entrenar una red se recolecta un set de datos, se decide la arquitectura a utilizar, se ajustan los parámetros y se ejecuta el aprendizaje durante 10.000 iteraciones, ¿cómo se podrían mejorar los resultados obtenidos?: ¿entrenarla durante más tiempo?, ¿aumentando el set de datos?, ¿cambiando la arquitectura?. Aunque la mayoría de estas acciones es difícil que empeoren el resultado, algunas podrían llegar a hacerlo. Y desde luego, todas ellas serían una perdida de tiempo si no se conoce el error a solventar.

3.5.1. Tipos de error

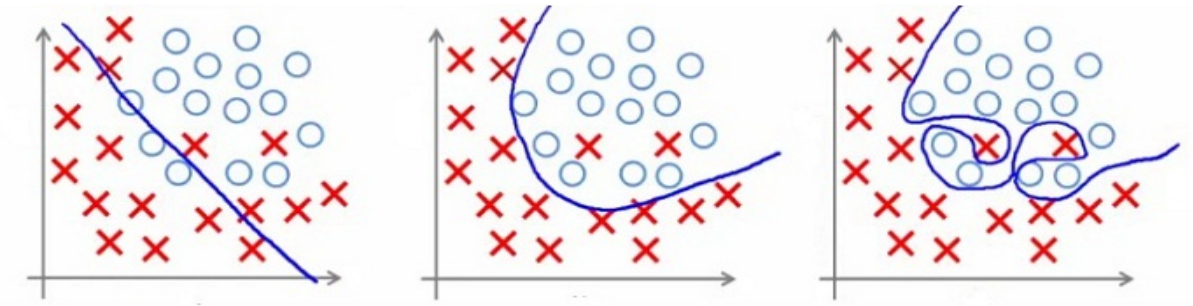
Las dos principales fuentes de error son el *bias* y la varianza, también llamados *overfitting* y *underfitting*. Saber reconocer en cual de los dos está errando es la mayor fortaleza que se puede tener a la hora de construir un modelo de *Machine Learning*. Se explicará brevemente el significado de ambos conceptos.

Los problemas de varianza se deben a que se han aprendido “tan bien” los datos de entrenamiento, que el modelo ya no es capaz de generalizar el conocimiento y aplicarlo a los casos de prueba. Esto se puede deber a que se haya entrenado la red durante demasiado tiempo, que la arquitectura sea muy compleja o que tendamos pocos datos de aprendizaje. Un problema de *bias* es todo lo contrario; aun no se ha aprendido los casos de entrenamiento y por lo tanto, ni se clasifican bien los de entrenamiento ni los de prueba. La mejor forma de comprender esto es con un ejemplo (Figura 3.7).

En la figura 3.7a podemos ver un ejemplo de regresión lineal (muy similar al funcionamiento de



(a) Regresión.



(b) Clasificación.

Figura 3.7: Ejemplos de *overfitting* y *underfitting* [27].

una Red Neuronal). El objetivo de este modelo es predecir el precio de una casa en función de su tamaño. Se puede ver a simple vista que "la realidad" se comporta similar a una función logarítmica, como en la gráfica del medio. La gráfica de la izquierda correspondería a un error de *bias*, porque se puede ver que una recta es demasiado simple para predecir el precio de una casa. En la función de la derecha, se puede ver que se ha intentado reducir tanto el error, que es muy probable que, cuando se intente probar este modelo con un caso real, el errores sea mucho más alto que en la gráfica del medio. Es decir, padece un error de varianza.

La figura 3.7b es un caso muy similar, solo que en este caso es un problema de clasificación. Se puede ver que, en este caso, tenemos dos parámetros de entrada (los dos ejes) y tenemos que decidir entre círculos y equis. Por poner un ejemplo, este modelo podría intentar predecir el sexo de una persona en función de su altura y su peso. Se puede ver que la línea que mejor describe la naturaleza es la gráfica del centro; pero, como en todo, hay excepciones. La gráfica de la izquierda es demasiado simple, y se puede esperar que clasifique mal gran número de ejemplos del set de prueba (*bias*). Sin embargo, la gráfica de la derecha clasifica bien el 100 % de los ejemplos de prueba, pero parece un modelo demasiado complicado para que la naturaleza se comporte así (varianza), y el próximo ejemplo que se acerque a la zona central corre un gran riesgo de ser mal clasificado.

Una manera de ver como evolucionan los errores de un modelo a lo largo de su aprendizaje sería representarlo. En la figura 3.8 se puede ver como se modifica el error tanto del conjunto de entrenamiento como de el conjunto de pruebas. El eje de abscisas corresponde al número de iteraciones que realiza la red durante el aprendizaje. El de ordenadas indica el error que comete cada conjunto. Al inicio del entrenamiento los dos errores están muy altos; acertar aquí sería puro azar. En esta zona, el modelo padece de un error de *bias*. Una vez que se va entrenando la red con más y más ejemplos, los errores van disminuyendo. Evidentemente, el error sobre el conjunto de entrenamiento disminuye más rápido, pues son los datos sobre los que esta aprendiendo la red. Aun así, en esta

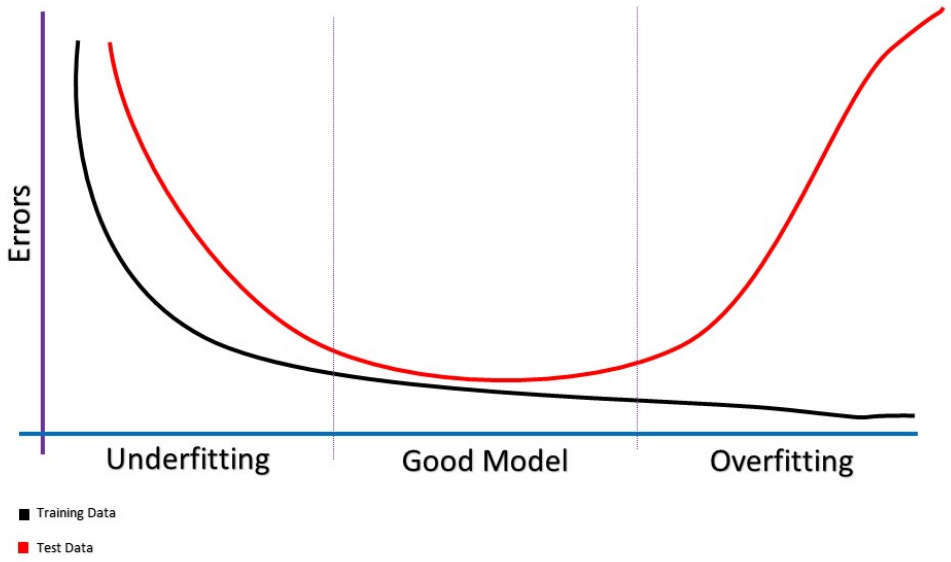


Figura 3.8: Curva de aprendizaje.

zona central, el modelo generaliza bien el conocimiento y clasifica de manera bastante similar el conjunto de prueba. Esta seria la zona idónea. Si seguimos entrenando más y más la red, el error del conjunto de entrenamiento seguirá disminuyendo cada vez mas, pero empezara a generalizar mal el conocimiento y a clasificar cada vez peor el conjunto de prueba. En esta zona, el conjunto sufre un problema de varianza. Con esta gráfica podemos hacernos a la idea de que tipo de correcciones se deben realizar para mejorar cada uno de los errores y como detectarlos.

3.5.2. Diagnóstico

Como hemos visto en la figura 3.8, cuando una red sufre de *bias*, los dos errores están próximos entre ellos y cometiendo un error alto. Cuando el modelo sufre de varianza, los errores están alejados entre ellos, y el error del conjunto de entrenamiento es bastante bajo. Estas son las ideas sobre las que nos centraremos para diagnosticar un modelo de *bias* y/o de varianza.

Lo primero que se debe realizar es marcar el mínimo error al que creemos que puede llegar nuestra red. Porque podríamos creer que estamos padeciendo de *bias*, pero en realidad el problema es tan complejo que ya estamos rozando una de las mejores soluciones a las que podemos aspirar, y es más grave el problema de varianza. Para definir este umbral, se suele utilizar el "nivel de rendimiento humano". Esta medida indicaría cual es el error que un grupo de humanos expertos cometería en este mismo problema. Si el problema es diferenciar entre gatos y perros, el "nivel de rendimiento humano" seria del 0 %, en el caso de que el problema sea predecir la evolución de la bolsa, este nivel estaría mucho más alto.

Una vez que tenemos definido el "nivel de rendimiento humano", solo nos quedaría medir los errores de nuestro modelos sobre ambos conjuntos. Una gran diferencia entre el error de entrenamiento y el de pruebas indicaría un error de varianza. La diferencia entre el error de entrenamiento

y el “nivel de rendimiento humano” nos indicaría el error de *bias* que padece el modelo. De esta manera, invertiremos nuestros recursos de manera mucho más eficiente.

3.5.3. Corrección

El *Machine Learning* es un proceso, ante todo, empírico. Es cierto que hay diversas ideas que nos permiten saber en qué estamos fallando y cómo solucionarlo. Pero al final es un proceso de “ensayo y error”, después de cada modificación tenemos que medir el error y analizar los resultados. Para ayudarnos en ese proceso, utilizaremos una técnica denominada ortogonalización. La idea detrás de esta técnica consiste en realizar modificaciones que solo mejoren el *bias* o la varianza, pero no los dos a la vez, para así poder ajustar mejor las diferentes técnicas y los diferentes parámetros.

Algo importante a la hora de mejorar un modelo, es definir qué es mejor. Así poder, de forma rápida, decidir si una modificación hizo que nuestra red mejorase o empeorase. Imagínese el lector que tenemos que realizar un problema de detección de objetos en el que el tiempo de ejecución es critico. ¿Qué tiene más importancia, mantener bajo el IoU o el tiempo de ejecución?. ¿Y cuánto de mejor es uno frente al otro?. Lo más recomendable en este punto sería definir una función que relacionase todas las métricas que nos interesan, para así poder comparar dos modelos de forma rápida y eficiente. Por ejemplo, si consideramos que el IoU es el doble de importante que la velocidad, nuestra función podría ser la siguiente: $IoU - 0{,}5 * \text{tiempo\_de\_ejecución}$. Otra alternativa podría ser marcar unos umbrales máximos de error en todas las métricas menos en una. Y mejorar la métrica libre al máximo, siempre que no superemos los umbrales. En el ejemplo anterior podríamos definir un máximo de un segundo para el tiempo de ejecución y mejorar el IoU todo lo que podamos sin superar este tiempo de ejecución.

A la hora de corregir los errores hay que intentar bajar el *bias* al máximo y una vez que este error este bajo, comenzar a acercar el error sobre el conjunto de prueba a ese nivel. Después, habría que probar como clasifica los errores en el mundo real, e intentar reducirlos al máximo. A continuación mostraremos una serie de posibles soluciones a los diferentes tipos de error.

Si nuestro modelo sufre de *bias* las principales soluciones podrían ser:

- Entrenar el modelo durante más tiempo. De esta manera se le permite continuar memorizando las características de los datos de entrada.
- Aumentar el tamaño de nuestro modelo. En el caso de ser una Red Neuronal, podríamos tanto añadir más neuronas en las capas ocultas como añadir más capas.
- Cambiar la arquitectura por una más compleja que sea capaz de ajustar funciones más difíciles.
- Reducir el valor del *learning rate* para que el entrenamiento sea más lento pero más preciso.
- En el caso de usar *mini-batch*, aumentar el tamaño del *batch* para que la red lleve siempre una dirección más precisa.

Existen más técnicas para mejorar el *bias*, pero lo importante es entender que el objetivo es apren-

der los datos de manera más intensa.

Para mejorar un modelo que sufre de varianza, podríamos realizar las siguientes mejoras:

- Aumentar el tamaño del conjunto de datos de entrenamiento.
- Añadir regularización a la función de coste.
- Como último recurso podríamos analizar la curva de aprendizaje y dejar de entrenar el modelo en el mejor punto. El problema de esta medida es que modifica tanto el *bias* como la varianza y no nos permite modificar los parámetros de forma independiente.

Por ultimo, comentar que si estamos teniendo un error muy desigual entre el conjunto de pruebas y la realidad, se puede deber a que la calidad de los datos sea baja. Una recomendación también puede ser revisar los errores cometidos "a mano". De esta manera se puede tener una idea mucho más precisa de por qué esta fallando el modelo. Por ejemplo, si en una Red Neuronal que debe diferenciar entre gatos y perros, la mayoría de los errores son gatos blancos confundidos por perros, sería una buena solución introducir más gatos blancos en el conjunto de datos.

3.6. Técnicas de Optimización

Los algoritmos de aprendizaje vistos hasta ahora se podrían considerar la base, la forma más sencilla entrenar una arquitectura concreta. Pero existen gran cantidad de técnicas de optimización que permiten tanto reducir el *bias* o la varianza, como acelerar el aprendizaje. Se comentarán a continuación algunas de las más importantes.

3.6.1. Regularización

Es una de las técnicas más utilizadas para reducir la varianza (si no la más). Se basa en dos ideas: que la realidad se suele ajustar con formulas sencilla y no con unas muy enrevesadas (como se puede ver en la figura 3.7) y que las formulas cuyos coeficientes (pesos en el caso de las Redes Neuronales) son menores, dan lugar a modelos más sencillos. De primeras no es muy intuitivo, pero que en la práctica da muy buenos resultados.

La forma de conseguir unos pesos menores es penalizando los valores altos, sumando a la función de coste un valor proporcional a la suma de todos los pesos de la red. De esta manera, al entrenar la Red Neuronal, se intentará reducir tanto el error como los pesos. Este valor proporcional depende de un nuevo hiper-parámetro que llamaremos λ; cuanto mayor sea, más sencilla será la función y por lo tanto menos sobre-ajustada. Hay que tener cuidado de no dar un valor demasiado alto de λ, pues esto podría aumentar el *bias*. La función de coste con regularización quedaría de la siguiente manera[3]:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(h(x^{(i)}), y^{(i)}) + \lambda \sum W^{[l]}_{n_1 n_2}$$

[3]No se escriben los tres sumatorios que acompañarían a $W^{[l]}_{n_1 n_2}$. Se deja indicado con un solo sumatorio que indica que habría que sumar todos los pesos existentes en la Red Neuronal.

3.6.2. *Batch, Mini-Batch* y *Stochastic Gradient Descent*

En la versión de *Gtradient Descent* que se presentó anteriormente, se utilizaba todo el conjunto de entrenamiento para calcular el error medio (con el que, más tarde, se actualizaban los pesos). Este es el denominado *Batch Gradient Descent*. Aunque este algoritmo sigue dando muy buenos resultados, para algún tipo de problema —o alguna etapa del aprendizaje– puede ser útil modificarlo. Comentaremos a continuación las dos modificaciones más usadas:

El denominado *Mini-Batch Gradient Descent* consiste en reducir el número de ejemplos analizados a la hora de calcular el error medio. Este conjunto de datos se denomina *batch*; su tamaño se considera un nuevo hiper-parámetro, y por lo tanto habrá que buscar el valor más apropiado. Los datos elegidos para un *batch* concreto están tomados al azar en cada iteración. Cuando se ha iterado sobre todo el conjunto de entrenamiento se dice que se ha realizado una etapa (lo que implicaría varias actualizaciones de pesos, más cuanto menor sea el tamaño del *batch*).

Con esta modificación ganamos velocidad de aprendizaje, aunque perdemos en precisión. Esto se debe a que cada uno de estos *batches* no nos llevará realmente al mínimo, sino a una aproximación de este; aunque realizará más actualizaciones de los pesos por unidad de tiempo. Decidir el tamaño del *batch* es una cuestión de compromiso entre estas dos cualidades. En la actualidad, es el *Gtradient Descent* más utilizado, aunque combinada con otra técnica denominada *momentum* (que veremos más adelante).

La última modificación, el *Stochastic Gradient Descent*, es, en realidad, un caso concreto del *Mini-Batch Gradient Descent* en el que el tamaño del *batch* es igual a uno. Esta modificación se utiliza, por ejemplo, cuando se obtienen nuevos datos en tiempo real, pero no se desea almecenarlos. Por lo tanto, cuando llega un nuevo dato, se memoriza y se descarta. En esta modificación toma especial importancia utilizar *momentum*.

3.6.3. *Transfer Learning*

El *Transfer Learning* se basa en la idea de que el conocimiento de un cierto problema puede ser útil para otro problema similar. En la práctica, esto se traduce en utilizar como datos de inicio los pesos de otra Red Neuronal encargada de solucionar un problema similar. Esto se utiliza para dos objetivos diferentes:

El primero consiste en acelerar el proceso de aprendizaje. Puesto que la red ya resuelve un problema similar, solo deberá aprender las diferencias entre ambos problemas. Muy útil para Redes Neuronales profundas, sobre todo si no se dispone de un máquina rápida donde entrenarla. Y en cualquier caso, hará del aprendizaje una labor más energética y por lo tanto, más económica. Además, al entrenar los modelos más rápidamente, permite probar en el mismo tiempo mayor número modelos diferentes; lo que siempre se traduce por mejores resultados.

El segundo es reducir la varianza. En ocasiones nuestra red no es capaz de generalizar suficiente los datos. Por esta razón, comenzar con un problema diferente o más general es muy útil a la hora de

reducir el *overfitting*. Para reducir aun más la varianza se podrían congelar las primeras capas, entrenado solo las capas más profundas. En Visión Artificial funciona muy bien porque, como se muestra en el *paper* titulado *"Visualizing and Understanding Convolutional Networks"* [38], las primeras capas de una Red Neuronal Convolucional memorizan características sencillas (una linea inclinada, una linea curva, un color, etc). Y no es hasta capas más profundas cuando el conocimiento se va especializando hacia características más concretas del problema a tratar. Por eso mismo, es muy probable que las primeras capas —sobretodo si es de una red entrenada con un problema muy genérico y de una manera muy concienzuda— coincidan para diferentes problemas. De nuevo, el número de capas a congelar es algo que se debe encontrar por "ensayo y error".

3.6.4. *Momentum*

Esta técnica ayuda a que la dirección de la derivada, en el *Gradient Descent*, apunte siempre en la dirección correcta, acelerando así el aprendizaje. Es uno de los algoritmos de optimización más populares, gran parte de los modelos más punteros utilizan esta técnica. Además, es muy útil para evitar quedar atascado en un mínimo local. Cosa que, por otra parte, en el caso de la Visión Artificial es muy poco probable, debido a el gran número de variables (sería necesario que la derivada de cada uno de estos pesos fuera cero).

Esto se consigue actualizando los pesos tanto con la función de coste calculada, como con un porcentaje de la iteración anterior. El hiper-parametro que determina el porcentaje de la iteración anterior utilizado se llamará β y la derivada de la iteración anterior se llamará $V_{\partial W}$ Por lo tanto, la fórmula se corresponde con:

$$W_i = W_i - \alpha[\beta V_{\partial W_i} + (1-\beta)\frac{\partial J(W_i)}{\partial W_i}]$$

De esta manera, se realiza una especie de media que va restando importancia a las derivadas según se alejan en el tiempo. Cuanto más grande sea el valor de β, menos influirá el valor actual y más suavizada estará la curva. En ocasiones se puede encontrar esta formula sin el término $(1-\beta)$ que multiplica a el calculo de la iteración actual, pero es más aconsejable esta versión porque así no hay que ajustar α después de cada modificación de β.

Es especialmente útil acompañado de un *Mini-Bath Gradient Descent* —más cuanto menor sea el tamaño del *batch*— porque permite tener en cuenta valores que no pertenecen a este *batch*. En el caso de utilizar el *Batch Gradient Descent*, seguiría siendo útil tanto para no quedar atascado en un mínimo local, como para acelerar el aprendizaje. Es también muy útil en algoritmos que modelan, en tiempo real, realidades varían con el tiempo (como podrían ser los gustos de una persona o los fallos de la maquinaria de una fabrica). Así, le damos más importancia a los datos actuales, permitiendo tener el modelo en consonancia con la realidad actual.

3.6.5. *Data Augmentation*

En ocasiones es necesario aumentar los datos (varianza alta) pero, o bien no se pueden conseguir, o el coste es tan alto que no merece la pena. Para solucionar este problema surge una técnica denominada *Data Augmentation*. Ésta consiste en aplicar pequeñas modificaciones a los datos actuales, tales que sigan siendo datos que se podrían encontrar en la realidad, pero que la red los perciba como datos diferentes.

En Visión Artificial se pueden encontrar muchos ejemplos de *Data Augmentation* muy intuitivos. Estos podrían ser: rotar una imagen, espejarla, ampliarla, cambiar levemente las tonalidades o incluso una combinación de las anteriores. En la practica, éstas técnicas se emplearían combinadas de manera aleatoria.

Con esta técnica podríamos multiplicar el tamaño del conjunto de datos de una manera muy económica. Hay que tener dos precauciones a la hora de utilizarla. La primera, no realizar cambios tan grandes que no se correspondan con la realidad que queremos modelar. La segunda, no realizar modificaciones tan pequeñas que sobre-ajusten tanto el conjunto de prueba como el conjunto de entrenamiento. Algo especialmente peligroso porque no se vería analizando la varianza; para solucionar esto se podría mantener el conjunto de pruebas con el 100 % de los datos originales.

Capítulo 4

Análisis

4.1. Especificación de requisitos

4.1.1. Requisitos de usuario

OBJ-1 - Reconocimiento de velocidad

El sistema deberá ser capaz de reconocer, en tiempo real, las señales de velocidad que afecten al vehículo en el que se encuentra.

OBJ-2 - Visión artificial

El sistema deberá realizar el reconocimiento a través de la información suministrada por una cámara colocada en la parte frontal del vehículo apuntando en dirección de la marcha.

OBJ-3 - Tipos de clientes

El sistema deberá ser capaz de utilizarse tanto a través de una aplicación Android como de un sistema embebido. Por lo tanto, debe ser ligera y estar implementada en C++.

OBJ-4 - Evolución del sistema

El sistema puede estar sujeto a gran número de modificaciones, por lo que debe ser fácil de modificar.

4.1.2. Requisitos funcionales

RF-1 - Detección de señales

El sistema deberá ser capaz de detectar la localización de señales de velocidad.

RF-2 - Detección de vehículos

El sistema deberá ser capaz de detectar la localización de vehículos.

RF-3 - Reconocimiento de señales

El sistema deberá ser capaz de reconocer la velocidad de la señal detectada.

4.1.3. Requisitos de información

INF-1 - Información sobre una imagen

El sistema deberá consultar la información sobre la imagen: dirección en la máquina.

INF-2 - Información sobre una detección

El sistema deberá construir la información de cada detección: Probabilidad, coordenada x, coordenada y, altura, anchura y clase a la que pertenece.

INF-3 - Información sobre una solución

El sistema deberá devolver la información sobre la solución: un conjunto de detecciones.

4.1.4. Requisitos no funcionales

RNF-01 - Error detección de señales

El sistema deberá ser capaz de detectar la localización de señales de velocidad con 0.5 de IoU de media en imágenes tomadas desde el interior de un vehículo y en dirección a la marcha.

RNF-02 - Error detección de vehículos

El sistema deberá ser capaz de detectar la localización de vehículos con 0.4 de IoU de media en imágenes tomadas desde el inferior de un vehículo y en dirección a la marcha.

RNF-03 - Error reconocimiento de señales

El sistema deberá ser capaz de reconocer la velocidad de la señal detectada con una precisión del 0.8 % de media en imágenes tomadas desde el inferior de un vehículo y en dirección a la marcha.

RNF-04 - Lenguaje C++

El sistema deberá estar desarrollado en C++.

RNF-05 - Reconocimiento en fotografía

El sistema deberá tomar como entrada para el análisis una fotografía.

RNF-06 - Escalabilidad

El sistema deberá permitir la detección de más tipos de objetos cambiando solo los archivos de configuración. [Sin necesidad de recompilar la aplicación.]

RNF-07 - Usabilidad

El sistema deberá ser accesible a través de una interfaz documentada.

RNF-08 - Celeridad

El sistema deberá ser capaz de realizar un reconocimiento en menos de dos segundos en una máquina con un i5 de tercera generación.

RNF-09 - Tamaño disco

El sistema deberá ocupar menos de 250MB en disco.

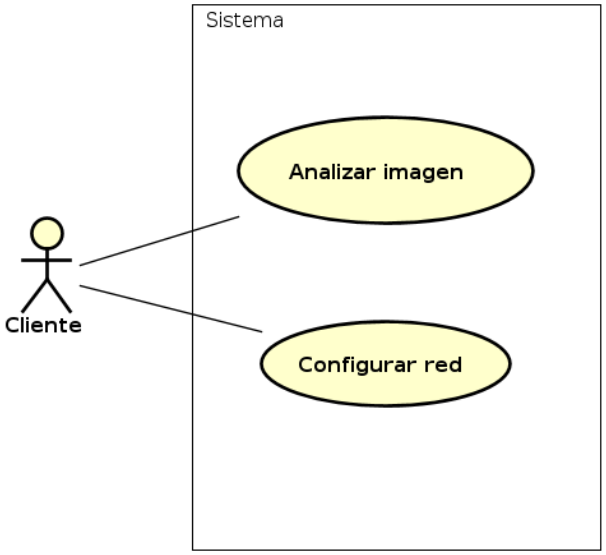


Figura 4.1: Casos de uso.

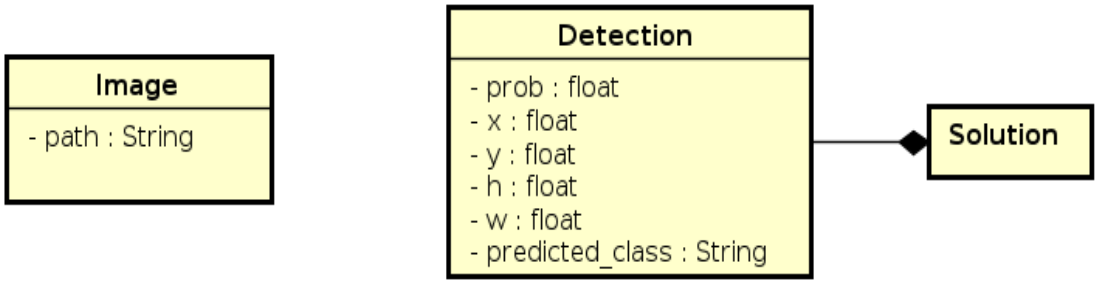


Figura 4.2: Diagrama de clases.

RNF-10 - Tamaño memoria

El sistema no puede superar 500MB de memoria durante su ejecución.

4.2. Casos de uso

La mejor forma de comprender la funcionalidad de un sistema —es decir, los requisitos funcionales— es mediante sus casos de uso (Figura 4.1). El sistema que se analiza en este trabajo tienen unas funciones muy concretas. Cómo se puede ver en el diagrama, lo único que pretende realizar el usuario con esta aplicación es configurar la red y, una vez configurada, suministrar imágenes para que sean analizadas.

4.3. Modelo del dominio

En la figura 4.2 se muestra el diagrama de clases que modela la información manejada por el sistema (los requisitos de información). Se puede dividir esta información en dos grupos: el *input*, es decir, la clase *Image* que conoce donde se encuentra la imagen en la máquina, y el *output* que contentiene la clase *Detection* y *Solution*, encargadas de almacenar la información del resultado del análisis. Como se puede ver, la clase *Solution* es una composición de clases *Detection*. Esta segunda se corresponde con cada uno de los objetos detectados por la red, almacenando: la probabilidad de que sea correcta (*prob*), sus coordenadas (x e y) normalizadas entre 0 y 1, su altura (h) y anchura (w), normalizadas también entre 0 y 1, y el nombre de la clase a la que pertenece (*predicted_class*).

Capítulo 5

Diseño

5.1. Algoritmo

Para comenzar a diseñar la solución, el primer paso que se realizó fue un estudio general de el ámbito del *Machine Learning*. Este estudio se inició sobre la base de conocimiento impartido en la carrera y el curso titulado *Machine Learning* [28] de la Universidad de Stanford. Después, se profundizó en el estudio del *Deep Learning* y de los algoritmos utilizados en Visión Artificial. Estos conocimientos se adquirieron con la ayuda del ciclo de cursos *Deep Learning Specialization* [26] y por el estudio de numerosos *papers*. Todos estos cursos están impartidos por Andrew Ng, un referente en *Machine Learning* (y cofundador de Coursera, por lo que se ve su compromiso al impartirlos). Los conocimientos, adquiridos durante este periodo, más relevantes para la comprensión de este trabajo, están expuesto en el capítulo 3. Este estudio ha sido una de las tareas principales para el desarrollo de este trabajo.

Como se puede ver en el capítulo 3, una vez se ha estudiado en profundidad los algoritmos disponibles, la decisión no puede ser otra que utilizar YOLO. Éste es, sin duda, el estado del arte dentro de los algoritmos de Visión Artificial. Ya no solos por sea el más rápido (requisito fundamental en este trabajo), sino porque también obtiene los mejores resultados.

Una vez el algoritmo está seleccionado, hay que decidir como implementarlo. En el mundo del *Machine Learning* hay gran número de *frameworks* y bibliotecas que nos pueden ayudar a implementar los algoritmos de manera más rápida y entrenarlos de manera mucho más eficiente (gracias a las implementaciones paralelas). Por esa razón, se descartó desde un inicio realizar una implementación manual del algoritmo. A la hora de decidir que *framework* o biblioteca utilizar, las características prioritarias fueron: que fuera compatible con C++ (requisito RNF-04), que soportase un entrenamiento en paralelo y que permitiera realizar *Transfer Learning*. Uno de los primeros en ser analizados fue TensorFlow [3], debido a su enorme popularidad. Fue desechado debido a que su interfaz para C++ es incompleta, pero, sobre todo, porque no existía ningún algoritmo YOLO previamente entrenado del que poder realizar *Transer Learning*. Después, se analizó OpenCV [5], una biblioteca muy conocida en el mundo de la Visión Artificial. Se comenzó a estudiar en profundidad, y se observó que, a pesar de las facilidades que brinda en cuanto a tratamiento de imágenes y de vídeo, el número

de algoritmos de *Machine Learning* es muy reducido y no proporciona ninguna herramienta para la construcción de una Red Neuronal personalizada. Por último se analizó Darknet [29], un *framework* desarrollado por el propio inventor de YOLO. Uno de los puntos fuertes de este *framework* era el gran número de arquitecturas diferentes. Dispone de las últimas arquitecturas (YOLO9000 o YOLOv3). También dispone de arquitecturas diseñadas especialmente para ser ligeras, pero con las ventajas de YOLO (YOLOv2-tiny y YOLOv3-tiny). Además, dispone de arquitecturas de muchos otros tipos de Redes Neuronales (AlexNet, LeNet o GoogLeNet). Y todas estas arquitecturas están acompañadas de la matriz de pesos entrenada en conjuntos muy grandes de conocimiento, como son COCO [22] y VOC [11], esto es ideal para poder realizar *Transfer Learning*. A parte, la arquitectura se define en un intuitivo fichero de configuración, lo que permite probar modificaciones en estas redes o incluso crear una arquitectura desde 0 (aunque esto eliminaría la posibilidad de realizar *Transer Learning*). Además, esta implementado con C, lo que facilita su utilización desde C++. En cuanto al paralelismo, este *framework* se puede compilar con las bibliotecas de CUDA y de openMP, permitiendo así ejecución y entrenamiento en paralelo tanto en gráficas como en procesadores. Permite la ejecución de los algoritmos hasta en cuatro gráficas en paralelo. Debido a la naturaleza, fácilmente paralelizable (no entraremos a comentar esto, porque se sale del alcance de este trabajo) de las Redes Neuronales, esto implica un gran avance. Más aun teniendo en cuenta que el proceso de mejora de estos algoritmos requiere numerosas pruebas, y si estas tardan demasiado, el proceso de implementación se podría alargar excesivamente. Como cosa negativa, la documentación de este *framework* es muy escasa y dispersa, por lo que aprender su funcionamiento iba a requerir mucho tiempo. A pesar de esto, eran tantas las ventajas que era preferible perder unas semanas leyendo código fuente y buscando en internet, que utilizar un *framework* diferente.

5.2. Arquitectura

Una idea que surgió en este proceso y repercutió en el resto del diseño fue la siguiente: Puesto que las diferentes señales de velocidad son muy similares entre ellas, diferenciándose solo por el número que contienen en su interior, podría ser interesante separar el reconocimiento en dos etapas. Hacer una red central que detectase dos clases: vehículos y señales de velocidad. Y después, suministrarle la señal de velocidad a otra red secundaria que diferenciase entre los distintos tipos de señales. Al separar así las tareas, nos aseguramos de que cada una de las redes puede ser más pequeña, ya que el conocimiento que deben aprender es más sencillo, y así lograr un resultado final que utilice menos recursos tanto de almacenamiento como de cómputo. También se empezaron a generalizar las responsabilidades de las redes y a pesar en el futuro de este software. Se empezó a ver una red como la central, que detectase cualquier tipo de objeto, y luego un conjunto de posibles redes interconectadas que detallasen el conocimiento suministrado por la principal. Pudiendo así ampliar el alcance de la red de forma sencilla. Fue entonces cuando se planteo la siguiente hipótesis: ¿Sería razonable realizar un desarrollo basado en componentes?

Este trabajo es el inicio de un sistema que podría llegar a ser realmente complejo. Tanto como una aplicación que permita el reconocimiento completo del entorno: coches, matriculas, semáforos, todo tipo de señales verticales y horizontales, peatones, etc. Como podría llegar a ser un sistema embebi-

do de ayuda a la conducción o incluso de conducción autónoma, coexistiendo con gran número de funcionales distintas. Por esta razón, es muy importante que sea un sistema en el que añadir nuevas funcionalidades sea realmente sencillo. Para esto, se necesita de un software estable, fácil de modificar y con una interfaz bien documentada. Estas no son cualidades de Darknet, por lo que se decidió crear unos componentes que lo encapsulen. Además, esto permite comunicar este modulo con aplicaciones implementadas en cualquier lenguaje de programación. En un inicio se planteo crear dos tipos de componentes, uno para problemas de clasificación y otro para problemas de detección, pero esto violaba el *Common-Clousure Principle* [23] puesto que una modificación en Darknet implicará la modificación de ambos componentes. Es decir, ambos componentes dependerían de Darknet, por lo que es mucho más interesante generalizar más la interfaz e incluir cualquier tipo de algoritmo de Visión Artificial en este componente. Además, la idea del desarrollo en componentes es interesante porque permite aislar a los clientes del componente de la implementación interna, por lo que una decisión de modificar el *framework* interno no afectaría a nadie más allá del componente. Incuso podrían coexistir componentes con implementaciones internas diferentes pero que compartiesen la misma interfaz, para de esta manera poder utilizarlos indistintamente por los clientes. Por esta razón, el diseño de la interfaz es fundamental, ya que es el contrato que "firmaremos" con los clientes, y que no deberíamos modificar nunca. Además, este componente es muy reutilizable, puesto que en cualquier programa que se necesite el uso de una Red Neuronal —por medio de DarkNet— será de gran ayuda. Evitando a los clientes de lidiar con las complicaciones de la integración de este *framework*.

Para diseñar la arquitectura del componente (Figura 5.1) se decidió utilizar el patrón arquitectónico Capas. De esta forma, se permite tener aisladas las zonas del componente más propensas a cambiar. Además, se deja abierta la opción de añadir casos de uso de una forma sencilla, ya que, como dice Robert C. Martin, el objetivo de la arquitectura software es dejar el mayor número de opciones abiertas durante el mayor tiempo posible [23]. Como se puede observar en el diagrama, la arquitectura se divide en tres capas. La primera es la encargada de dar entrada y salida a la aplicación. También se encarga de conocer y traducir el formato de la información tanto en el interior como en el exterior del componente y de realizar las transformaciones pertinentes entre ambos. Se utiliza el patrón *Facade* [12], suministrando una interfaz única para la utilización de todo el componente. Es importante prestar especial atención al diseño de esta interfaz, puesto que es la más sensible a los cambios y sería conveniente que pudiera permanecer inalterada independientemente de la implementación interna. Esta capa consta también de la clase encargada de implementar esta interfaz y de comunicarse con los controladores. En la siguiente capa se encuentra un controlador por cada caso de uso. Por el momento, uno para la configuración de la red y otro para el análisis de una imagen. La capa más profunda es la que contiene el modelo. Consta de las clases encargadas de almacenar la solución (*Solution* y *Detection*) y de la clase encargada de comunicarse con Darknet (*CVNetCore*).

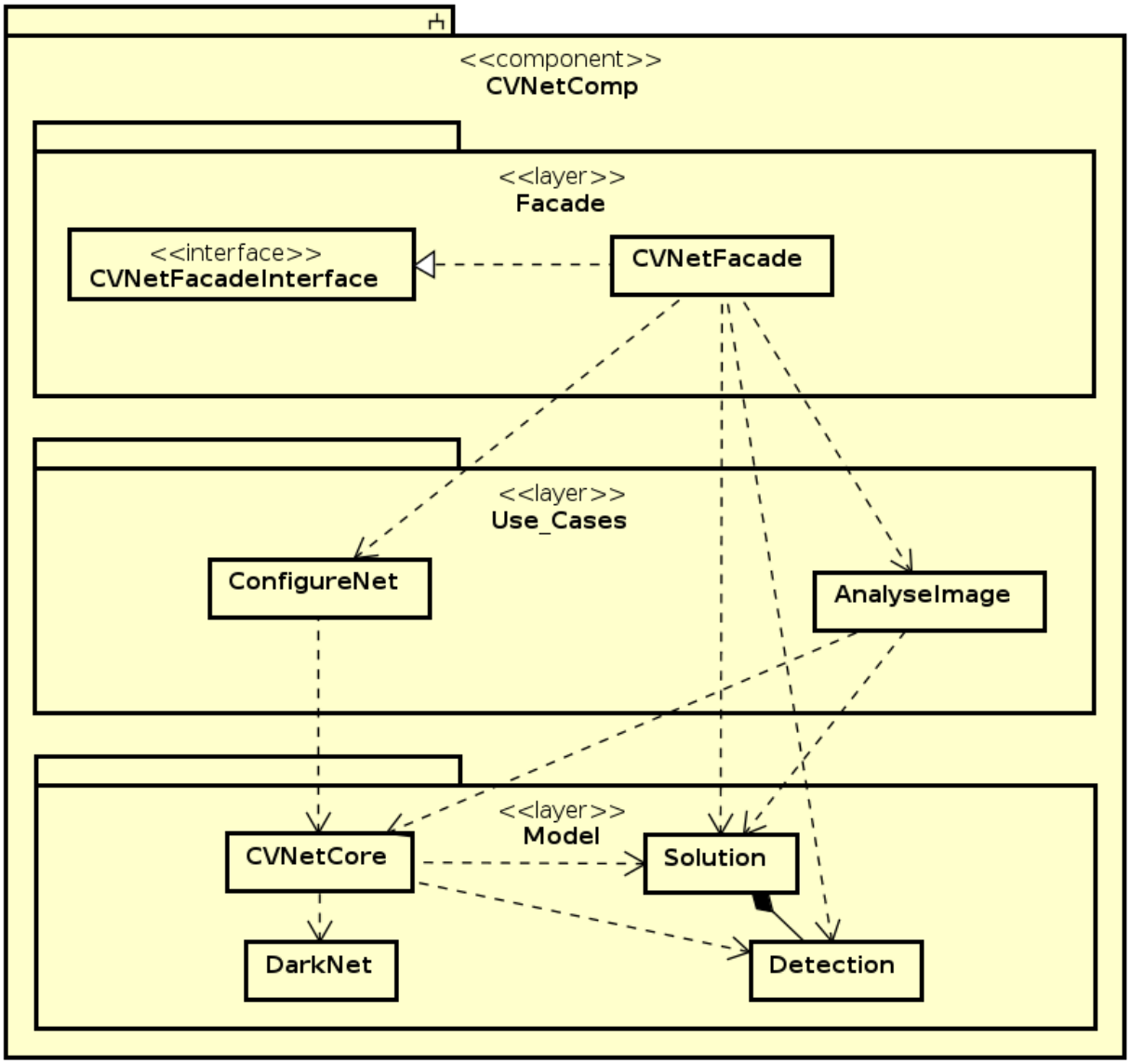


Figura 5.1: Arquitectura.

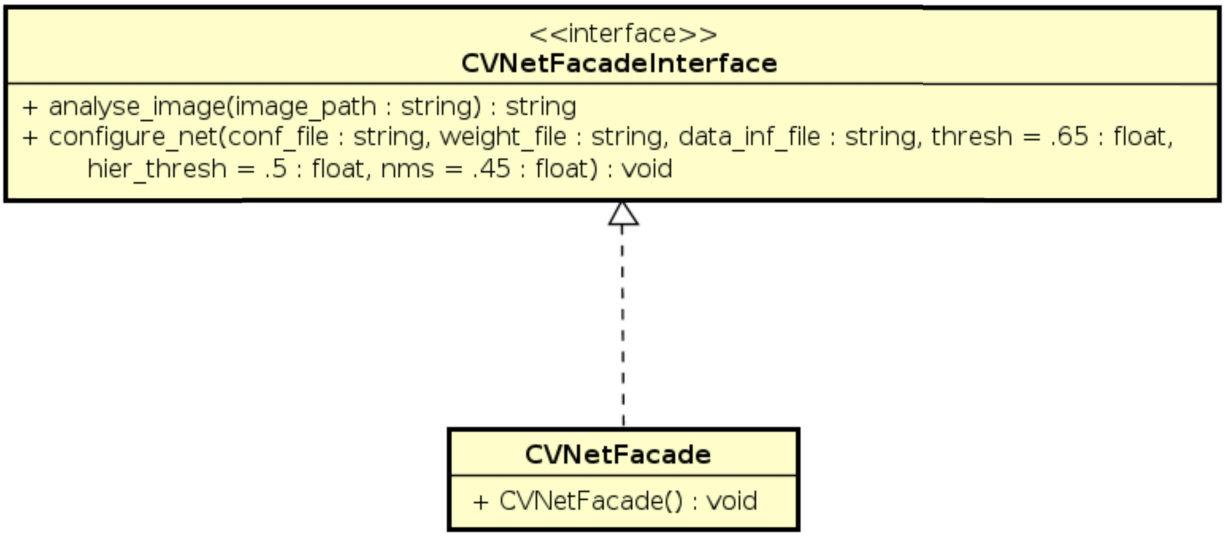


Figura 5.2: Clases implicadas en la fachada.

5.3. Interfaz externa

Como se ha comentado, una de las partes más importantes de un componente es diseñar cuidadosamente su interfaz externa. Esto se debe a que es la parte que utilizarán los clientes y, por lo tanto, deberá ser intuitiva y fácil de utilizar. Además, cualquier cambio realizado en la interfaz implicaría actualizar todas las clases dependientes de esta, es decir, los clientes. Esto implicaría que muchos clientes se planteasen actualizar el componente si esto implica tener que modificar software ya estable. Por esta razón, la interfaz debe ser suficientemente general como para permitir gran número de modificaciones internas sin implicar cambios en ésta.

En la figura 5.2 se puede observar las clases implicadas en la fachada. Como se puede ver, la clase *CVNetFacadeInterface* es una interfaz software con dos métodos: *analyse_image* y *configure_net*. La clase *CVNetFacade* implementa esta interfaz y consta solo de un constructor. Éste es el encargado, como veremos más adelante, de desplegar todo el componente. Se explicarán a continuación los métodos implicados en la interfaz:

El método *analyse_image*, como su propio nombre indica, es el encargado de analizar una imagen. Recibe como parámetro la dirección de la imagen a analizar y devuelve una cadena de texto con la solución. Ésta está en formato JSON y consta de una lista de detecciones. Cada una de estas detecciones tiene como claves los atributos de la clase *Detection*, que son: *prob*, *x*, *y*, *h*, *w* y *predicted_class*. Se utiliza un JSON para que sea independiente del lenguaje en el que esté implementado el cliente.

El método *configure_net* es el encargado de configurar la red. Es bastante dependiente de la implementación interna, concretamente de Darknet, aunque es bastante intuitivo, por lo que se decidió dejar de esta manera. Aun así, podría ser interesante, en el futuro, estudiar en profundidad los *frameworks* y bibliotecas más utilizados para conseguir un método de configuración más general. Otra opción podría ser —en el caso de modificar la implementación— que se adaptase ésta a estos mismos parámetros, dejamos esta decisión para actualizaciones venideras. Actualmente los que se utilizan son:

- *conf_file* es el archivo de configuración de la red. El listing 4.1 muestra un ejemplo de configuración. Comentaremos por encima los campos más importantes. El campo denominado *batch* define el tamaño del *batch* en el *Gradient Descent*. *Width* y *height* definen el tamaño de la imagen de entrada. *Channels* el número de canales o colores de la imagen, es decir, la profundidad. *Momentum* y *learning_rate* definen los dos hiper-parámetros comentados en el capitulo 3. Después, cada una de las etiquetas *[convolutional]* inician la configuración de una capa. Ésta incluye todos los parámetros estudiados en el capítulo 3, por lo que no merece la pena detenerse a explicarlo. De la misma forma ocurre con la etiqueta *[maxpool]*. Por último, al final del archivo, se encuentra un campo llamado *classes*, que debe coincidir con el número de clases existentes en el modelo (en este caso serían 2).
- *weight_file* se corresponde con el archivo que contiene la matriz de pesos entrenada con Darknet.
- *data_inf_file* es el archivo en el que se incluye toda la configuración respectiva a los datos. Esta sería: *classes* indicando el número de clases a clasificar, *train* indicando la ruta de

un fichero que lista todas las imágenes de entrenamiento, *backup* indicando el directorio en el que se guardarán periódicamente los *backups* durante el entrenamiento y *names* indicando la ruta a un archivo con el nombre de todas las clases que se analizan.

- *thresh* es un parámetro opcional que se utiliza como *threshold* a la hora de decidir si la probabilidad de una detección es suficiente como para sacarla por la salida. En caso de no suministrar este parámetro se toma como valor 0,65.
- *hier_thresh* es un parámetro opcional que se utiliza como *threshold* máximo a la hora de decidir si una detección es valida. En el caso de ser menor que *thresh* se ignora. Si no se suministra este parámetro se toma como valor 0,5.
- *nms* es un parámetro opcional que se utiliza como *threshold* en la fase de *non-max suppression* definiendo como de parecidos deben ser dos detecciones para tomarse por la misma. Si no se suministra este parámetro se toma como valor 0,45.

```
[net]
batch=64
subdivisions=2
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2
```

```
[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
```

```
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

############

[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=512
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=425
activation=linear

[region]
anchors = 0.57273, 0.677385, 1.87446, 2.06253, 3.33843, 5.47434, 7.88282, 3.52778,
   9.77052, 9.16828
bias_match=1
classes=80
coords=4
num=5
```

```
softmax=1
jitter=.2
rescore=0

object_scale=5
noobject_scale=1
class_scale=1
coord_scale=1

absolute=1
thresh = .6
random=1
```

Listing 5.1: Archivo de configuración de la red

5.4. Realización de Casos de Uso

Una vez que la arquitectura estaba propuesta y se comenzaron a analizar los casos de uso, se observó que podían surgir errores a la hora de utilizar el componente. Estos errores son: intentar analizar una imagen en una red sin configurar e intentar configurar una red con archivos erróneos. Se tuvieron en cuenta dos soluciones posibles, definir un retorno especial para los errores o lanzar una excepción. Se decidió utilizar excepciones para, de esta manera, no tener que complicar la interfaz del componente con códigos de error y poder delegar fácilmente la corrección del error a la clase encargada de solucionarlo (que estará, en los dos casos, fuera del componente). Estas excepciones se definieron como *logic_error*, acompañadas de un mensaje explicativo que indica la razón de la excepción.

Durante el diseño de la clase *CVNetCore* surgió un problema: Se debía decidir cómo y en qué momento construirla, teniendo en cuenta que debe haber como máximo una instancia de ésta y que debe estar disponible siempre que se quiera realizar el caso de uso Analizar imagen. Se valoraron dos soluciones: crear la clase en el despliegue del componente o utilizar el patrón *Singleton* [12] y que fueran los controladores de caso de uso los que solicitasen una instancia. La primera solución aseguraba que siempre iba a haber una instancia de ésta y, de esta forma, se elimina la posibilidad de intentar analizar una imagen con la red sin configurar. Pero se añadía una excepción en el despliegue (archivos de configuración erróneos), algo que nunca es una buena práctica. Además, el mayor problema era que había que añadir parámetros al despliegue del componente, cosa que complicaba la interfaz y la hacía demasiado concreta. Por esta razón, se decidió utilizar el patrón *Singleton* y añadir la configuración de la red como un caso de uso aparte. De esta manera, se permite reconfigurar la red las veces que sean necesarias.

Se muestran a continuación los diagramas de secuencia que muestran tanto el despliegue del componente como los dos casos de uso. Se puede ver también el momento en el que podría saltar una excepción en cada uno de estos casos de uso.

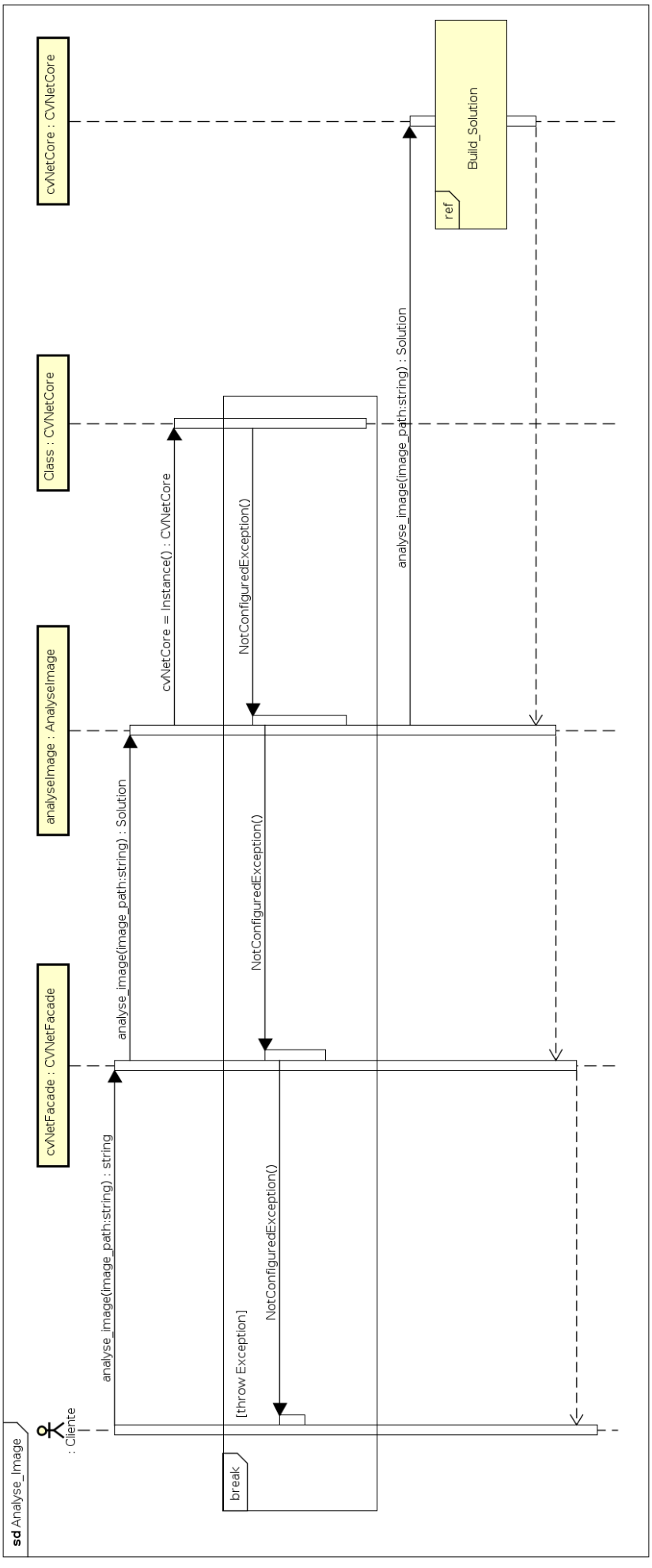


Figura 5.3: Caso de uso Analizar imagen.

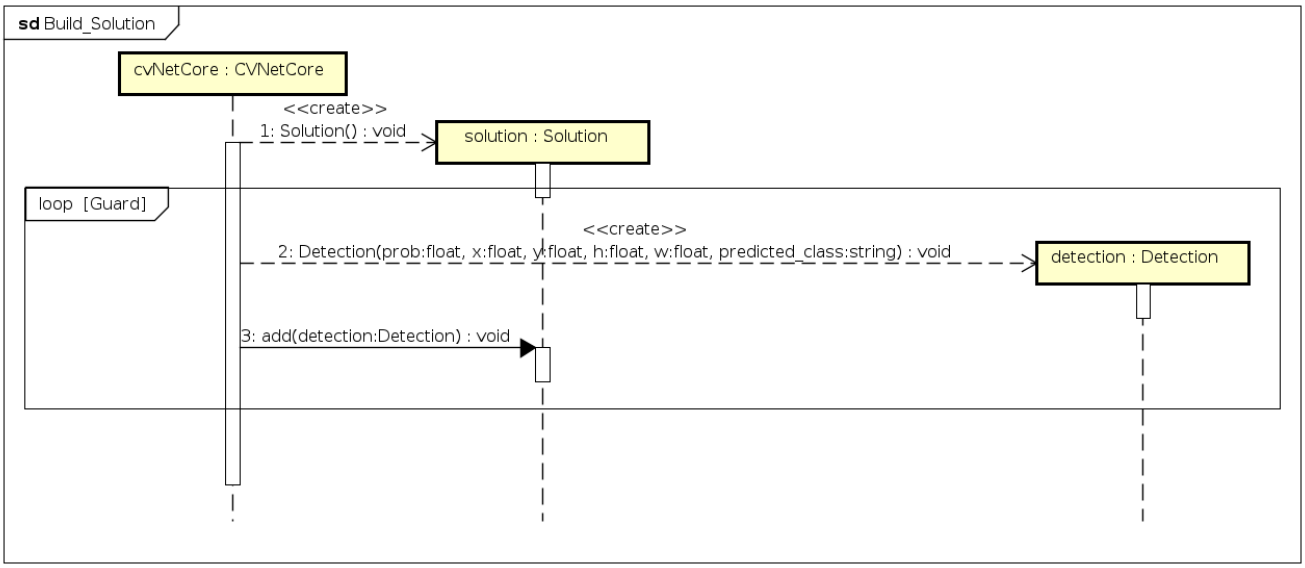


Figura 5.4: Subsecuencia Build_Solution.

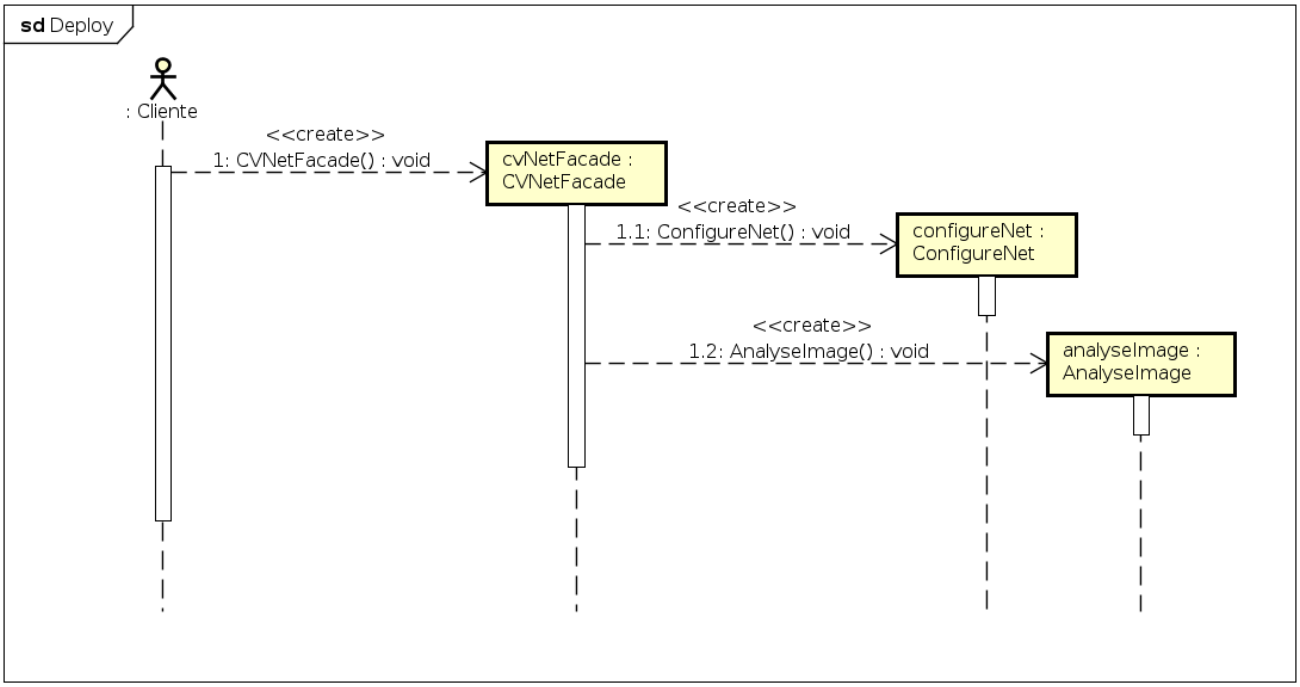


Figura 5.5: Despliegue del componente.

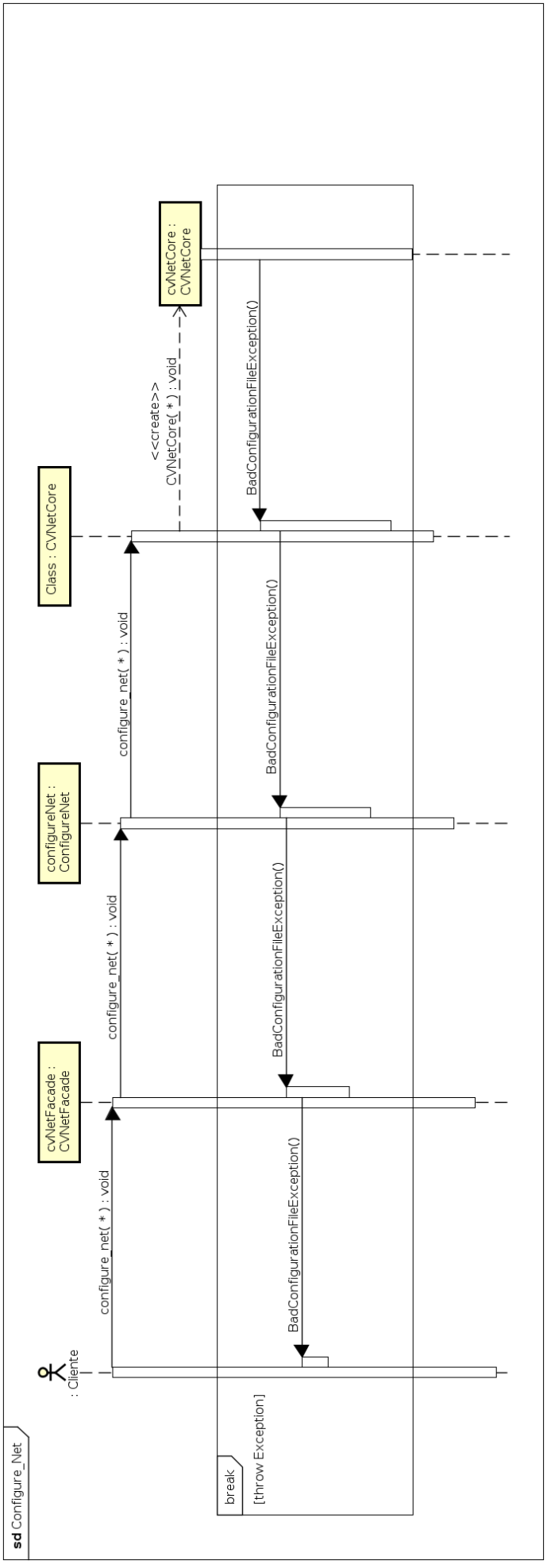


Figura 5.6: Caso de uso Configurar red.

En las figuras 5.3 y 5.4 se muestran las interacciones que se llevan a cabo entre las diferentes clases a la hora de analizar una imagen. Sin embargo, no se muestra la interacción entre *CVNetCore* y Darknet puesto que añade mucha complejidad al diagrama, siendo esto lo que se pretende evitar. Se puede ver, como se ha comentado anteriormente, que puede saltar una excepción si se pretende analizar una imagen con la red sin configurar. Esta excepción se produciría a la hora de pedirle la instancia a la clase *CVNetCore*. A la hora de construir la solución, la condición del bucle depende de la interacción con Darknet, por lo que no se especifica. Como se puede imaginar, este bucle se ejecuta mientras haya detecciones que almacenar.

A continuación, en la figura 5.5 se puede ver la sencillez del despliegue del componente. Cuando un cliente crea una instancia de la fachada, ésta se encarga de crear los controladores de los casos de uso. Estos controladores pueden acceder a la clase *CVNetCore* a través del patrón *Singleton*.

Por último, la figura 5.6, se muestra el proceso por el cual se configura la red. La orden de configurar la red se comunica desde la fachada hasta la clase *CVNetCore*, la cual construye una instancia, como indica el patrón *Singleton*. Los parámetros se han eliminado del diagrama para reducir su tamaño, pero serían: *const std::string &conf_file*, *const std::string &weight_file*, *const std::string &data_inf_file*, *const float thresh*, *const float hier_thresh* y *const float nms*. Todos los parámetros son constantes para hacer la interfaz más general, permitiendo pasar como parámetro tanto valores constantes como variables, así como referencias o literales. Además, todos los parámetros de tipo std::string se han definido como referencias para que no se construya un nuevo objeto en cada llamada a esta función, sino que se pase la dirección de memoria, ahorrando así tiempo y espacio.

Capítulo 6

Implementación y Pruebas

Se junta en este capítulo la parte de implementación y de pruebas ya que, debido a la naturaleza del proceso de creación de un modelo de *Machine Learning*, sería improductivo exponer estas dos fases de manera independiente. Se presentará primero la parte referente al componente software, más escueta debido a no tener ninguna complicación manifiesta, y después todo el proceso mediante el cual se ha seleccionado y entrenado el modelo, mucho más extenso pues es la parte principal de este trabajo.

6.1. Componente *CVNetComp*

La implementación del componente se realizó en C++. La conexión con Darknet fue relativamente sencilla ya que C y C++ son muy compatibles. En cuanto a la compilación, este *framework* ofrece dos posibilidades. Una compilación estática de la biblioteca por medio del fichero *libdarknet.a* o una compilación dinámica por medio de *libdarknet.so*. Se eligió la segunda opción ya que el tamaño del ejecutable se reduce en medio gigabyte. Esto tomará especial importancia en el caso de que se utilice el componente más de una vez en la misma máquina. Con esta segunda opción hay que incluir el fichero *libdarknet.so* en la carpeta */lib*, o el sistema operativo no será capaz de encontrarla en tiempo de ejecución. Para ayudar en la compilación, el componente incluye un *Makefile* encargado de compilar todos los módulos y un archivo de prueba que ejemplifica el funcionamiento del componente.

Tras la implementación completa se realizaron unas pruebas de carga para comprobar que Darknet liberaba automáticamente los recursos empleados. Esta prueba consistió, simplemente, en generar un bucle que reconfigurase la red una y otra vez y con el comando *top* observar los recursos utilizados por el proceso. Se vio claramente que los recursos aumentaban de manera continuda, por lo que se implementó un destructor para la clase *CVNetCore* encargado de liberar toda la memoria asignada dinámicamente por Darknet. Tras esto, se volvió a ejecutar la prueba y se vio que los recursos utilizados permanecían constantes.

6.2. Modelo de *Machine Learning*

Donde más tiempo se dedicó de la implementación fue al entrenamiento de la Red Neuronal. Este proceso, como se ha comentado con anterioridad, es mayormente empírico. A pesar de esto, con un buen análisis de errores, es posible intuir que modificaciones de los parámetros mejorarán los resultados de la red.

Lo primero que se realizó fue etiquetar un conjunto de datos con el que entrenar la red. Aunque puede no parecerlo, éste es uno de los pasos más importantes a la hora de entrenar un modelo. Se buscaron imágenes de la carretera realizadas desde la parte delantera del vehículo. Se encontró un conjunto de datos muy interesante de 900 imágenes, The German Traffic Sign Detection Benchmark [17]. También se tomaron numerosas imágenes, sobretodo en condiciones que no aparecían en el conjunto encontrado (lloviendo, de noche, etc). Después, se busco un programa que permitiera etiquetar las imágenes de forma sencilla, se eligió LabelImg [35], un proyecto de software libre fácil de utilizar. A continuación, se etiquetó minuciosamente las imágenes para que no hubiera ningún fallo en el conjunto de entrenamiento. Se iniciaron las pruebas con estas 900 imágenes, quedando abierta la posibilidad de aumentar este conjunto de datos si el análisis de los errores mostrase que es insuficiente. Este conjunto se separó en dos, el conjunto de entrenamiento y el conjunto de desarrollo. El primero, como su propio nombre indica, se utiliza para entrenar la red. El segundo se utiliza para probar la red sobre un conjunto desconocido, para poder estimar como se comportará en "la realidad". De esta manera se pueden ajustar los diferentes parámetros teniendo en cuenta el *bias* y la varianza. Algunos autores recomiendan crear un tercer conjunto —denominado conjunto de prueba— utilizado simplemente para la prueba final de un algoritmo en la que medimos como se comporta. Es necesario porque al utilizar el conjunto de desarrollo para modificar los parámetros podemos llegar a sobreajustarlo un poco. Se decidió no utilizar este tercer conjunto ya que los datos no son abundantes y no se presentará a ningún concurso en el que se necesite una precisión total del error, simplemente hacer que funcione lo mejor posible. Para separar los datos se seleccionó el 90 % para el conjunto de entrenamiento y 10 % para el conjunto de desarrollo. Por supuesto, los datos se eligieron aleatoriamente para que ambos conjuntos tuvieran la misma distribución.

Lo siguiente que se realizó, antes de comenzar el entrenamiento propiamente dicho, fue definir como se mediría el error y hasta donde se podría aspirar a llegar. Como se ha explicado con anterioridad, es importante definir el error al que aspira el modelo para así tener una medición más exacta del *bias*. Para definir este umbral se utiliza el "nivel de rendimiento humano" y se establecerá en un 0 % de error. Esto se debe a que cualquier persona es capaz de reconocer un vehículo o una señal dentro de una imagen. En cuanto a que error se desea medir, hay que tener en cuenta las tres cualidades que se pretenden optimizar en este modelo: el tiempo de ejecución, el tamaño y el IoU. Se decidió marcar un umbral para los dos primeros objetivos y mejorar todo lo que se pueda el IoU siempre y cuando no se superen estos umbrales. Como tamaño máximo se estableció 250MB —que es bastante grande, pero no se pretende que esto restrinja al modelo ya que es el objetivo menos prioritario— y como tiempo máximo de análisis de dos segundos, ya que si se observan las señales cada dos segundo, se tienen varias oportunidades de reconocerla antes de que ésta sea válida.

Una vez que los datos estaban preparados y estaba bien definido el objetivo, se comenzó con las

pruebas. Durante todas las pruebas se fue analizando la curva de aprendizaje. Esta curva muestra el error (IoU) tanto en el conjunto de entrenamiento como en el de desarrollo, para observar así si se está cometiendo algún error en el entrenamiento (es decir, si el error en el conjunto de entrenamiento no disminuye), comienza a sobreajustarse (los errores en ambos conjuntos comienzan a separarse a mayor velocidad), si la arquitectura de la red no es capaz de memorizar mejor los datos (es decir, el error en el conjunto de entrenamientos deja de disminuir o lo hace muy lentamente), etc.

Con la primera prueba, la cual detallaremos más adelante, surgieron las primeras complicaciones. Se instaló un Debian lo más ligero posible (sin siquiera interfaz gráfica) en un i7 de tercera generación. Se instaló un servidor ssh y se configuró un DNS dinámico para poder ejecutar las pruebas en cualquier momento y desde cualquier dispositivo capaz de albergar un cliente ssh (es decir, hasta un teléfono móvil). Se lanzó la primera prueba paralelizando la ejecución con OpenMP en los cuatro núcleos. Se midieron los tiempos de cada *batch* de 64 imágenes y se obtuvo una media de 300 segundos por *batch*. Lo que implica que un entrenamiento de 10.000 *batches* —número algo escaso para cada prueba— se tendría que esperar más de un mes. Aparte de esto, a las cinco horas de entrenamiento los cuatro núcleos se encontraban a más de cien grados, temperatura crítica para un procesador. Es decir, habría que incluir pausas en los entrenamientos para enfriar la máquina, en otras palabras, más tiempo de entrenamiento. Para un trabajo con un alcance aproximado de cuatro meses y con el número de pruebas necesarias, estos tiempos eran totalmente inadmisibles.

Por esta razón, se decidió buscar una máquina alternativa para entrenar la red. Se contacto con Arturo González y Diego Llanos, del Grupo Trasgo, a través de Benjamín Sahelices, tutor de la universidad. El Grupo Trasgo ofreció, amablemente, una máquina con cuatro GPUs Titan Black, las cuales ejecutaban cada *batch* en diez segundos de media. Con estos tiempos, el ejemplo anterior tardaría menos de un día en memorizar los 10.000 *batches*. Esta ayuda fue fundamental en el desarrollo del trabajo, pues sin esta capacidad de computo no hubiera sido posible realizar las pruebas necesarias.

El primer modelo que se entreno fue un YOLOv2-tiny, una versión reducida de YOLOv2 [30] (También llamado YOLO9000). Como se puede ver en el cuadro 6.1, esta arquitectura consta de 15 capas. Como la mayoría de las arquitecturas de Visión Artificial, se intercalan capas convolucionales con capas *maxpool*. Las primeras van reduciendo la matriz en anchura y las segundas van aumentando su profundidad. Por último comentar que la última capa es de tamaño 13x13x35 debido a que la imagen se divide en 169 cuadrantes (13x13) y cada uno de estos cuadrantes consta de 5 *anchor boxes* y, como se comento con anterioridad, estos necesita de siete posiciones para almacenar su información (cuatro para las coordenadas y el tamaño, uno para la probabilidad y dos para las probabilidades de cada clase) de ahí que la profundidad sea de 35.

Se comenzó a entrenar este modelo realizando un *Transfer Learning*, que proporciona la propia página de Darknet, sobre el conjunto de datos COCO. Antes de nada, se midió el tamaño de la matriz de pesos, pues es el fichero más pesado, y el tiempo de análisis en varias imágenes para ver si se podía continuar con esta arquitectura. El fichero pesaba 42MB, muy lejos del umbral auto-establecido, y las imágenes se clasificaban en 1,33 segundos, también dentro de la marca. Una vez que se comprobó que la arquitectura era válida, se le asignó a los hiperparámetros los siguientes valores: un *Learning Rate* de 0,001 y un *Momentum* de 0,9 y se comenzó a iterar sobre los datos. Como el aprendizaje estaba siendo muy lento, a las 5000 iteraciones se decidió aumentar el *Learning Rate*, se probó con

Tipo de capa	Tamaño salida	Tamaño filtro	stride	padding
Entrada	416x416x3	-	-	-
Convolucional (Leaky ReLU)	416x416x16	3x3	1	1
Pooling (max)	208x208x16	2x2	2	0
Convolucional (Leaky ReLU)	208x208x32	3x3	1	1
Pooling (max)	104x104x32	2x2	2	0
Convolucional (Leaky ReLU)	104x104x64	3x3	1	1
Pooling (max)	52x52x64	2x2	2	0
Convolucional (Leaky ReLU)	52x52x128	3x3	1	1
Pooling (max)	26x26x128	2x2	2	0
Convolucional (Leaky ReLU)	26x26x256	3x3	1	1
Pooling (max)	13x13x256	2x2	2	0
Convolucional (Leaky ReLU)	13x13x512	3x3	1	1
Pooling (max)	13x13x512	2x2	1	1
Convolucional (Leaky ReLU)	13x13x1024	3x3	1	1
Convolucional (Leaky ReLU)	13x13x512	3x3	1	1
Convolucional (Linear)	13x13x35	1x1	1	1

Cuadro 6.1: Arquitectura YOLOv2-tiny.

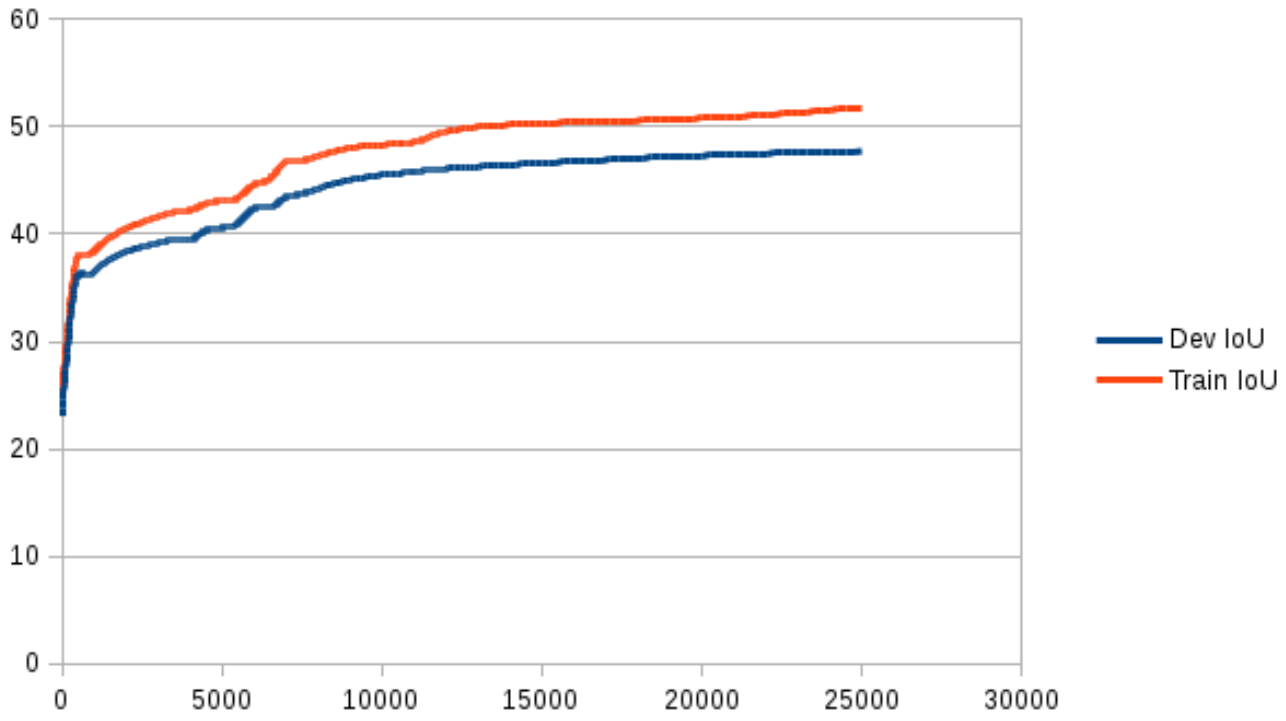


Figura 6.1: Curva de aprendizaje de la primera prueba.

Tipo de capa	Tamaño salida	Tamaño filtro	stride	padding
Entrada	416x416x3	-	-	-
Convolucional (Leaky ReLU)	416x416x16	3x3	1	1
Pooling (max)	208x208x16	2x2	2	0
Convolucional (Leaky ReLU)	208x208x32	3x3	1	1
Pooling (max)	104x104x32	2x2	2	0
Convolucional (Leaky ReLU)	104x104x64	3x3	1	1
Pooling (max)	52x52x64	2x2	2	0
Convolucional (Leaky ReLU)	52x52x128	3x3	1	1
Pooling (max)	26x26x128	2x2	2	0
Convolucional (Leaky ReLU)	26x26x256	3x3	1	1
Pooling (max)	13x13x256	2x2	2	0
Convolucional (Leaky ReLU)	13x13x512	3x3	1	1
Pooling (max)	13x13x512	2x2	1	1
Convolucional (Leaky ReLU)	13x13x1024	3x3	1	1
Convolucional (Leaky ReLU)	13x13x256	1x1	1	1
Convolucional (Leaky ReLU)	13x13x512	3x3	1	1
Convolucional (Linear)	13x13x35	1x1	1	1

Cuadro 6.2: Arquitectura YOLOv3-tiny.

0,01 y los resultados fueron buenos, la red había aprendido correctamente y más rápido. Se probó también con 0,1 y la función divergió. Se probo un termino medio, 0,03, pero tampoco dio buenos resultados. Así que se continuó el entrenamiento con 0,01. Como se puede ver en la figura 6.1, el entrenamiento se aceleró sustancialmente en este punto. Se continuó entrenando hasta las 25000 iteraciones. Hasta el momento no se sobreajustó la red, pero el aprendizaje comenzó a relentizarse cada vez más. Llegando a aumentar solo un 1 % tras 5000 iteraciones. En este punto el error era claramente de *bias*, por lo que la únicas soluciones eran o seguir entrenando o probar con otra arquitectura. Se decidió probar con otra arquitectura y observar la velocidad con la que aprendía. Pudiendo decidir a posteriori si continuar con esta arquitectura o con la nueva.

Durante el desarrollo de este trabajo, el equipo de Darknet presento una nueva arquitectura llamada YOLOv3 [31], la siguiente iteración de la que se está utilizando. Y junto con ésta, vino también la versión reducida. Esta no incluye grandes cambios, pero debido a que la solución más prometedora era una nueva arquitectura y esta acababa de salir a la luz, probarla era algo necesario. Como se puede ver en el cuadro 6.2, la arquitectura es prácticamente la misma. La mayoría de las mejoras son pequeños cambios que se ha ido viendo durante el último año que dan buenos resultados (se pueden ver todos detallados en el *paper*). Ahí se observa que todas las modificaciones has sido pequeños cambios como sustituir el cálculo del error mediante la suma de los errores cuadrados a una regresión logística.

Este modelo se comenzó a entrenar de la misma manera. Se midió el tamaño y la velocidad sobre el modelo obtenido por *Transfer Learning*. El fichero de la matriz de pesos ocupaba 33MB y las imágenes se clasificaban, por termino medio, en 1,25 segundos; todo dentro de los márgenes. Des-

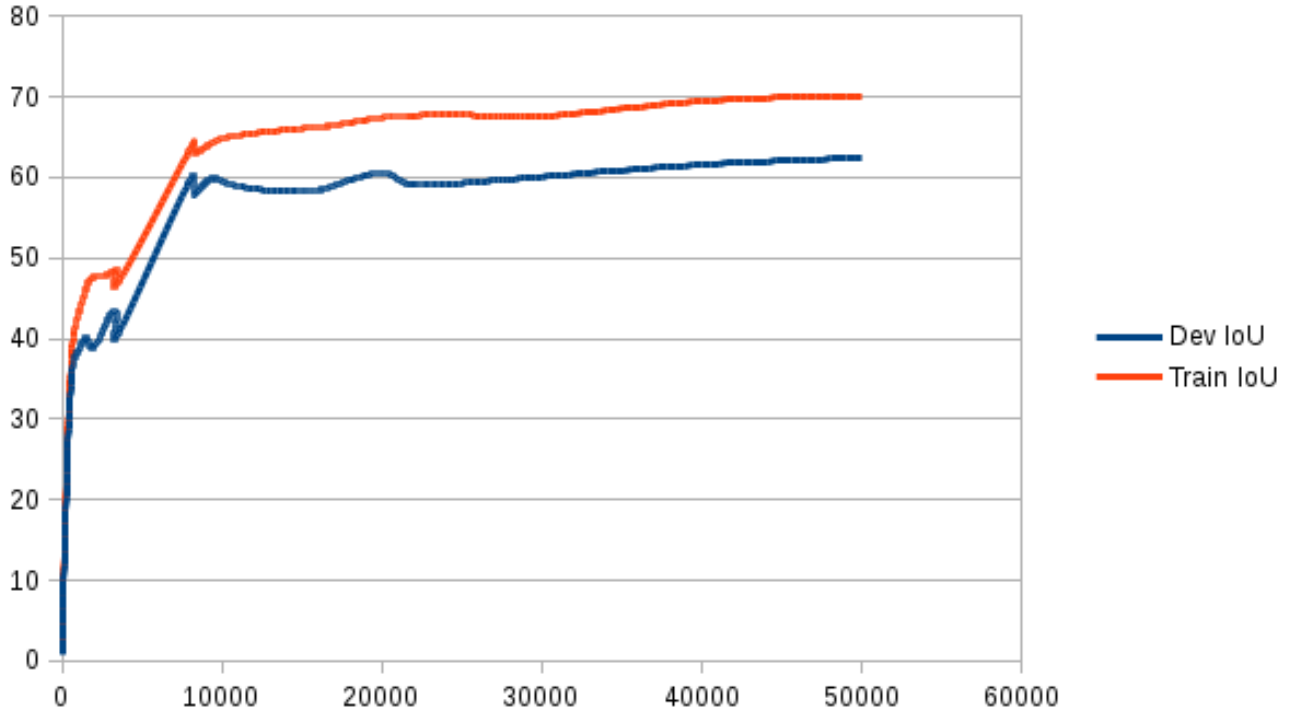


Figura 6.2: Curva de aprendizaje de la segunda prueba.

pues, se estableció un *Learning Rate* de 0,001 y un *Momentum* de 0,9. Además, también se aumento el *Learning Rate* a 0,01, solo que esta vez se hizo a las 3500 iteraciones. Se puede ver claramente el acelerón que pega la gráfica en ese momento. Los resultados parecían muy prometedores pues estaban diez puntos por encima de la primera prueba. Por esta misma razón, el entrenamiento se prolongó hasta las 50.000 iteraciones. En este punto se realizaron unas pruebas manuales de como clasificaba esta red las imágenes y se comparó con como lo hacía la anterior. Estas pruebas consistieron, simplemente, en coger un gran número de imágenes del conjunto de pruebas y clasificarlas con ambas redes. Lo que se observó es que la segunda red clasificaba mejor los vehículos, y de ahí ese mejor IoU, pero era incapaz de clasificar ninguna señal de velocidad, pero al ser menos numerosas no afectan tanto a la medida del error. Así que, a pesar de tener un IoU más bajo, se decidió continuar con la primera arquitectura.

Como habíamos comentado antes, la única solución que podíamos emplear sobre esta primera arquitectura era o probar una arquitectura nueva —cosa que resultó fallida— o entrenar la red durante más tiempo (Figura 6.3). Así que esta fue la decisión que se tomó: continuar con el entrenamiento hasta que la red fuera incapaz de aprender nada más. Esto ocurrió aproximadamente a las 30.000 iteraciones, pero en las siguientes 30.000 se consiguió que mejorase otro punto. Cuando ya no era capaz de generalizar con mayor precisión las imágenes, a las 70.000 iteraciones, se disminuyó el *Learning Rate* a 0.001 de nuevo y se emplearon otras 20.000 iteraciones; para de esta manera conseguir aproximar de una manera más precisa los últimos entrenamientos. Puesto que no mostró una especial mejora se redujo a 0.0001 y se entrenó durante 10.000 iteraciones más, como esto tampoco consiguió mejorar en nada el IoU se llegó a la conclusión de que esta arquitectura no es capaz de mejorar su aprendizaje, por lo que se detuvo ahí el entrenamiento de forma permanente.

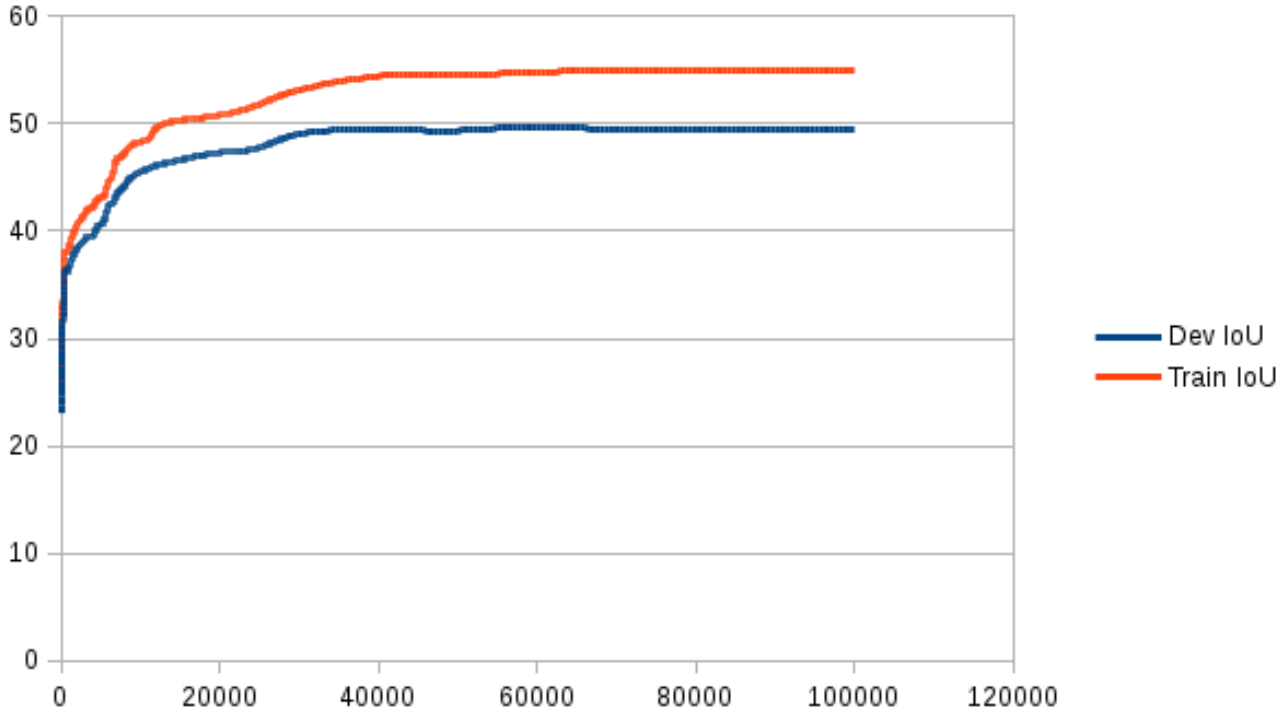


Figura 6.3: Curva de aprendizaje de la última prueba.

6.3. Resultados obtenidos

Se muestran a continuación algunas imágenes —del conjunto de prueba— clasificadas por esta Red Neuronal para poder observar como han sido los resultados obtenidos en el trabajo. Se intentan mostrar unas imágenes en las que haya señales que se pudieran confundir, como la figura 6.10, imágenes con objetivos lejanos, como las figuras 6.6, 6.8 o 6.9, algunas en las que haya gran densidad de vehículos, como las figuras 6.4 o 6.8 —pues aquí es donde ocurren la mayoría de los fallos— y una más oscura, como la figura 6.8, para ver como se comporta el modelo en esta situación. Además, como se puede ver en las figuras 6.6 y 6.7, un aumento del *threshold* de la fase de *non-max supression* podría ser conveniente para esta Red Neuronal, pero dado que es un parámetro modificable desde la interfaz del componente, lo dejamos a elección del cliente.

De estas pruebas se puede inferir que un punto muy positivo de este modelo es que clasifica muy bien las señales de velocidad y solo falla con los vehículos lejanos y cuando hay mucha densidad de los mismos. Se puede concluir que los objetivos propuestos para esta Red Neuronal están superados con éxito.

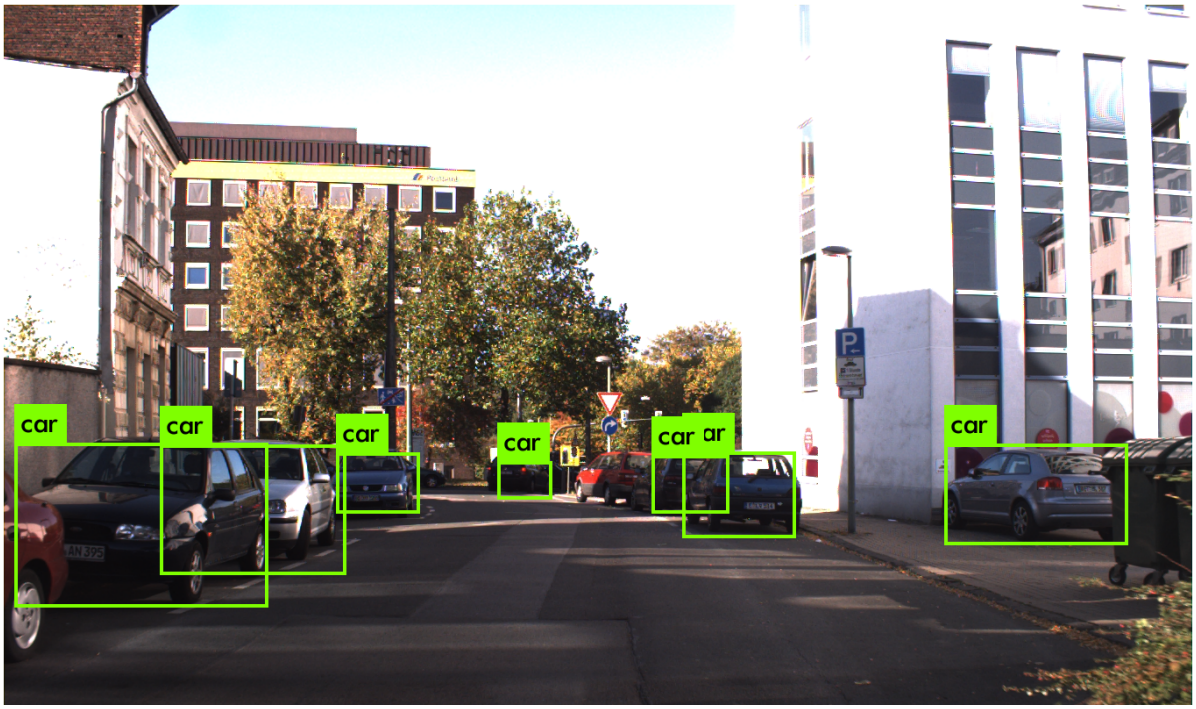


Figura 6.4: Imagen 32.jpg

Figura 6.5: Imagen 61.jpg

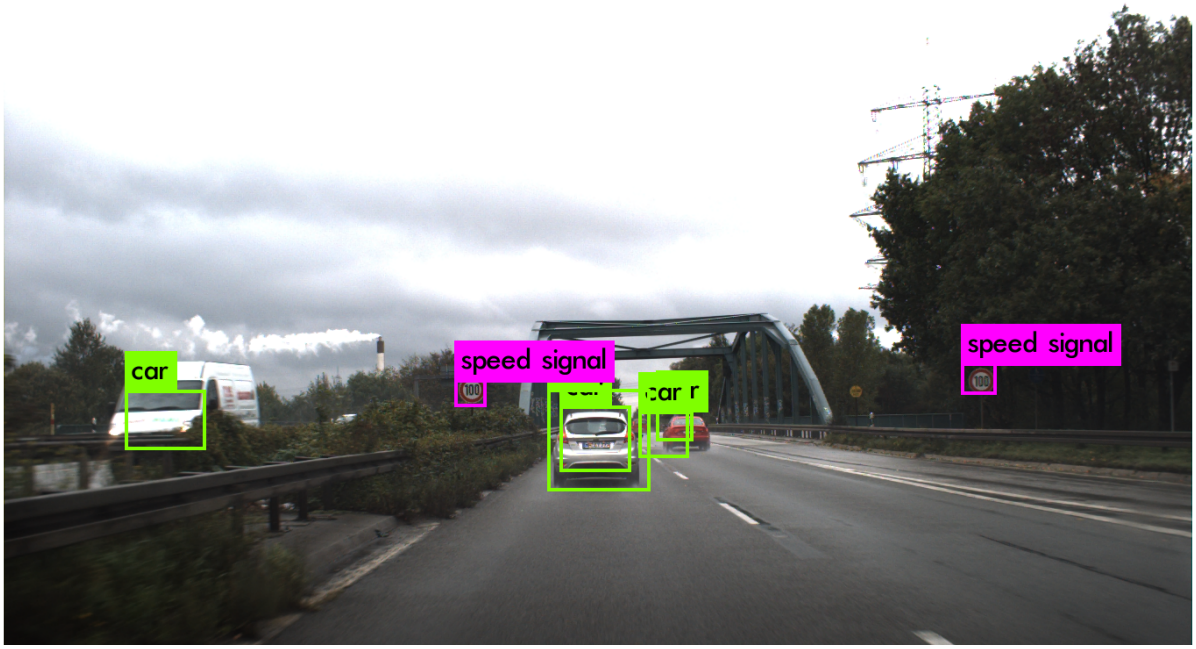


Figura 6.6: Imagen 123.jpg

Figura 6.7: Imagen 220.jpg

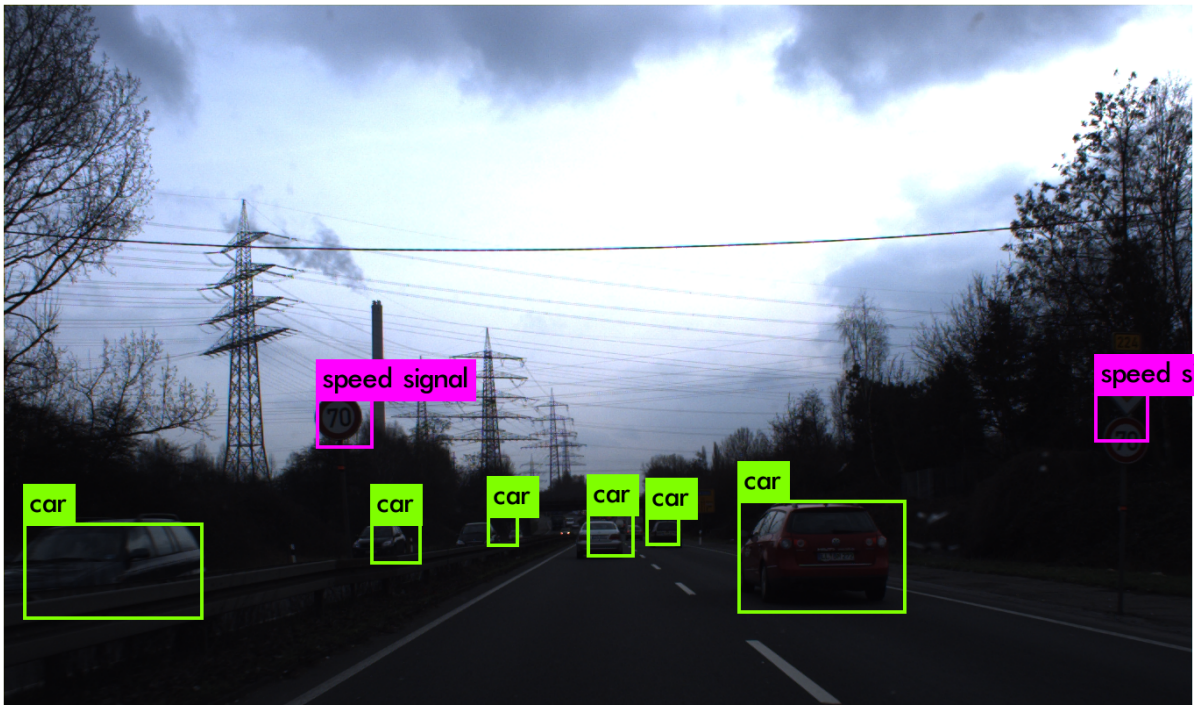


Figura 6.8: Imagen 405.jpg

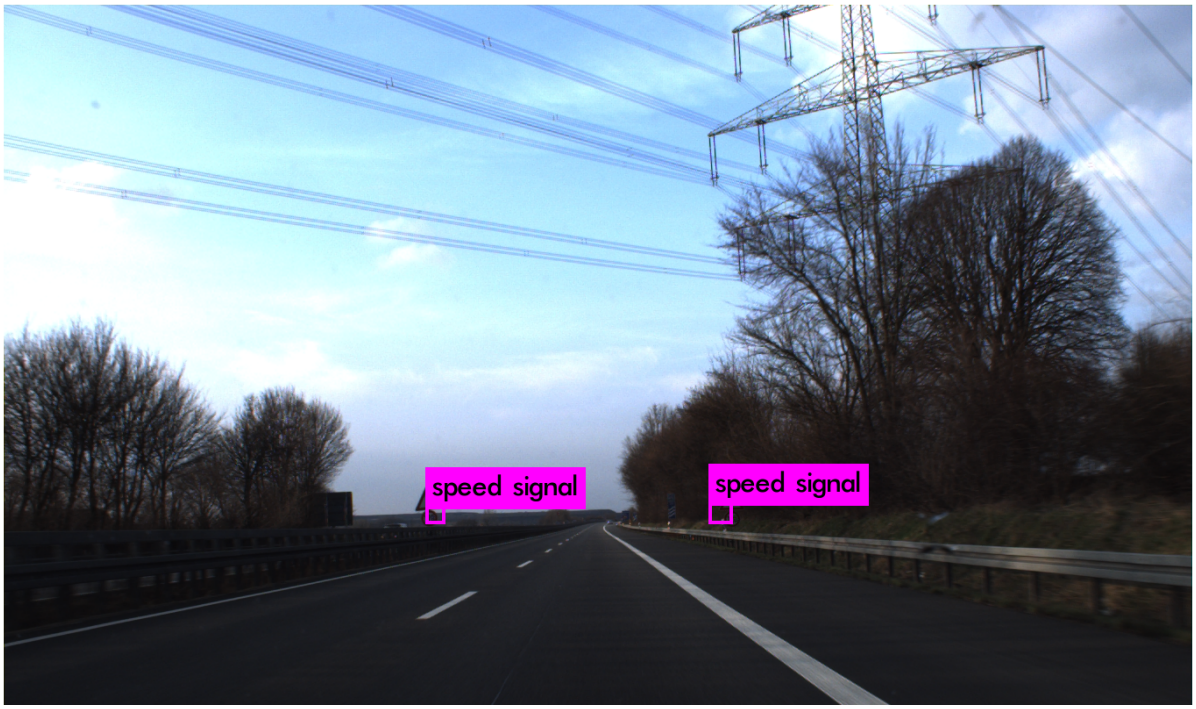


Figura 6.9: Imagen 443.jpg

Figura 6.10: Imagen 712.jpg

Capítulo 7

Conclusiones

En este trabajo se ha realizado un estudio en profundidad de los algoritmos punteros dentro del ámbito de la Visión Artificial, así como de todas las herramientas existentes para entrenar y mejorar estos modelos para conseguir que memoricen los datos que se le suministren de la forma más precisa posible. Además, se ha puesto en práctica todos estos conocimientos entrenando una Red Neuronal capaz de localizar señales de velocidad y vehículos. Por último, se ha implementado un componente capaz de albergar este modelo —y cualquier otra Red Neuronal— y de presentar una interfaz sencilla de utilizar.

Tras la realización de este trabajo, se puede afirmar que los objetivos que se han podido abarcar se han cumplido ampliamente. No obstante, ha habido uno que no se ha podido afrontar. Éste es el objetivo de analizar una señal, una vez detectada, y determinar a qué velocidad concreta restringe la marcha (RF-3). Esto se ha debido a que las metas propuestas inicialmente sobrepasaban los recursos de este proyecto. A pesar de esto, se puede llevar a cabo con facilidad mediante la utilización de este componente y de los conocimientos acera de Redes Neuronales Convolucionales presentados en el capítulo 3.

Del desarrollo de este proyecto se extraen gran número de conclusiones y aprendizajes, se comentarán los más importantes. En cuanto al desarrollo del componente, se ratifica una vez más que el proceso de construcción de una solución software es un proceso de ingeniería, y que hasta los componentes más pequeños requieren de tiempo de análisis y de una compresión completa del problema. Además, un buen diseño de la solución permite que este problema sea robusto y que escale con facilidad. Otra de las conclusiones que se extrae es que diseñar y entrenar un modelo de *Machine Learning* es un proceso complejo que requiere de tiempo y gran capacidad de computo para obtener buenos resultados. Por lo que no se debe subestimar su carga de trabajo.

También se concluye que, efectivamente, estos algoritmos de Visión Artificial han alcanzado, en los últimos años, tal velocidad que permiten analizar la realidad en tiempo real. Esto es una gran noticia, pues abre un gran número de nuevos caminos y aplicaciones en las que se pueden utilizar —como la que se presenta en este trabajo, que tradicionalmente se había abordado desde el GPS— pero como todo avance tecnológico merece una reflexión, pues ya son muchos los proyectos de dudosa ética que se están presentando en este ámbito. Especialmente por nuestra parte, los desarrolladores,

que somos los que tenemos en nuestra mano decidir para qué utilizar estos conocimientos; debemos pararnos un momento a pensar a dónde queremos que toda esta tecnología nos lleve.

7.1. Lineas de trabajo futuras

En las futuras iteraciones sobre este trabajo, hay tres objetivos que se deberían analizar con prioridad. El primero debería ser retomar el objetivo que queda sin completarse, y por medio del *framework* Darknet, entrenar una red capaz de reconocer a que señal de velocidad se corresponde cada imagen. Después, utilizar el componente presentado en este trabajo para albergar esa nueva red. Como segundo objetivo, se podría presentar una interfaz más genérica para el método *configure_net*, puesto que es demasiado dependiente de Darknet (aunque esto permite utilizar las diferentes arquitecturas que este *framework* suministra de una manera mucho más sencilla). Por ultimo, sería interesante reducir el tamaño de la biblioteca *libdarknet.so*, pues se ha observado que contiene gran cantidad de archivos sin importancia, que tras un minucioso estudio del código se podrían eliminar y reducir enormemente su tamaño.

Una vez que estos objetivos estén analizados, los siguientes pasos son prácticamente ilimitados. Partiendo de la red suministrada se podría tomar un camino hacia una aplicación móvil de asistencia a la conducción, a un ordenador de abordo o incluso a un coche de conducción autónoma. Por supuesto, todos estos posibles itinerarios requieren de la incorporación de nuevos módulos, cosa que se facilita con el componente *CVNetComp*. Además de para esto, este componente puede ser utilizado para cualquier tipo de problema de Visión Artificial, en donde solo la imaginación del cliente pondría los límites. De esta manera, se podría añadir a la potencia de Darknet, la facilidad de uso, la robustez y la independencia de lenguaje de este componente.

Bibliografía

[1] Learning, then talking. *The New York Times* (1988).

[2] Matsuda, el robot candidato a la alcaldía de un distrito de tokio que promete acabar con la corrupción y "justicia para todos". *La Sexta* (Abr 2018).

[3] ABADI, M., AGARWAL, A., BARHAM, P., BREVDO, E., CHEN, Z., CITRO, C., CORRADO, G. S., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., GOODFELLOW, I., HARP, A., IRVING, G., ISARD, M., JIA, Y., JOZEFOWICZ, R., KAISER, L., KUDLUR, M., LEVENBERG, J., MANÉ, D., MONGA, R., MOORE, S., MURRAY, D., OLAH, C., SCHUSTER, M., SHLENS, J., STEINER, B., SUTSKEVER, I., TALWAR, K., TUCKER, P., VANHOUCKE, V., VASUDEVAN, V., VIÉGAS, F., VINYALS, O., WARDEN, P., WATTENBERG, M., WICKE, M., YU, Y., AND ZHENG, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] BELLO, I., ZOPH, B., VASUDEVAN, V., AND LE, Q. V. Neural optimizer search with reinforcement learning. Tech. rep., Sep 2017.

[5] BRADSKI, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[6] CASTLE, N. Supervised vs. unsupervised machine learning. `hhttps://www.datascience.com/blog/supervised-and-unsupervised-machine-learning-algorithms`.

[7] CORONA, S. La robot sophia: "los humanos son las criaturas más creativas del planeta pero también las más destructivas". *El País* (Abr 2018).

[8] DEEPMIND. Alphago zero: Learning from scratch. `https://deepmind.com/blog/alphago-zero-learning-scratch/`.

[9] DESHPANDE, A. The 9 deep learning papers you need to know about. `https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html`.

[10] DESHPANDE, A. A beginner's guide to understanding convolutional neural networks part 2. `https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/`.

[11] EVERINGHAM, M., VAN GOOL, L., WILLIAMS, C. K. I., WINN, J., AND ZISSERMAN, A. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision 88*, 2 (June 2010), 303–338.

[12] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[13] GIRSHICK, R. B. Fast R-CNN. *CoRR abs/1504.08083* (2015).

[14] GIRSHICK, R. B., DONAHUE, J., DARRELL, T., AND MALIK, J. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR abs/1311.2524* (2013).

[15] GOOGLECLOUD. A history of machine learning. `https://cloud.withgoogle.com/build/data-analytics/explore-history-machine-learning/`.

[16] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *CVPR* (2016), IEEE Computer Society, pp. 770–778.

[17] HOUBEN, S., STALLKAMP, J., SALMEN, J., SCHLIPSING, M., AND IGEL, C. Detection of traffic signs in real-world images: The German Traffic Sign Detection Benchmark. In *International Joint Conference on Neural Networks* (2013), no. 1288.

[18] KOCH, C. How the computer beat the go master. *Scientific American* (2016).

[19] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[20] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*. 1998, pp. 2278–2324.

[21] LIN, M., CHEN, Q., AND YAN, S. Network in network. *CoRR abs/1312.4400* (2013).

[22] LIN, T., MAIRE, M., BELONGIE, S. J., BOURDEV, L. D., GIRSHICK, R. B., HAYS, J., PERONA, P., RAMANAN, D., DOLLÁR, P., AND ZITNICK, C. L. Microsoft COCO: common objects in context. *CoRR abs/1405.0312* (2014).

[23] MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA, 2017.

[24] MCCARTHY, J. Arthur samuel: Pioneer in machine learning. `http://infolab.stanford.edu/pub/voy/museum/samuel.html`.

[25] NG, A. Convolutional neural networks. `https://www.coursera.org/learn/convolutional-neural-networks`. Accessed 12/03/18.

[26] NG, A. Deep learning specialization. `https://www.coursera.org/specializations/deep-learning`. Accessed 01/03/18.

[27] NG, A. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. `https://www.coursera.org/learn/deep-neural-network`. Accessed 06/03/18.

[28] NG, A. Machine learning. https://www.coursera.org/learn/machine-learning. Accessed 20/12/17.

[29] REDMON, J. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2018.

[30] REDMON, J., AND FARHADI, A. YOLO9000: better, faster, stronger. *CoRR abs/1612.08242* (2016).

[31] REDMON, J., AND FARHADI, A. Yolov3: An incremental improvement. *CoRR abs/1804.02767* (2018).

[32] REN, S., HE, K., GIRSHICK, R. B., AND SUN, J. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR abs/1506.01497* (2015).

[33] SIMONYAN, K., AND ZISSERMAN, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[34] SZEGEDY, C., LIU, W., JIA, Y., SERMANET, P., REED, S., ANGUELOV, D., ERHAN, D., VANHOUCKE, V., AND RABINOVICH, A. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)* (2015).

[35] TZUTALIN. Labelimg. https://github.com/tzutalin/labelImg, 2015.

[36] UJJWALKARN. A quick introduction to neural networks. https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/.

[37] V, A. S. Understanding activation functions in neural networks. https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

[38] ZEILER, M. D., AND FERGUS, R. Visualizing and understanding convolutional networks. *CoRR abs/1311.2901* (2013).

Apéndice A

Contenido del CD

- memoria.pdf
- Conf_files: Archivos de configuración de la Red Neuronal.
- CVNetComp: Implementación del componente.
- Images: Conjunto de pruebas.