Detección de vehículos en circulación mediante visión artificial y redes neuronales convolucionales

Detection of vehicles in circulation by artificial vision and convolutional neural networks

Fernando Pérez Gutiérrez

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin Máster en Ingeniería Informática

Jun. de 2020

Calificación: 9.5/10

Director:

Gonzalo Pajares Martinsanz

Autorización de difusión

Fernando Pérez Gutiérrez

Fecha

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Detección de vehículos en circulación mediante visión artificial y redes neuronales convolucionales", realizado durante el curso académico 2019-2020 bajo la dirección de Gonzalo Pajares Martinsanz en el Departamento de Ingeniería del Software e Inteligenica Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El presente trabajo está orientado a la detección de vehículos en movimiento utilizando secuencias de imágenes consecutivas (vídeos) sobre las que se aplican diferentes técnicas de visión artificial con el fin de obtener datos acerca del tránsito de vehículos en entornos urbanos bajo el contexto de las ciudades inteligentes.

En lo que se refiere a la detección del movimiento, se formula una propuesta computacional basada en una rama de la psicología, concretamente la percepción visual. Sobre esta propuesta teórica, se aplican algoritmos de detección de movimiento relativo entre un observador (la cámara) y la escena, esto es, flujo óptico.

Además, basado en la información obtenida de los algoritmos de flujo óptico, se obtienen las regiones de interés, estas son, regiones de la imágen donde se encuentran los objetos en movimiento. Partiendo de esta información, se hace uso de las redes neuronales convolucionales (CNN, *Convolutional Neural Network*) como herramienta específica dentro del aprendizaje profundo, para obtener información de los objetos detectados, véase, su categoría.

De esta forma, se diseñan un total de tres modelos de CNN, AlexNet, ResNet y YOLO, con hasta cinco variantes en el caso de ResNet. Así mismo, las arquitecturas mencionadas son entrenadas bajo un conjunto de datos adaptado al contexto de la detección y clasificación de vehículos.

Finalmente, se integran todas funcionalidades un sistema completo con la capacidad para discernir entre distintos tipos de vehículos según su categoría y pertenencia a las diferentes clases establecidas en las CNN.

Con todos los experimentos realizados, se hace a su vez, un análisis del diseño de las diferentes CNN y de los métodos de detección de movimiento en relación a su desempeño y sus casos de uso.

Palabras clave

Visión por computador, aprendizaje profundo, red neuronal convolucional, flujo óptico y ciudades inteligentes.

Abstract

The present work is oriented to the detection of moving vehicles using sequences of consecutive images (videos) applying different artificial vision techniques in order to obtain data about vehicle traffic in urban environments under the context of smart cities.

As far as motion detection is concerned, a computational proposal is formulated based on visual perception, a psychology branch. On this theoretical proposal, relative motion detection algorithms are applied between an observer (the camera) and the scene, that is, optical flow.

In addition, based on the information obtained from optical flow algorithms, the regions of interest are calculated. These are regions of the image where the objects in movement are located. With this information, the convolutional neural networks (CNN) are used as a specific tool within the deep learning. All of this with the objetive of obtaining information of the detected objects, for example, its category.

A total of three models of CNN, AlexNet, ResNet and YOLO, with up to five variants in the case of ResNet, are designed. Also, the mentioned architectures are trained under a dataset adapted to the context of vehicle detection and classification.

Finally, all functionalities are integrated into a complete system with the ability to distinguish between different types of vehicles according to their category and belonging to the different classes established in the CNN.

With all the experiments carried out, an analysis of the design of the different CNNs and motion detection methods is made in relation to their performance and use cases.

Keywords

Computer vision, deep learning, convolutional neuronal network, optical flow and smart cities.

Índice general

Índice de figuras

Agradecimientos

Quiero agradecer la ayuda y el apoyo recibido por parte de un gran número de personas sin las cuales este proyecto no se habría llevado a cabo.

En primer lugar, agradecer a Gonzalo Pajares por guiarme y ayudarme en todo lo que ha podido a lo largo del proyecto. Gracias a él me he adentrado en un campo de la inteligencia artificial (la visión artificial) que me apasiona y que seguramente que no hubiera podido llegar a conocer en tanta profundidad sin su ayuda, sus guías, sus correcciones, su dedicación al proyecto y en general, a enseñar. Todo ello en un curso muy convulso en el que la comunicación online ha sido clave y las circunstancias no han sido idóneas, por tanto, agradecer ahora más que nunca su gran labor.

En segundo lugar, a los profesores que me han dado clase tanto en la carrera como en el master, que me han permitido formarme y tener las cualidades y competencias que han hecho posible este proyecto.

Por último, agradecer a mi familia, y especialmente a mi hermano, por apoyarme y motivarme en todos los proyectos en los que me he ido embarcando.

Capítulo 1

Introducción

El desarrollo tecnológico computacional, tanto en capacidad de procesamiento como en posibilidad de almacenamiento en memoria, así como los avances teóricos, han propiciado el desarrollo de nuevos algoritmos.

Lo cual junto con la creciente necesidad de resolver problemas cada vez más complejos, han creado un caldo de cultivo perfecto para la aplicación de la inteligencia artificial en distintos ámbitos, entre ellos, el procesamiento de imágenes y vídeos, haciendo así, que en cierto sentido se pueda conseguir que una máquina posea una alta capacidad de interpretación de las imágenes y en definitiva aproximarse al sistema de percepción visual humano.

Dada la facilidad con la que "vemos", en 1960 los investigadores en inteligencia artificial pensaron que sería relativamente sencillo desarrollar sistemas dotados de visión artificial. Sin embargo, tras años de investigación, se ha comprobado que ésta es una tarea muy compleja.

Así mismo, estos sistemas de visión por computador tratan de resolver algunos de los problemas más importantes en la actualidad. Desde la detección y clasificación de patrones o la predicción de movimiento hasta problemas compuestos como la conducción automática o la implantación de ciudades inteligentes ("smart cities").

Este trabajo, se enmarca en el contexto del procesamiento de imágenes y secuencias consecutivas de imágenes (vídeos), o más concrétamente, en la detección de vehículos en circulación por medio de algoritmos de flujo óptico, para su posterior clasificación haciendo uso de redes neuronales convolucionales.

Este planteamiento sirve para determinar el flujo de vehículos en tránsito en un determinado punto, como parte de los sistemas a implantar en las futuras ciudades inteligentes.

1.1. Estado del arte

Actualmente existen muchos proyectos parecidos que tratan de abordar el problema de la detección de vehículos en movimiento, en tiempo real.

Algunos de los proyectos más relevantes desarrollados en el mismo contexto son:

A Vehicle Detection Approach using Deep Learning Methodologies [1]: Los autores del artículo tratan de resolver el mismo problema que en este trabajo, pero la aproximación que se realiza se basa en el uso de redes neuronales de tipo R-CNN [1]. Por tanto, si bien, dicho artículo y este proyecto, en lo que respecta a las CNN resuelven el problema siguiendo los mismos principios, en cuanto al método de obtención de regiones de interés son completamente diferentes, puesto que en el artículo se hace uso de algoritmos genéricos de selección de regiones y en este proyecto se utilizan las premisas conocidas de este problema concreto (los vehículos siempre se encuentran en movimiento). No obstante, ambas aproximaciones son completamente válidas.

Efficient Scene Layout Aware Object Detection for Traffic Surveillance [2]: Artículo ganador del concurso MIO-TCD[2][3], donde el objetivo era conseguir un sistema capaz de detectar y clasificar vehículos de 11 categorías diferentes y compararlos con el resto de proyectos[3]. En este artículo se aborda el problema haciendo primero un análisis de la escena que permite detectar objetos en la misma para posteriormente realizar la clasificación. Además, es importante destacar el rendimiento que se ha obtenido en este proyecto, siendo superior a otros conseguidos con FasterRCNN [4] y YOLOv1[5].

YOLO: Son las siglas de "You Only Look Once" y se presentó por primera vez en 2016 en el artículo *You only look once: Unified, real-time object detection* [5] y cuya última versión

[1] Las R-CNN (regions with CNN) son redes neuronales convolucionales en la cuál se utilizan algoritmos de selección de regiones a enviar a la red neuronal.

[2] The MIOvision Traffic Camera Dataset (MIO-TCD) Challenge.

[3] Para este proyecto se ha utilizado una versión reducida del dataset aquí proporcionado.

ha sido YOLOv3[6]. Estos artículos proponen una forma completamente diferente de resolver el problema, basada en observar la imágen una única vez, haciendo uso de una red neuronal que permite predecir las regiones de interés o *bounding boxes* y asociarles una clase con probabilidades de detección de los objetos. Esto permite obtener un sistema robusto y fluido (pudiendo alcanzar el procesamiento en tiempo real).

Conceptualmente YOLO sigue los siguientes pasos para resolver el problema de detección y clasificación de objetos en una imagen:

1. Dividir la imágen en una cuadrícula de SxS.
2. En cada celda predice N posibles *bounding boxes* y calcula la probabilidad (nivel de certidumbre)[4].
3. Se elimininan todas las *bounding boxes* que no superen un mínimo de probabilidad.
4. Se aplica "non-max suppression" para evitar detectar el mismo objeto múltiples veces con varias *bounding boxes* muy solapadas.

Como puede fácilmente deducirse, tanto YOLO como otros métodos genéricos de detección de objetos basados en redes neuronales convolucionales, analizan las imágenes en su conjunto y formulan propuestas de detección de objetos proporcionando una probabilidad de detección, que en el caso del presente trabajo serían vehículos en movimiento. La diferencia fundamental de la propuesta que se plantea en este trabajo con respecto a dichas propuestas reside en el hecho de que la propuesta de los objetos se determina por su movimiento, de forma que a la red neuronal se le proporcionan solamente regiones en las que se detecta movimiento para su clasificación, siendo aquí donde reside la novedad de la propuesta. La detección de esas regiones en movimiento se realiza aplicando técnicas específicas de visión por computador, más concretamente métodos de detección del flujo óptico.

[4]Por tanto, se calculan SxSxN posibilidades.

1.2. Objetivos

Con este trabajo se propone la creación de un sistema a nivel conceptual capaz de detectar vehículos en movimiento, a través de secuencias de imágenes, esto es, vídeos. Para ello se utilizan técnicas de flujo óptico, haciendo uso de dos algoritmos específicos, a saber: Lukas-Kanade y Gunner Furnebarck. Además, se probará la eficiencia y la eficacia de los mismos en este sistema concreto.

Con dicha información, el sistemá deberá ser capaz de generar secciones de cada fotograma donde se localiza cada uno de los vehículos en movimiento. Estas imágenes serán enviadas a una API Rest conectada a diferentes modelos de redes neuronales convolucionales (Resnet50 y Alexnet) con la intención de clasificar el tipo de vehículo que aparece en las mismas.

Por último se compararán los resultados obtenidos con cada una de las arquitecturas de las redes neuronales tanto en eficiencia, tiempo de respuesta, cómo en porcentaje de aciertos.

Así mismo, durante este proceso se realizará:

- Análisis y comparación de métodos de detección del flujo óptico en el contexto propuesto, seleccionando así el más apropiado.
- Selección de modelos de red y redefinición de capas.
- Comparación entre los modelos de red.
- Integración de las distintas tecnologías.

El plan de trabajo previsto tiene como base los objetivos específicos reseñados, contemplando el mismo número de fases que objetivos, y con una planificación equilibrada por semanas a lo largo del tiempo de desarrollo del proyecto.

1.3. Motivación

Este proyecto surge de la idea de resolver problemas actuales complejos tales cómo: recomendación de rutas alternativas en carretera para evitar atascos, detección temprana y captura de vehículos sospechosos, generación de estadísticas de vehículos por carretera, generación de predicciones de tráfico, etc. Todos estos problemas tienen un requisito en común, la extracción de datos relativos al movimiento de vehículos en vías urbanas e interurbanas.

La solución de este requisito tiene muchas aproximaciones con arquitecturas completamente diferentes, desde grandes redes de sensores, lo cuál se traduce en una costosa implantación en la sociedad así como cortes de carreteras para su colocación, hasta el uso de los teléfonos móviles que hagan la función de sensores portátiles. No obsante, estas ideas, requieren de una gran inversión o acceso a los dispositivos de muchos usuarios, y aún así estas soluciones no brindarán la posibilidad de obtener información sobre las características de los vehículos (tipo de vehículos, marca, etc.).

Con todo esto en mente, surge la necesidad de crear un sistema basado en la detección por medio de cámaras estáticas situadas en las cercanías de las carreteras, conectadas a uno o más servidores centrales que realicen el procesamiento de los datos.

De esta manera se conseguirían obtener todos los datos deseados a la vez que se evitan algunas desventajas del resto de aproximaciones. En efecto, implementar este sistema no requiere cortes de carreteras, ni una gran red de sensores, ni acceso a los dispositivos de miles de usuarios.

Así mismo, desde el principio este sistema se ha planteado con la intención de poder adaptarse a diferentes contextos, si bien las restricciones deben ser similares. Por ejemplo, cambiando el modelo, podría utilizarse en cámaras de seguridad para detectar y reconocer a personas en busca y captura.

Conviene reseñar que el desarrollo que se plantea es a nivel conceptual, dado que no existe posibilidad de implantación de un sistema real en espacios públicos por razones obvias. En cualquier caso, los análisis sobre los resultados permiten situarse en disposición de trasladar

el sistema a un entorno real, sin más que adaptar las tecnologías al mismo.

1.4. Organización de la memoria

La memoria está organizada en 4 capítulos, que complementan a este mismo. Concretamente en el capítulo 2 se describen las técnicas aplicadas a nivel teórico, incluyendo los métodos de detección de flujo óptico y las redes neuronales utilizadas. En el capítulo 3 se expone el diseño de la aplicación desde el punto de vista de la implementación e integración de los métodos utilizados. En el capítulo 4 se analizan los resultados obtenidos en la aplicacion de las diferentes tecnologías. Finalmente, en el capítulo 5 se incluyen las conclusiones y trabajo futuro.

Capítulo 2

Métodos utilizados

En este capítulo se describen en detalle todos los conceptos teóricos en los que se basa este proyecto para dar una solución al problema de la detección y clasificación de vehículos en movimiento de manera eficiente.

Los métodos se encuadran en los dos grandes bloques que se indican a continuación. A su vez, en cada uno de ellos se describen los métodos específicos que lo conforman.

1. Obtención de regiones de interés (ROI).
    - Cómputo del flujo óptico para detección de la magnitud y dirección del movimiento a nivel de píxel.
    - Utilización de técnicas de umbralización sobre el resultado del flujo óptico y obtención de regiones de interés a través de algoritmos de búsqueda de contornos.
2. Clasificación de de los objetos detectados en las regiones de interés por medio de Redes Neuronales Artificiales (RNA).

El capítulo se estructura como sigue. En la sección 2.1 se introduce el algoritmo de Shi-Tomasi como detector de puntos de interés en ROIs, que se puede utilizar en conjunción con los métodos de detección de flujo óptico que se abordan en la sección 2.2 (Lucas Kanade y Farneback). En la sección 2.3 se aborda el concepto de Redes Neuronales Artificiales, junto con la definición de las operaciones más relevantes involucradas en ellas. Finalmente, en la sección 2.4 se abordan plenamente las redes neuronales convolucionales.

2.1. Algoritmo Shi-Tomasi

El algoritmo de Shi-Tomasi[7] es un algoritmo de detección de esquinas y está basado en el de Harris[8] aunque alterando una función de la misma.

Para una mejor comprensión del mismo, es necesario entender cuáles son las características que definen las esquinas en una imagen. Las esquinas son puntos relevantes en una imagen caracterizados por grandes variaciones de intensidad en todas las direcciones. Por tanto, considerando una ventana [1] se calcula la variación en intensidad de los píxeles en dicha ventana y con ello se obtiene una puntuación dada. De este modo, se consideran esquinas aquellos píxeles donde esta puntuación supere un determinado umbral.

La diferencia entre el algoritmo de Harris y el de Shi-Tomasi se basa únicamente en la función utilizada para calcular dicha puntuación.

Por útlimo, es importante destacar que este algoritmo no es un detector de flujo óptico pero se puede utilizar en conjunto con el algoritmo de Lucas Kanade (explicado en la subsección 2.2.1) para seleccionar puntos de interés.

2.2. Flujo Óptico

El Flujo Óptico en secuencias de imágenes es el patrón de movimiento que se obtiene de los objetos provocado por el movimiento relativo entre el observador y la escena. En el contexto de este proyecto, el observador es la cámara y la escena son los objetos grabados.

James J. Gibson introdujo el concepto de Flujo Óptico[9] en la década de 1940 tras realizar experimentos con aviadores durante la II Guerra Mundial. Dichos experimentos le llevaron a la conclusión de que los estudios previos acerca de la percepción de profundidad apenas ayudaban a mejorar al piloto en sus aterrizajes. Así se originó su teoría de percepción espacial que enfatizaba las superficies continuas (en el caso del piloto, el suelo) en el que los objetos reposaban (tradicionalmente se hacía énfasis en dichos objetos).

[1] Píxeles cercanos a examinar en conjunto para encontrar una esquina.

En el ámbito de la computación[10], los diferentes algoritmos de flujo óptico intentan calcular el movimiento entre dos fotogramas consecutivos para cada píxel.

De esta forma se puede considerar un píxel como $I(x, y, t)$ siendo I la intensidad, x e y las coordenadas (en anchura y altura del píxel respectivamente) y t el tiempo. Además, (dx, dy) es la distancia que se ha movido el píxel en el siguiente fotograma transcurrido el tiempo dt. Por tanto, asumiendo que el valor de intensidad de esos píxeles no varía significativamente:

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

Aproximando por series de Taylor, eliminando terminos comunes y dividiendo por dt se obtiene la siguiente ecuación:

$$f_x u + f_y v + f_t = 0 \tag{2.1}$$

Siendo:

$$f_x = \frac{\partial f}{\partial x};\ f_y = \frac{\partial f}{\partial y}$$

$$u = \frac{dx}{dt};\ v = \frac{dy}{dt}$$

Ésta es la ecuación del flujo óptico. En ella f_x y f_y son gradientes espaciales de la imagen, mientras que f_t es el gradiente a lo largo del tiempo. No obstante, u y v son las incógnitas a resolver y que definen el flujo óptico, siendo respectivamente las componentes horizontal y vertical con respecto a las imágenes. Por tanto, dado que no se puede resolver matemáticamente un problema con una ecuación y dos incógnitas, existen varios algoritmos que tratan de resolverlo.

2.2.1. Algoritmo Lucas Kanade

El algoritmo de Lucas Kanade[11] asume que el flujo es esencialmente constante para los píxeles cercanos al píxel objetivo (axioma). Resolviendo así las ecuaciones de flujo óptico 2.1 para los píxeles cercanos y aplicándoles el criterio de mínimos cuadrados.

Con esta estrategia de combinar información de varios píxeles (y siempre que se cumpla el axioma) se resuelve la ambigüedad de la ecuación de flujo óptico (dos incógnitas para una

ecuación).

La solución final es:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i^2} \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix} \tag{2.2}$$

De esta forma, se obtiene una de las soluciones más utilizadas para resolver el problema del flujo óptico. Aún así este algoritmo plantea nuevos problemas.

Lucas Kanade se aplica sobre puntos concretos[2] de la imagen, esto implica la utilización de algún tipo de método de selección de puntos de interés. Los puntos considerados más relevantes son los contornos de los objetos (se ha comprobado que éstos son los mejores puntos para aplicar el método de Lucas Kanade).

Existen muchos algoritmos de detección de contornos, si bien para este proyecto se ha utilizado el algoritmo de Shi-Tomasi (cuyos fundamentos teóricos se explican en la sección 2.1).

Por otro lado, surge otro problema. El algoritmo de Shi-Tomasi se tiene que aplicar sobre un fotograma concreto, encontrando así los píxeles relevantes del mismo.

Pero, dado que constantemente entrarán y saldrán vehículos de la zona de visión de la cámara, necesariamente surge la pregunta ¿cada cuántos fotogramas debe el sistema realizar una nueva búsqueda de píxeles relevantes?

La respuesta a esta pregunta depende enteramente del promedio de tiempo que se mantengan los vehículos en la zona de visión. Pero éste es un dato del que no se dispone información a priori desde el punto de vista de aplicación del método, por tanto, en el contexto de este proyecto se decidió tras diversas pruebas de ensayo y error, aplicarlo cada 2 segundos, teniendo en cuenta el número de *frames* por segundo.

En la figura 2.1 se muestra un ejemplo de aplicación del algoritmo de Lucas Kanade, considerando un tamaño de vecindad 5x5 (ventana) para el cálculo de las derivadas espaciales y temporales. En cada unos de los objetos (vehículos) en movimiento se observan, en forma

[2]Es por esto que pertenece al conjunto de los algoritmos de flujo óptico dispersos.

de flechas sobre cada uno de los vehículos, los vectores de flujo, a partir de los cuales se puede determinar tanto su magnitud (módulo) como su dirección y sentido.

Figura 2.1: Ejecución del algoritmo Lucas Kanade.

A este problema hay que añadirle el hecho de que el algoritmo de Lucas Kanade junto con el algoritmo de Shi-Tomasi sólo permiten obtener y realizar un seguimiento de los píxeles en movimiento, pero no ayudan a obtener de una manera intuitiva las regiones de interés[3] (objetivo final de la primera fase del proyecto).

Es debido a estos problemas por lo que se optó por una aproximación diferente. Analizar todos los puntos de la imagen para obtener regiones en movimiento (explicado en la subsección 2.2.2) y sobre ellas aplicar técnicas de la umbralización (para reducir el ruido) y finalmente algoritmos de detección de contornos (sobre el resultado de la umbralización) para obtener las regiones de interés (explicado en la subsección 2.2.3).

[3]Puesto porque no se puede saber a priori cuáles de los píxeles seleccionados pertenecen a un mismo objeto.

2.2.2. Algoritmo Gunnar Farneback

Algoritmo desarrollado en 2003 por Gunnar Farneback[12]. A diferencia del algoritmo de Lucas Kanade (explicado en la subsección 2.2.1), el algoritmo de Gunnar Farneback calcula el flujo óptico para todos los píxeles de la imagen[4].

El algoritmo trata de aproximar los conjuntos de píxeles que pertenecen a una misma región[5] utilizando polinomios cuadráticos en dos fotogramas consecutivos. Obteniendo la siguiente señal expresada en un sistema de coordenadas local:

$$f(x) \sim x^T \mathbf{A} x + \mathbf{b}^T x + c \tag{2.3}$$

Siendo $\mathbf{A}$ una matríz simétrica, $\mathbf{b}$ un vector y c un escalar. Los coeficientes se aproximan por el criterio de mínimos cuadrados ponderados para todos los píxeles dentro de un entorno de vecindad a uno dado. La ponderación consta de dos componentes, certeza y aplicabilidad.

La certeza está asociada a los valores de los píxeles en la vecindad. Además, fuera de la imagen la certeza es cero, de esta forma estos píxeles no tienen impacto en el coeficiente de estimación del flujo.

La aplicabilidad determina el peso relativo de los píxeles en la vecindad basada en su posición dentro de la misma, de esta forma se consigue dar más peso al píxel central y hacer que disminuya radialmente desde dicho píxel a medida que se aleja del central. Además, el tamaño de la ventana de aplicabilidad determinará la escala de las estructuras que se quieran capturar por el coeficiente de expansión.

A partir de aquí, y suponiendo una vecindad que se ajuste a la ecuación 2.3 explicada anteriormente[6] se puede calcular un desplazamiento global $\mathbf{d}$ obteniendo la nueva señal f_2

[4] Es por esto que pertenece al conjunto de los algoritmos de flujo óptico densos.

[5] En su artículo[12], Gunnar Farneback se refiere a estos cómo los píxeles del mismo "vecindario".

[6] En la práctica es poco realista que una vecindad cumpla exáctamente la ecuación 2.3, pero es útil a modo explicativo.

tal que:

$$\begin{aligned} f_2(x) = f(x-d) &= (x-d)^T A(x-d) + b^T(x-d) + c \\ &= x^t Ax + (b-2Ad)^T x + d^T Ad - b^T d + c \\ &= x^T A_2 x + b_2^T x + c_2 \end{aligned} \tag{2.4}$$

Igualando los coeficientes con f_x se obtiene:

$$A_2 = A \tag{2.5}$$

$$b_2 = b - 2Ad \tag{2.6}$$

$$c_2 = d^T Ad - b^T d + c \tag{2.7}$$

Así, gracias a la ecuación 2.6 se puede resolver el desplazamiento $\mathbf{d}$ como sigue:

$$2A\mathbf{d} = -(b_2 - b) \tag{2.8}$$

$$d = -\frac{1}{2}A^{-1}(b_2 - b_1) \tag{2.9}$$

De esta forma se puede obtener un vector $\mathbf{d}$ para cada píxel con la estimación de la dirección y magnitud del movimiento en dicho píxel.

En la figura 2.2 se muestra el resultado de aplicar el algoritmo de Farneback, observándose las magnitudes en función de la tonalidad de color que envuelven a cada objeto en movimiento.

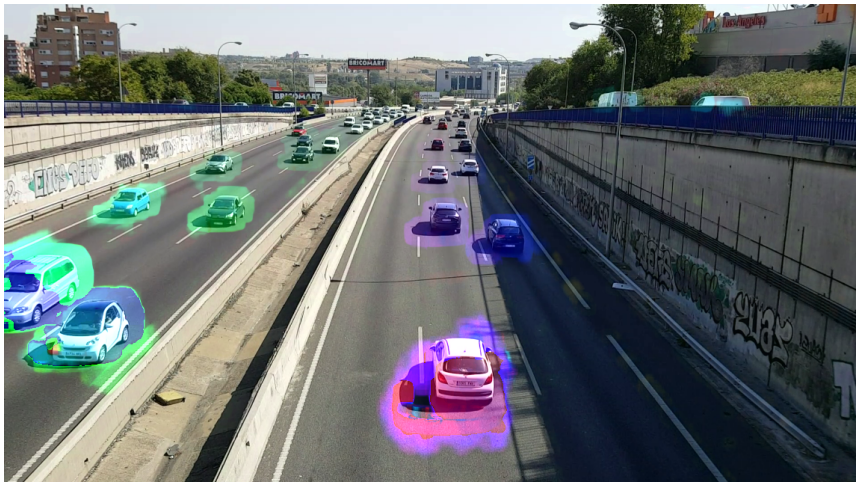


Figura 2.2: Ejecución del algoritmo Farneback.

Finalmente es importante destacar, que todo esto se aplica sobre un caso teórico ideal. Para resolver el problema en la práctica, el algoritmo de Gunnar Farneback requiere otro tipo de ajustes y estimaciones para reducir el error y el ruido (tal y como se analiza en su artículo[12]).

2.2.3. Obtención de Regiones de Interés

Partiendo de una matríz, en la cuál cada elemento hace referencia a la magnitud del movimiento de un píxel con respecto al fotograma previo, siendo los valores mínimo y máximo, 0 y 255 respectivamente [7] se pueden aplicar diversas técnicas sencillas de umbralización con el fin de eliminar ruido[8].

Para este proyecto y dada la naturaleza del problema es suficiente con utilizar umbralización binaria[9]. Esto implica que todos los píxeles cuyo valor de intensidad sea inferior a un umbral previamente fijado, $\mathbf{T}$, se le asigna el valor 0 (negro) y al resto se le asigna el valor 255 (blanco).

A partir de este punto, conviene añadir que existen diferentes técnicas con las que proceder, habiéndose optado por obtener los bordes delimitadores de cada objeto y con ellos extraer las coordenadas menor y mayor en uno de los ejes horizontal y vertical. Consiguiendo así una región de interés con forma rectangular, que permite realizar un recorte sobre la imagen original constituyendo la entrada a la red neuronal para clasificar el objeto que contiene.

El algoritmo de detección de contornos se explica en detalle en Suzuki[13].

[7] De esta forma se puede visualizar fácilmente dicha matríz en escala de grises. Siendo las regiones blancas las de mayor movimiento y las negras las de menor.

[8] Dicho ruido aparece por movimientos pequeños (que se desean ignorar) o sombras.

[9] Naturalmente, existen algoritmos más complejos, como la umbralización adaptativa o el método de Otsu.

2.3. Redes Neuronales Artificiales

Las redes neuronales artificiales son sistemas conceptualmente basados en el comportamiento de las neuronas de los seres vivos. Estos sistemas son capaces de encontrar patrones en conjuntos de datos.

Inicialmente el sistema debe ser entrenado con conjuntos de datos que tengan tanto la entrada como la salida deseable por parte del sistema. La entrada son todas aquellas muestras sobre las que se quiere encontrar un patrón y la salida es el valor a predecir para dichas muestras.

En un ejemplo práctico (sacado de Aldabas-Rubira[14]), la entrada puede ser una matríz de píxeles con valores 0 o 1 (blanco o negro), mientras que la salida clasifica las entradas en 11 patrones. Siendo éstos, los números de 0 al 9, guión y punto, tal y como se puede ver en la figura 2.3.

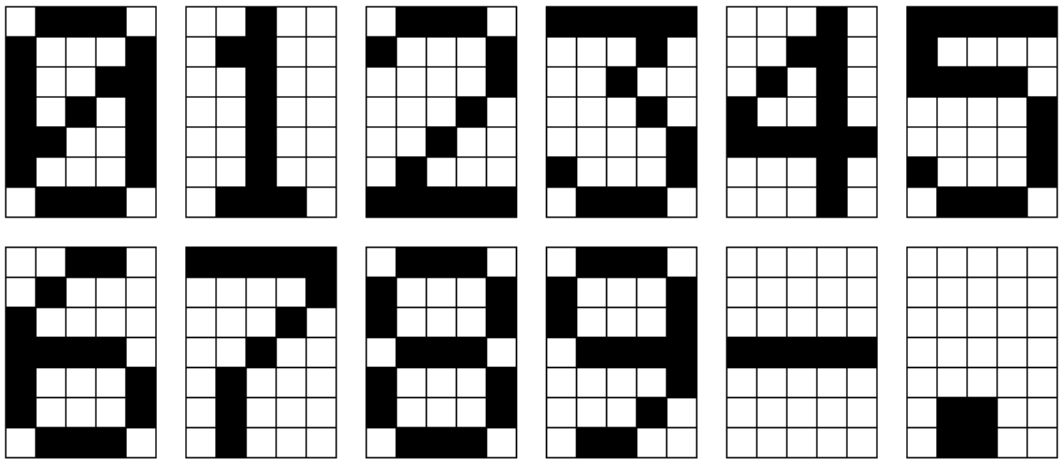


Figura 2.3: Representación gráfica de los patrones de los caracteres numéricos.

A partir de este punto un sistema correctamente entrenado debería, no sólo ser capaz de reconocer sobre el conjunto de datos con el que se ha entrenado, sino también sobre datos nuevos del mismo contexto. De esta forma, con los datos de la figura 2.3 como conjunto de entrenamiento, debería reconocer la nueva entrada (figura 2.4) como el patrón “1”.

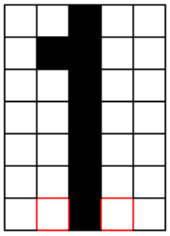


Figura 2.4: Muestra de testeo con el patrón “1” incompleto.

En este punto es necesario entender cómo una red neuronal artificial es capaz de generalizar a través de un conjunto de datos limitado.

Una vez establecidos los fundamentos de una red neuronal, es necesario explicar en detalle algunos conceptos computacionales que permiten obtener el comportamiento previamente explicado.

2.3.1. Métodos de optimización

La optimización consisten en minimizar o maximizar una función *f(x)* ajustando el parámetro *x*. En la práctica, se habla únicamente de minimización puesto que maximizar *f(x)* es equivalente a minimizar *-f(x)*.

Cuando se está minimizando, a esta función se le denomina **cost function**, **loss function** o **error function**. Así mismo, el valor de x que consigue la minimización se identifica con el símbolo *.

$$x^* = arg\ min\ f(x) \tag{2.10}$$

Para encontrar dicho valor existen diferentes métodos, no obstante, uno de los más utilizados en el ámbito de las redes neuronales es el basado en el gradiente.

La derivada de una función en un punto x proporciona la pendiente de la función en dicho punto. Por tanto, la derivada permite escalar un pequeño cambio en la entrada para obtener el correspondiente cambio en la salida:

$$f(x + \epsilon) \approx F(x) + \epsilon f'(x) \tag{2.11}$$

De esta forma, la derivada es útil para minimizar una función porque indica cómo cambiar

x para conseguir una pequeña mejora en la salida y. Por ejemplo, se sabe que,

$$f(x - \epsilon * (f'(x)) < f(x) \tag{2.12}$$

para un ϵ lo suficientemente pequeño. Se puede entonces disminuir *f(x)* moviendo x en pequeños pasos con el signo opuesto a la derivada. A este proceso se le conoce como **descenso de gradiente**.

La figura 2.5 muestra un ejemplo de cómo el descenso de gradiente utiliza derivadas de una función (2.13 y 2.14) descendiendo hacia un mínimo.

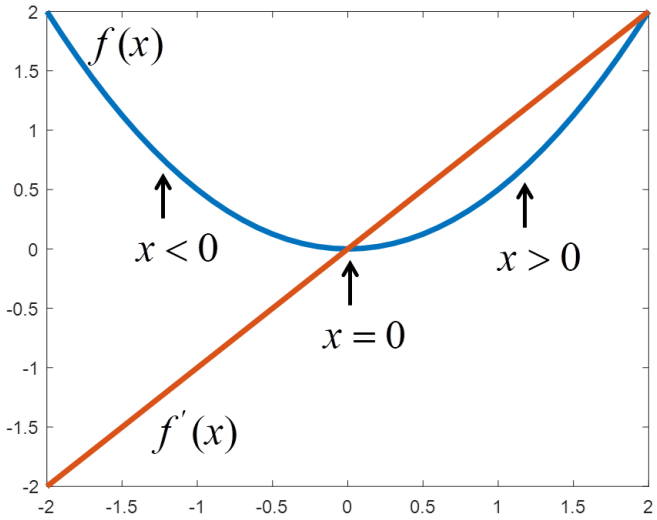


Figura 2.5: Gradiente descendente.

$$f(x) = \frac{1}{2}x^2 \tag{2.13}$$

$$f'(x) = x \tag{2.14}$$

Con esta información se puede inferir el siguiente comportamiento:

$$\begin{cases} f \text{ } disminuye \text{ } hacia \text{ } la \text{ } derecha & si \quad x < 0, f'(x) < 0 \\ f \text{ } disminuye \text{ } hacia \text{ } la \text{ } izquierda & si \quad x > 0, f'(x) > 0 \\ el \text{ } gradiente \text{ } se \text{ } detiene & si \quad x = 0 \end{cases} \tag{2.15}$$

Es importante destacar que cuando *f'(x) = 0* la derivada no proporciona información acerca de la dirección en la que se debe desplazar la x. A este tipo de puntos se les conoce como *puntos críticos* o *puntos estacionarios*.

Además, un mínimo local es un punto donde *f(x)* es menor que todos los puntos vecinos, de modo que no es posible disminuir *f(x)* con esta estrategia de pasos infinitesimales.

Por tanto, y dado que pueden existir múltiples mínimos locales (especialmente cuando la entrada a la función es multidimensional), optimizar la función para obtener un mínimo global puede llegar a ser una tarea muy costosa.

En el ámbito del aprendizaje profundo esta tarea de optimización es especialmente compleja dado que se pueden llegar a utilizar muchas dimensiones en casos reales. Por lo tanto, en general es necesario conformarse con un valor que sea muy bajo pero no necesariamente óptimo en ningún sentido formal. Ver figura 2.6 cómo ejemplo ilustrativo.

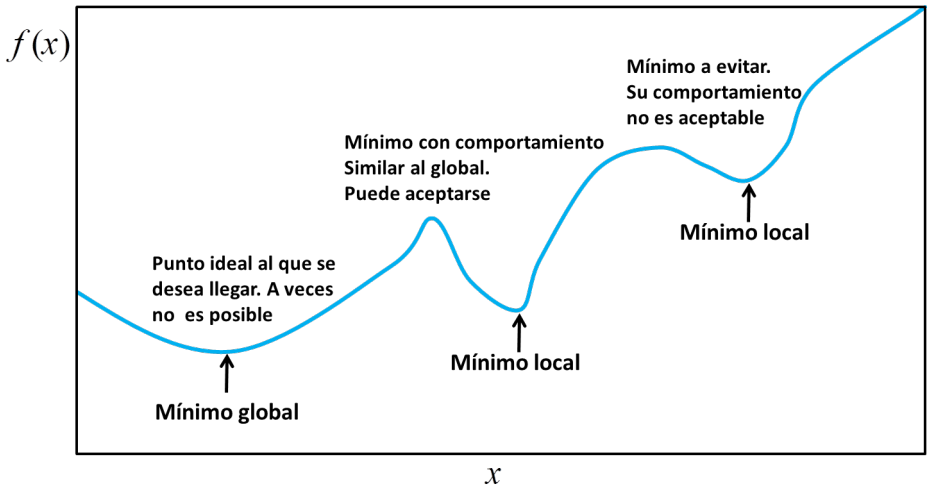


Figura 2.6: Puntos críticos.

2.3.2. Stochastic Gradient Descent (SGD)

Cuando se habla de aprendizaje automático la funcion a minimizar tiene la forma:

$$J(w) = \frac{1}{n}\sum_{i-1}^{n} n^2 J_i(w) \tag{2.16}$$

El parámetro w que minimiza *J(w)* es el que debe estimarse. J_i se asocia con la i-ésima

observación en el conjunto de datos utilizados para el ajuste (fase de entrenamiento). El término estocástico proviene del hecho relativo a la selección de las muestras (observaciones) para el ajuste de forma aleatoria. *$J_i(w)$* es el valor de la función de coste en el i-ésimo ejemplo y *$J(w)$* es el **riesgo empírico**.

El descenso de gradiente se usa para minimizar esta función de forma iterativa (con iteraciones t):

$$w(t+1) = w(t) - \epsilon \frac{1}{n} \sum_{i=1}^{n} \nabla J_i(w) \tag{2.17}$$

De forma iterativa el método recorre el conjunto de entrenamiento y realiza la actualización anterior para cada muestra de entrenamiento. Se pueden realizar varios pasos sobre el conjunto de entrenamiento hasta que el algoritmo converja. Además, para evitar ciclos los datos se pueden seleccionar aleatoriamente en cada paso (criterio estocástico). Estas muestras así seleccionadas constituyen un **batch**. Las implementaciones típicas usan una razón de aprendizaje adaptativa para que el algoritmo converja.

En pseudocódigo, el descenso de gradiente estocástico sigue los siguientes pasos:

1. Seleccionar vector inicial de parámetros w (puede ser aleatoriamente) y la marcar la razón de aprendizaje ϵ.

2. Repetir hasta que se consigue un mínimo aproximado

    2.1. Seleccionar aleatoriamente ejemplos en el conjunto de entrenamiento

    2.2. Para i = 1,2,...,n, hacer: $w(t+1) = w(t) - \epsilon \nabla J_i(W)$

Se puede leer más sobre uso de batches y optimización en la fase de entrenamiento en Ioffe y Zegedy[15]. Más información sobre descenso de gradiente y ejemplos en el Lewis[16].

2.3.3. Funciones de activación

La función de activación en una neurona artificial o perceptrón es el análogo al **potencial de acción** en las redes neuronales biológicas. Su función es determinar cuán estimulada está una neurona ante un determinado estímulo.

En el campo de las redes neuronales artificiales, una función de activación clásica es la función sigmoidal:

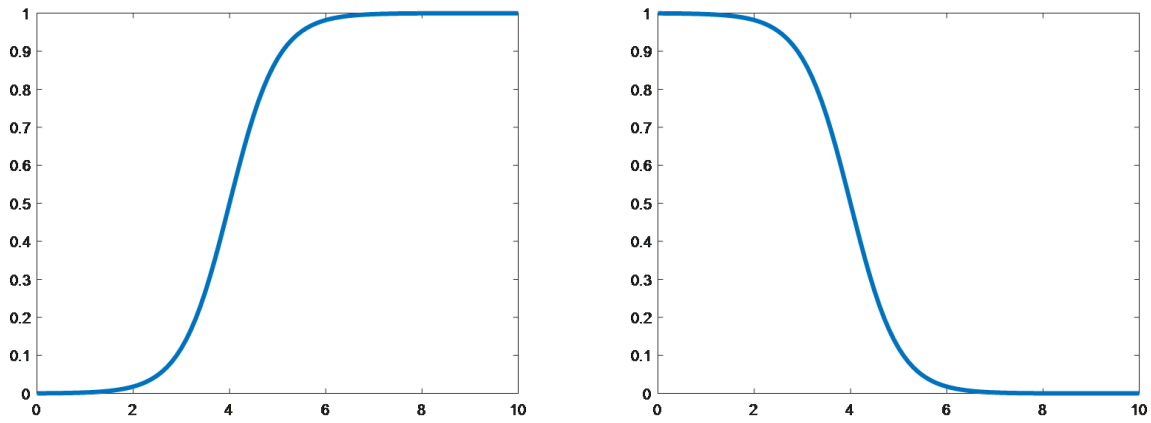
$$f(a, x, c) = \frac{1}{1 + e^{-a(x-c)}} \tag{2.18}$$

Dependiendo del signo del parámetro a, la función sigmoide se abre hacia la izquierda o hacia la derecha, siendo apropiada para representar conceptos tales como "muy grande" o "muy negativo".

La función sigmoide proyecta salidas en el intervalo [0,1] para números reales y posee los siguientes problemas:

- Cuando el valor de la función de activación se aproxima a los extremos (0 ó 1), el gradiente de la función tiende a 0, lo que repercute en el ajuste de los pesos de las redes.
- El valor medio de la función de salida no es 0 y por tanto, los pesos tiendan a ser positivos.

En la figura 2.7 se muestra la representación gráfica de la función sigmoide con distintos parámetros.



(a) a = 2; c = 4 (b) a = -2; c = 4

Figura 2.7: Función sigmoide

Los problemas mencionados previamente provocan una convergencia lenta de los parámetros afectando a la eficiencia del entrenamiento. Es por esto que existen otras funciones de activación y no hay una respuesta correcta para todos los problemas.

La función *tanh* (figura 2.8) proporciona salidas reales en el rango [-1,+1], siendo una variante de la función *sigmoide*, esto es:

$$tanh(x) = 2sigmoid(2x) - 1 \tag{2.19}$$

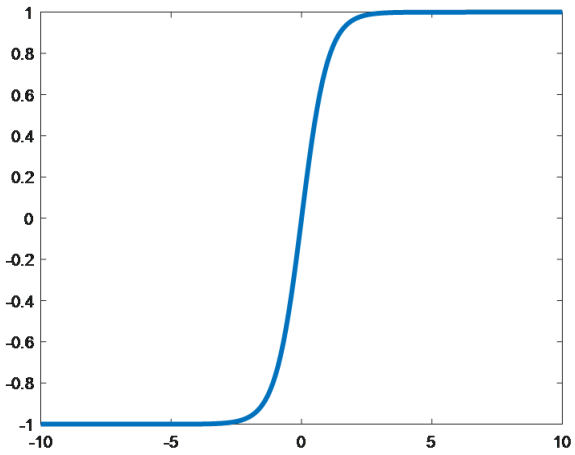


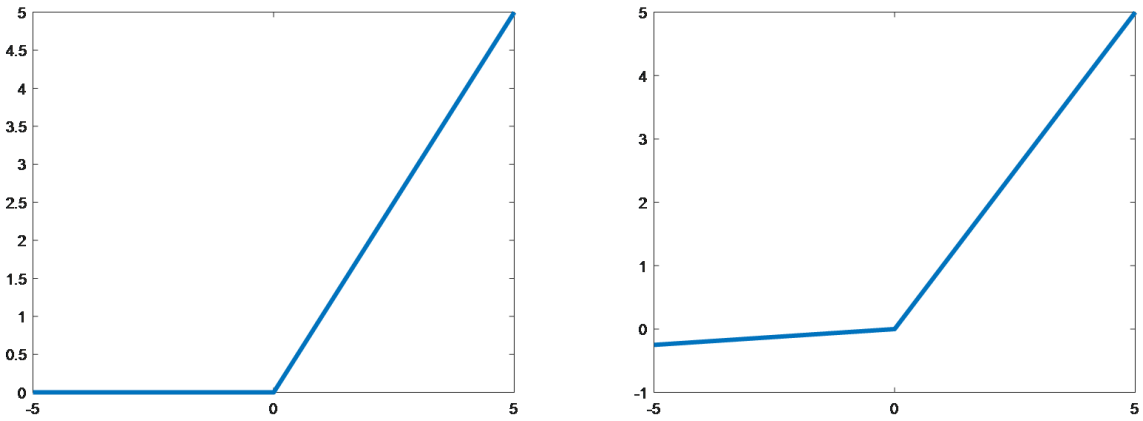
Figura 2.8: Función tangente hiperbólica.

Esta función presenta el mismo problema de la saturación del gradiente que la función sigmoidal.

La función ReLU (*Rectified Linear Unit*)[17][18][19][20], representada en la figura 2.9a tiene las siguientes características:

- Gradiente no saturado, por el hecho de que x >0. Por lo tanto el problema de la dispersión del gradiente en el proceso de propagación inversa se ve aliviado y los parámetros en la primera capa de la red neuronal pueden actualizarse rápidamente.
- Baja complejidad computacional debido a su propia definición.

- Como desventaja, una neurona ReLU puede dejar de ser útil cuando recibe un gradiente negativo alto durante la retropropagación que le impide aprender más porque su derivada es cero cuando la entrada es menor a cero. Esto se puede evitar inicializando cuidadosamente los pesos o utilizando la variante de la función ReLU conocida como Leaky ReLU (ReLU con "fugas") representada en la figura 2.9b. Dicha variante hace que la derivada no sea cero para valores negativos, y por tanto pueda seguir aprendiendo incluso cuando recibe gradientes negativos.



(a) ReLU
$f(x) = max(0, x)$

(b) Leaky ReLU
$f(x) = max(0{,}01x, x)$

Figura 2.9: Funciones lineales

2.4. Redes Neuronales Convolucionales

Las redes neuronales convolucionales son un tipo especializado de redes neuronales. Cuentan con una topología basada en rejilla para el procesamiento de datos, tales como series temporales o imágenes, siendo rejillas de una y dos diemensiones respectivamente.

A lo largo de este capítulo se siguen especialmente las referencias[21] [22].

El término *convolucional* hace referencia a la operación matemática de *convolución*[10]. De

[10]La operación de convolución se aborda aquí desde el punto de vista de las redes neuronales, y por tanto no se corresponde exactamente con el mismo concepto aplicado en otros ámbitos tales como el procesamiento de señales o desde el concepto matemático puro.

esta forma puede decirse que las CNN (Convolutional Neural Network) son redes neuronales que utilizan la convolución (en lugar de la multiplicación de matrices), en al menos una de sus capas.

Además, en casi todas las CNN se aplica una operación denominada *pooling*, la cual, en términos generales, consiste en una agrupación de píxeles mediante alguna operación de agrupación realizada sobre una ventana en los resultados intermedios obtenidos tras la convolución.

2.4.1. Capas de Convolución

En lineas generales, la operación de convolución involucra dos funciones con valores reales como argumento.

Supóngase que se tiene una fuente de luz variable cuya intensidad se recibe mediante un sensor que proporciona una salida en una determinada posición x, en el tiempo t, esto es $x(t)$[11]. Debido a la variabilidad de la fuente mencionada, se pueden obtener diferentes lecturas en distintos instantes de tiempo. Si a esto se le añade que la captura de la señal por el sensor puede estar contaminada por un cierto ruido, para obtener una señal más precisa, lo acertado es realizar un promedio de la salida con varias medidas. Si además tenemos en cuenta que las medidas más recientes son más relevantes que las alejadas en el tiempo, el promedio puede ponderarse concediendo así más peso a las más recientes. Esto puede hacerse mediante la función de promedio $w(a)$[12], donde a representa el alejamiento de la medida en el tiempo. Si se realiza esta operación de promedio ponderado en cada instante de tiempo t, se obtiene una función nueva promediada como sigue:

$$s(t) = \int x(a)w(t-a)da \tag{2.20}$$

A esta operación se le denomina **convolución** y se denota como sigue,

[11] Tanto x como t son valores reales.

[12] Hay que matizar que w necesita ser una función de densidad de probabilidad. Adenás w debe ser cero para todos los argumentos negativos (para evitar la toma de valores en el futuro, algo que no es materialmente posible).

$$s(t) = \int (x * w)(t) \tag{2.21}$$

En el ámbito de las CNN, el primer argumento x para la convolución se conoce como entrada (*input*) y el segundo w, como núcleo (*kernel*) de convolución. La salida s se define generalmente como mapa de características (*feature map*).

Desde el punto de vista computacional, y siguiendo con el ejemplo de la fuente de luz, las medidas son discretas, de forma que se realizan medidas en intervalos de tiempo determinados, por ejemplo, cada segundo. De esta forma, el tiempo t puede tomar sólo valores enteros, por tanto x e y se definen sólo en el tiempo t con valor entero, quedando por tanto, la convolución discreta como sigue.

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.22}$$

En inteligencia artificial, la entrada es un vector o matriz multidimensional de datos, y el núcleo es, generalmente, un vector o matriz multidimensional de parámetros, que se ajustan, mediante el proceso de aprendizaje. Dichas estructuras multidimensionales se conocen como tensores. Puesto que cada elemento de la entrada y del núcleo deben almacenarse separadamente, los valores que no pertenecen a esos elementos deben considerase nulos, lo que en la práctica es una suma finita de valores en un número finito de elementos.

Por otra parte, las convoluciones se realizan sobre más de un eje a la vez, tal es el caso de las imágenes I, cuando constituyen las entradas, que se trata de estructuras bi-dimensionales a tratar con un núcleo de entrada bi-dimensional K.

$$S(i,j) = (I * K)(i,j) = \sum_{m} \sum_{n} I(m,n)K(i-m, j-n) \tag{2.23}$$

Una propiedad de la convolución es que es conmutativa [13], por tanto, la expresión anterior puede escribirse de forma equivalente como sigue:

[13]La propiedad conmutativa de la convolución se interpreta como una forma de reflejar el núcleo respecto de la entrada, de forma que a medida que el índice m aumenta, el índice en la entrada crece a la vez que disminuye en el núcleo.

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.24}$$

Esta segunda forma de expresarlo resulta en una menor variación en los valores m y n, lo que permite obtener una implementación más eficiente.

En el contexto de las CNN es muy importante el concepto de correlación cruzada:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.25}$$

La figura 2.10 muestra un ejemplo de convolución con el núcleo K, sin reflexión de núcleo, aplicado sobre un tensor de dos dimensiones, que bien puede representar una imagen, I. La convolución se representa con el solapamiento total del núcleo y obteniendo una imagen cuyos bordes externos quedan sin valores, por tanto, obteniendo una imagen de menor dimensión que la original (en el caso de la figura se pasa de tener dimensión 6x7 a 4x5). No obstante, si se desea obtener una imagen de la misma dimensión, lo que se debe hacer es ampliar la imagen original con dos filas (arriba y abajo) y dos columnas (izquierda y derecha) con ceros, procesándola con el núcleo solapado[14].

Como se ha indicado previamente, la convolución en la figura 2.10 es de dos dimensiones (2-D). También podría ser de tipo 3-D, por tanto el núcleo sería un cuboide y se desplazaría a través de las dimensiones largo, ancho y alto del mapa de características.

Por ejemplo, si se tiene una imagen RGB y por tanto con tres canales para la capa de entrada se utilizan núcleos de tipo cuboide (con tres dimensiones). Esto se puede generalizar a convoluciones de tipo N-D, siendo N el número de dimensiones.

Además, en la figura anterior, el desplazamiento del núcleo es de una posición a la siguiente, no obstante, el desplazamiento puede ser de más de una unidad, a esto se le conoce como *stride*.

Así pues la colección de núcleos que definen una convolución discreta tiene una forma que corresponde a alguna permutación del tipo (n, m, k_1, ... , k_N), siendo:

[14] A esta operación se le conoce como zero-padding.

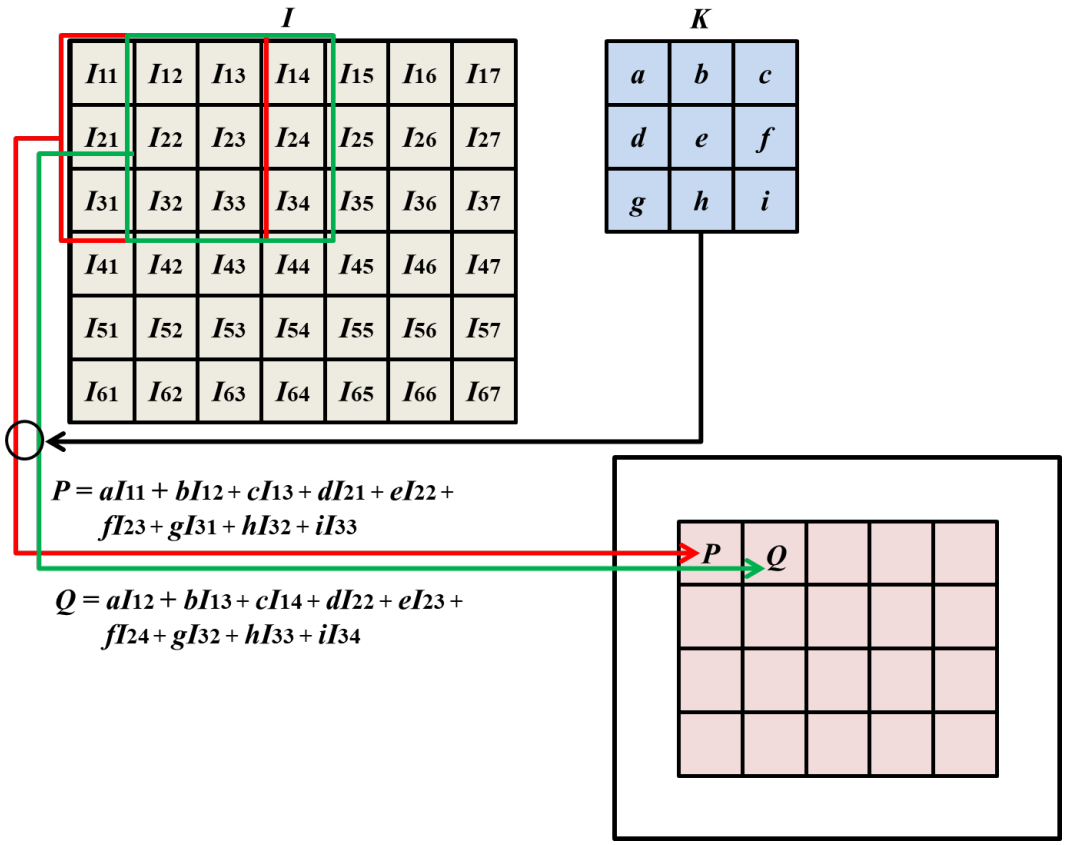


Figura 2.10: Ejemplo de convolución con dos dimensiones

n ≡ número de mapas de características de salida,

m ≡ número de mapas de características de entrada,

k_j ≡ dimensión del núcleo a lo largo del eje j.

La dimensión o_j de una capa de convolución a lo largo del eje j tiene las siguientes propiedades:

i_j ≡ dimensión de entrada a lo largo del eje j,

k_j ≡ dimensión del núcleo a lo largo del eje j,

s_j ≡ *stride* a lo largo del eje j,

p_j ≡ número de ceros concatenados al comienzo y al final del eje j (zero-padding),

Generalmente, es habitual que los valores de i, k, s y p en todos los ejes tengan el mismo valor.

La figura 2.11 muestra un ejemplo ilustrativo sobre el proceso de convolución paso a paso. Como se puede observar, se obtiene el producto entre cada elemento del núcleo y el elemento de la entrada sobre el que se solapa. Finalmente los resultados se suman para obtener la salida en la posición actual.

Dicho procedimiento se repite utilizando diferentes núcleos para formar tantos mapas de características como se deseen.

En la figura 2.12 se representa una operación de convolución a partir de tres mapas de características de entrada para obtener cuatro mapas de características de salida, utilizando una colección de núcleos w de 3x3x3x4. De esta forma, el mapa de características 1 se convoluciona con el núcleo $w_{1,1}$, el mapa de características 2, con el $w_{1,2}$ y el mapa de características 3 con el núcleo $w_{1,3}$ de forma que los resultados se suman elemento a elemento para formar el primer mapa de características de salida.

Repitiendo este mismo procedimiento para las otras 3 figuras se obtendrían los 3 mapas de características de salida restantes. Finalmente se agrupan de forma conjunta los cuatro mapas de salida.

Extrapolando, para un volumen de entrada con dimensiones de tensor HxWxC, donde H y W son las dimensiones alto y ancho y C el número de canales, si se aplican N filtros de dimensión k_h x k_w x C, representados como k_h x k_w x C x N se obtiene un volumen con N canales[15].

2.4.2. Capas de Pooling (agrupación o concentración)

En las redes neuronales, las capas de *pooling* proporcionan una importante invarianza a pequeñas traslaciones de la entrada. Existen varias operaciones de este tipo, no obstante, las dos más utilizadas son:

[15]El coste computacional de esta operacion viene dado por: H $\cdot W \cdot C \cdot k_h \cdot k_w \cdot N$

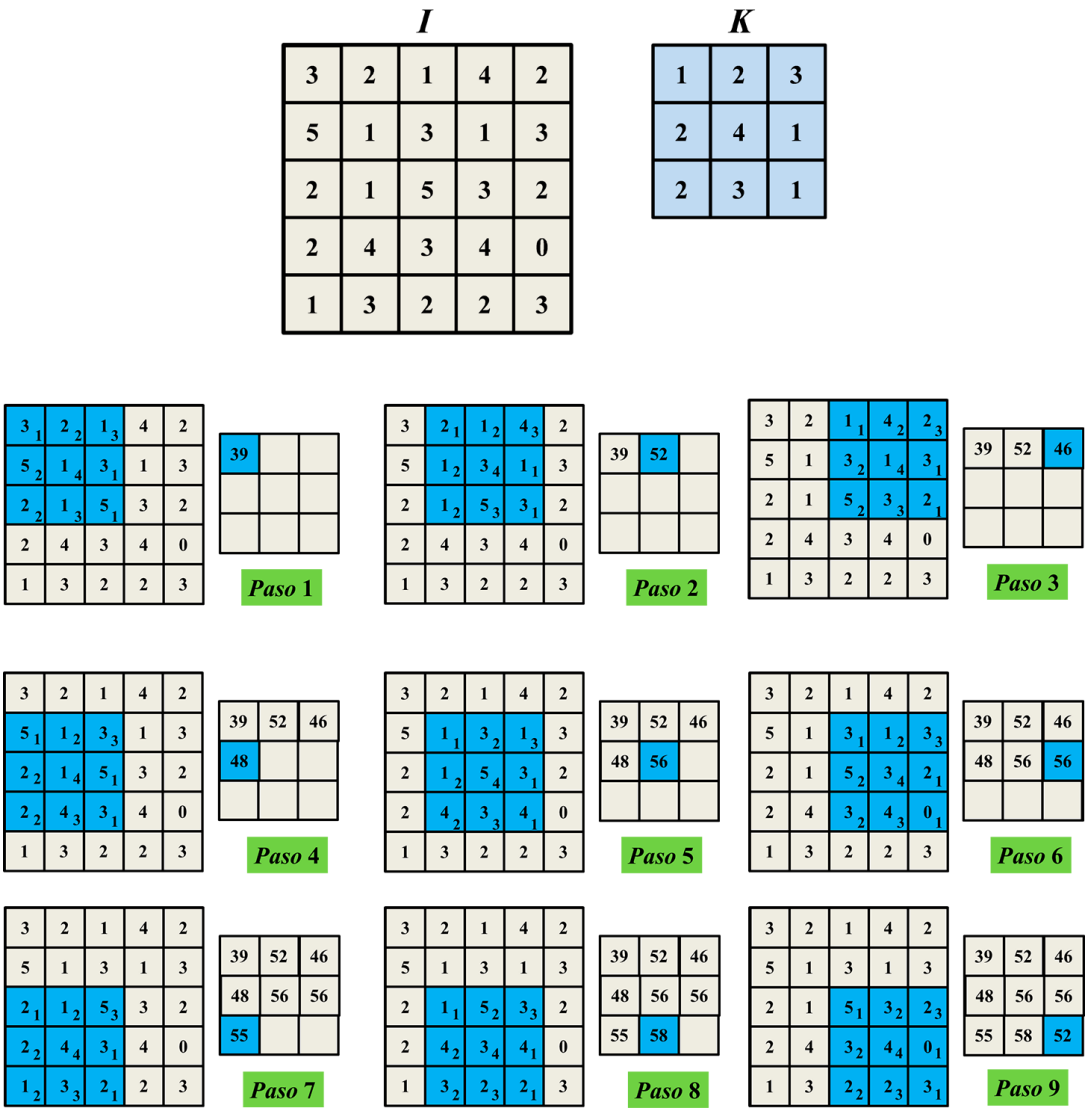


Figura 2.11: Ejemplo de convolución bi-dimensional discreta con: $N = 2, i_1 = i_2 = 5, k_1 = k_2 = 3, s_1 = s_2 = 1, p_1 = p_2 = 0$

1. **Max Pooling**: Consiste en dividir la entrada en ventanas produciendo como salida el máximo de la ventana.

2. **Average Pooling**: Procedimiento equivalente al de *max pooling* pero produciendo como salida la media de la ventana.

Generalmente, esta operación se aplica para reducir la dimensionalidad, de forma que en cada paso, la posición de la ventana se actualiza de acuerdo a los desplazamientos (*strides*).

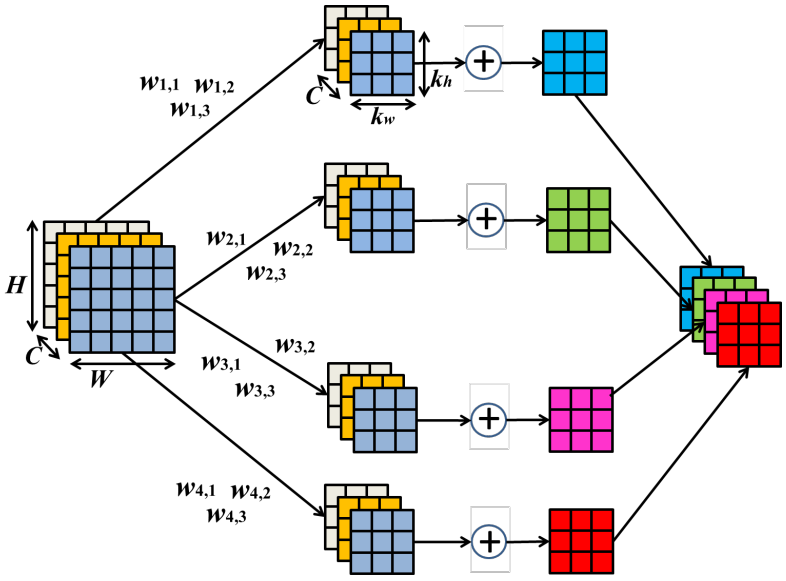
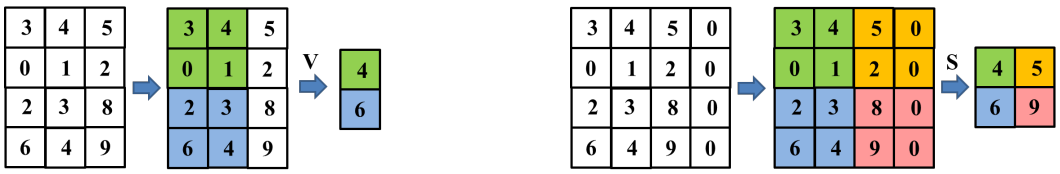


Figura 2.12: Forma alternativa de interpretación de los desplazamientos del núcleo

Cuando la ventana se encuentra situada en los bordes, algunos de los elementos de la ventana pueden quedar fuera de los elementos de entrada. Por tanto para obtener los valores de las regiones de borde, la entrada puede extenderse con valores de 0, tal y cómo se podía hacer en la convolución. Tal y como se ha explicado previamente, a esta operación se le denomina zero-padding. No obstante no es obligatorio realizar esta operación. En la figura 2.15 se pueden ver dos ejemplos con *stride* = 2, sin 2.13a y con zero-padding 2.13b.



(a) Max pooling sin zero-padding

(b) Max pooling con zero-padding

Figura 2.13: Max pooling con s = 2

2.4.3. Capas de Dropout

La figura 2.14 muestra los tres tipos de ajustes que pueden surgir durante el entrenamiento de una red neuronal.

De estos tres casos, el de la izquierda *underfitting* consiste en que por una falta de datos de entrenamiento, el sistema no es capaz de generalizar correctamente (es demasiado simple). El de la derecha *overfitting* es un polinomio demasiado complejo que se ajusta demasiado a los datos de entrenamiento, y por tanto, no es capaz de generalizar con la entrada de nuevos datos. El del centro, *correcto* es el resultado que se busca, el sistema es suficientemente complejo como para ajustarse correctamente a los datos de entrenamiento y a la vez suficientemente robusto como para ser capaz de generalizar y por tanto, ajustarse correctamente también a los datos de testeo.

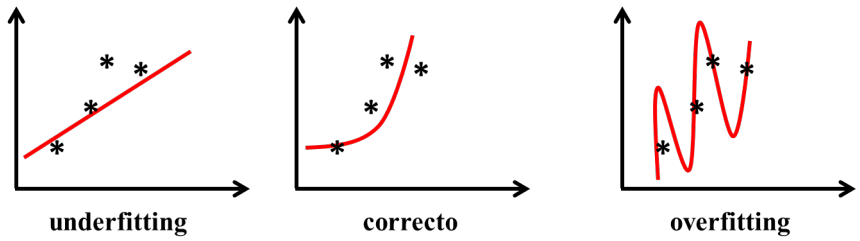


Figura 2.14: Forma alternativa de interpretación de los desplazamientos del núcleo

Un método para evitar el *overfitting* es aplicar *regularización*. Esto consiste en modificar la función objetivo a minimizar añadiendo términos adicionales que penalizan pesos con valores elevados.

Así por ejemplo, dada la función J se le puede añadir el término $\lambda f(\theta)$ de forma que $f(\theta)$ crece a medida que el argumento θ también crece y λ es el coeficiente de regularización.

El valor de este parámetro determina la "protección" contra el *overfitting*, de forma que si es cero, no afecta de ninguna manera, por el contrario si es demasiado grande el modelo dará prioridad a mantener θ lo más pequeño posible, al tratar de encontrar los valores de los parámetros que mejor se ajustan al conjunto de datos de entrenamiento.

El tipo de regularización más común es la regularización L_2. Se puede implementar aumentando la función de error con la magnitud cuadrada de todos los pesos en la red neuronal. Es decir, para cada peso w en la red neuronal se agrega $1/2\lambda w^2$ a la función de error.

De forma intutitiva la regularización L_2 penaliza fuertemente los pesos con valores altos, de esta forma la red neuronal tenderá a utilizar todas sus entradas en alguna medida en lugar de algunas de ellas en gran medida. Es debido a esta tendencia que la regularización L_2 también se conoce comúnmente como disminución de peso.

No obstante, en el ámbito de las redes neuronales, para paliar el efecto del overfitting se propone anular (*dropout*) determinado tipo de neuronas, incluidas sus salidas, para evitar el problema del *overfitting* en el proceso de entrenamiento [16].

La selección de las unidades a anular se realiza de forma aleatoria. Así la red resultante es la que mantiene las unidades que sobreviven al dropout. No obstante, en lugar de eliminar neuronas, también cabe la posibilidad de mantenerlas asignándoles un hiperparámetro de forma aleatoria que es en realidad una probabilidad p de supervivencia durante la fase de entrenamiento. Esto se hace con el fin de disminuir la influencia del peso en la fase de testeo.

En la figura 2.15a se muestran nodos anulados (en rojo) y con el peso de activación $w_{(5,6)}$ atenuado por la probabilidad p durante la fase de testeo.

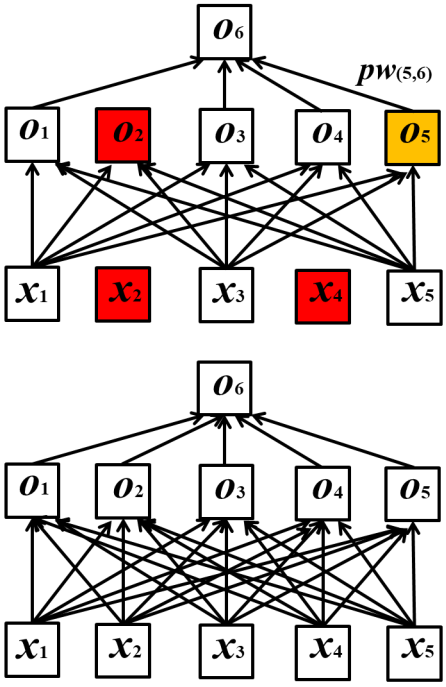
Además, el dropout puede aplicarse tras la salida de una determinada capa como por ejemplo, tras un *pooling* como ocurre en el ejemplo gráfico mostrado en la figura 2.15b.

2.4.4. Normalización

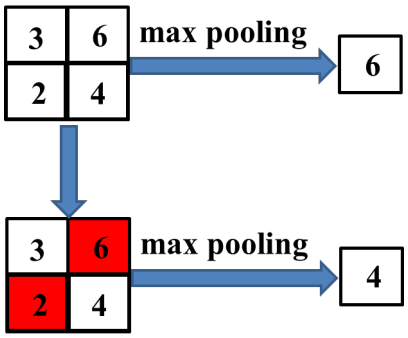
Como se ha descrito en la subsección 2.3.3, la función de activación ReLU tiene la propiedad de que no requiere normalización para prevenir la saturación[24]. Siempre que algunos ejemplos del entrenamiento produzcan una entrada positiva a una unidad ReLu dicha neurona continuará aprendiendo.

No obstante, se puede aplicar una normalización local que ayuda a la generalización. Si se denota por $a^i{}_{x,y}$ la actividad de una neurona obtenida por aplicación del núcleo i en la posición (x,y) y se aplica la nolinealidad ReLU, la actividad de la respuesta normalizada

[16] Dicho concepto proviene del artículo: *Dropout: a simple way to prevent neural networks from overfitting*[23].



(a) Ejemplo de dropout



(b) Dropout y max pooling

Figura 2.15: Max pooling tras dropout

$b^i{}_{x,y}$ cumple la siguiente expresión

$$b^i_{x,y} = \frac{a^i_{x,y}}{(k + \alpha \sum_{j=\text{máx}(0,i-n/2)}^{\text{mín}(N-1,i+n/2)} (a^i_{x,y})^2)^\beta} \tag{2.26}$$

donde la suma se extiende sobre n mapas adyacentes generados por los núcleos en la misma posición (x,y) y N es el número total de núcleos en dicha capa. A su vez, los mapas se ordenan de manera arbitraria antes de que comience el aprendizaje.

Este tipo de respuesta normalizada inmplementa una forma de inhibición lateral similiar al encontrado en neuronas reales, creando así, una competición para actividades elevadas frente a salidas de neuronas obtenidas usando diferentes núcleos.

Las constantes k, n, α y β son hiperparámetros cuyos valores se determinan usando un conjunto de validación. En el trabajo de Krizhevsky col.[24] se proponen los siguientes valores $k = 2$, $n = 5$ $\alpha = 10^{-4}$ y $\beta = 0.75$. En este mismo artículo, se indica que esta normalización se realiza en ciertas capas tras la aplicación de ReLU.

2.4.5. Función Softmax

La función *softmax* o función exponencial normalizada se define mediante la equación 2.27 y se emplea generalmente en las últimas capas ocultas de las redes neuronales para proyectar un vector n-dimensional, x de valores reales en un vector n-dimensional *softmax(x)* de valores reales en el rango [0,1].

$$softmax(x)_i = \frac{exp(x_i)}{\sum_{j-1}^{n} exp(x_i)} \tag{2.27}$$

Suponiendo una hipotética red neuronal con cuatro neuronas (n = 4) en la capa de salida, obteniendo un modelo con los siguientes valores x = (1.2, 2.0, 5.6, 3.1) los valores generados por la función son: $softmax(x) = (0{,}011, 0{,}024, 0{,}892, 0{,}073)$. La suma de todos estos elementos es la unidad.

De esta forma, aplicando la función exponencial a cada elemento x_j del vector de entrada x y normalizando esos valores por la suma de las exponenciales se obtiene un vector de salida cuya suma es exactamente 1 (y por tanto normalizado).

Es posible todavía modificar los exponentes de las exponenciales con el fin de crear funciones de distribución de probabilidad más concetradas alrededor de posiciones de los valores de entrada mayores, lo cual se consigue con $\alpha x_i > 0$ o $-\alpha x_i > 0$ tal y como se puede ver en las equaciones 2.28 y 2.29.

$$softmax(x)_i = \frac{exp(\alpha x_i)}{\sum_{j-1}^{n} exp(\alpha x_i)} \tag{2.28}$$

$$softmax(x)_i = \frac{exp(-\alpha x_i)}{\sum_{j-1}^{n} exp(-\alpha x_i)} \tag{2.29}$$

2.4.6. Red neuronal AlexNet

Alexnet[25] es una CNN diseñada por Alex Krizhevsky que compitió en el reto *Large Scale Visual Recognition Challenge (ILSVRC)*[26] en 2012. Se pudo comprobar que la red tenía un

gran porcentaje de acierto y un alto rendimiento[17].

La figura 2.16 muestra la estructura de la red AlexNet así como las operaciones involucradas, identificando las mismas con indicación de las dimensiones de filtros en las capas convolucionales y totalmente conectadas, para las operaciones de *Convolución* (*conv*), *ReLU* (*relu*), *Normalización* (*norm*), *Pooling* (*pool*), *dropout*(*drop*) o *softmax*.

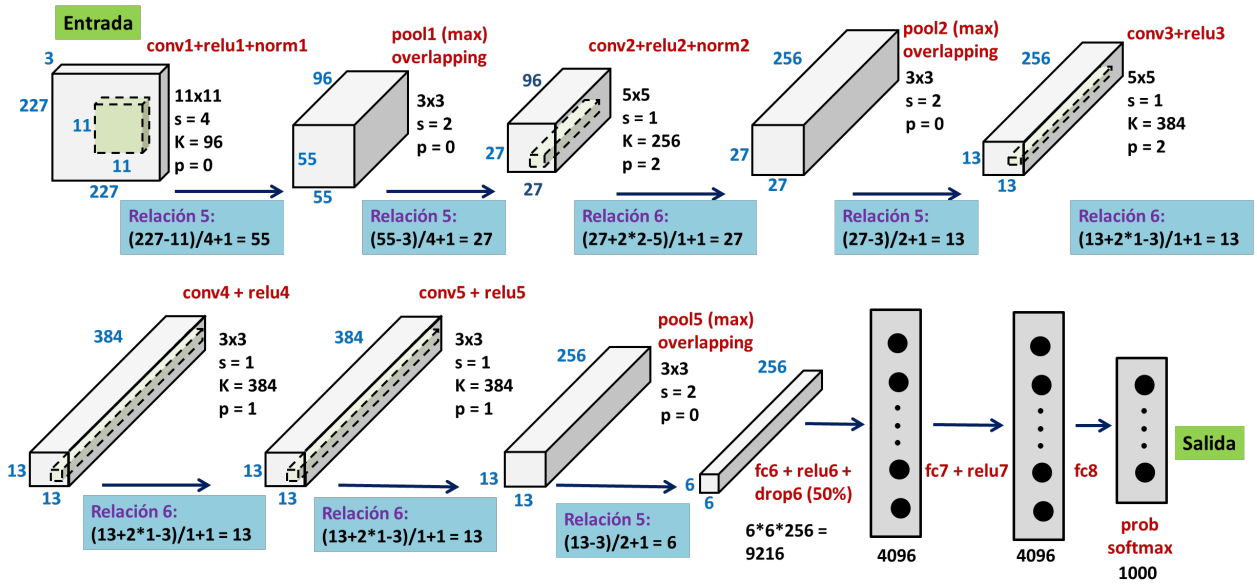


Figura 2.16: Estructura de capas de AlexNet

Los pesos que se aprenden son exactamente los de las cinco capas de convolución y los de las tres capas totalmente conectadas, independientemente de que en éstas últimas se aplique o no *dropout*.

Obsérvese que como pesos a aprender se incluyen en cada capa de convolución un valor de *bias* por filtro y que es un parámetro de ajuste adicional, por ejemplo, similar al término independiente b en líneas rectas (y = ax + b).

Así mismo, se indican los números de los parámetros de entrada i, k, p, y s.

Además, en algunas implementaciones, tras la función *relu7* se realiza una operación de *dropout* del 50 % modificando las dimensiones de la salida de dicha capa.

[17]Es importante destacar, que en el momento, era bastante costoso desde el punto de vista computacional, pero se hizo factible con la utilización de unidades de procesamiento gráfico (GPU) durante el entrenamiento.

2.4.7. Red neuronal ResNet (Deep Residual Learning)

Residual Neural Network o ResNet, es un tipo de CNN que surge a partir del análisis empírico en relación al número de capas de los diferentes modelos, ya que cuanto más profundas sean las redes neuronales mayor dificultad para el entrenamiento.

ResNet surge a partir de la idea de conseguir una mayor facilidad para realizar el entrenamiento, es decir, mayor facilidad en el proceso de optimización, donde surge la propuesta formulada bajo el paradigma *Deep Residual Learning*[27]. Según este trabajo, en ocasiones, cuando la profundidad de la red aumenta, la precisión se satura, llegando a degradarse rápidamente[18]. La idea de ResNet consiste en la introducción de lo que se conoce como cortocircuito, o en términos originales *identity shortcut connection* consistente en saltar una o más capas de la red original (ver figura 2.17).

El objetivo es que en lugar de esperar a que las capas se acoplen directamente hacia una proyección subyacente deseada, se deja explícitamente que estas capas se ajusten a una proyección residual. Formalmente, denotando la proyección subyacente deseada como *H(x)* se permite que las capas no lineales acopladas se ajusten a otra proyección *F(x) = H(x) -x*; de esta forma, la proyección original se pantea como *F(x) + x* bajo la hipótesis de que si una proyección de identidad es óptima, resultaría más fácil llevar el residuo *F(x)* a cero, que ajustar una proyección de identidad mediante un bloque de capas no lineales.

Dicho de otra forma, es más fácil encontrar una solución tal que *F(x) = 0* en lugar de *F(x) = x* usando una pila de capas no lineales como función.

Así mismo, la función *F(x)* se le conoce como residuo. De esta forma las capas apiladas (bloque) no deberían degradar el rendimiento de la red, porque simplemente pueden apilar asignaciones de identidad[19].

Es importante destacar que la red posee tanto las capas apiladas no lineales que producen

[18] No necesariamente causada por *overfitting*, a veces, añadir más capas a la red conduce a mayor error en el entrenamiento.

[19] Una capa que no hace nada en la red actual, por tanto, la arquitectura resultante funcionaría de la misma manera.

la salida F(x) como el cortocircuito que produce la identidad x.

La formulación de $F(x) + x$ puede implemetnarse mediante redes neuronales con propagación hacia adelanta (*feedforward*) con conexiones cortocircuitadas que saltan una o más capas.

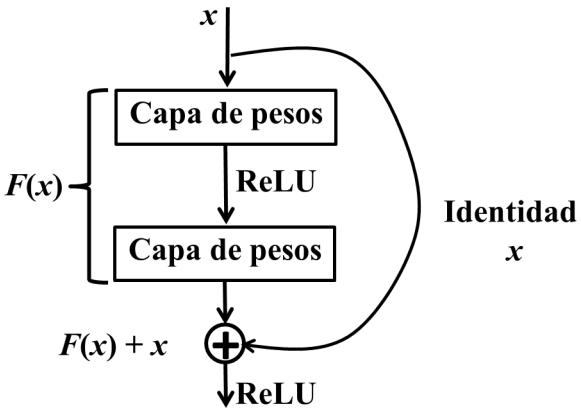


Figura 2.17: Bloque ResNet

En la figura 2.17 las conexiones cortocircuitadas realizan simplemente una proyección de identidad, de forma que sus salidas se añaden a la salida de las capas apiladas. En definitiva, a la hora de relizar la operación de identidad $H(x) = F(x) + x$, el objetivo es que la red aprenda cualquier función $F(x)$ de forma que ésta sea cero, $F(x) = 0$.

Además, conviene señalar que las conexiones de identidad cortocircuitadas no añaden parámetros extra ni complejidad computacional. La red neuronal completa puede seguir entrenándose de principio a fin con el Descenso de Gradiente Estocástico, 2.3.2, mediante retropropagación para conseguir lo indicado previamente.

Así mismo, considérese $H(x)$ como una proyección subyacente para ser ajustada por unas pocas capas apiladas, y por tanto no necesariamente la red completa, siendo x las entradas a la primera de estas capas.

Para las capas apiladas se aplica el aprendizaje residual, obteniendo un bloque completo como el mostrado en la figura 2.17, que se define como sigue:

$$y = F(x, \{W_i\}) + x \tag{2.30}$$

Siendo x e y los vectores de entrada y salida de las capas consideradas. Como se ha indicado anteriormente la función $F(x, \{W_i\})$ representa la proyección residual que debe aprenderse.

Por ejemplo, siguiendo con el trabajo del artículo[27] y considerando la figura 2.17 que tiene dos capas, $F \equiv W_2\theta(W_1x)$, donde θ indica la capa ReLU, omitiendo los pesos *bias* por simplicidad. La operación *F(x) + x* se realiza por una adición elemento a elemento. En el ejemplo mencionado, la segunda operación no lineal, ReLU se lleva a cabo después de la suma.

Para poder realizar la adición, es importante que las dimensiones de F y x sean iguales. Si esto no se cumple, es necesario realizar una proyección lineal W_S para las conexiones cortocircuitadas para establecer la correspondencia deseada, quedando la siguiente expresión:

$$y = F(xm\{W_i\}) + W_Sx \tag{2.31}$$

En cualquier caso, en la ecuación 2.30 se pondría igualmente una matriz W_S, aunque según el artículo[27], se considera inefectivo en función de los resultados obtenidos empíricamente, siendo suficiente la identidad.

Además, aunque en las notaciones anteriores se refieren a capas totalmente conectadas por simplicidad, esto es aplicable a capas convolucionales. La función $F(x, \{W_i\})$ puede representar múltiples capas convolucionales. La operación de adición elemento a elemento se realiza sobre los dos mapas de características canal por canal.

Con carácter ilustrativo en la figura 2.18 se muestra un ejemplo de un tramo de red con distintos bloques y con dos cortocircuitos, A y B. La correspondiente función residual comprende 2 y 3 capas respectivamente. Se consideran cinco volúmenes ordenados del 1 al 5 con las dimensiones y operaciones indicadas. Para cada volumen se identifican las correspondientes dimensiones, siendo respectivamente las siguientes: $55\times55\times96$, $27\times27\times96$, $27 \times 27 \times 256$, $13 \times 13 \times 256$ y $13 \times 13 \times 384$. Así mismo, al lado de cada volumen se indican los parámetros utilizados en las operaciones definidas, éstos son, núcleos de convolución d, *stride* (s), *padding* (p) y número de núcleos (K) generando los volúmenes con las dimensiones

correspondientes mediante las operaciones indicadas en dichas relaciones.

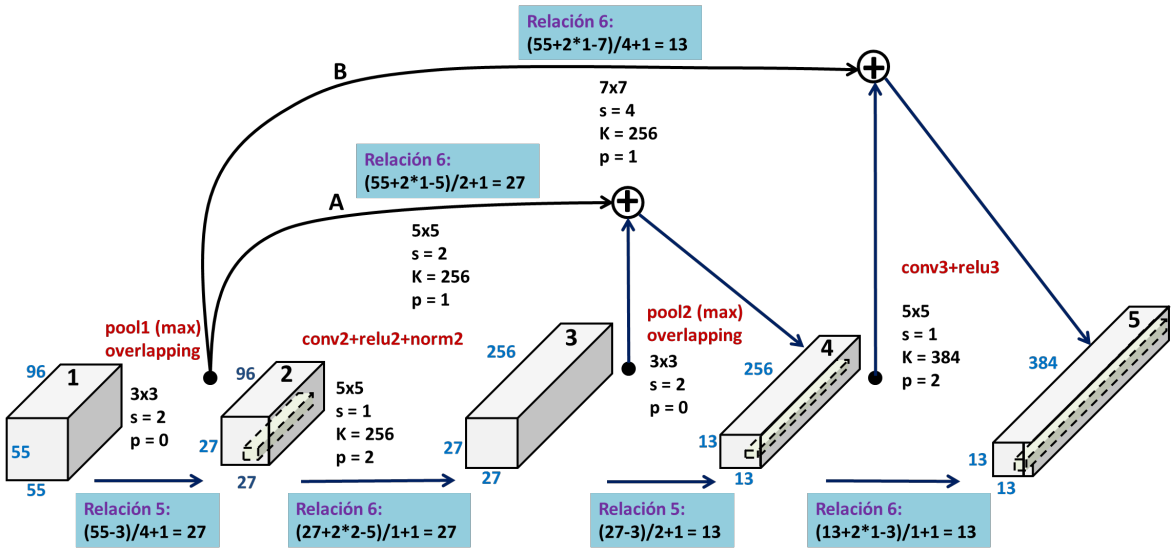


Figura 2.18: Ejemplo ilustrativo ResNet

Por otro lado, en el cortocircuito A se aplica la relación 6 sobre el volumen 1 con los parámetros indicados para obtener a la salida (previo a la operación suma) un volumen de dimensión $27 \times 27 \times 256$, cuyas dimensiones se corresponden exactamente con las del volumen 3, pudiéndose en este caso realizar la operación de suma elemento a elemento entre este volumen y el volumen 3 en cada uno de los 256 canales de sendos volúmenes, obteniendo como resultado, un bloque de las mismas dimensiones que conforman la suma. En el caso del cortocircuito B, se aplica también la relación 6, con dimensiones de salida $13 \times 13 \times 256$, que a su vez también coincide con las salidas del volumen 4, por tanto, se puede realizar la operación elemento a elemento.

Siguiendo este planteamiento, quedan diseñados dos tramos de red ResNet, tal y como se puede ver en la figura 2.19. Es importante destacar, que no conviene utilizar los dos cortocircuitos simultáneamente en la misma arquitectura, se opta por uno o por otro. En ambos casos, durante el entrenamiento se ajustan los pesos que conforman las uniones con el objetivo de conseguir, a través de dichos ajustes que la función residual F en los cortocircuitos sea cero.

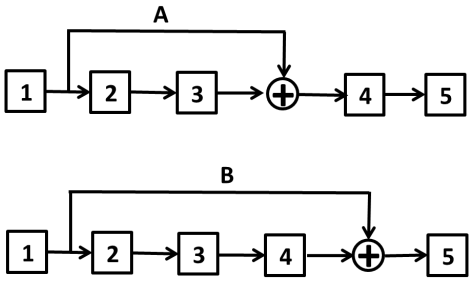


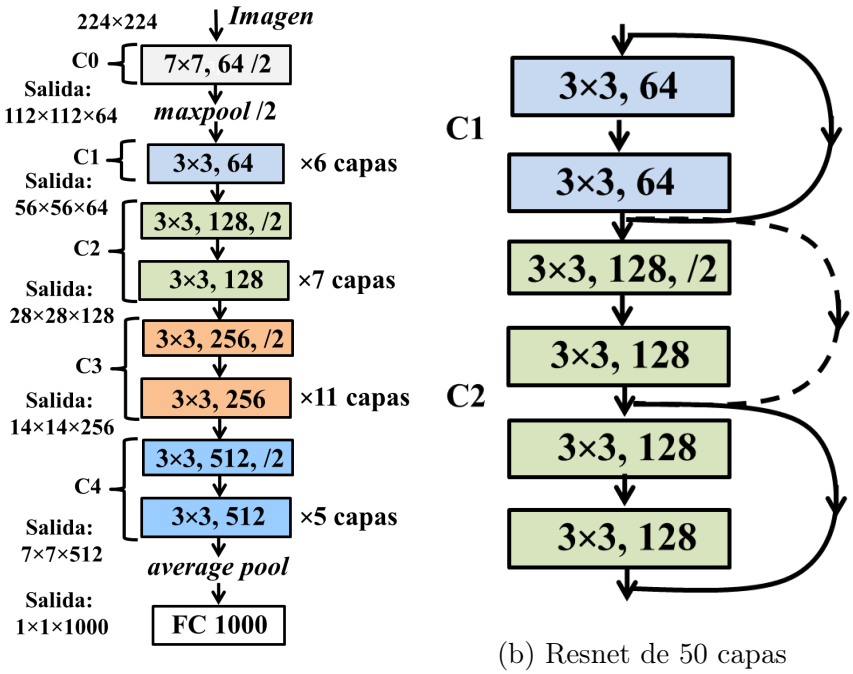
Figura 2.19: Tramos de ResNet

Bajo estos conceptos se han realizado diferentes variantes de ResNet. Entre ellas, ResNet-50, inspirada en VGG-19[28], ésta es una red base de 34 capas tal y como aparece en la figura 2.20a. Obsérvese que consta de cinco módulos *C0* (1 capa), *C1* (6 capas), *C2* (8 capas), *C3* (12 capas) y *C4* (6 capas), todo ello, hace un total de 33 capas, más la capa *FC* (*fully connected*).

En estas representaciones, la nomenclatura viene dada por *(M, $k \times k$, K)* donde *M* expresa la dimensión del mapa de características de la capa de salida; *maxpool* se refiere a la operación de *pooling* con la operación máximo, el símbolo */2* indica la operación desplazamiento o *stride* ($s = 2$), *FC 1000* (*fully connected*) con clasificación para 1000 categorías de clasificación y *average pool* (*pooling* promediado).

A partir de esta red, se crea ResNet-50, que como su nombre indica consta de 50 capas, otenidas reemplazando cada bloque de dos capas en la red de 34 con el bloque de 3 capas con una arquitectura similar a la del bloque mostrado en la figura 2.21, si bien variando los tamaños de los núcleos de convolución y el número de filtros.

Suponiendo una imagen de entrada de dimensión 224×224, mediante la aplicación de convolución con *stride* 2, se obtiene un mapa de características de salida de dimensión $112 \times 112 \times 64$, de forma que a su vez tras la operación *maxpool* con $s = 2$, generando un mapa de características de $28 \times 28 \times 128$. En los bloques C3 y C4 se generan las salidas de mapas $14 \times 14 \times 256$ y $7 \times 7 \times 512$ respectivamente, tras las convoluciones indicadas y considerando los dos desplazamientos, de nuevo con $s = 2$. Se llega así, a la capa FC con



(a) Modelo de 34 capas

(b) Resnet de 50 capas

Figura 2.20: Comparación de los modelos

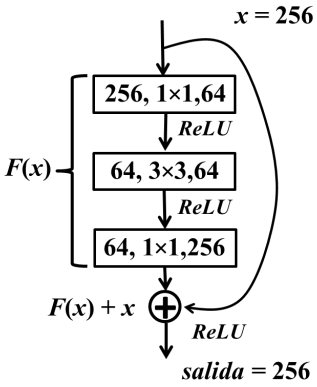


Figura 2.21: Bloque ResNet

1000 unidades de salida.

En la figura 2.20b se muestra un esquema de implementación de ResNet con indicación de las capas conectadas, mostrando las transiciones entre bloques con flechas discontinuas. Por tanto, disponiendo de 4 bloques, con un total de 32 capas, si se agrupan éstas de dos en dos se obtienen 16 parejas de capas, sustituyendo éstas por bloques ResNet con tres capas,

se obtendría un total de 48 capas, que sumadas a las otras dos capas restantes, se obtiene un total de 50 capas, que es el número indicado previamente.

Análogamente a ResNet-50, existen ResNet-18, ResNet-101 y ResNet-152[20]. Observese la tabla 2.1 para ver en detalle como se obtiene el número de capas indicado.

Nombre capa	**Dimensión de salida**	**34 capas (plana)**	**50 capas (ResNet)**	**101 capas (ResNet)**	**152 capas (ResNet)**
C0	112×112	7×7, 64, $s = 2$			
C1	56×56	3×3, *maxpool*, $s = 2$			
		$\begin{bmatrix} 3\times3 & 64 \\ 3\times3 & 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 64 \\ 3\times3 & 64 \\ 1\times1 & 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 64 \\ 3\times3 & 64 \\ 1\times1 & 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 64 \\ 3\times3 & 64 \\ 1\times1 & 256 \end{bmatrix} \times 3$
C2	28×28	$\begin{bmatrix} 3\times3 & 128 \\ 3\times3 & 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1 & 128 \\ 3\times3 & 128 \\ 1\times1 & 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1 & 128 \\ 3\times3 & 128 \\ 1\times1 & 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1 & 128 \\ 3\times3 & 128 \\ 1\times1 & 512 \end{bmatrix} \times 8$
C3	14×14	$\begin{bmatrix} 3\times3 & 256 \\ 1\times1 & 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 & 256 \\ 3\times3 & 256 \\ 1\times1 & 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1 & 256 \\ 3\times3 & 256 \\ 1\times1 & 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1 & 256 \\ 3\times3 & 256 \\ 1\times1 & 1024 \end{bmatrix} \times 36$
C4	7×7	$\begin{bmatrix} 3\times3 & 512 \\ 3\times3 & 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 512 \\ 3\times3 & 512 \\ 1\times1 & 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 512 \\ 3\times3 & 512 \\ 1\times1 & 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1 & 512 \\ 3\times3 & 512 \\ 1\times1 & 2048 \end{bmatrix} \times 3$
	1×1	*average pool*, FC 1000, *softmax*			

Tabla 2.1: Modelos de redes de diferentes capas y dimensiones del mapa de características de salida

Como se puede ver, las capas convolucionales tienen en su mayoría filtros de dimensión 1×1 y 3×3 con un diseño relativamente simple. De esta forma, para el mismo tamaño de mapa de características de salida, las capas tienen el mismo número de filtros; y cuando el tamaño del mapa de características se reduce a la mitad, el número de filtros se duplica.

2.4.8. YOLO: You Only Look Once

YOLO pertenece a la categoría de métodos en los que la imagen se procesa una vez, en lugar de procesar regiones por separado, siendo una de sus principales ventajas la velocidad de proceso. Hasta el momento se han desarrollado tres versiones:

1. Redmon y col., 2016[5].

[20]Éstas no son todas las variantes, pero se hace hincapié en éstas puesto que son las que se han utilizado en este proyecto.

2. Redmon y Farhadi, 2017[29], cuyo objetivo es incrementar la velocidad de procesamiento de la versión previa y a la vez, hacerla más robusta.

3. Redmon y Farhadi, 2018[6], mejorando nuevamente la versión de 2017.

A continuación se explica en detalle los fundamentos del funcionamiento de YOLO poniendo el enfásis en su primera versión.

El esquema de esta versión es relativamente simple. Una única CNN predice múltiples *bounding boxes*, junto con sus correspondientes probabilidades de clase. El entrenamiento se realiza sobre las imágenes completas, no sobre subimágenes de las mismas, procediendo a optimizar el proceso de detección. Bajo este planteamiento se consigue una alta velocidad de procesamiento con buenos resultados de precisión, a la vez se obtienen representaciones generalizables de los objetos.

El procedimiento consiste en dividir la imagen en una rejilla conteniendo $S \times S$ celdas. Si el centro de un objeto cae sobre una de tales celdas, dicha celda es la responsable de la detección de ese objeto. Cada celda de la rejilla, predice B *bounding boxes* y valores de confianza para los mismos. Dichos valores de confianza expresan la seguridad del modelo de que el *box* contenga un objeto y también la seguridad respecto de la precisión.

Formalmente se define la confianza como $P(objeto)IoU^{verdadero}_{predicho}$. Si no existe el objeto en esa celda, el valor de confianza es cero y en caso de que exista, lo que se desea es que el valor de confianza sea igual a IoU (*Intersection over Union*) entre el *box* (rectángulo) predicho y el *ground truth*.

Cada celda de rejilla, también predice C probabilidades condicionadas de clase conteniendo un objeto, $P(C_i|objeto)$. Sólo se predice un conjunto de probabilidades de clase (C_i) por celda de rejilla, independientemente del número de *bounding boxes*, B. En el momento del test se multiplica la probabilidad condicionada de clase y las predicciones de confianza de la *box* individual, que proporciona los valores de confianza específicos de clase para cada

rectángulo. De este modo se obtiene la siguiente fórmula:

$$Valor\ confianza\ de\ clase = P(C_i|objeto)P(objeto)IoU^{verdadero}_{predicho} = P(C_i)IoU^{verdadero}_{predicho} \tag{2.32}$$

Siendo:

- $P(objeto)$ es probabilidad de que el *box* (rectángulo) contenga un objeto.
- $IoU^{verdadero}_{predicho}$ es la IoU entre el rectángulo predicho y el *ground truth*.
- $P(C_i|objeto)$ es la probabilidad de que el objeto pertenezca a la clase C_i, dado que el objeto esté presente.
- $P(C_i)$ es la probabilidad de que el objeto pertenezca a la clase C_i.

Esos valores de confianza determinan tanto la probabilidad de que la clase aparezca en el rectángulo y cómo de bien el rectángulo predicho se ajusta al objeto.

En definitiva, el modelo de detección se plantea como un problema de regresión, de forma que la imagen se divide en una rejilla de dimensión $S \times S$ celdas y para cada celda de la rejilla se predicen B *bounding boxes*, así como los valores de confianza para esos *boxes* y C probabilidades condicionadas de clase para un objeto dado. Esas predicciones se codifican como un tensor de dimensión $S \times S \times (5B + C)$.

Así mismo, cada celda de la rejilla predice un único objeto, por ejemplo, en la figura 2.22, la rejilla marcada en amarillo trata de predecir el objeto peluche a través del *bounding box* cuyo centro cae dentro de la celda de la rejilla (círculo indicado). Como se ha explicado previamente, cada celda define un número de *bounding boxes* prefijado (B) para localizar de la mejor manera posible el objeto. Cada *box* tiene asociado un grado de confianza. Si bien, a pesar de la existencia de los B *bounding boxes* una celda sólo detecta un objeto.

En lo que respecta al diseño de la red neuronal, en el trabajo de Redmon y col.[5] se propone un modelo de arquitectura inspirado en GoogleNet[30].

Dicho modelo consta de 24 capas convolucionales seguidas por 2 totalmente conectadas. Además, se sustituyen los módulos *inception* utilizados en GoogleNet por capas de reducción

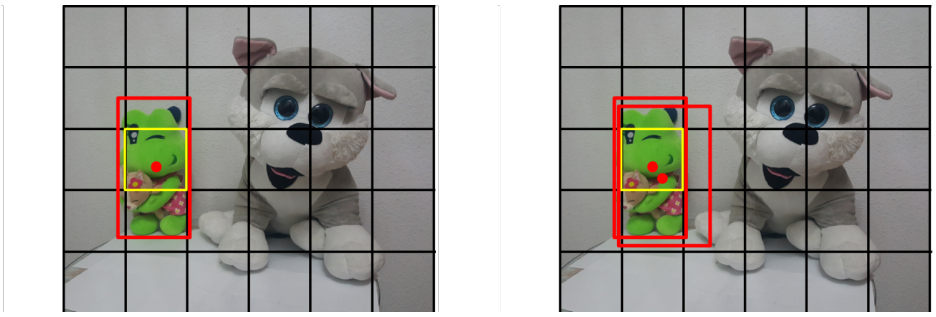


Figura 2.22: Rejilla de celdas y bounding boxes definidos por una celda

1×1 seguidos por capas de convolución de dimensión 3×3. De esta forma, alternando capas de convolución 1×1 se reducen las dimensiones del espacio de características a partir de las capas precedentes. Se puede ver la red completa en la figura 2.23, produciendo un tensor de salida de dimensión $3 \times 3 \times 30$.

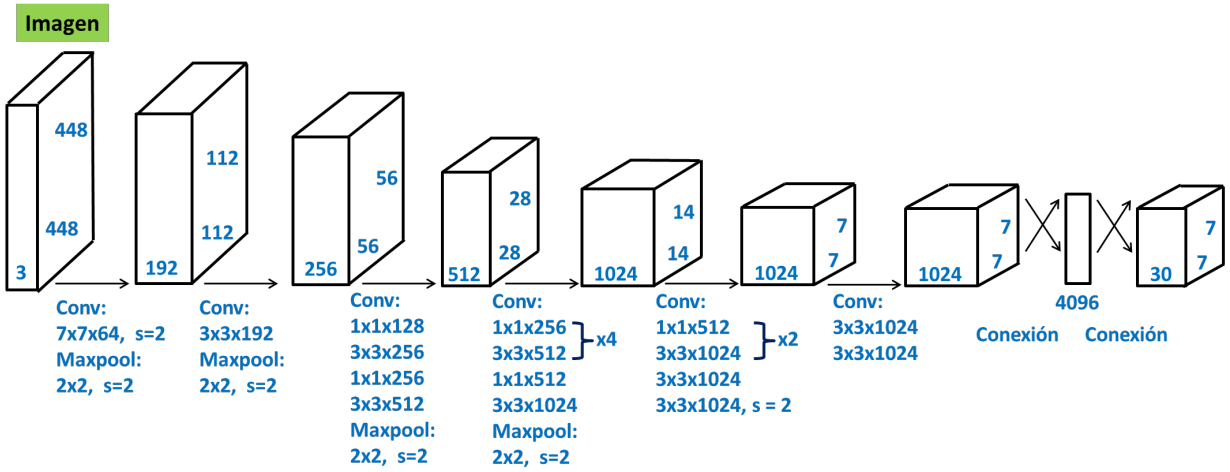


Figura 2.23: Red neuronal en YOLO

Capítulo 3

Diseño y desarrollo de la aplicación

En este capítulo se describe en detalle el proceso seguido para completar el diseño y desarrollo de la aplicación.

El diseño consta de tres módulos básicos, a saber:

1. Captura de vídeo, con los vehículos en movimiento.
2. Flujo óptico, para detección del movimiento mediante los métodos con aplicación de los algoritmos de Lucas-Kanade y Farneback.
3. Redes neuronales convolucionales (AlexNet y ResNet) para clasificación de los objetos en movimiento.

Tras la obtención de las secuencias de imágenes procedentes del vídeo, que en este caso simula la captura de imágenes en el tiempo, tal y como se realizaría en un sistema real, se aplican los métodos de detección de movimiento indicados, basados en la técnica del flujo óptico. La identificación de áreas o zonas en la imagen con fuertes componentes de movimiento, expresadas a través de la magnitud de los vectores de flujo, permiten determinar las regiones donde se ha producido un movimiento significativo. Estas regiones determinan lo que se ha denominado regiones de interés, tal y como se describen en la sección 2.2.3, y constituyen las entradas a las redes neuronales correspondientes para que éstas determinen el tipo de vehículo de que se trata, clasificándolo como preteneciente a una de las categorías especificadas.

De este modo, en la figura 3.1 se puede ver un esquema simplificado del flujo completo de la aplicación.

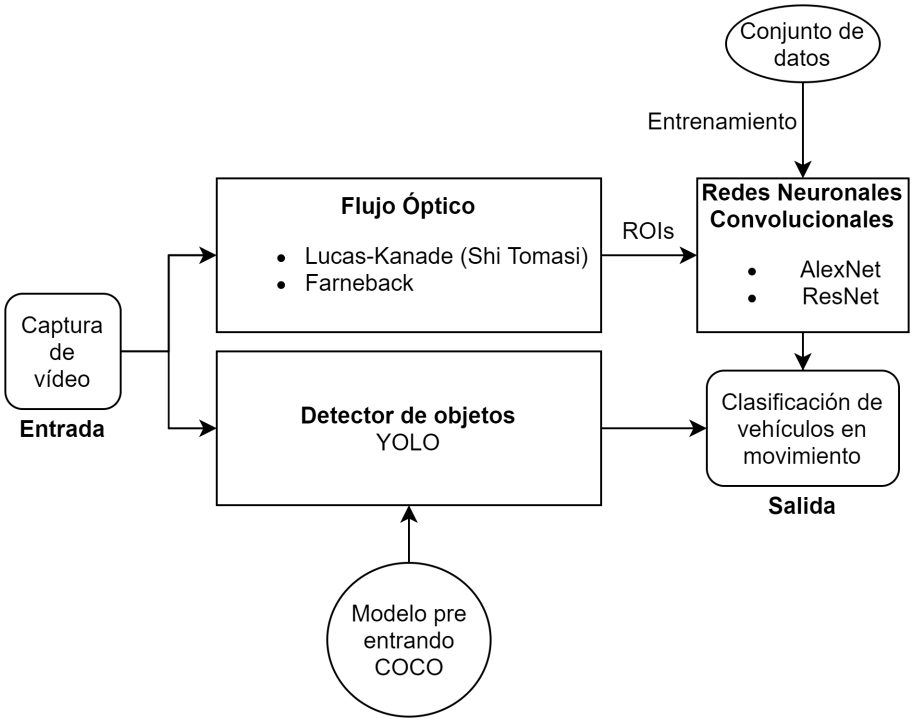


Figura 3.1: Diagrama de flujo del proyecto.

Para llevar a cabo el diseño propuesto es necesario identificar y determinar las tecnologías necesarias para ello. A continuación se explican las que constituyen el núcleo base para el desarrollo.

Se ha hecho uso del sistema operativo *GNU Linux* en su distribución *Ubuntu 18.04* debido a las facilidades en la instalación de drivers y librerías necesarias para el proyecto.

Así mismo se ha utilizado *Python* como lenguaje de programación principal para el desarrollo del sistema, así como *Bash* para el desarrollo de los scripts auxiliares de instalación y automatización de tareas sencillas.

Se ha utilizado *OpenCV*[1][32] para realizar el procesamiento de las imágenes y la ejecución de los algoritmos de flujo óptico (ver sección 2.2).

[1]Tanto los paquetes de *C++*, como la versión no oficial de *Python*[31].

Además, basada en la librería OpenCV, por simplicidad, se ha hecho uso de la librería *cv2-tools*[33] desarrollada en *Python*.

También se ha utilizado el ecosistema *TensorFlow*[34] para realizar el diseño, entrenamiento, visualización y uso de los modelos de red neuronal de tipo AlexNet y ResNet (ver subsecciones 2.4.6 y 2.4.7 respectivamente).

Por último, se ha hecho uso de *Git*[35] como sistema de control de versiones.

3.1. Fases de desarrollo

En el ámbito práctico y siguiendo el esquema de diseño previo, este proyecto se ha desarrollado siguiendo las fases enumeradas a continuación y explicadas en profundidad en su correspondiente subsección. Cabe destacar, que en todas estas fases se ha generado al menos un fichero *Python* con su respectivo sistema de ejecución aislada[2].

1. Desarrollo de un sistema básico de conexión a sucesiones de imágenes (tanto vídeos como streamings).

2. Implementación de algoritmos de flujo óptico.

   - Lucas Kanade[3] 2.2.1 haciendo uso del algoritmo de Shi-Tomashi 2.1.
   - Gunnar Farneback 2.2.2.

3. Generación de las regiones de interés partiendo del algoritmo de Gunnar Farneback.

4. Implementación y uso de *YOLOv3* como detector de objetos. Es importante destacar que *YOLO* no es compatible con los algoritmos de flujo óptio previamente mencionados, puesto que, como se ha explicado en la sección 2.4.8, YOLO utiliza la imagen completa como entrada a la red neuronal. Aún así, con fines académicos, se ha implentado en su totalidad y ha sido de gran utilidad para realizar comparaciones de

[2] Así mismo, se puede usar el argumento *-h* en todos los ficheros para ver una pequeña explicación de las opciones que ofrece.

[3] Con opción a cambiar el tamaño de ventana dinámicamente.

rendimiento con el sistema propuesto en este proyecto. Así mismo, se ha hecho uso del modelo preentrenado *COCO*[36] (*Common Objects in COntext*).

5. Selección y preprocesado del dataset de entrenamiento y validación para las redes neuronales AlexNet y ResNet.

6. Implementación y entrenamiento de redes neuronales.

   - AlexNet 2.4.6
   - ResNet 2.4.7

7. Integración de *TensorFlow Extended*[37] (TFX), para poder servir los modelos de redes neuronales hechas con TensorFlow a través de una API Rest.

8. Integración con *TensorBoard*[38].

Al final de este proceso el resultado es un sistema con el cual fácilmente se puede entrenar una nueva versión de uno de los modelos existentes y que éste se actualice automáticamente sin necesidad de detener el sistema en ningún momento.

Así mismo, el diseño es de naturaleza flexible de forma que, se puede crear una nueva funcionalidad, bien para computar el flujo óptico o bien un nuevo modelo de red neuronal e integrarlo sin que éste afecte al conjunto del sistema previamente diseñado[4].

Esto ocurre porque la estructura hace una clara diferenciación entre los algoritmos de flujo óptico y las redes neuronales, así como sus modelos entrenados.

3.1.1. Conexión a sucesiones de imágenes

Se ha creado la clase `VideoController` haciendo uso de las funcionaledes de *OpenCV* y *cv2-tools*. Esta sirve a modo de estructura básica que permita establecer la conexión con dispositivos (por ejemplo, webcam) o ficheros de vídeo, así como la implementación de funcionalidades útiles para el resto de algoritmos.

[4] Aunque en este caso el sistema si tendría que reiniciarse para cargar la nueva estructura de red neuronal.

3.1.2. Implementación de algoritmos de flujo óptico

Como se ha mencionado previamente, las regiones de interés (ROI) constituyen el elemento esencial como entrada a la correspondiente red neuronal. Para ello, se han hecho uso de algoritmos de flujo óptico, concretamente, Lucas-Kanade 2.2.1 y Farneback 2.2.2 (aunque no son los únicos existentes), los cuáles utilizan aproximaciones diferentes (flujo óptico disperso frente a flujo óptico denso, respectivamente). En cualquier caso, tras diversos experimentos de ensayo y error, se optó por utilizar únicamente el algoritmo de Farneback debido a que Lucas-Kanade no calcula el desplazamiento para todos los píxeles, sino para los puntos de interés que selecciona el algoritmo de Shi-Tomashi 2.1, esto implica que, si bien, se puede obtener el movimiento de un objeto, no se pueden diferenciar los píxeles pertenecientes a este mismo objeto, y por tanto, no se pueden obtener las regiones de interés necesarias en las siguientes etapas.

En cualquier caso, para la implementación de Lucas-Kanade, Shi-Tomashi y Farneback se han hecho uso de las clases `LucasKanade_OF` [5] y `Dense_OF`, que a su vez, heredan de la clase mencionada previamente `VideoController`.

3.1.3. Implementación de YOLOv3

Tal y como se ha mencionado previamente, YOLO no puede aprovechar el conocimiento obtenido sobre la imagen a través de los algoritmos de flujo óptico. Esto es, porque YOLO no recibe un fragmento de imagen como entrada a la red neuronal, sino la imagen completa de la misma para procesarla en una única llamada, tal y como se explica en la subsección 2.4.8.

No obstante, YOLO es una de las mejores alternativas en cuanto a eficiencia y porcentaje de acierto al sistema propuesto en este proyecto. Por lo tanto, es útil, plantearse la implementación del mismo para poder realizar una comparación con la alternativa propuesta en este trabajo.

[5] De manera transparente, utiliza el algoritmo de Shi-Tomashi para obtener nuevos puntos de interés cada 60 fotogramas.

Para lograrlo, se ha hecho uso del proyecto *Darknet*[39], su modelo preentrenado *COCO* (capaz de detectar 80 clases diferentes, entre ellas, vehículos) y su integración con *OpenCV*.

Como resultado, se ha obtenido la clase `Yolo` que a su vez hereda de `Dense_OF`.

3.1.4. Selección y obtención del conjunto de datos

La obtención de un dataset adecuado es una de las etapas más críticas para el correcto funcionamiento de una red neuronal. Es por esto, que hay que elegir el mismo, cuidadosamente.

En este proyecto inicialmente se optó por implementar un sistema para obtener fragmentos de imágenes etiquetadas a partir de vídeos. Para ello, el programa selecciona fotogramas f distanciados entre sí por ϵ fotogramas, eligiendo por tanto los fotogramas $f(0), f(\epsilon), f(2\epsilon), etc.$ En el contexto de este proyecto, mediante ensayo y error, se optó por fijar $\epsilon = 50$ debido a que permite obtener por un lado, suficientes imágenes y por otro, imágenes suficientemente diferentes[6]. Sobre estos fotogramas seleccionados, se le solicita de manera gráfica al usuario que seleccione todos los objetos de cada categoría (aprendizaje supervisado) para poder generar así las subimagenes clasificadas.

En la teoría, este sistema de generación de datasets es completamente válido, pero en la práctica esta solución requiere de una gran dedicación de horas y un conjunto de vídeos (para generar dichas imágenes) suficientemente amplio y con fotogramas suficientemente variados para que la red neuronal pueda ser entrenada correctamente. Esto es debido a que demasiadas imágenes sobre el mismo fondo, o con la misma iluminación dificultan el entrenamiento de una red neuronal capaz de generalizar correctamente, y por tanto, si el resto de los pasos se hicieran correctamente se acabaría obteniendo una red neuronal con *overfitting* 2.14.

Debido a las limitaciones, se optó por la utilización de datasets previamente generados en otros proyectos. En este caso, el dataset MIO-TCD[3][40], que cuenta, con 11 categorías

[6]Es importante tener en cuenta los fotogramas por segundo (FPS) del vídeo original, en este caso, 30 FPS.

(diferentes tipos de vehículos, peatones y fondo). De este conjunto de datos se han suprimido tres categorías poco relevantes para este proyecto y se ha seleccionado un número similar de imágenes de cada categoría, en concreto 1982 imágenes para todas las categorías salvo coche, que cuenta con 3964 imágenes.

Se pueden observar todas las categorías utilizadas así como un ejemplo de imágen para cada una de ellas en la figura 3.2.

(a) Bicicleta (b) Bus (c) Camión (d) Camión articulado (e) Camioneta

(f) Coche (g) Fondo (h) Motocicleta (i) Van

Figura 3.2: Ejemplo de las categorías del conjunto de datos MIO-TCD[40]

El conjunto de las imágenes seleccionadas se ha dividido de manera aleatoria en dos subconjuntos en una proporción de 8 a 2, el conjunto de entrenamiento y el conjunto de validación respectivamente.

Así mismo, se ha realizado un post procesado sobre todas las imágenes para que sus dimensiones coincidan con la entrada de ambas redes neuronales (AlexNet y ResNet).

3.1.5. Implementación y entrenamiento de redes neuronales

Como se ha mencionado previamente, para la implementación de las redes neuronales, se ha hecho uso del ecosistema de TensorFlow. Esto es, porque TensorFlow es uno de los estándares en el campo del aprendizaje automático. Ofreciendo, no sólo una manera eficaz

y versátil de implementar redes neuronales, sino también, la capacidad de visualizar el funcionamiento de las mismas o el dataset de entrenamiento haciendo uso de TensorBoard. Así mismo, TensorServer permite servir los modelos de manera flexible a través de un API de muy alto rendimiento a la vez que permite guardar un control de versiones en lo que a los modelos entrenados respecta.

Todo ello, no solo agiliza el proceso de desarrollo, sino que al mismo tiempo ofrece mejores rendimientos que si tuviera que implementarse desde cero.

Teniendo todo esto en cuenta, a continuación se explica el proceso de implementación de las redes neuronales AlexNet y ResNet.

3.1.5.1. AlexNet

Para la implementación de AlexNet se siguió la teoría explicada en la subsección 2.4.6. Así mismo se diseñó con la suficiente versatilidad para que fácilmente se pueda adaptar el tamaño de las imágenes que recibe como *input*, y el número de categorías que debe detectar.

El proceso de entrenamiento de AlexNet llevó un total de 14 iteraciones exitosas[7], realizando adaptaciones en el número de épocas, tamaño del *batch* y tamaño de las imágenes que se recibirían como input tanto en la fase de entrenamiento como en la fase de clasificación.

Finalmente, se optó por un modelo con aproximadamente 20000 imágenes de tamaño 227×227 entrenado en un total de 40 épocas en *batches* de 16 imágenes.

El tamaño de *batch* se decidió especialmente para tratar de realizar el entrenamiento de la forma más eficiente posible, pero sin agotar la memoria de la tarjeta gráfica. El número de épocas se fijó al valor mencionado, puesto que a partir de la época 35 el sistema ya no realizaba un incremento sustancial en el porcentaje de aciertos.

3.1.5.2. ResNet

Para la implementación de ResNet se siguió la teoría explicada en la subsección 2.4.7.

[7]En el sentido, de que el entrenamiento ha acabado correctamente, no en porcentajes elevados de acierto, eso se puede ver en detalle en el capítulo 4.

En este caso, se implementaron de manera simultánea 5 modelos sustentados bajo la misma teoría, éstos son, ResNet-18, ResNet-34, ResNet-50, ResNet-101 y ResNet-152. De todos ellos se ha puesto el énfasis en el modelo ResNet-50.

Todo el proceso de entrenamiento llevó un total de 21 iteraciones hasta alcanzar un modelo final. Durante las mismas, se seleccionó la variante de ResNet-50, se optó por imágenes de 64×64, 40 épocas[8] y *batches* de 32 imágenes.

3.1.6. Integración con TensorFlow Extended

Debido a que los modelos de redes neuronales han sido diseñados en TensorFlow, la integración con *TensorFlow Extended* no ha sido especialmente complicada.

Así mismo, y con el fin de que las redes neuronales queden completamente abstraídas del resto del código se diseñó un script encargado de servir las mismas de manera pública a través de Tensorserver y un método `send_frame_serving_tf` que realiza la función de intermediario para comunicar las redes neuronales con el resto del código.

De este modo, se obtiene un sistema ágil y flexible con facilidad para sustituir los modelos de redes neuronales que se utilizan, por otros, diseñados en TensorFlow o incluso programados haciendo uso de otras librerías o lenguajes de programación, siempre y cuando dichos modelos sean accesibles a través de una API Rest.

3.1.7. Integración de todos los componentes previos

En este punto del proyecto, se diseñó la clase `DenseClassifier` que a su vez hereda de `Dense_OF` 3.1.2 y aprovecha las funcionalidades previamente programadas, principalmente, el cálculo del flujo óptico usando el algoritmo de Farneback. Sobre el resultado del mismo, se aplican técnicas de umbralización para eliminar el ruido. Posteriormente, se aplica el algoritmo de búsqueda de contornos. De este modo, si el algoritmo se ha ejecutado correctamente, cada contorno equivale a un objeto, y puesto que se tiene la forma del mismo, se pueden obtener también las coordenadas máximas y mínimas en cada eje que delimita el propio

[8] Al igual que AlexNet, cerca de la época 35 no experimenta mejora sustancial en el porcentaje de aciertos.

objeto. De esta forma, se filtra una vez más dicha región, de modo que, sólo sean válidas las regiones de un tamaño aceptable de acuerdo al tipo de imágenes utilizadas en el contexto de este proyecto.

En este caso, y dado los ángulos de la cámara, es suficiente con especificar regiones de al menos, 20 píxeles tanto de altura como anchura y de menos de la mitad del tamaño del fotograma o imagen que contiene los objetos. Esto es, porque en general regiones tan grandes significan que se ha movido la cámara en lugar de los objetos[9], caso para el cual el algoritmo de Farneback no funciona correctamente.

A continuación, las subimágenes que han pasado el filtro correspondientes a cada objeto en movimiento en el fotograma actual, se envían a la red neuronal, independientemente de que sea AlexNet 3.1.5.1, ResNet 3.1.5.2 u otra, haciendo uso del método explicado previamente `send_frame_serving_tf` 3.1.6. Con la respuesta del mismo, se muestran gráficamente superpuestos sobre la imagen original todos los objetos etiquetados con su categoría correspondiente a excepción de los objetos detectados con la categoría *background* o fondo, puesto que en este caso, se opta por no etiquetar ni recuadrar el objeto en movimiento[10].

Por último, y a modo de resumen, en la figuras 3.3, se represantan dos diagramas UML con la relación entre todas las clases previamente mencionadas. Cabe destacar que este diagrama no incluye ningún *script* en el que no se han hecho uso de clases.

[9] Al sistema le afecta mucho cualquier tipo de movimiento de cámara, por pequeño que sea.
[10] Se asume que ha sido un falsa detección de objeto en movmimento por parte del algoritmo de Farneback.

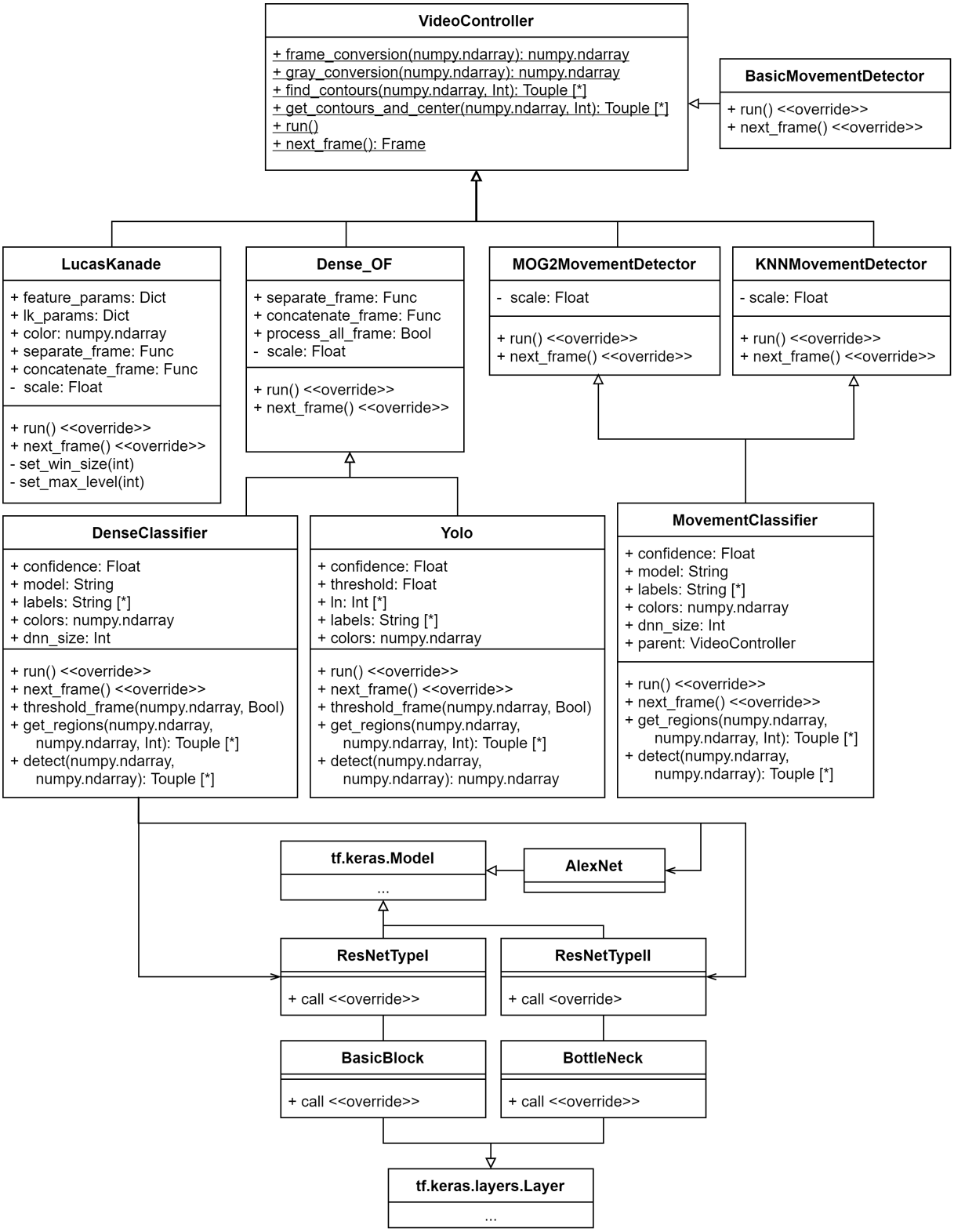


Figura 3.3: Diagrama UML del proyecto.

Capítulo 4

Resultados

Teniendo encuenta los fundamentos teóricos y el desarrollo, es necesario realizar un análisis de los resultados, así como la comparación entre las diferentes técnicas utilizadas. Para ello, este capítulo se divide en dos secciones principales, flujo óptico y redes neuronales.

Así mismo, es importante destacar que todas las pruebas de rendimiento se han realizado en el mismo ordenador y en las condiciones lo más parecidas posibles.

4.1. Flujo óptico

En lo que a los algoritmos de flujo óptico respecta, se han realizado mediciones de tiempo de ejecución y fotogramas por segundo obtenidos en cada algoritmo.

Para realizar dichas mediciones, se han utilizado 20 vídeos con una resolución original de 1920×1080, codificados con el códec H.264 y renderizados a 30 fotogramas por segundo (FPS). Así mismo, dichos vídeos cuentan con una tasa de bits de vídeo de 17015 Kbps (no importan los bits de audio puesto que para este proyecto, éstos se descartan) y una duración promedio de 1 minuto y 6 segundos.

De esta forma, se han ejecutado los algoritmos de Lucas-Kanade 2.2.1 y Farneback 2.2.2 utilizando los vídeos previamente especificados en diferentes resoluciones, éstas son, 1920×1080, 1440×810, 960×540 y 480×270, o lo que es lo mismo, en los escalados 1, 0.75, 0.5 y 0.25 respectivamente sobre los vídeo originales. Cabe destacar que para realizar estas pruebas no se han visualizado el resultados obtenidos en los vídeos, esto es para que

la reproducción de los vídeos, manejo de la ventana o tratamiento de los píxeles con el único fin de visualizar los resultados, no afecte al rendimiento final obtenido, puesto que este tratamiento no forma una parte intrínseca de estos algoritmos.

Con este experimento el objetivo es extrapolar la tendencia de estos algoritmos en función del número de píxeles en cada fotograma.

Los resultados obtenidos quedan especificados en la figura 4.1. Estos resultados son valores promediados sobre los vídeos utilizados para el análisis. Como se puede observar, el aumento de la resolución afecta al algoritmo de Farneback de manera exponencial. Esto tiene sentido puesto que este algoritmo evalúa todos los píxeles y para cada uno de ellos comprueba sus vecinos. Por otro lado, en el algoritmo de Lucas-Kanade se puede observar una ligera mejora en cuanto al tiempo y los fotogramas por segundo al reducir el número de píxeles por fotograma. No obstante esta diferencia no es especialmente relevante. Esto es debido a que pese a aumentar el número de píxeles por fotograma, no se ha aumentado el número de puntos de interés seleccionados con el algoritmo de Shi-Tomasi 2.1 y por tanto, los píxeles a evaluar son los mismos. La diferencia en tiempos de ejecución se debe al manejo de matrices (fotogramas) de mayor o menor tamaño.

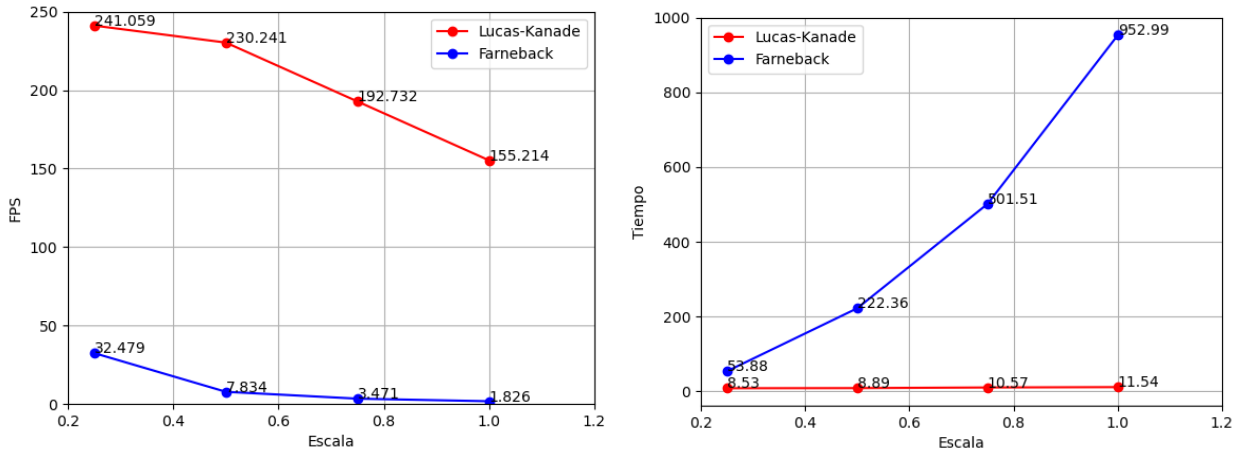


Figura 4.1: Gráfica de tiempo y FPS en función a la escala del vídeo.

En términos generales los porcentajes de aciertos se sitúan en torno al 96 % y 94 % en Farneback y Lucas-Kanade respectivamente. Estos porcentajes se han establecido por un

experto humano, que determina para cada fotograma los objetos que él considera verdaderamente en movimiento de forma que cuando el algoritmo correspondiente los detecta como tal, se anota un acierto. A la vista de estos resultados se puede fácilmente inferir el adecuado comportamiento de estos algoritmos.

No obstante a continuación se explican algunos casos particulares, que no entrarían en el cómputo mencionado previamente.

En la figura 4.2 se puede observar cómo el algoritmo de Farneback ha detectado movimiento en la totalidad de la imagen[1]. Esto se deduce por la aparición de tonalidades diferentes al negro que indica ausencia total de movimiento, de forma que cualquier otra tonalidad indica presencia de movimiento, tanto más acusado cuanto más intensa es ésta. Esto es porque dicho sistema es muy sensible al movimiento de la cámara[2]. En la figura previamente mencionada la cámara se ha movido levemente, provocando así que el sistema detecte el movimiento de los píxeles en la totalidad de la imagen, produciendo así un error que se propaga a las siguientes fases del sistema (detección de regiones de interés 2.2.3 y clasificación con la red neuronal).

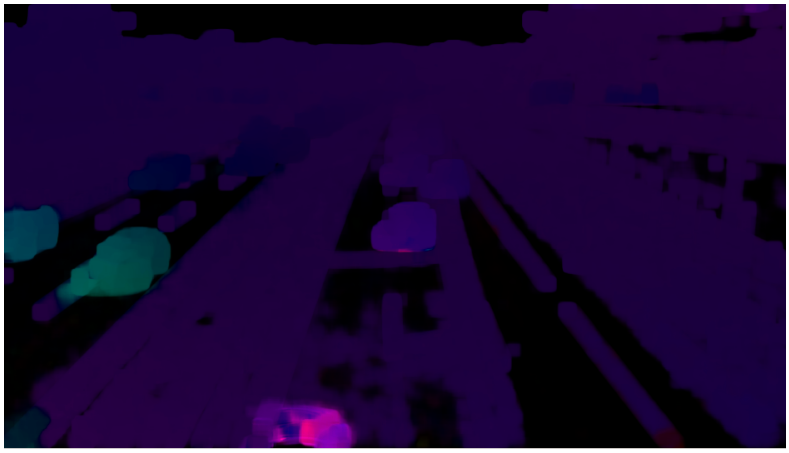


Figura 4.2: Ejemplo de fallo con el algoritmo de Farneback.

[1]Se utiliza el algoritmo previamente mencionado en este proyecto de traducción de dirección e intensidad a un color, aunque en este caso no se está solapando con la imagen original.

[2]Esto ocurre igualmente en el algoritmo de Lucas-Kanade, pero dado que éste sólo observa ciertos píxeles es más difícil de percibir

En la figura 4.3 se puede observar la traducción realizada de la magnitud y dirección del movimiento detectada por el algoritmo de Farneback a una imagen en blanco y negro, reservando el color blanco para los objetos sobre los que se ha determinado la presencia de algún movimiento y el negro para el resto de la imagen. Además, sobre este resultado, se aplica un filtro para evitar secciones demasiado pequeñas (ruido). Concretamente se aplica un filtrado consistente en una operación de apertura morfológica, para eliminar esas pequeñas regiones, que no representan vehículos en movimiento. No obstante, a pesar de esto, se puede observar el contorno de varios vehículos solapados entre sí (dos en la parte izquierda de la imagen y otros dos en la parte central). Esto es algo que fácilmente puede ocurrir en un caso real y es una característica especialmente crítica en este sistema debido a que estos casos producen que el algoritmo de obtención de regiones de interés 2.2.3 detecte ambos vehículos (o en general objetos) como uno solo, produciendo a su vez el envío de una imágen incorrecta a la red neuronal que se esté utilizando.

Figura 4.3: Ejemplo de fallo en la obtención de ROIs.

4.2. Redes neuronales

En cuanto al entrenamiento de las redes neuronales, tal y como se ha visto en la subsección 3.1.4, se ha utilizado un dataset con 9 categorías y un total de 19820 imágenes.

Los resultados de todos los modelos de redes neuronales propuestos se pueden ver en

la tabla 4.1. De esos datos, los más relevantes son *Validation Loss* y *Validation Accuracy*, pudiendo observar que el modelo que mejor resuelve este problema ha sido ResNet-18 y el que por ResNet-101.

Modelo	**Loss**	**Accuracy %**	**Validation Loss**	**Validation Accuracy %**
AlexNet	0.2139	87.91	2.8340	58.32
ResNet-18	0.0568	98.34	1.9122	68.55
ResNet-34	0.0175	99.46	2.8784	63.92
ResNet-50	0.1350	94.40	1.8462	62.55
ResNet-101	1.2362	53.12	8.9631	39.91
ResNet-152	Sin capacidad de cómputo			

Tabla 4.1: Resultados de entrenamiento de los modelos de red neurconal

Hay que tener en cuenta que los resultados obtenidos en un caso real, frente a los resultados de validación tienden a ser menores. Esto es debido a que las imágenes del conjunto de datos de entrenamiento y validación no cubren exáctamente los ángulos y resolución de los vídeos utilizados. Así mismo la sub-imagen obtenida de los algoritmos de flujo óptico es reescalada para adaptarse a la entrada de la red neuronal. Esto en general, no es una solución óptima, pues se pierde el concepto de escala de los objetos y puede perderse resolución o deformese ligeramente.

Además se ha realizado un análisis de todos los modelos de manera análoga a como se explicó en la sección 4.1, pero únicamente con fotrogramas de tamaño 1440×810, o lo que es lo mismo, con escala 0,5. De esta forma, en la tabla 4.2 se puede observar el rendimiento obtenido con los diferentes modelos así como el dato aproximado de tiempo dedicado a la red neuronal (restando el tiempo dedicado a los algoritmos de flujo óptico). Se muestran los modelos de red en la primera columna, los FPS para cada modelo en la segunda columna, así como el los tiempos en segundos, dedicados a la red neuronal y en total en la tercera y cuarta columna respectivamente. Como se ha explicado anteriormente, el cómputo del tiempo dedicado a la red neuronal es aproximado. Se ha calculado conociendo el número total de fotogramas de los vídeos utilizados (se ha limitado a 1980) y los fotogramas por segundo que se obtuvieron utilizando el algoritmo de Farneback (de manera independiente)

Modelo	FPS	Tiempo dedicado a la red neuronal (s)	Tiempo total (s)
AlexNet	1.479	1095.12	1347.86
ResNet-18	5.366	116.25	368.99
ResNet-34	5.148	131.87	384.61
ResNet-50	4.833	156.94	409.68
ResNet-101	4.111	228.89	481.63
ResNet-152	Sin capacidad de cómputo		

Tabla 4.2: Resultados de rendimiento de los modelos de red neuronal

sobre los vídeos, éstos son en promedio, 7.834 fotogramas, tal y como se puede ver en la figura 4.1. Por tanto 1980/7,834 = 252,74 es el tiempo aproximado que se dedica al algoritmo de Farneback sobre el tiempo global obtenido de cada red neuronal. De esta forma, sólo queda sustraer 252,74 a cada tiempo global para obtener un tiempo aproximado dedicado a la detección en cada modelo.

Esto se ha podido hacer, puesto que los vídeos utilizados han sido los mismos en todos los casos y las condicienes de prueba han sido lo más parecidas posibles. Así mismo, a continuación se exponen en detalle los resultados obtenidos con cada modelo.

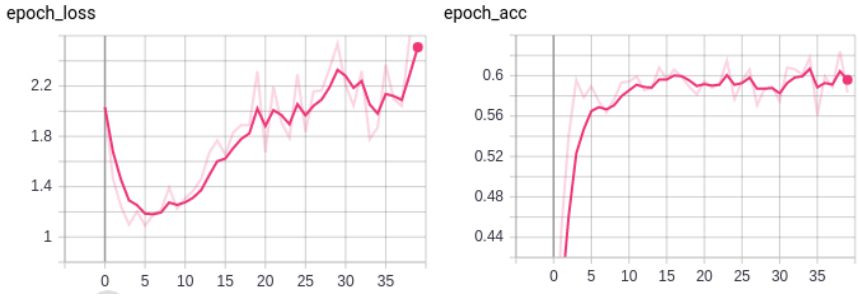
4.2.1. AlexNet

El tiempo promedio de entrenamiento de la red neuronal AlexNet ha sido de 29 minutos y 31 segundos utilizando el algoritmo de optimización Adam[41] con un ratio de aprendizaje de 0,001 y con *batches* de 16 imágenes.

Se puede ver la evolución del entrenamiento en la figura 4.4[3] se observa el progreso del entrenamiento a lo largo de las 40 épocas, tanto en los resultados de la función de error 4.4a como en la precisión de la misma 4.4b.

Cabe destacar que en cada gráfica muestran dos líneas. La más suavizada hace referencia al error y precisión en el conjunto de entrenamiento, mientras que la otra hace referencia al error y precisión en el conjunto de validación.

[3]Gracias a la integración con TensorBoard.



(a) Error por época (b) Precisión por época

Figura 4.4: Gráficas del entrenamiento de AlexNet.

Como se puede observar, aproximadamente a partir de la época 15, el sistema no mejora sustancialmente en cuanto a la precisión, sin embargo, la función de error empeora el resultado. Habitualmente esto significa que el sistema está experimentando *overfitting*. Se puede corroborar observando, en la tabla 4.1, la diferencia entre el *Accuracy* y el *Validation Accuracy*.

Esta es una tendencia que se da no sólo en el modelo AlexNet sino también en ResNet en todas sus variantes. Este problema se da principalmente por disponer de un conjunto de datos de entrenamineto idóneo para el proyecto, puesto que el *dataset* cuenta con imágenes de resoluciones muy dispares y desde ángulos muy variados (no está planteado especialmente para una cámara en posición superior a los vehículos y alineada con la dirección de la carretera).

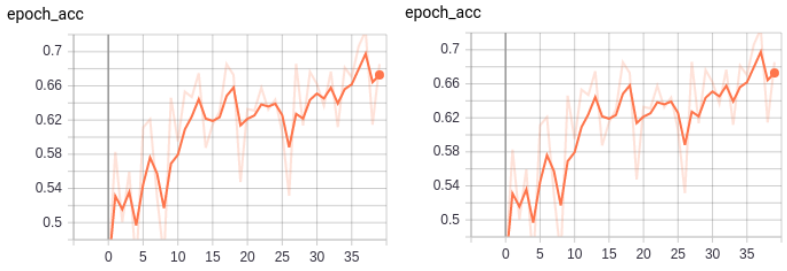
4.2.2. ResNet

En cuanto al entrenamiento de ResNet, no hay que olvidar que se han implementado 5 variantes del mismo, éstas son ResNet-18, ResNet-34, ResNet-50, ResNet-101 y ResNet-152, si bien esta última no ha podido utilizarse en este proyecto por no disponer de una tarjeta gráfica con suficiente memoria para entrenarla.

De esta forma, los tiempos de entrenamiento utilizando *batches* de 32 imágenes para cada uno de los modelos han sido: ResNet-18: 29 minutos y 31 segundos; ResNet-34: 32 minutos y 47 segundos; ResNet-50: 39 minutos y 11 segundos; ResNet-101: 1 hora, 10 minutos y 28

segundos. Estos tiempos tienen sentido puesto que a mayor complejidad de la red neuronal más tiempo requiere para su entrenamiento.

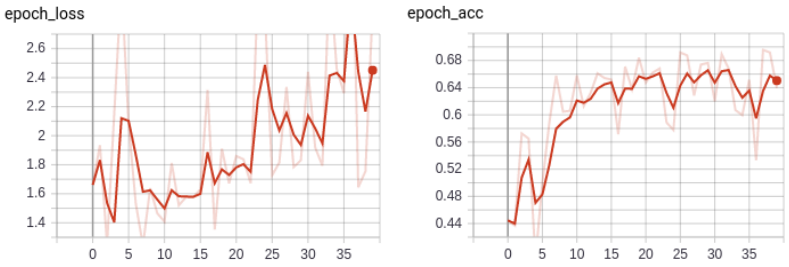
Además, a continuación en las gráficas 4.5a y 4.5b del proceso de entrenamiento de todos los modelos siguiendo la misma nomenclatura que en las gráficas de AlexNet 4.2.1.



(a) Error por época (b) Precisión por época

Figura 4.5: Gráficas del entrenamiento de ResNet-18.

De nuevo, en estas gráficas se observa una tendencia general a que los modelos dejen de mejorar en torno a la época 15 (a excepción de ResNet-18 y ResNet-50 que mejoran aproximadamente hasta la época 35).

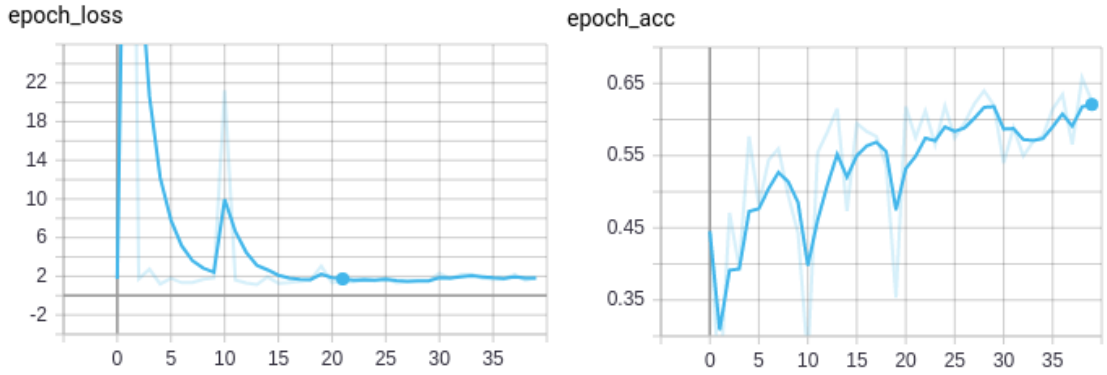


(a) Error por época (b) Precisión por época

Figura 4.6: Gráficas del entrenamiento de ResNet-34.

Además, resulta especialmente interesante el resultado obtenido con el modelo de ResNet-50, dado que, como se puede observar en la figura 4.7a, éste no está empeorando la función de error (o función de coste). Puede deberse a que el modelo es capaz de generalizar mejor las estructuras subyacentes en las imágenes[4].

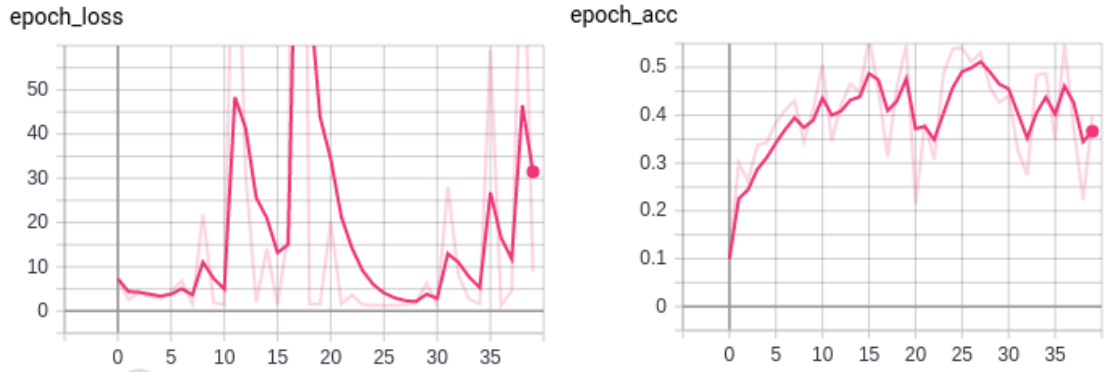
[4]Es decir, ha aprendido a clasifcar las imágenes de estas categorías.



(a) Error por época

(b) Precisión por época

Figura 4.7: Gráficas del entrenamiento de ResNet-50.



(a) Error por época

(b) Precisión por época

Figura 4.8: Gráficas del entrenamiento de ResNet-101.

4.2.3. YOLO: You Only Look Once

Como se ha explicado previamente se ha utilizado la red neuronal Yolov3 con el modelo prentrenado COCO, por lo tanto, no se tienen tiempos estimados de entrenamiento. Así mismo, en el caso de YOLO no se ha conseguido hacer uso de la GPU para la medición de los tiempos. Esto es así debido a la dificultad intrínseca de realizar la instalación de OpenCV de manera customizada habilitando los módulos *DNN* para conectarse a CUDA.

Por otro lado, no es útil comparar el porcentaje de acierto del modelo con los demás propuestos en este proyecto, puesto que los *datasets* utilizados son diferentes.

En cualquier caso, a continuación se exponen los datos referentes al rendimiento conse-

guido con Yolov3 haciendo uso de la *CPU*. Es importante destacar que todos los datos han sido el promedio de los resultados obtenidos en 20 vídeos diferentes.

El modelo de red neuronal se ha ejecutado a una velocidad de 2.42 fotogramas por segundo, obteniendo así un rendimiento superior a AlexNet (con 1.479 FPS). Dedicando por tanto, en promedio un total de 818.18 segundos a la red neuronal.

Es importante destacar que las pruebas de rendimiento se han realizado sin mostrar los resultados por pantalla, para que esto no afecte a la medición de los tiempos.

4.2.4. Resultados de detección de vehículos

En la tabla 4.3 se especifican los resultados obtenidos con cada modelo haciendo uso del flujo completo del sistema. Para ello se han utilizado 10 vídeos con una duración promedio de 5 segundos y a razón de 30 FPS, lo que da lugar a 150 fotogramas. De esta forma se ha obtenido un porcentaje de aciertos para cada modelo de red neuronal. Estos porcentajes se han establecido por un experto humano, que determina para cada fotograma los objetos que han sido correctamente clasificados. Es importante destacar que para este cálculo sólo se han tenido en cuenta las clasificaciones realizadas sobre sub-imágenes correctamente seleccionadas. Es decir, ignorando los casos particulares mencionados en la subsección 4.1.

Modelo	**Accuracy %**
AlexNet	57.00
ResNet-18	61.04
ResNet-34	62.83
ResNet-50	61.02
ResNet-101	42.33
ResNet-152	Sin capacidad de cómputo
Yolov3	99.21

Tabla 4.3: Resultados de clasificación de vehículos

En la figura 4.9 se puede observar un ejemplo de detección y selección correcta de vehículos utilizando el modelo de red neuronal ResNet-34, que es un ejemplo de frame como los utilizados en el cómputo de los resultados mostrados en la tabla 4.3.

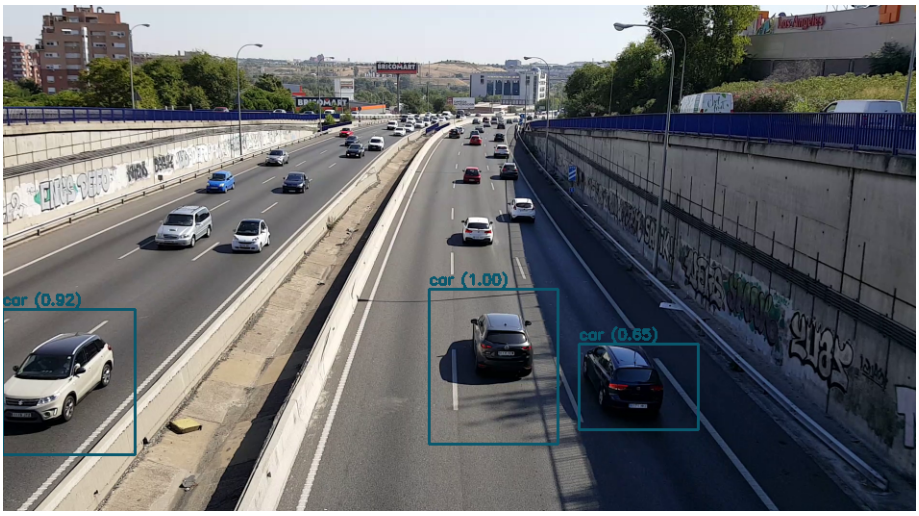


Figura 4.9: Ejemplo correcto de detección en el flujo completo del sistema.

Así mismo, se ha estudiado los momentos en los que los modelos detectan mejor y peor la categoría de vehículo correcta.

Como se puede ver en la figura 4.10, se ha observado que los modelos no tienen la capacidad de inferir correctamente la categoría de los objetos detectados cuando hay más de un vehículo en la sub-imagen seleccionada. No obstante tal y como se ha mencionado previamente estos casos no se han contabilizado para el porcentaje de aciertos.

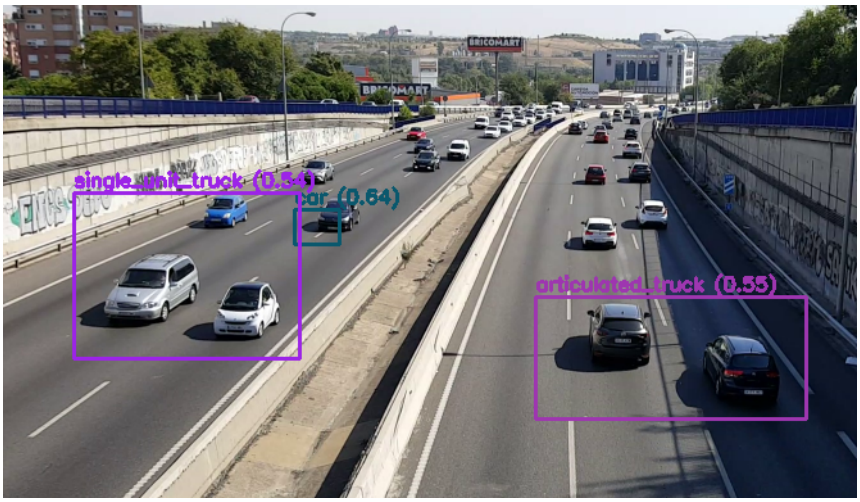
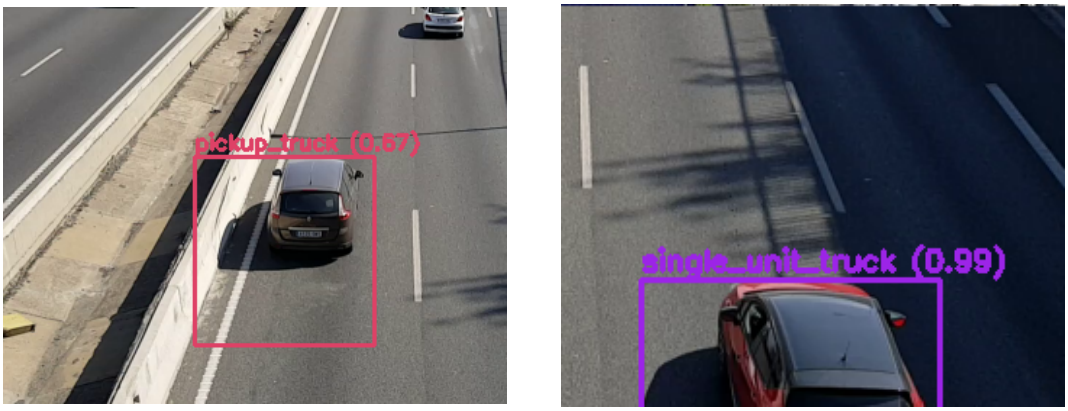


Figura 4.10: Ejemplo de fallo en la detección por ROI con múltiples vehículos.

Las otras dos circunstancias donde los vehículos se detectan especialmente mal, han sido,

en primer lugar, cuando las regiones seleccionadas eran sustancialmente mayores a la región asociada al vehículo[5] como se puede ver en la figura 4.11a, y en segundo lugar, cuando no se puede ver el vehículo completo en el fotograma actual, tanto si es en la imagen, como en la sub-imagen. Esto es un problema especialmente cuando los vehículos se encuentran entrando o saliendo del plano de la cámara. La figura 4.11b) muestra una sub-imagen con error de detección por vehículo incompleto (por la parte inferior la imágen no ha sido cortada, sino que es el límite de la cámara).



(a) ROI demasiado grande

(b) Objeto incompleto

Figura 4.11: Fallos comunes de detección

En el resto de circunstancias no se han detectado porcentajes de acierto muy superiores o muy inferiores a los datos especificados en la tabla 4.3. Del mismo modo, no se ha detectado ninguna categoría que haya sido clasificada errónemente en un porcentaje significativamente mayor que el resto.

[5]Estos datos si entran en el cómputo del porcentaje de acierto.

Capítulo 5

Conclusiones y trabajo futuro

En relación a los temas abordados en este trabajo, a continuación se indican los aspectos más relevantes que merecen especial consideración en relación a los resultados obtenidos, así como posibles líneas de trabajo futuro.

5.1. Conclusiones generales

En cuanto a los objetivos específicos planteados en la sección 1.2, éstos se han alcanzado con resultados satisfactorios, por tanto, se puede concluir que el trabajo desarrollado cumple con las expectativas planteadas para la prueba de concepto desarrollada.

En lo que a los algoritmos de flujo óptico respecta, se han conseguido implementar con éxito y realizar comparaciónes de los mismos, tanto en casos de uso como eficiencia. En el contexto de este proyecto se optó especialmente por el algoritmo de Farneback 2.2.2 dado que este ofrece la posibilidad de obtener la forma de los objetos y extraer de ahí las regiones de interés. No obstante hay que tener en cuenta que el algoritmo de Farneback conlleva mayor coste computacional que el algoritmo de Lucas-Kanade 2.2.1, es por esto que siempre que los requisitos del sistema lo permitan es una mejor opción frente al de Lucas-Kanade.

De igual forma, dentro de este mismo ámbito (flujo óptico), el sistema ha mostrado ser robusto y estable aunque con ciertos problemas difícilmente solventables. Estos afectan principalmente al algoritmo de Farneback y suceden cuando se solapan dos objetos en movimiento o cuando se mueve la cámara. Aún así, en la sección 5.2 se explican alternativas

de futuro que podrían atenuar estos problemas.

Por otro lado, en lo que a las redes neuronales respecta, el resultado ha sido satisfactorio, pues se ha conseguido realizar un sistema robusto, escalable e independiente del resto del proyecto. Además, frente a la propuesta inicial de implementar las redes neuronales AlexNet 2.4.6 y ResNet 2.4.7, se ha implementado con éxito el detector de objetos YOLOv3 2.4.8, aunque este último no se ha conseguido ejecutar haciendo uso de GPU de manera integrada con OpenCV. Hay que aclarar que sí se ha conseguido como librería separada de OpenCV, haciendo uso de Darknet, no obstante se ha considerado que la inclusión de esta versión daría lugar a un proyecto con malos patrones de diseño y mal estructurado, por tanto, se optó únicamente por la integración de OpenCV sin tarjeta gráfica, aunque esto conlleva no poder realizar las comparaciones adecudas en rendimiento frente al sistema propuesto.

5.2. Trabajo futuro

El proyecto desarrollado hace uso de diversas tecnologías y, sobre todo, se basa en dos fundamentos teóricos (detección de movimiento en imágenes y aprendizaje profundo), por tanto, cuenta con multiples líneas de investigación o mejora sobre la versión actual. A continuación se mencionan algunas de las áreas más importantes donde se puede mejorar o complementar diferentes aspectos involucrados en el proyecto.

5.2.1. Generación del conjunto de datos

Se hizo uso de un conjunto de datos previamente generado y únicamente se hicieron modificaciones sobre el mismo. Esto tiene bastantes limitaciones en cuanto al ángulo de las imágenes que definen los vehículos, la resolución o las categorías diferenciadas. Este es un punto de especial importancia que podría llevar a la implementación del sistema en un entorno real. Sabiendo el ángulo de la cámara que se va a utilizar para detectar vehículos, así como la resolución de la imágen se puede generar un conjunto de datos particularizado para el entorno de aplicación, que sin duda redundará en el incremento de las tasas de éxito.

5.2.2. Flujo óptico

Cabe destacar que en cuanto al flujo óptico se ha utilizado el algoritmo de Farneback 2.2.2 en el flujo completo del sistema, quedando Lucas-Kanade descartado por no poderse obtener la forma de los objetos detectados (y por tanto la región de interes o ROI).

No obstante, se pueden utilizar ciertas aproximaciones partiendo del algoritmo de Lucas-Kanade, tales como, generar regiones de interés utilizando los píxeles destacados [1] calculando el centro de las mismas. De este modo se obtendría un sistema mucho más eficiente en tiempo de ejecución. Con respecto a los resultados previamente mencionados en el capítulo 4 se obtendría un sistema casi 30 veces más rápido (en lo que a flujo óptico respecta) utilizando una resolución de 1440×810.

Por otro lado, no se han explotado todas las variantes existentes en algoritmos de flujo óptico, tales como el algoritmo de Horn-Schunck[42] que quizá podrían ser de utilidad dado que está orientado a resolver lo que en este contexto se conoce como el problema de apertura, derivado del paso de escenas reales 3D a las imágenes 2D y que en ocasiones genera ambigüedades en la resolución de las ecuaciones que definen el flujo óptico.

5.2.3. Redes neuronales

En este proyecto, se ha planteado en todo momento la entrada de sub-imágenes en formato RGB como entrada para las redes neuronales. No obstante, existen ciertas alternativas a plantear. Una de ellas sería la extracción de bordes de las imágenes (tanto en el entrenamiento como en la detección). De esta forma, se ayuda a la red neuronal a discernir lo que es importante de lo que no, puesto que ya, otros factores, como el color o la textura no afectan a la hora de decidir la categoría del objeto detectado.

Del mismo modo, tampoco se han utilizado modelos de redes neuronales alternativos a AlexNet y ResNet[2], si bien existe una amplia variedad de ellos que también han demostrado buenos resultados en el ámbito de la visión artificial, tales como, GoogleNet[30].

[1] Obtenidos del algoritmo de Shi-Tomasi 2.1.

[2] Aunque si se ha implementado con la intención de poder añadir fácilmente nuevos modelos.

5.2.4. Otras mejoras

El resultado final del presente trabajo es un sistema capaz de detectar y categorizar vehículos en movimiento, no obstante, no se ha avanzado en el sentido de utilizar los resultados con un fin concreto. De esta forma, computando el número de vehículos detectados en una determinada ubicación o momento se pueden plantear futuros trabajos que se dediquen al ánalisis del tráfico, predicción de atascos u otras mediciones típicas de las *smart cities*.

Capítulo 6

Introduction

The development of computer technology (both in processing capacity and in the possibility of memory storage) as well as theoretical advances led to the development of new algorithms.

All this, along with the growing need to solve increasingly complex problems have created a perfect breeding ground for the application of artificial intelligence in various fields, including image and video processing. In this way, it is possible to achieve a machine with a high capacity to interpret images and, in short, to get closer to the human visual perception.

Because of the ease with which we "see", in 1960 artificial intelligence researchers thought it would be relatively simple to develop systems with artificial vision. However, after years of research, it has been proven that this is a very complex task.

Also, these computer vision systems try to solve some of the most important problems today. From pattern detection and classification or motion prediction to composite problems such as automatic driving or the implementation of smart cities.

This work is framed in the context of image processing and consecutive image sequences (video). More specifically, in the detection of vehicles in circulation by using optical flow algorithms, for their subsequent classification using convolutional neural networks (CNN).

This approach serves to determine the flow of vehicles in transit at a given point. These are the systems to be implemented in future smart cities.

6.1. State of the art

There are currently many similar projects trying to address the problem of detecting vehicles in motion, in real time.

Some of the most relevant projects developed in the same context are:

A Vehicle Detection Approach using Deep Learning Methodologies[1]: The authors of the article try to solve the same problem, but the approach is based on the use of R-CNN type neuronal networks[1]. The article and this project, as far as CNNs are concerned, solve the problem following the same principles. However, the method used to obtain regions of interest is completely different, since the article uses generic algorithms to select regions and this project uses the known premises of this specific problem (vehicles are always in motion). However, both approaches are completely valid.

Efficient Scene Layout Aware Object Detection for Traffic Surveillance[2]: Winning article of the MIO-TCD[2][3] where the objective was to achieve a system capable of detecting and classifying vehicles in 11 different categories to find the best system. In this article the problem is addressed in two steps. First by making an analysis of the scene that allows to detect objects in it and then by making the classification. It is important to highlight the performance obtained in this project, being superior to others achieved with FasterRCNN[4] and YOLOv1[5].

YOLO: It is an acronym for "You Only Look Once" and was first presented in 2016 in the article[5] and whose latest version was YOLOv3[6]. These articles propose a completely different way of solving the problem only observing the image once. The system makes use of a neural network that allows to predict the regions of interest and associating them with probabilities of object detection.

The main difference of the proposal in this work with respect to the previously mentioned projects lies in the fact that the objects are determined by their movement. In this way, the

[1]R-CNN (regions with CNN) are convolutional neural networks in which algorithms are used to select regions to be sent to the neural network.

[2]The MIOvision Traffic Camera Dataset (MIO-TCD) Challenge.

neural network is only provided with regions in which movement is detected for classification purposes. This is where the novelty of the proposal lies. The detection of these moving regions is done by applying computer vision techniques, more specifically optical flow detection methods.

6.2. Objetives

This work proposes the creation of a system at the conceptual level capable of detecting moving vehicles, through image sequences (videos). For this purpose, optical flow techniques are used (Lukas-Kanade and Gunner Furnebarck algorithms). In addition, their efficiency and effectiveness will be tested on this particular system.

With this information, the system should be able to generate sections of each frame where each of the moving vehicles are located. These images will be sent to an API Rest connected to different models of convolutional neural networks (Resnet50 and Alexnet) with the intention of classifying the type of vehicle that appears in them.

Finally, the results obtained with each of the neural network architectures will be compared in terms of efficiency, response time and success rate.

Also, during this process, the following objectives will be tracked:

- Analysis and comparison of optical flow detection methods in the proposed context. Selecting the most appropriate one.
- Network model selection and layer redefinition.
- Comparison between network models.
- Integration of the different technologies.

The work planning is based on the specific objectives outlined, contemplating the same number of phases as objectives. All distributed in a balanced way by weeks throughout the development time of the project.

6.3. Motivation

This project arises from the idea of solving current complex problems such as: recommendation of alternative road routes to avoid traffic jams, early detection and capture of suspicious vehicles, generation of road vehicle statistics, generation of traffic forecasts, etc. All these problems have a common requirement, the extraction of data relating to the movement of vehicles on urban and interurban roads.

The solution to this requirement has many approaches with completely different architectures. From large sensor networks, which means costly implementation in society, to the use of mobile phones that act as portable sensors. However, these ideas require a large investment or access to the devices of many users, and even so these solutions will not provide the possibility of obtaining information about the characteristics of the vehicles (type of vehicles, brand, etc.).

With all of this in mind, the need arises to create a system based on detection by using static cameras located near the roads, connected to one or more central servers that perform the data processing.

In this way it would be possible to obtain all the desired data while avoiding some disadvantages of the other approaches. Indeed, implementing this system does not require roadblocks, nor a large sensor network neither access to the devices of thousands of users.

Likewise, from the beginning this system has been designed with the intention of being able to adapt to different contexts, although the restrictions must be similar. For example, by changing the model, it could be used in security cameras to detect and recognize wanted or missing people.

It should be noted that development has been considered at the conceptual level. Since there is no possibility of implementing a real system in public spaces for obvious reasons. In any case, the analyses of the results allow us to be in a position to transfer the system to a real environment.

Capítulo 7

Conclusions and future work

In relation to the subjects addressed in this paper, the most relevant aspects that deserve special consideration in relation to the results obtained, as well as possible lines of future work, are indicated below.

7.1. General conclusions

The specific objectives set out in the section 1.2 have been achieved with satisfactory results. It can be concluded that the work carried out meets the expectations set for the proof of concept developed.

As far as the optical flow algorithms are concerned, they have been successfully implemented and compared in use cases and efficiency. In the context of this project, the Farneback 2.2.2 algorithm was chosen because it offers the possibility of obtaining the shape of the objects and extracting the regions of interest from them. However, it must be taken into account that the Farneback algorithm carries a higher computational cost than the Lucas-Kanade 2.2.1 algorithm. Because of this whenever the system requirements allow it, it is a better option than the Lucas-Kanade algorithm.

In the same way, within this same scope (optical flow), the system has shown to be robust and stable although with certain problems difficult to solve. These mainly affect the Farneback algorithm and happen when two moving objects overlap or when the camera is moved. Even so, in the section 5.2, future alternatives are explained that could mitigate

these problems.

On the other hand, as far as the neural networks are concerned, the result has been satisfactory. A robust, scalable and independent system has been achieved from the rest of the project. In addition, as opposed to the initial proposal to implement the AlexNet 2.4.6 and ResNet 2.4.7 neural networks, the object detector YOLOv3 2.4.8 has been successfully implemented. However, it has not been possible to run YOLOv3 using GPU in an integrated way with OpenCV. It is necessary to clarify that it has been achieved as a separate library from OpenCV, using Darknet. Nevertheless it has been considered that the inclusion of this version would give rise to a project with bad design patterns and badly structured. Therefore, it was chosen to integrate OpenCV without graphics card, although this entails not being able to make the appropriate comparisons in performance against the proposed system.

7.2. Future work

The project developed makes use of various technologies and, above all, is based on two theoretical foundations (motion detection in images and deep learning). Consequently, multiple lines of research or improvement on the current version reamin open. The following are some of the most important areas where different aspects involved in the project can be improved or complemented.

7.2.1. Dataset generation

A previously generated dataset was used and only minor changes were made to it. This has quite a few limitations in terms of the angle of the images that define the vehicles, the resolution or the categories it can detect. This is a significant point that could lead to the implementation of the system in a real environment. Knowing the angle of the camera that will be used to detect vehicles, as well as the resolution of the image, can generate a dataset customized for the application environment. This will undoubtedly result in increased success

rates.

7.2.2. Optical flow

It should be noted that the Farneback 2.2.2 algorithm has been used in the complete flow of the system. Lucas-Kanade was discarded because the shape of the detected objects could not be obtained (and therefore the region of interest or ROI).

However, certain approximations can be used from the Lucas-Kanade algorithm, such as generating regions of interest using the highlighted pixels [1] by calculating their center. This would result in a much more efficient system at run time. With regard to the results previously mentioned in the chapter 4, using a resolution of 1440×810 the system obtained would be almost 30 times faster (in terms of optical flow).

On the other hand, not all the existing variants in optical flow algorithms have been explored, such as the Horn-Schunck[42]. This algorithm could perhaps be useful since it is oriented to solve what in this context is known as the aperture problem, derived from the traduction from real 3D scenes to 2D images. This sometimes generates ambiguities in the resolution of the equations that define optical flow.

7.2.3. Neural networks

In this project, only the input of sub-images in RGB format has been considered as input for the neural networks. However, there are certain alternatives to be considered. One of them would be the extraction of borders from the images (both in training and in detection). In this way, the neural network is helped to discern what is important from what is not, since other factors, such as colour or texture, do not affect the decision on the category of the detected object.

Similarly, alternative neural network models to AlexNet and ResNet have not been used [2], although there is a wide variety of them that have also shown good results in the field of

[1] obtained from the Shi-Tomasi 2.1 algorithm.
[2] Although it has been implemented with the intention of being able to easily add new models.

machine vision, such as GoogleNet[30].

7.2.4. Other improvements

The final result of this work is a system capable of detecting and categorizing moving vehicles. However, no progress has been made in using the results for a specific purpose. In this way, by computing the number of vehicles detected in a certain location or time, future work on traffic analysis, traffic jam prediction or any other typical smart city measurements can be developed.

Bibliografía

[1] Abdullah Asim Yilmaz, Mehmet Serdar Guzel, Iman Askerbeyli, and Erkan Bostanci. A vehicle detection approach using deep learning methodologies. *arXiv preprint arXiv:1804.00429*, 2018.

[2] Tao Wang, Xuming He, Songzhi Su, and Yin Guan. Efficient scene layout aware object detection for traffic surveillance. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 53–60, 2017.

[3] Mio-tcd: A new benchmark dataset for vehicle classification and localization. http://podoce.dinf.usherbrooke.ca/challenge/dataset/. Accessed: 2020-05-22.

[4] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[5] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[6] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[7] Jianbo Shi et al. Good features to track. In *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pages 593–600. IEEE, 1994.

[8] Christopher G Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[9] James J Gibson. The perception of the visual world. 1950.

[10] Steven S. Beauchemin and John L. Barron. The computation of optical flow. *ACM computing surveys (CSUR)*, 27(3):433–466, 1995.

[11] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. 1981.

[12] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Scandinavian conference on Image analysis*, pages 363–370. Springer, 2003.

[13] Satoshi Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer vision, graphics, and image processing*, 30(1):32–46, 1985.

[14] Emiliano Aldabas-Rubira. Introducción al reconocimiento de patrones mediante redes neuronales. *IX Jornades de Conferències d'Enginyeria Electrònica del Campus de Terrassa, Terrassa, España, del 9 al 16 de Diciembre del 2002*, 2002.

[15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[16] JP Lewis. Creation by refinement: A creativity paradigm for gradient descent learning networks. In *International Conf. on Neural Networks*, pages 229–233, 1988.

[17] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[18] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107:3–11, 2018.

[19] Diganta Misra. Mish: A self regularized non-monotonic neural activation function. *arXiv preprint arXiv:1908.08681*, 2019.

[20] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[22] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[25] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and<0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[26] Large scale visual recognition challenge (ilsvrc). http://www.image-net.org/challenges/LSVRC/. Accessed: 2020-05-22.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[28] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[29] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

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

[31] Wrapper package for opencv python bindings. https://pypi.org/project/opencv-python/. Accessed: 2020-05-26.

[32] Open source computer vision library. https://opencv.org/. Accessed: 2020-05-26.

[33] cv2-tools library source code. https://github.com/fernaper/cv2-tools. Accessed: 2020-05-26.

[34] Tensorflow. https://www.tensorflow.org/. Accessed: 2020-05-26.

[35] Git. https://git-scm.com/. Accessed: 2020-05-26.

[36] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[37] Tensor flow extended (tfx). https://www.tensorflow.org/tfx. Accessed: 2020-06-09.

[38] Tensorboard. https://www.tensorflow.org/tensorboard. Accessed: 2020-06-10.

[39] Darknet. https://pjreddie.com/darknet/. Accessed: 2020-06-01.

[40] Zhiming Luo, Frederic Branchaud-Charron, Carl Lemaire, Janusz Konrad, Shaozi Li, Akshaya Mishra, Andrew Achkar, Justin Eichel, and Pierre-Marc Jodoin. Mio-tcd: A new benchmark dataset for vehicle classification and localization. *IEEE Transactions on Image Processing*, 27(10):5129–5141, 2018.

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

[42] Berthold KP Horn and Brian G Schunck. Determining optical flow. In *Techniques and Applications of Image Understanding*, volume 281, pages 319–331. International Society for Optics and Photonics, 1981.

Apéndice A

Manual de instalación

En este anexo se recogen los pasos de instalación del proyecto en un sistema Linux con una distribución basada en Debian. Se ha probado concretamente en Ubuntu 18.04.

Requisitos del sistema:

- Tarjeta gráfica GPU compatible con CUDA con al menos 3 GB de memoria RAM.
- Espacio en disco de al menos 6 GB para poder descargar el conjunto de datos de entrenamiento y los modelos de red neuronal pre entrenados (una vez instalado ocupará aproximadamente 3 GB, pero durante el proceso de instalación requiere el doble).

Pasos de instalación:

- Instalar CUDA y cuDNN siguiendo las instrucciones de Nvidia.
- Instalar Tensorflow 2.
- Instalar OpenCV para Python y C++. Preferiblemente con los módulos de Nvidia GPU, CUDA y cuDNN.
- Descargar el código del proyecto, ejecutando:
  `git clone https://github.com/fernaper/tfm-artificial-vision`

- Entrar a la carpeta del proyecto y dar permisos de ejecución a los ficheros con la extensión *.sh* de la siguiente forma:

  `chmod +x *.sh`

- Ejecutar el instalador:

  `./installer.sh`

  El instalador seguirá los siguientes pasos:

  1. Crear todos los directorios necesarios dentro del repositorio.
  2. Descargar un archivo comprimido con los modelos pre entrenados y el conjunto de datos utilizado.
  3. Extraer el archivo comprimido y borrarlo.
  4. Añadir enlaces simbólicos a los archivos descargados.
  5. Instalar el proyecto como una librería de Python.
  6. Mostrar información de ejecución y uso del proyecto.

Además, la mayoría de los ficheros con extensión *.py* se pueden ejecutar con el parámetro `-h` para ver todas las opciones que ofrecen.