ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

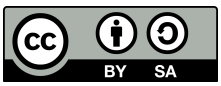
Análisis de Aprendizaje Profundo con la plataforma Caffe

Autor: Nuria Oyaga de Frutos

Tutor: Inmaculada Mora Jiménez

Cotutor: José María Cañas Plaza

Curso académico 2016/2017



"The way to get started is to quit talking and begin doing"

-Walt Disney-

A mi rosa mas bonita,
la más bella del rosal,
gracias por enseñarme,
lo que es vivir y luchar.

Agradecimientos

En primer lugar quiero dar las gracias a mis tutores, Inmaculada y José María, por haberme introducido en este mundo e inculcarme la pasión que ellos mismos tienen, además de por su guía y apoyo durante todos estos meses de desarrollo del trabajo.

Por otro lado, quiero agradecer a mi compañero de proyecto David Pascual, por ayudarme en momentos en los que las cosas no salen y facilitarme toda su ayuda para conseguir terminar el trabajo, además de por su infinita paciencia en momentos de estrés.

Por supuesto, gracias a todos mis compañeros de carrera que durante esta etapa me han acompañado, especialmente a los que puedo considerar amigos: Vanessa, Miguel Ángel, Isa y Mireya, por haber compartido conmigo el estrés y la alegría de superar poco a poco todos los cursos. Además, gracias al resto de mis amigos, los que estaban antes de empezar esta étapa: Nazaret, Sara, Arancha, Iván, Sarai, José Javier, Silvia, Raquel, Bea y Katia, y los que empezaron la etapa conmigo pero fueron por otros caminos: Jorge, Andrés y Julia, porque sin ellos y sus momentos de desconexión no habría llegado hasta el final.

Gracias a toda mi familia por apoyarme y animarme incondicionalmente durante todo este tiempo, y por sentirse siempre orgullosos de mí, proporcionándome toda la fuerza necesaria para continuar. Especialmente quiero mencionar a mis padres, Iñigo y Pilar, y a mis hermanas, Elena y Arancha, que son los que más han sufrido mis malos ratos y han celebrado todos mis exitos, estando día tras día a mi lado.

Por último quiero dar las gracias a mi pareja, Gonzalo, por su ayuda prestada, por soportar mis malos ratos, por compartir mis alegrías, y por hacerme sentir grande con todas mis metas conseguidas.

¡Muchísimas gracias a todos!

Resumen

En los últimos años, la investigación para conseguir que las máquinas perciban escenas y estímulos visuales con la robustez y rapidez que lo hacen los humanos, ha sido altamente desarrollada. En este aspecto, el campo de la Inteligencia Artificial, y en concreto nuevos esquemas de aprendizaje máquina como el aprendizaje profundo (*Deep Learning*) con redes neuronales convolucionales, han experimentado un claro avance, permitiendo por ejemplo la identificación robusta de personas o la detección fiable de vehículos. A diferencia de otros, este aprendizaje no requiere de una etapa externa de extracción de características, pues la extracción se realiza en la propia red. Este trabajo analiza las redes neuronales convolucionales mediante el uso de la plataforma Caffe, que permite el entrenamiento de estas redes y facilita la implementación en aplicaciones propias.

En visión artificial, las redes neuronales pueden ser utilizadas para la resolución de dos problemas. Por un lado, pueden clasificar el contenido de una imagen en una categoría de entre un conjunto de clases indicadas durante el aprendizaje. Por otro lado, pueden detectar estímulos dentro de la imagen, obteniendo como salida un conjunto de cajas delimitadoras que indiquen la presencia de determinados estímulos.

En este trabajo se ha construido un componente en Python que clasifica números manuscritos utilizando una red neuronal de Caffe. También se ha estudiado el efecto que entrenar la red con diferentes bases de datos tiene sobre las prestaciones del clasificador. Se ha utilizado la base de datos MNIST, compuesta por imágenes de dígitos manuscritos. A esta base de datos se le ha aplicado un preprocesamiento para lograr una red más robusta. En concreto, se han considerado transformaciones de escalado, rotación, traslación y contaminación con ruido aditivo. Posteriormente, se ha aplicado un filtro de bordes de Sobel para diseñar una red que premita resolver el problema independientemente de la procedencia de las imágenes (base de datos MNIST, imágenes captadas con una cámara). Adicionalmente, tambien se ha explorado la detección de estímulos interesantes con redes neuronales de Caffe.

Índice general

Índice de figuras

Índice de tablas

Acrónimos

BIDMC Beth Israel Deaconess Medical Center.

CNN Redes Neuronales Convolucionales.

COCO Common Objects in Context.

IA Inteligencia Artificial.

ISBI Simposio Internacional sobre Imágenes Biomédicas.

JSON JavaScript Object Notation.

lmdb Lightning Memory-Mapped Database.

MNIST Modified National Institute of Standards and Technology.

NIST National Institute of Standards and Technology.

ReLU Rectified Linear Unit.

RLE Run-Length Encoding.

RNA Red Neuronal Artificial.

SSD Single Shot MultiBox Detector.

VA Visión Artificial.

VOC Visual Objects Classes.

Capítulo 1

Introducción y objetivos

El objetivo general de este trabajo se sitúa en entender el funcionamiento del aprendizaje profundo con redes neuronales, utilizando para ello una de las plataformas existentes para el desarrollo de las mismas, Caffe.

En este capítulo se situará el trabajo en el marco existente en la actualidad, explicando de manera genérica en qué consiste el aprendizaje profundo, el por qué del uso de una determinada plataforma y los problemas que es posible abordar con esta técnica. Además, se expondrán los objetivos concretos de este proyecto, la metodología empleada para alcanzarlos y un pequeño resumen de cómo se ha estructurado el trabajo.

1.1. Contexto y motivación

Desde que los primeros ordenadores fueron programados, el ser humano se ha planteado la posibilidad de conseguir que estas máquinas adquieran inteligencia, logrando que realicen tareas propias de las personas. Ejemplos de estas tareas son automatizar el trabajo de rutina, entender el habla o las imágenes, hacer diagnósticos en medicina y apoyar la investigación científica básica. Hoy en día, la Inteligencia Artificial (IA) [2], como se denomina al campo que desarrolla estas tareas, cada vez adquiere más presencia, con un alto potencial por sus muchas aplicaciones prácticas y temas de investigación activos.

En los orígenes de la IA se abordaron y resolvieron problemas que no son de resolución inmediata para los seres humanos, pero relativamente sencillos para los ordenadores, problemas que pueden describirse mediante una lista de reglas formales. Una meta actual,

aún más ambiciosa, consiste en resolver tareas que son fáciles de realizar para las personas pero difíciles de describir formalmente.Se trata de problemas que son resueltos de manera rápida e intuitiva por el ser humano como, por ejemplo, el reconocimiento visual de las personas.

La IA abarca varios campos y en este trabajo el foco estará puesto sobre la Visión Artificial (VA). La VA trata de analizar y procesar imágenes del mundo real de tal manera que un ordenador pueda establecer conclusiones sobre las mismas. La idea básica es tratar de trasladar a una máquina la forma en que el ser usa sus ojos y su cerebro para comprender el mundo que le rodea, permitiendo a las mismas tomar decisiones y actuar según la situación. Para lograr el aprendizaje de las máquinas es necesario aplicar conocimientos de distintos campos como la geometría, la estadística, la física y otras disciplinas.

Para conseguir materializar el concepto de la VA se hace uso de las Redes Neuronales Artificiales (RNA). Estas redes están compuestas por capas de neuronas interconectadas que tratan de imitar el comportamiento de las mismas en el cerebro del ser humano. La neurona biológica, consta de un cuerpo celular (soma) del que surge un denso árbol de ramificaciones (dendritas) y una fibra tubular (axón). Esta estructura se asemeja a un procesador de información simple formado por un canal de entrada, similar a las dentritas, un procesador, cuya función se asemeja a la del soma, y un canal de salida, equiparable al axón [1]. Además, entre las neuronas biológicas se establecen una serie de conexiones unidireccionales, conocidas como sinapsis, que son emuladas en el sistema artificial por el conjunto de pesos. En la Figura 1.1 se muestra la similitud entre ambos tipos de neuronas. Una vez entendida la unidad de proceso, se puede definir una RNA como la conexión de este elemento de diferentes formas, dando lugar a diferentes tipos de redes. Para este trabajo tiene interés la red multicapa, que se compone de dos o más capas de neuronas que están conectadas entre ellas, realizando un procesado no lineal de los datos que se presentan a la entrada.

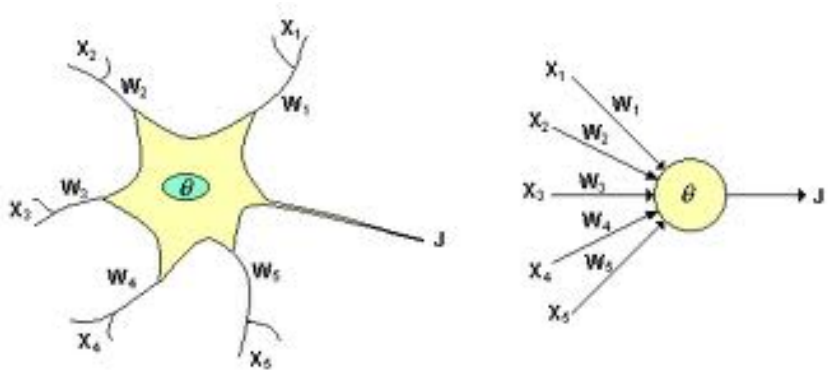


Figura 1.1: Comparación de neurona artificial (derecha) y biológica (izquierda). Imagen obtenida de [1].

Tras definir el contexto general, este trabajo se centra en el Aprendizaje Profundo, situado dentro del marco de la IA, que permite obtener el conocimiento entrenando a las máquinas con ejemplos, sin necesidad de realizar una extracción de características previa, sino que es la propia red que se obtiene la encargada de ello. En la Figura 1.2, se sitúa esta solución en el marco de la IA y sus diferentes divisiones.

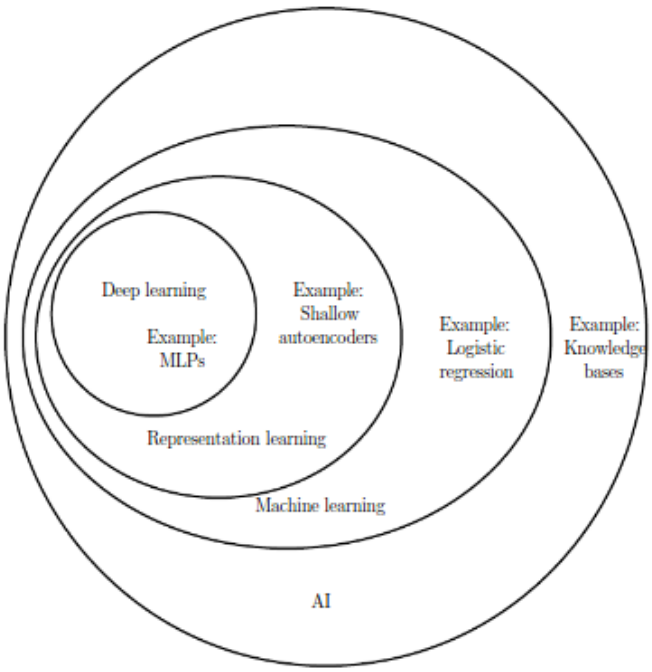


Figura 1.2: Diagrama de Venn que muestra el marco de la IA. Imagen obtenida de [2].

En términos generales, la IA contiene al Aprendizaje Máquina (*Machine Learning*) definido como la capacidad de las computadoras para adquirir sus propios conocimientos, extrayendo patrones de datos sin procesar. A su vez, en el interior de la misma se sitúa el Aprendizaje de la Representación (*Representation Learning*), que utiliza el aprendizaje máquina para descubrir, no sólo el mapeo de la representación a la salida, sino también la representación misma. Finalmente, en este último bloque estaría situado el Aprendizaje Profundo (*Deep Learning*), que permite a la computadora construir conceptos complejos a partir de conceptos más sencillos.

Dentro del Aprendizaje Profundo es posible aplicar diversas técnicas. La técnica más común y la que será empleada en este trabajo son las Redes Neuronales Convolucionales (CNN), que pretenden simular el funcionamiento del cerebro humano para establecer conclusiones sobre los datos introducidos a la misma utilizando filtros convolucionales. La propiedad más importante del aprendizaje con redes neuronales es la capacidad de generalización, que será estimada gracias a una base de datos específica, denominada de test. Se trata de la capacidad que tiene una red de identificar una muestra que no era conocida para ella.

Las CNN tienen una estructura especial, mostrada en la Figura 1.3. Está compuesta, generalmente, por una capa convolucional, que implementa una operación de convolución, y una capa de submuestreo o agrupación, que genera características invariantes calculando estadísticas de las activaciones de convolución a partir de un pequeño campo receptivo. Cada neurona en una capa oculta se conectará a un pequeño campo de la capa anterior, denominado campo receptivo local. En las CNN, las neuronas están organizadas en múltiples capas ocultas paralelas, denominadas mapas de características, de tal manera que cada neurona en un mapa de características está conectada a un campo receptivo local. Para cada mapa de características, todas las neuronas comparten el mismo parámetro de peso que se conoce como filtro o *kernel* [3].

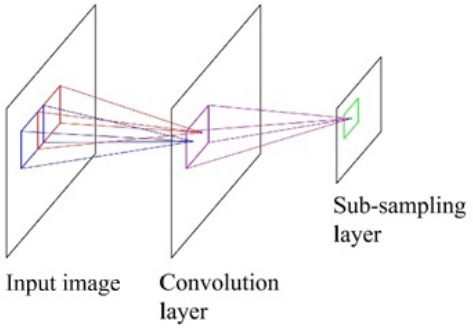


Figura 1.3: Estructura de CNN. Imagen obtenida de [3].

En la actualidad existen múltiples disciplinas que tratan de incluir esta tecnología en su ámbito de trabajo para obtener resultados de forma más rápida y precisa. Una de las disciplinas que mas esfuerzos pone en la investigación de este campo es la medicina, cuyo principal objetivo es lograr diagnosticar y tratar, e incluso prevenir enfermedades como el cáncer. Un ejemplo de ello fue llevado a cabo por el *Beth Israel Deaconess Medical Center (BIDMC)* y la Escuela de Medicina de Harvard, que consiguió el premio máximo en dos categorías del Simposio Internacional sobre Imágenes Biomédicas (ISBI) para la detección del cáncer de mama. El equipo, utilizando la plataforma Caffe, comenzó entrenando su máquina con cientos de diapositivas marcadas para indicar qué partes tienen células cancerosas y cuáles normales, identificando qué tipo de diapositivas eran más complicadas y alimentando el sistema con muestras más difíciles. Con este método, se obtuvo un acierto 92 por ciento y, aunque todavía no compite con los patólogos humanos, cuya precisión es el 96 por ciento, se obtiene un gran avance en este campo [4].

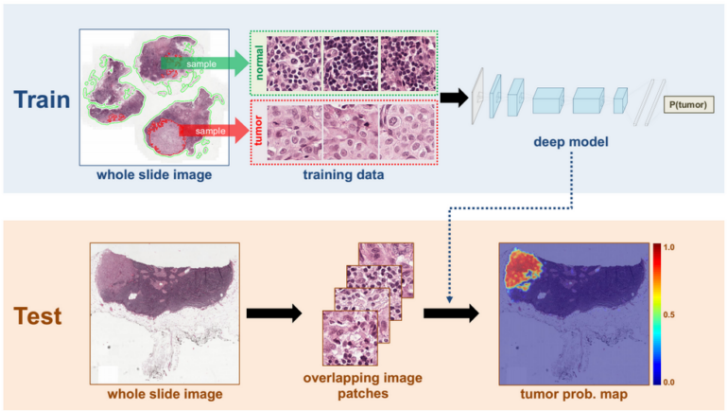


Figura 1.4: Marco utilizado para la detección del cáncer de mama. Imagen obtenida de [4].

Un segundo ejemplo que está a la orden del día es la aplicación en la conducción autónoma. La idea de que un coche pueda manejarse en la carretera por sí mismo, sin necesidad de la existencia de un conductor que realice el trabajo, es tan llamativa que existen múltiples organizaciones que tratan de abarcar este problema. Existen tres paradigmas, mostrados en la Figura 1.5, que permiten abarcar el problema de la conducción autónoma: (1) el enfoque de percepción mediada, que analiza una escena entera para tomar una decisión de conducción; (2) el enfoque de reflejo de comportamiento, que directamente mapea una imagen de entrada a una acción de conducción por un regresor; y (3) el enfoque basado en la percepción directa para estimar la posibilidad para conducir. Este último fue desarrollado por la Universidad de Princeton se propone mapear una imagen de entrada a un pequeño número de indicadores de percepción clave, que se relacionan directamente con la disponibilidad de un estado de carretera o tráfico para la conducción. Esta representación proporciona un conjunto de descripciones compactas pero completas de la escena, permitiendo a un controlador conducir de forma autónoma.

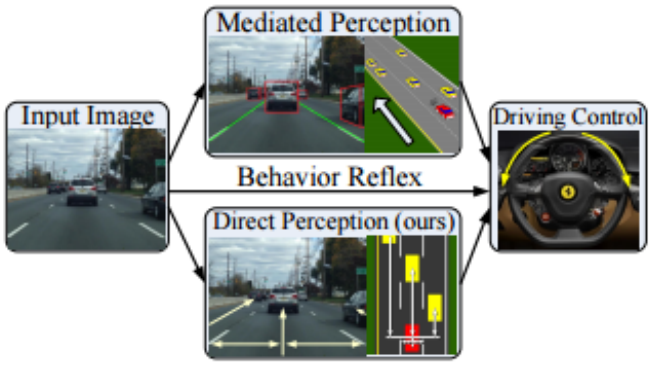


Figura 1.5: Paradigmas de la conducción autónoma. Imagen obtenida de [5].

Otro ejemplo de aplicación es la traducción automática que, a pesar de haber existido durante mucho tiempo, el aprendizaje profundo consigue los mejores resultados en la traducción automática de texto e imágenes. Gracias a las CNN es posible identificar las imágenes que tienen letras y dónde se situan éstas en la escena. Una vez identificadas, pueden convertirse en texto, traducirlo y recrear la imagen con esa traducción. Esta técnica es conocida como traducción visual instantánea [6].

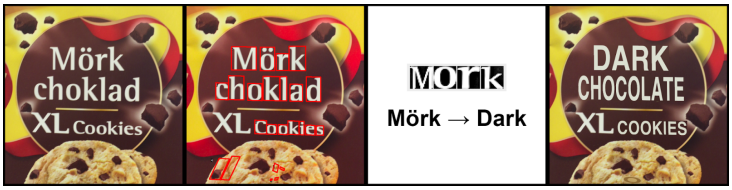


Figura 1.6: Ejemplo de traducción visual instantánea. Imagen obtenida de [6].

Un último ejemplo de aplicación de esta tecnología se basa en adición automática de sonidos a películas mudas, donde el sistema debe sintetizar sonidos para que coincidan con un vídeo silencioso. Para esta tarea el sistema se entrena utilizando 1000 ejemplos de vídeo con el sonido de un tambor golpeando diferentes superficies y creando diferentes sonidos. El modelo de aprendizaje profundo asocia los fotogramas de vídeo con una base de datos de sonidos pre-grabados para seleccionar el sonido que mejor se adapte a lo que está sucediendo en la escena y lo reproduzca [7].

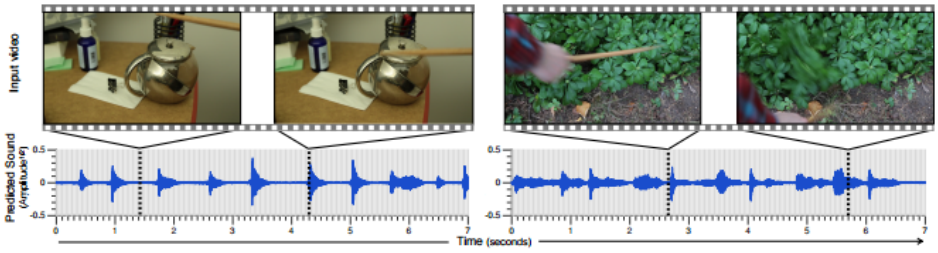


Figura 1.7: Ejemplo adición de sonidos a películas mudas. Imagen obtenida de [7].

Además de las aplicaciones explicadas anteriormente existen muchísimas otras que surgen de las propias inquietudes del ser humano: reconocimiento facial, videovigilancia, identificación de posiciones... Sin embargo, ninguna de ellas ha sido desarrollada por completo, dejando una gran vía de investigación sobre el tema.

Este trabajo se centrará en la clasificación y detección de imágenes estáticas o en movimiento. La diferencia entre ambas aplicaciones es que la detección permite identificar en una única imagen diferentes objetos, independientemente del tamaño y la posición en la que se encuentren, mientras que la clasificación identifica una imagen entrante a la red como perteneciente a una clase determinada.

Por último, para implementar lo explicado anteriormente existen múltiples plataformas que facilitan el entrenamiento y la implementación de estas redes. TensorFlow, Keras, Theano, Caffe, Lassagne o Torch son algunos de los ejemplos más conocidos de estas plataformas. En este trabajo se utilizará la plataforma Caffe, una de las más veteranas. La elección de esta plataforma radica en el gran número de modelos pre-entrenados que proporciona la misma para poder implementar en las aplicaciones ahorrando tiempo al nuevo desarrollador. Además está centrado en la VA, lugar hacia el que se enfoca este trabajo, y, en caso de querer entrenar una nuevo red, resulta bastante rápido.

1.2. Objetivos

Una vez contextualizado en trabajo, el objetivo general es claro: se pretende ahondar en el mundo de la IA, en concreto en técnicas de Aprendizaje Profundo y CNN, para obtener resultados que puedan resultar interesantes y desarrollar una aplicación que permita implementar estas técnicas con la mayor exactitud posible. Este objetivo general ha sido articulado en varios subobjetivos concretos:

- **Estudio de la plataforma Caffe.** Se pretende estudiar la plataforma escogida para un correcto uso de la misma y la obtención de las redes que sean necesarias.
- **Desarrollo de componente que permita la clasificación de dígitos**, integrando una red entrenada con Caffe en el mismo.
- **Desarrollo de un banco de pruebas**, que permita agilizar la obtención de parámetros de evaluación de las distintas redes entrenadas.
- **Estudio y mejora de redes neuronales para la clasificación de dígitos.** Se realizarán diversas pruebas para tratar de alcanzar la red más robusta posible y utilizarla en el componente desarrollado.
- **Primera aproximación a la detección de Caffe.** Se estudiará la detección Single Shot MultiBox Detector (SSD) con Caffe para estímulos interesantes para aplicaciones futuras como personas, coches, bicicletas o señales de tráfico.

En la Figura 1.8 se desglosa, por semanas, el tiempo que ha llevado cada una de las tareas para la consecución de los objetivos.

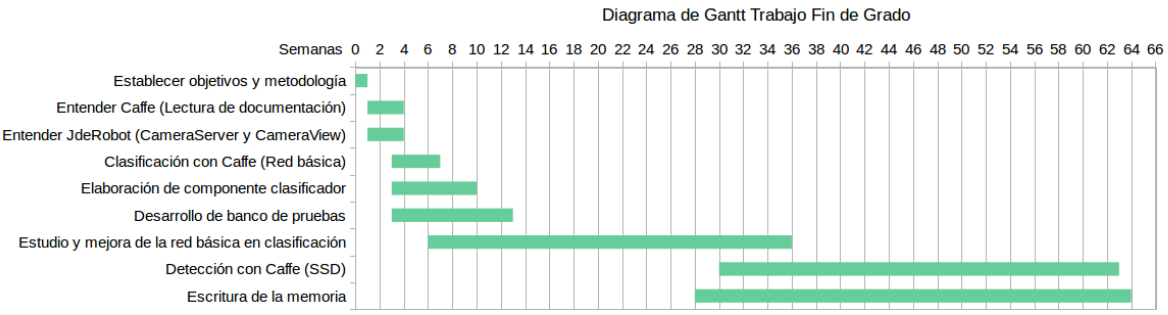


Figura 1.8: Diagrama de Gantt del trabajo realizado.

1.3. Metodología

Para la consecución de los objetivos marcados se han utilizado varias herramientas que han permitido el seguimiento del trabajo por todos los miembros del grupo, permitiendo a todos ellos estar al tanto del trabajo realizado y aportar los comentarios o correcciones que se consideren necesarias.

Como herramienta principal, se establecieron reuniones semanales con todos los miembros del grupo, José María Cañas, Inmaculada Mora y David Pascual, que permitieron poner en común y discutir los resultados obtenidos y acordar las vías a seguir durante la siguiente semana.

Como herramientas complementarias, se utilizó una bitácora y un repositorio de Git Hub, que permitieron agilizar las reuniones al tener acceso de antemano al trabajo realizado. La bitácora se desarrolló empleando la *Wiki* de JdeRobot[1], redactando cada semana el trabajo realizado y los resultados obtenidos. El repositorio de Git Hub[2] permitió a todos los miembros del grupo acceder al código desarrollado, probarlo y establecer correcciones sobre el mismo.

El enfoque de desarrollo que se ha llevado a cabo es similiar al enfoque en espiral. Se establecen cuatro fases principales que son recorridas en forma de caracola. En primer

[1]http://jderobot.org/Noyaga-tfg
[2]https://github.com/RoboticsURJC-students/2016-tfg-nuria-oyaga

lugar se marcaron los objetivos a alcanzar, seguido de una fase de análisis de riesgo para evaluar qué problemas era posible encontrarse al empezar el desarrollo. Tras evaluar las amenazas se procedió al propio desarrollo del trabajo y una última fase de evaluación del resultado obtenido. No existe un número fijo de iteraciones, produciéndose tantas como sean necesarias para obtener un resultado adecuado, incrementando en cada iteración el alcance del trabajo.

1.4. Estructura de la memoria

Para mostrar el trabajo realizado y los objetivos conseguidos se elabora la memoria con la estructura mostrada a continuación.

En el **Capítulo 1, Introducción y objetivos,** se sitúa el trabajo en el marco actual de la tecnología, la IA y la sociedad en general. A continuación se establecen las metas que se pretenden alcanzar con el mismo.

El **Capítulo 2, Infraestructura,** describe todo el *software* utilizado en el proyecto, incluyendo la principal plataforma, Caffe, y los conjuntos de datos empleados para la obtención y evaluación de las redes neuronales. Además, se explican los diferentes parámetros que serán empleados para evaluar las prestaciones de las redes creadas.

En el **Capítulo 3, Clasificación con Aprendizaje Profundo,** se expone todo el trabajo realizado para abordar el problema de la clasificación de dígitos con CNN. En concreto, se describe el funcionamiento de Caffe en la clasificación, el componente creado, y las pruebas realizadas para conseguir una red robusta, con sus correspondientes resultados.

En el **Capítulo 4, Detección con Aprendizaje Profundo,** se proporciona la información necesaria para realizar la detección con Caffe. Se muestran algunos resultados de pruebas preliminares, realizadas para una mejor comprensión del proceso de detección.

Por útlimo, el **Capítulo 5, Conclusiones y líneas futuras,** resume las conclusiones obtenidas en el trabajo y establece un posible plan de actuación futuro para continuar con la investigación en el tema que se aborda en el trabajo.

Capítulo 2

Infraestructura

En este capítulo se expondrán los principales componentes software utilizados, centrados principalmente en la conexión con la cámara y el desarrollo, entrenamiento y test de la red neuronal. Además, se expone una descripción de las bases de datos de las que se partirá para realizar las distintas pruebas sobre la red neuronal. Estas bases de datos serán luego modificadas y adaptadas para el problema concreto que se plantee, permitiendo obtener diversas conclusiones acerca de la relación entre el entrenamiento y el comportamiento de la propia red y, así, emplear la más adecuada. Por último, serán expuestos los parámetros empleados para evaluar el impacto del aprendizaje en las redes neuronales y que permitirán escoger la red más adecuada para el problema.

2.1. Software

2.1.1. Entorno JdeRobot

JdeRobot[1] es una plataforma de software libre que facilita la tarea de los desarrolladores en el campo de la robótica, visión por computador y otras áreas con sensores y actuadores.

Está escrito en su mayoría en el lenguaje C ++ y proporciona un entorno de programación basado en componentes distribuidos, de tal manera que una aplicación está formada por una colección de varios componentes asincronos y concurrentes. Permite la ejecución de los distintos componentes en diferentes equipos, estableciendo una conexión

[1] `http://jderobot.org`

entre ellos mediante el *middleware* de comunicaciones ICE. Además, ofrece gran flexibilidad a la hora de desarrollar las aplicaciones, ya que estos componentes pueden escribirse en C ++, Python, Java, etc. y todos ellos interactúan a través de interfaces ICE explícitas.

Esta plataforma incluye una gran variedad de herramientas y librerías para la programación de robots, y de una amplia gama de componentes previamente desarrollados para realizar tareas comunes en este ámbito. Este proyecto únicamente se centra en la utilización de uno de sus componentes para facilitar la obtención de las imágenes, *CameraServer*, en la versión 5.5 del entorno.

Camera Server

Es un componente que permite servir a las aplicaciones las imágenes de un número determinado de cámaras, ya sean reales o simuladas, o de un archivo de vídeo. Internamente utiliza *gstreamer* para el manejo y el procesamiento de las diferentes fuentes de vídeo.

Para su uso es necesario editar su fichero de configuración, adaptándolo a las necesidades concretas que plantee la máquina. Dentro de este fichero se permite especificar los siguientes campos:

- Configuración de la red, donde se indica la dirección del servidor que va a recibir la petición.
- Número de cámaras que se servirán.
- Configuración de las cámaras. Se podrán modificar los siguientes campos para cada cámara:
  - Nombre y breve descripción
  - URI: string que define la fuente de vídeo
  - Numerador y denominador del *frame rate*
  - Altura y anchura de la imagen
  - Formato de la imagen
  - Invertir o no la imagen

2.1.2. Entorno Caffe

Caffe[2] [14] es una plataforma de aprendizaje profundo que permite el desarrollo, entrenamiento y evaluación de redes neuronales. Incluye, además, modelos y ejemplos previamente trabajados para un mejor entendimiento de las redes neuronales. Es una plataforma de software libre, escrita en C ++, que utiliza la librería CUDA para el aprendizaje profundo y permite interfaces escritas en Python o Matlab.

Esta plataforma es interesante por múltiples factores. Además de incluir diferentes ejemplos y modelos ya entrenados, lo que ofrece agilidad a la hora de empezar a entender el funcionamiento de las redes neuronales, es destacable la velocidad que ésta ofrece para el entrenamiento de las redes y su posterior evaluación, ya que está prevista con varios indicadores que permiten evaluar la propia red y compararla con otras.

Su base se encuentra en las redes neuronales convolucionales explicadas en el Capítulo 1, utilizando un entrenamiento por lotes. En concreto, su estructura y funcionamiento básico queda explicado en la Figura 2.1.

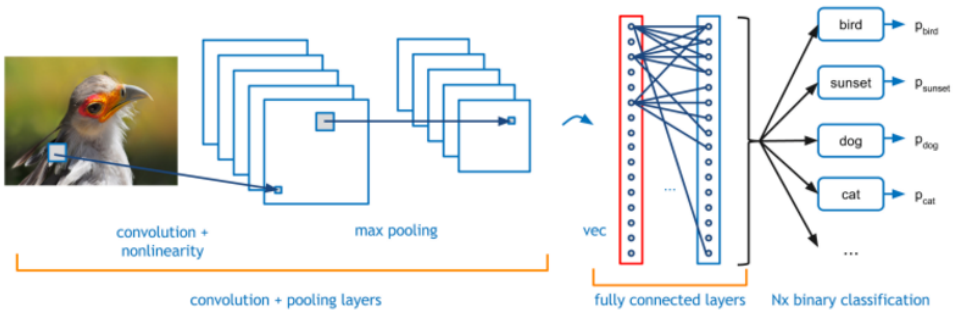


Figura 2.1: Estructura y funcionamiento básico de red en Caffe. Figura obtenida de [8]

La plataforma utiliza una serie de capas (*layers*), que, según su configuración y la distinta conexión entre ellas, permite la creación de diferentes redes neuronales. Estas capas se dividen en varios grupos, en función del tipo de entrada, el tipo de salida o la función que realiza cada una de ellas. Este trabajo no utiliza todas las capas existentes en

[2] `http://caffe.berkeleyvision.org/`

la plataforma. A continuación se explicarán cada una de las capas empleadas, clasificadas según al grupo que pertenecen.

Data Layers

Su uso se centra en la introducción de datos a la red neuronal, y estarán situadas siempre en la parte inferior de la misma. Estos datos provienen de diferentes vías que pueden ser: bases de datos eficientes como LMDB, utilizada en este trabajo, directamente desde la memoria o desde archivos en disco en HDF5 o formatos de imagen comunes.

Dentro de esta capa es posible, además de especificar la ruta de los datos y el tamaño del lote (*batch*), indicar la fase en la que se utilizarán los datos, entrenamiento o evaluación, así como algunos parámetros de transformación para el preprocesamiento de la imagen. En concreto, en este trabajo se utilizarán datos de entrada para ambas fases y un factor de escala para establecer el rango de las imágenes en [0,1].

Vision Layers

Este tipo de capas, típicamente toman una imagen de entrada y producen otra de salida, de forma que, aplicando una operación particular a alguna región de la entrada, se obtiene la región correspondiente de la salida. Caffe dispone de varias capas de este estilo, a continuación se comentan las dos utilizadas en el trabajo.

Convolution Layer

Realiza la convolución de la imagen de entrada con un conjunto de filtros de aprendizaje, cada uno produciendo un mapa de características en la imagen de salida. Se deben especificar datos como el número de salidas, el tamaño del filtro, el desplazamiento entre cada paso del filtro, y la inicialización y relleno de los pesos y sesgo (*bias*).

Pooling Layer

Combina la imagen de entrada aplicando una operación dentro de las regiones

definidas por el filtro, siendo su finalidad la reducción del muestreo. Se especifican parámetros como el tipo de *pooling* a realizar, máximo, promedio o estocástico, el tamaño del filtro o el desplazamiento entre cada paso del filtro.

Common Layers

Inner Product

Calcula un producto escalar con un conjunto de pesos aprendidos, y, de manera opcional, añade sesgos. Trata la entrada como un simple vector y produce una salida en forma de otro, estableciendo la altura y el ancho de cada *blob* en 1. Se establece el número de salidas, y la inicialización y relleno de los pesos y sesgo.

Dropout

Durante el entrenamiento, únicamente, establece una porción aleatoria del conjunto de entrada a 0, ajustando el resto de la magnitud del vector en consecuencia, evitando así el sobreajuste. Se debe indicar el ratio en un valor del 0 a 1, que indicará el porcentaje de muestras que se ignorarán.

Activation / Neuron Layers

En general estas capas son operadores de elementos que toman un *blob* inferior y producen uno superior del mismo tamaño. Existen varias capas con este funcionamiento en la plataforma, en concreto se empleará la *Rectified Linear Unit (ReLU)*.

ReLU

Utiliza la función $y = max(0, x)$, cuya gráfica se define en la Figura 2.2, donde x es la entrada a la neurona.

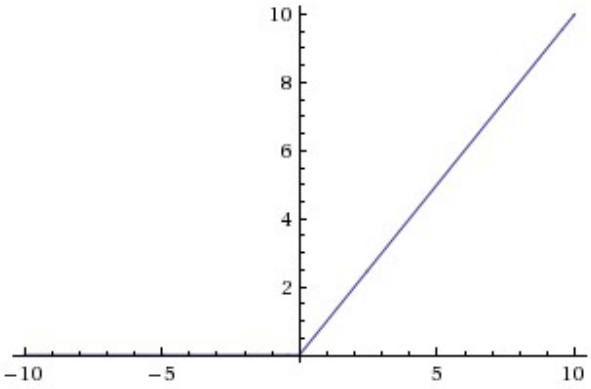


Figura 2.2: Función de activación *ReLU*.

Loss Layers

El cálculo de la pérdida permite el aprendizaje mediante la comparación de la salida con un objetivo y la asignación de un coste para minimizarla. Se calcula mediante el paso hacia adelante. Existen diferentes medidas de las que se destacan dos.

Softmax with Loss

La función *softmax* se utiliza a menudo en la capa final de un clasificador basado en redes neuronales. Se trata de una función que modifica un vector K-dimensional de valores reales arbitrarios a un vector K-dimensional de valores reales en el rango (0, 1] que suman 1.

Esta capa es conceptualmente idéntica a una capa de *softmax*, la cual calcula la función con el mismo nombre, seguida por una capa de pérdida logística multinomial, proporcionando un gradiente numéricamente más estable. Se calcula la pérdida (*loss*) como:

$$E = \frac{-1}{N} \sum_{n=1}^{N} \log(\hat{p}_{n,l_n})$$

Siendo N el número total de muestras, $\hat{p}$ las probabilidades de cada etiqueta para cada muestra y l_n las etiquetas existentes. Se definen las etiquetas existentes como $l_n \in [0, 1, 2, ..., K-1]$, siendo K el total de clases. Adicionalmente, se debe multiplicar todo por -1 ya que se aplica el logaritmo a una probabilidad, oscilante entre 0 y 1, obteniendo un resultado negativo, y el que se desea obtener debe ser positivo.

Accuracy

Esta capa calcula únicamente la tasa de acierto de la red, es decir, el número de aciertos en la clasificación referenciado al número total de muestras analizadas. Se calcula como:

$$\frac{1}{N}\sum_{n=1}^{N}\delta\{\hat{l}_n = l_n\}$$

Donde $\hat{l_n}$ es la etiqueta que la red decide en la clasificación y, al igual que en el caso anterior, N es el número total de muestras y l_n las etiquetas existentes. Por último, la función $\delta\{x\}$ se define como:

$$\delta\{\text{condición}\} = \begin{cases} 1 & \text{si condición} \\ 0 & \text{resto} \end{cases}$$

Por último, además de las capas y parámetros definidos anteriormente, Caffe permite el desarrollo de un solucionador (*solver*) en el que se podrán ajustar parámetros como el número de iteraciones totales que se ejecutarán, el de evaluación que se van a realizar, cada cuantas iteraciones se realizará esa evaluación, o se sacarán redes intermedias.

Para Caffe, el número de iteraciones no se corresponde con el número de veces que la red recorre la base de datos al completo, sino como las veces que se pasa por cada lote al completo. Esto viene dado porque, debido a la amplia dimensión de las bases de datos necesarias para desarrollar el aprendizaje profundo, según se explica en el Capítulo 1, será necesaria una división de la misma en pequeños lotes para que ordenador no se bloquee en el tratamiento de las mismas. De esta manera, se define el número de épocas, es decir, el número de veces que se recorre de manera completa la base de datos, con la siguiente expresión:

$$\text{N.Epocas} = \frac{\text{Tamaño lote de entrenamiento} \times \text{Total iteraciones}}{\text{Muestras entrenamiento}}$$

2.1.3. OpenCV

OpenCV[3](*Open Source Computer Vision Library*) es una biblioteca de software de visión artificial y de aprendizaje automático. Fue construido para proporcionar una infraestructura común para aplicaciones de visión por computadora y para acelerar el uso

[3]http://opencv.org/

de la percepción de la máquina en los productos comerciales. Tiene interfaces C ++, C, Python, Java y MATLAB y soporta Windows, Linux, Android y Mac OS. OpenCV se inclina principalmente hacia aplicaciones de visión en tiempo real y se aprovecha de un conjunto de instrucciones "*Single Instruction, Multiple Data*" (SIMD), MMX y *Streaming SIMD Extensions* (SSE) cuando están disponibles.

La biblioteca cuenta con más de 2500 algoritmos optimizados, que incluye un amplio conjunto de clásicos y avanzados algoritmos de visión por computadora y aprendizaje de máquina. Estos algoritmos pueden ser utilizados para diferentes funciones como, por ejemplo:

- Detectar y reconocer rostros
- Identificar objetos
- Clasificar acciones humanas en videos
- Realizar un seguimiento de movimientos de cámara
- Rastrear objetos en movimiento
- Encontrar imágenes similares de una base de datos

En este trabajo esta librería se utiliza en su version 2.4.9.1 con la interfaz de Python, y será de ayuda en el desarrollo del componente y la aplicación de las diferentes transformaciones a la base de datos.

2.1.4. PyQt

PyQt[4] es un conjunto de enlaces Python v2 y v3 para el *framework* de aplicaciones Qt. se ejecuta en todas las plataformas soportadas por Qt incluyendo Windows, OS X, Linux, iOS y Android. Qt es un conjunto de bibliotecas de C++ y herramientas de desarrollo que incluye abstracciones independientes de la plataforma para interfaces gráficas de usuario, redes, hilos, Unicode, entre otras muchas más. PyQt permite al programador obtener las ventajas de Qt, teniendo todo su poder, y Python, siendo capaz de explotarlo

[4]https://riverbankcomputing.com/software/pyqt/intro

con su simplicidad.

En este trabajo se emplea la versión 4, PyQt4[5], que implementa 440 de las clases de Qt como un conjunto de módulos Python y soporta las plataformas Windows, Linux, UNIX y MacOS/X. Comprende una serie de componentes diferentes divididos en una serie de módulos de extensión Python, instalados en el paquete PyQt4 Python, y una serie de programas de utilidad.

Para el desarrollo de este trabajo se empleará esta herramienta para desarrollar la parte gráfica del componente que permite la visualización de las imágenes y los resultados. Para ello se utilizan dos de los módulos Python que contiene PyQt4:

- ***QtCore***, que contiene las clases principales no GUI, incluyendo el bucle de eventos y el mecanismo de señal y ranura de Qt. También incluye abstracciones independientes de la plataforma para Unicode, subprocesos, archivos asignados, memoria compartida, expresiones regulares y configuración de usuario y aplicación.
- ***QtGui***, que contiene la mayoría de las clases GUI.

2.1.5. DroidCam

DroidCam[6] es una aplicación que permite convertir un dispositivo móvil en una cámara web, estableciendo una conexión mediante WiFi/LAN, modo servidor wifi, o USB. Esta aplicación es muy usada para establecer videoconferencias a través de plataformas como Skype o Google+, entre otras aplicaciones. En este trabajo será usada para obtener el flujo de vídeo desde un dispositivo distinto a la webcam del ordenador, haciendo más sencillo el manejo del mismo.

La aplicación funciona con un componente cliente en el ordenador que instala los controladores de la cámara web y conecta el equipo con el dispositivo Android, que deberá tener instalada la misma aplicación.

[5]http://pyqt.sourceforge.net/Docs/PyQt4/
[6]https://www.dev47apps.com/

Entre sus características principales destacan:

- Incluye sonido e imagen
- Conexión por diferentes medios
- Uso de otras apliaciones con DroidCam en segundo plano
- Cámara IP de vigilancia con acceso MJPEG

2.2. Bases de datos

Para realizar el entrenamiento y la evaluación de las CNN es necesario disponer de unas bases de datos cuyo contenido se corresponda con el estímulo de interés en el problema. A continuación se exponen las tres bases de datos empleadas en el trabajo para obtener las redes en el desarrollo del mismo.

2.2.1. MNIST

La base de datos *Modified National Institute of Standards and Technology (MNIST)* [7] está formada por diferentes imágenes con números escritos a mano y consta de un conjunto de entrenamiento de 60.000 ejemplos y otro de prueba de 10.000 ejemplos. Es una buena base de datos para personas que quieren probar técnicas de aprendizaje y métodos de reconocimiento de patrones en datos del mundo real, mientras que dedican un mínimo esfuerzo a preprocesar y formatear.

Se trata de un subconjunto de una más grande, *National Institute of Standards and Technology (NIST)*, en la que las imágenes originales en blanco y negro fueron normalizadas en el tamaño para encajar en un cuadro de 20x20 píxeles, preservando su relación de aspecto. Las imágenes obtenidas contienen niveles de gris como resultado de la técnica anti-aliasing utilizada por el algoritmo de normalización. Estas imágenes se centraron en una de 28x28 calculando el centro de masa de los píxeles y trasladando la imagen para situar este punto en el centro del campo 28x28.

[7] http://yann.lecun.com/exdb/mnist/

En la Figura 2.3 se puede observar una muestra de cada uno de los dígitos que se almacenan, en este caso, en la base de datos de test, siendo de las mismas características la de entrenamiento.

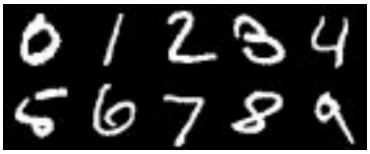


Figura 2.3: Muestras de base de datos MNIST.

Fue construida a partir de la Base de Datos Especial 3 y la Base de Datos Especial 1 del NIST, que contienen imágenes binarias de dígitos manuscritos.NIST originalmente designó SD-3 como su conjunto de entrenamiento y SD-1 como su conjunto de pruebas. Sin embargo, SD-3 es mucho más limpio y más fácil de reconocer que SD-1. Esto es debido a que SD-3 fue recogido entre los empleados de la Oficina del Censo, mientras que el SD-1 fue recogido entre los estudiantes de secundaria. Dado que para una buena extracción de conclusiones es necesario que el resultado sea independiente de la elección del conjunto de entrenamiento y de prueba entre el conjunto completo de muestras, fue necesaria la elaboración de un nuevo conjunto en el que ambas bases de datos estuviesen representadas de manera equitativa. Además, se aseguraron de que los conjuntos de escritores en el de entrenamiento y el de prueba son disjuntos.

2.2.2. COCO

Microsoft *Common Objects in Context (COCO)* [8] es un gran conjunto de datos de imágenes diseñado para la detección de objetos, segmentación y generación de subtítulos [15]. Alguna de las características principales de este conjunto de datos son:

- Múltiples objetos en cada imagen
- Más de 300.000 imágenes
- Más de 2 millones de instancias
- 80 categorías de objetos

[8] http://mscoco.org/

Esta plataforma se ha desarrollado para varios retos, en concreto es de interés el reto de la detección, establecido en 2016. Se utilizan conjuntos de entrenamiento, prueba y validación con sus correspondientes anotaciones. COCO tiene tres tipos de anotaciones: instancias de objeto, puntos clave de objeto y leyendas de imagen, que se almacenan utilizando el formato de archivo *JavaScript Object Notation (JSON)* y comparten estructura de datos establecida en la Figura 2.4.

```
{
  "info"          :info,
  "images"        :[image],
  "annotations"   :[annotation],
  "licenses"      :[license],
}

info{
  "year"          :int,
  "version"       :str,
  "description"   :str,
  "contributor"   :str,
  "url"           :str,
  "date_created"  :datetime,
}

image{
  "id"            :int,
  "width"         :int,
  "height"        :int,
  "file_name"     :str,
  "license"       :int,
  "flickr_url"    :str,
  "coco_url"      :str,
  "date_captured" :datetime,
}

license{
  "id"            :int,
  "name"          :str,
  "url"           :str,
}
```

Figura 2.4: Estructura básica de anotaciones en COCO.

Para la detección son de interés las anotaciones de instancias de objetos, cuya estructura se muestra en la Figura 2.5.Cada anotación de instancia contiene una serie de campos, incluyendo el ID de categoría y la máscara de segmentación del objeto. El formato de segmentación depende de si la instancia representa un único objeto ($iscrowd = 0$), en cuyo caso se utilizan polígonos, o una colección de objetos ($iscrowd = 1$), en cuyo caso se utiliza *Run-Length Encoding (RLE)*. Debe tenerse en cuenta que un único objeto puede requerir múltiples polígonos, y que las anotaciones de la multitud se utilizan para etique-

tar grandes grupos de objetos. Además, se proporciona una caja delimitadora para cada objeto, cuyas coordenadas se miden desde la esquina superior izquierda de la imagen y están indexadas en 0. Finalmente, el campo de categorías almacena el mapeo del ID de categoría a los nombres de categoría y supercategoría.

```
annotation{
  "id"              :int,
  "image_id"        :int,
  "category_id"     :int,
  "segmentation"    :RLE or [polygon],
  "area"            :float,
  "bbox"            :[x,y,width,height],
  "iscrowd"         :0 or 1,
}

categories[{
  "id"              :int,
  "name"            :str,
  "supercategory"   :str,
}]
```

Figura 2.5: Estructura de instancias de objetos en COCO.

Figura 2.6: Muestras de base de datos COCO. Imagen obtenida de [9].

2.2.3. VOC

El objetivo del desafío de *Visual Objects Classes (VOC)* [10] es investigar el desempeño de los métodos de reconocimiento en un amplio espectro de imágenes naturales. Para ello,

se requiere que los conjuntos de datos VOC contengan variabilidad significativa en términos de tamaño del objeto, orientación, pose, iluminación, posición y oclusión. También es importante que los conjuntos de datos no muestren sesgos sistemáticos, por ejemplo, favoreciendo imágenes con objetos centrados o una buena iluminación. Del mismo modo, para garantizar un entrenamiento y una evaluación precisa, es necesario que las anotaciones de imagen sean consistentes, precisas y exhaustivas para las clases especificadas.

Teniendo claros estos conceptos, en 2007, se llevó a cabo una recolección de imágenes, como las mostradas en la Figura 2.7, formando el conjunto de datos *VOC2007* [16]. Este conjunto dispone de dos grandes bases de datos, una de ellas compuesta por un conjunto de validación y otro de entrenamiento, y la otra con un único conjunto de test. Ambas bases de datos contienen alrededor de 5000 imágenes en las que se representan, apróximadamente, 12.000 objetos diferentes, por lo que, en total, este conjunto dispone de unas 10000 imágenes en las que se reperesentan unos 24000 objetos. En el año 2012 se modifica este conjunto, aumentando a 11530 el número de imágenes con representación de 27450 objetos diferentes [17].

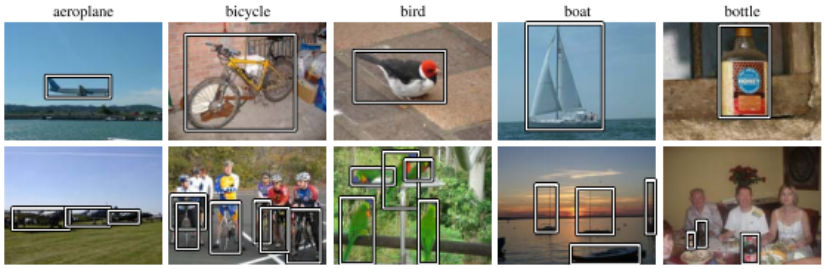


Figura 2.7: Ejemplos de imágenes en VOC. Imagen tomada de [10]

.

Dado que la finalidad de este conjunto de datos es permitir el desarrollo tanto de la clasificación de objetos como la detección de los mismos, será necesario que estas imágenes contengan una serie de anotaciones. Entre otras cosas, estas anotaciones contienen dos atributos importantes para ambas tareas:

- Para la **clasificación**, se debe indicar la clase de objeto que es. Los objetos de este conjunto estan clasificados en 20 clases diferentes. En la Figura 2.8 se puede observar la división que se hace de cada una de las clases y las distintas clases existentes.

- Para la **detección**, será necesario indicar, para cada objeto, la *bounding box*. Se trata de un cuadro delimitador alineado con el eje que rodea la extensión del objeto visible en la imagen, permitiendo identificar el objeto en la imagen.

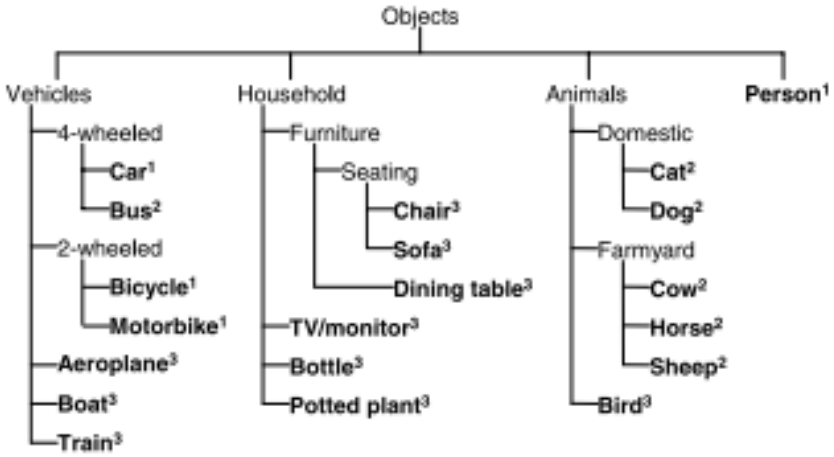


Figura 2.8: Estructura de las clases en Visual Objects Classes (VOC)2007. Imagen tomada de [10]. El superíndice de cada una de las clases indica el año de inclusión en el desafío: 2005^1, 2006^2, 2007^3.

2.3. Evaluación de prestaciones

Existen multitud de parámetros que permiten la evaluación de las redes neuronales, sin embargo, en este proyecto, todas las comparaciones se centrarán en cinco de ellas: *accuracy*, *loss*, matriz de confusión, *precision* y *recall* [18].

Las dos primeras fueron explicadas en la Sección 2.1.2, por lo que no se ahondará más sobre ellas. Las tres restantes serán explicadas más profundamente, y de manera totalmente teórica, a continuación.

2.3.1. Matriz de confusión

Una matriz de confusión resume el desempeño de clasificación de un clasificador con respecto a algunos datos de prueba. Es una matriz bidimensional, indexada en una dimensión por la clase verdadera de un objeto y en la otra por la clase que el clasificador ha estimado. Un caso especial de la matriz de confusión se utiliza a menudo con dos clases,

una designada la clase positiva y la otra a la clase negativa. En este contexto, las cuatro células de la matriz se designan como verdaderos positivos (TP), falsos positivos (FP), negativos verdaderos (TN) y falsos negativos (FN), según se muestra en la Figura 2.9 [19].

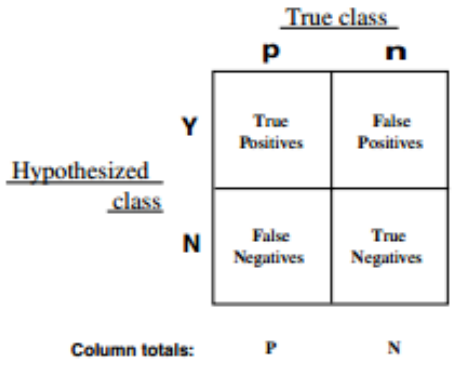


Figura 2.9: Matriz de confusión para clasificación binaria. Imagen tomada de [11]

.

En una clasificación multiclase, esta matriz se aumenta de tal forma que exista una columna para cada clase posible real y una fila para cada clase posible predicha, según se muestra en la Figura 2.10.

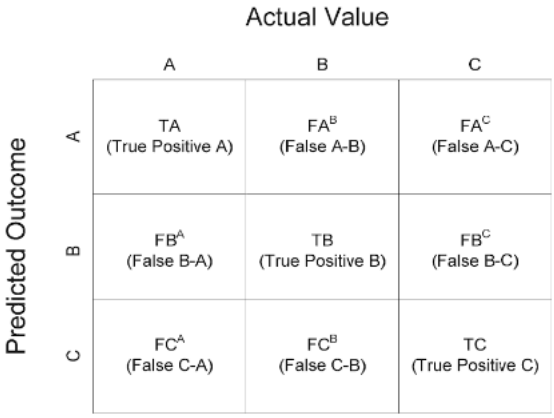


Figura 2.10: Matriz de confusión para clasificación multiclase. Imagen tomada de [12]

.

En este caso no existe el concepto de falso negativo, sino que se obtiene, en cada celda, el número de veces que se predijo la clase correspondiente a la fila de la misma, al producirse la clase correspondiente a la columna en la que se sitúa. De esta manera, la

diagonal, formaría los aciertos que se produjeron en la clasificación, y el resto de celdas las equivocaciones cometidas [12].

2.3.2. *Precision*

El valor de *precision*, en clasificación binaria, se define como una proporción de positivos verdaderos (TP) y el número total de positivos predichos por un modelo [11]. En el caso de clasificación multiclase se deberá de tener en cuenta la existencia de varias clases y se utilizará la siguiente expresión, adaptada de [11]:

$$P = \frac{TP_X}{TP_X + FP_X}$$

Donde:

- TP_X se corresponde el número de verdaderos positivos para la clase X, es decir, el valor de aciertos correspondiente para dicha clase, situado en la diagonal.
- FP_X se corresponde el número de falsos positivos para la clase X, es decir, el número de veces que se predijo dicha clase sin haber sido producida, correspondiente con la suma del resto de celdas en la misma fila.

2.3.3. *Recall*

El valor de *recall*, en clasificación binaria, se define como una proporción de positivos verdaderos (TP) y el número total de positivos que se producen [11]. Al igual que en el caso anterior, al considerar la clasificación multiclase se adapta la expresión de [11] para considerar todas las clases posibles, de esta forma se obtiene la expresión:

$$R = \frac{TP_X}{TP_X + FN_X}$$

Donde TP_X se corresponde con lo explicado en la sección anterior y FN_X se corresponde con los falsos negativos para la clase X, es decir, el número de veces que se predijo erróneamente otra clase habiéndose producido X, correspondiente con la suma del resto de la columna.

Capítulo 3

Clasificación con Aprendizaje Profundo

En este capítulo se expondrá el trabajo realizado para la comprensión del problema de clasificación de imágenes utilizando la plataforma Caffe de aprendizaje profundo. Para ello se ha elaborado un componente en Python que permite la clasificación de dígitos del 0 al 9 en tiempo real integrando una red generada por Caffe. Además, ha sido mejorado gracias a un amplio estudio sobre las variantes posibles aplicadas a las redes entrenadas.

3.1. Clasificador de dígitos

La primera tarea que se abarca en el proyecto es el desarrollo de un componente en Python para la clasificación de dígitos manuscritos entre 0 y 9 en tiempo real, materializando en una aplicación concreta el problema de la clasificación de imágenes con aprendizaje profundo. Para desarrollar esta aplicación es útil, previamente, un entendimiento de una primera red básica, que será la encargada de realizar la clasificación. En esta sección se explica el procedimiento seguido para el entendimiento de la red y el desarrollo del propio componente.

3.1.1. Red básica

La red que se emplea está orientada a la clasificación de números, utilizando en el entrenamiento la base de datos numérica MNIST, explicada en la Sección 2.2.1, y en la que se entra en detalle a continuación.

La base de datos MNIST proporciona dos conjuntos de datos, uno de entrenamiento y otro de test para evaluar la capacidad de generalización, pero, a diferencia de otros conjuntos, no dispone de una de validación para evaluar las prestaciones del modelo durante el entrenamiento. Por ello, el primer paso consiste en dividir la base de datos de entrenamiento en dos, obteniendo un conjunto de validación a partir del que ofrece MNIST para el entrenamiento. Para esta tarea, se desarrolla un script, *create_validation_database.py*, que divide la base de datos de entrenamiento original en dos, el 80 % para entrenamiento y el 20 % restante para validación. En la Tabla 3.1, se muestra un resumen de la estructura final de ambas bases de datos, así como la estructura de la base de datos de test, que no sufre ninguna modificación.

	Set entrenamiento			**Set test**
Dígito	**Total**	**80 %**	**20 %**	**Test**
0	5923	4738	1185	980
1	6742	5393	1349	1135
2	5958	4767	1191	1032
3	6131	4905	1226	1010
4	5842	4674	1168	982
5	5421	4337	1084	892
6	5918	4734	1184	958
7	6265	5012	1253	1028
8	5851	4681	1170	974
9	5949	4759	1190	1009
Total	60000	48000	12000	10000

Tabla 3.1: Estructura de conjuntos de datos.

Se puede comprobar, como era de esperar, que no todos los dígitos tienen la misma presencia, siendo mayor el número de muestras en los dígitos que pueden generar mayor confusión, como el 1 o el 7, y menor en los que son más claros como el 0. Por ello, es muy importante respetar las proporciones existentes en cada dígito al dividir la base de datos para no alterar la naturaleza de la base de datos original. Para mantener esta proporción se

calcula el porcentaje sobre el total de cada dígito y no sobre el total del conjunto de datos.

Tras tener claro los diferentes conjuntos de datos que se emplearán en adelante, se procede al entrenamiento de la red. Para entrenar una red, Caffe proporciona tres archivos que se editan para adaptar la red al problema concreto que se aborde. A continuación, se explicará cada uno de esos archivos, siguiendo el orden que fue necesario hasta conseguir la red completamente entrenada.

3.1.1.1. Definición de la red

Caffe utiliza el archivo *lenet_train_test.prototxt* para la especificación de todos los parámetros que son necesarios en el entrenamiento de la red, es decir, en este documento se definen las imágenes que se emplean, la propia estructura de la red y la forma en la que se analizan las imágenes proporcionadas, todo ello empleando diferentes capas (*layers*).

La primera línea de este documento es utilizada para indicar el nombre que se le quiere dar a la red, según se muestra a continuación.

```
name: "LeNet"
```

En concreto, esta red recibe el nombre de LeNet, un tipo de red que es conocida por un buen funcionamiento en las tareas de clasificación de dígitos y que, por lo general, consta de una capa convolucional seguida por una capa de agrupamiento (*pooling*), repetido dos veces. Tras ellas, se incluyen dos capas totalmente conectadas similares a los perceptrones multicapa convencionales. Con Caffe la estructura habitual de la red LeNet se ve ligeramente modificada, utilizando una función de activación lineal en lugar de sigmoidal.

Tras la definición del nombre se definen dos capas de datos, una de ellas correspondiente a los datos de entrenamiento y la otra correspondiente a los datos que se utilizan para realizar la evaluación durante el entrenamiento, obteniendo datos de *accuracy* y *loss* con el conjunto de validación. A continuación se muestra un ejemplo de cómo se define esta capa de datos, en concreto, en fase de entrenamiento.

```
layer {
    name: "mnist"
    type: "Data"
    top: "data"
    top: "label"
    include {phase: TRAIN}
    transform_param {scale: 0.00390625}
    data_param {
        source: "examples/mnist/mnist_train_lmdb"
        batch_size: 64
        backend: LMDB }
}
```

Los parámetros de transformación (*transform_param*) indican el preprocesamiento de la imagen antes de comenzar el entrenamiento, y éstos deben coincidir en ambas fases, ya que si se evaluase la red con una transformación de la imagen distinta a la aplicada en el entrenamiento los resultados obtenidos no serían realistas. En este caso, se utiliza un factor de escala, indicado con el nombre *scale*, que establece el rango de la imagen en [0,1].
En esta red se utilizan dos capas de datos que difieren en la fase en la que se utilizan los datos (entrenamiento o evaluación de prestaciones de la red), el tamaño del lote, siendo 64 muestras para el entrenamiento y 100 para la evaluación, y la ruta de la que se cogen los datos.

A continuación, se comienzan a definir las capas del entrenamiento propiamente dicho. Se intercala una capa de convolución con una de agrupamiento y se repite la estructura dos veces.

En la capa de convolución, explicada en la Sección 2.1.2, se define que el tamaño del filtro será de 5x5 y que se obtienen 20 salidas en la primera de ellas. En la segunda, sin embargo, se obtienen 50 salidas. Además, se define el algoritmo "Xavier" para la inicialización de los pesos, que determina automáticamente la escala de inicialización basada en el número de entradas y de las neuronas de salida, y la inicialización del sesgo mediante una constante que por defecto es 0. Esta estructura se define de la siguiente

forma en el documento de Caffe:

```
layer {
    name: "conv1"
    type: "Convolution"
    bottom: "data"
    top: "conv1"
    param {lr_mult: 1}
    param {lr_mult: 2}
    convolution_param {
        num_output: 20
        kernel_size: 5
        stride: 1
        weight_filler {type: "xavier"}
        bias_filler {type: "constant"}
        }
}
```

La capa de agrupamiento, también explicada en la Sección 2.1.2, es alimentada por la capa de convolución anterior y alimenta a la siguiente en caso de que la haya. A continuación se muestra un ejemplo de cómo se define esta capa.

```
layer {
    name: "pool1"
    type: "Pooling"
    bottom: "conv1"
    top: "pool1"
    pooling_param {
        pool: MAX
        kernel_size: 2
        stride: 2 }
}
```

Se definen en esta capa un tamaño de filtro de 2x2, un intervalo de dos muestras entre cada aplicación del filtro, por lo que no hay solape, y el método del máximo para realizar el agrupamiento.

En caso de ser la última de las capas de agrupamiento sus salidas serán la entrada de las capas completamente conectadas, *InnerProduct*, explicadas en la Sección 2.1.2. En concreto se establecen dos, cuya definición en el documento queda de la siguiente manera:

```
layer {
    name: "ip1"
    type: "InnerProduct"
    bottom: "pool2"
    top: "ip1"
    param {lr_mult: 1}
    param {lr_mult: 2}
    inner_product_param {
        num_output: 500
        weight_filler {type: "xavier"}
        bias_filler {type: "constant"}
    }
}
```

En estas capas se definen 500 salidas para la primera de ellas, y tantas como clases se tengan, en la segunda. La aplicación que se está desarrollando pretende clasificar los dígitos del 0 al 9, por lo que esta última capa debe tener 10 salidas.

Las capas completamente conectadas están separadas entre sí por una capa de activación, en este caso lineal, llamada *ReLU*. Esta capa fue explicada en la Sección 2.1.2 y tiene la siguiente forma en el documento:

```
layer { name: "relu1"
    type: "ReLU"
    bottom: "ip1"
    top: "ip1"}
```

La capa de activación no dispone de ningún parámetro modificable ya que la plataforma proporciona directamente la función por su identificador único.

Para terminar la estructura de la red básica, Caffe permite la opción de añadir capas que muestren parámetros de evaluación de la red que se está entrenando. Éstas se deben añadir como una capa más a continuación de las anteriormente explicadas y se define su estructura de la siguiente forma:

```
layer {
    name: "accuracy"
    type: "Accuracy"
    bottom: "ip2"
    bottom: "label"
    top: "accuracy"
    include {phase: TEST}
}

layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip2"
    bottom: "label"
    top: "loss"
}
```

Estas dos capas permiten obtener valores de precisión y pérdidas cada ciertas iteraciones, siendo marcado este valor en el documento que se explicará a continuación, el solucionador.

En la Figura 3.1 se puede observar un esquema de la estructura definida en este apartado, los valores de interés y cada una de las entradas y salidas de las capas. Para obtener esta figura se ha ejecutado un código proporcionado por la propia plataforma, que, mediante el archivo que define la estructura explicado anteriormente, dibuja la red. Para ello se debe ejecutar el comando mostrado a continuación.

```
$ caffe/python/draw_net.py <netprototxt_filename> <out_img_filename>
```

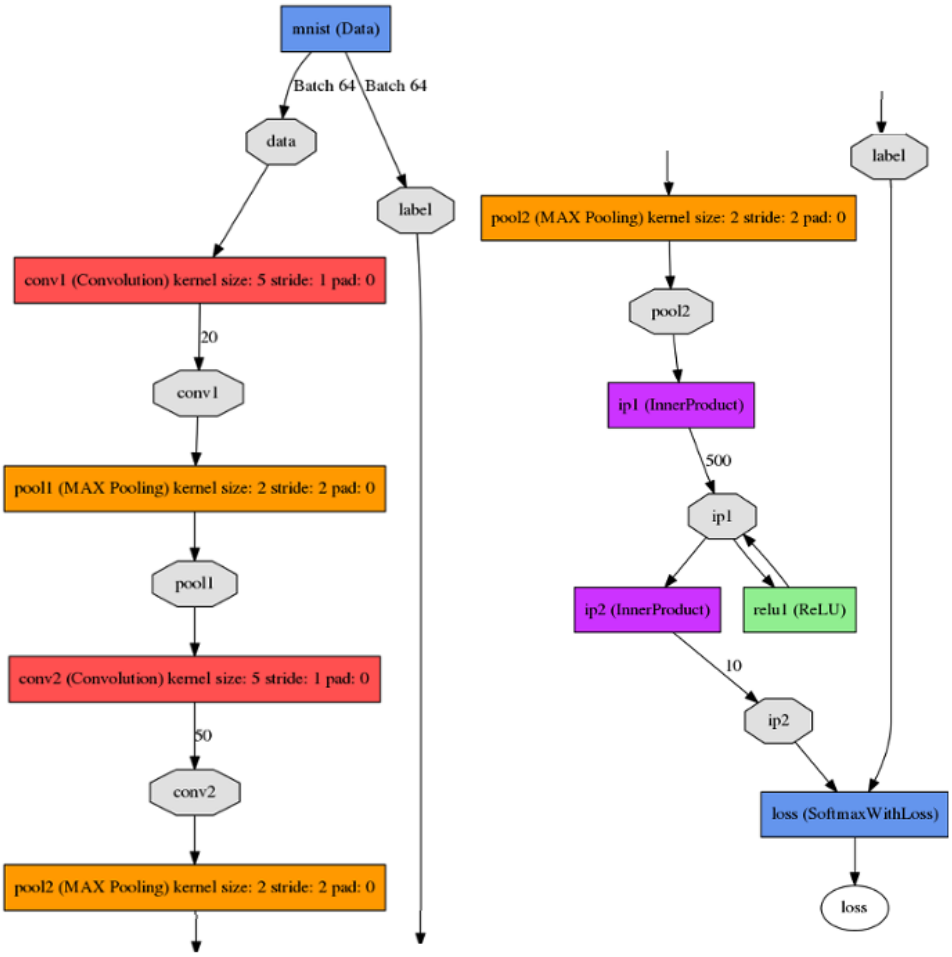


Figura 3.1: Red básica LeNet MNIST.

3.1.1.2. Definición del solucionador

Para esta tarea se utiliza el archivo de Caffe *lenet_solver.prototxt*. En este documento se definen parámetros como la estructura de red que se utiliza, definida en el apartado anterior, y el número de iteraciones que se ejecutan durante el entrenamiento de la red, cuya explicación se aportó en la Sección 2.1.2. Además, en ese mismo capítulo, se explican el resto de parámetros que se manejan en este proyecto, como la evaluación de la red o las redes intermedias que se almacenan. A continuación se muestra el aspecto de este solucionador en Caffe, destacando los parámetros más significativos.

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt"
# test_iter specifies how many forward passes the test should carry
# out.
# In the case of MNIST, we have test batch size 100 and 100 test
# iterations, covering the full 10,000 testing images.
test_iter: 100
# Carry out testing every 500 training iterations.
test_interval: 500
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 100
# The maximum number of iterations
max_iter: 10000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
# solver mode: CPU or GPU
solver_mode: CPU
```

Se debe poner especial atención en parámetros como *net*, que define la ruta al documento de la sección anterior en el que se define la estructura de red; *test_interval*, que marca cada cuántas iteraciones se realiza la evaluación de la red en el entrenamiento; *max_iter*, que define el número de iteraciones totales para finalizar el entrenamiento; *snapshot*, que indica cada cúantas iteraciones se crea un archivo con la red intermedia correspondiente; y, finalmente, *snapshot_prefix*, que marca la ruta en la que se almacenan los archivos. La forma en la que se almacenan los archivos se corresponde con la ruta indicada hasta el último "/", siendo lo posterior el nombre deseado, completado con el

número de iteración correspondiente. En el ejemplo mostrado, el archivo se almacena en la ruta *"examples/mnist/"* y el nombre de la red final es *"lenet_iter_10000"*.

Tras definir el solucionador se procede a la ejecución de un archivo que comience con el entrenamiento de la red y permita obtener, finalmente, el modelo entrenado para usar en la aplicación.

3.1.1.3. Ejecución de la red

Para comenzar con el entrenamiento de la red, una vez definida la estructura y y el solucionador, se deben ejecutar los siguientes comandos:

```
cd $CAFFE_ROOT
./examples/mnist/train_lenet.sh
```

El archivo que se ejecuta contiene información sobre qué solucionador se debe implementar y el modo de ejecución, de la forma que se muestra a continuación.

```
#!/usr/bin/env sh
set -e

./build/tools/caffe train
  --solver=examples/mnist/lenet_solver_validation.prototxt
```

Este archivo permite añadir una nueva línea mediante la que se obtiene, en la ruta marcada, un archivo de *log* con información sobre el entrenamiento y la evaluación. Esta línea se añade tras indicar el solucionador de la forma que se muestra a continuación.

```
2>&1 | tee .../NombreArchivoLog.log $@
```

En la Figura 3.2 se muestra la información que se observa en el terminal al ejecutar el entrenamiento, siendo ésta la misma que queda registrada en el archivo *log* indicado.

Figura 3.2: Ejecución de entrenamiento de red LeNet MNIST.

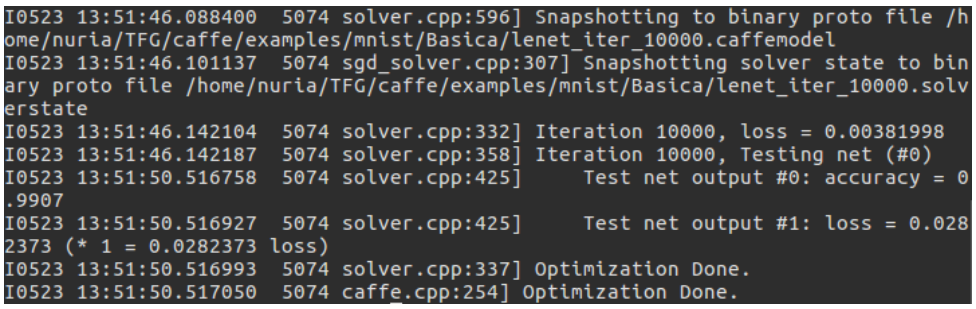


Figura 3.3: Fin de entrenamiento de red LeNet MNIST.

Tras terminar el entrenamiento, mostrado en la Figura 3.3, se obtiene el archivo con la red neuronal entrenada, almacenado según la ruta que se indicó en el solucionador, que podrá ser utilizada en la herramienta que sea de interés.

Los parámetros de pérdidas y precisión calculados durante el entrenamiento para ambas fases quedan almacenados en el archivo *log* generado, y son divididos en las dos fases, entrenamiento y evaluación, para su análisis. Para ello se ejecuta el archivo *parse_log.sh* proporcionado por la plataforma en su carpeta *tools/extra*. En la Figura 3.4 se muestra el aspecto de estos archivos desglosados.

#Iters	Seconds	TrainingLoss	LearningRate
0	4.543065	2.39246	0.01
100	10.843418	0.212262	0.00992565
200	17.127633	0.152028	0.00985258
300	23.413865	0.150457	0.00978075
400	30.395256	0.0463222	0.00971013
500	41.022588	0.111935	0.00964069
600	47.257046	0.101373	0.0095724
700	53.451002	0.191312	0.00950522
800	59.698984	0.198037	0.00943913
900	65.914914	0.185254	0.00937411
1000	76.655816	0.0961904	0.00931012
1100	82.881708	0.00524919	0.00924715
....			
9600	691.993744	0.00282211	0.00603682
9700	698.204811	0.00327492	0.00601382
9800	704.430812	0.0124943	0.00599102
9900	710.674634	0.00725178	0.00596843
10000	716.892889	0.00381998	

(a)

#Iters	Seconds	TestAccuracy	TestLoss
0	0.000828	0.0389	2.45624
500	36.547207	0.9718	0.088367
1000	72.189569	0.9822	0.0551621
1500	107.783205	0.9841	0.0472525
2000	143.383918	0.9863	0.0418842
2500	178.980020	0.9843	0.0484483
3000	214.905048	0.9864	0.0395492
3500	250.514330	0.9869	0.0407612
4000	286.113136	0.9894	0.029773
4500	322.125512	0.987	0.0361792
5000	358.348772	0.9893	0.0319051
5500	394.239689	0.9886	0.0330068
6000	430.236060	0.9906	0.0282792
6500	466.125142	0.9902	0.0299727
7000	502.317760	0.9906	0.027419
7500	537.902366	0.9897	0.0311683
8000	573.707601	0.9906	0.0275084
8500	609.346996	0.9901	0.028508
9000	645.584828	0.9904	0.0273948
9500	681.330111	0.9899	0.0338555
10000	716.892972	0.9907	0.0282373

(b)

Figura 3.4: Archivos log de: (a) Entrenamiento, (b) Evaluación.

3.1.2. Componente JdeRobot clasificador

Se desarrolla un componente escrito en Python que, mediante la ayuda del *Camera Server* de JdeRobot, comentado en la Sección 2.1.1, y la red explicada en la Sección 3.1.1, es capaz de clasificar un dígito mostrado a la cámara en tiempo real, encendiendo una bombilla que se corresponde con el número obtenido. En la Figura 3.5 se muestra un diagrama de bloques con el diseño de este componente y todas sus entradas y salidas.

Debido a la magnitud de la tarea a realizar, se optó por dividir el programa en dos hilos que serán explicados a continuación. Uno de ellos se encarga del aspecto gráfico de la aplicación, mostrando la imagen obtenida por la cámara, la imagen procesada para la clasificación, y la iluminación de la bombilla correspondiente. El segundo hilo, se encarga de gestionar la captación de la cámara, mediante la conexión con el componente *Camera Server*, así como el proceso de clasificación utilizando la red entrenada. Todo el código correspondiente a esta aplicación se encuentra disponible en Github [1].

[1]https://github.com/RoboticsURJC-students/2016-tfg-nuria-oyaga

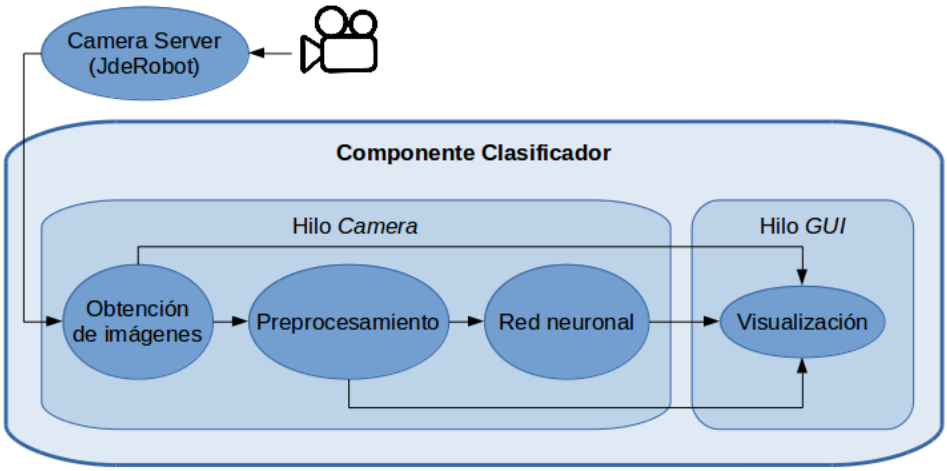


Figura 3.5: Esquema del componente.

3.1.2.1. Hilo *Camera*

El hilo fundamental de la aplicación, que se encarga de la lógica de la misma mediante la adquisición de la imagen y su posterior procesamiento, está referenciado por el nombre *Camera*.

Al comienzo de la ejecución se inicializa un objeto Cámara, mediante el constructor *Camera()*, que será el encargado de gestionar las acciones anteriormente nombradas. En esta inicialización se indica qué cámara se va a utilizar, referenciada de manera externa mediante un archivo de configuración que se indica en la ejecución de la aplicación. La propiedad que indica la cámara en este archivo está enlazada con el componente *Camera Server* de JdeRobot y es la siguiente:

```
Numberclassifier.Camera.Proxy=cameraA:default -h localhost -p 9999
```

Otro aspecto importante que se maneja en la inicialización de la cámara es la especificación y carga de la red que se empleará para la clasificación. Este aspecto se realiza mediante las siguientes líneas:

```
model_file = '.../lenet.prototxt'
pretrained_file = '.../lenet_iter_10000.solverstate'
self.net = caffe.Classifier(model_file, pretrained_file,
                            image_dims=(28, 28), raw_scale=255)
```

Con este código se realizan las tres acciones necesarias para establecer la red que se utiliza. En primer lugar, se indica cuál será el modelo empleado para la clasificación. Este modelo es un archivo porporcionado por Caffe de manera homóloga al *lenet_train_test.prototxt*, con la excepción de que la capa de datos no recurre a archivos almacenados sino que utiliza imágenes que serán insertadas en la ejecución de la red. El resto de datos deben ser exáctamente iguales a la estructura de la red entrenada para que no se produzcan errores. En segundo lugar, se indica la red entrenada que se utiliza en la ejecución, el archivo obtenido al finalizar el entrenamiento según se indicó en la Sección 3.1.1. Por último, se crea la red ejecutable, es decir, se crea un objeto que será utilizado por la aplicación cada vez que se quiera realizar la clasificación. Para esta creación es necesario indicar que se trata de una red para la clasificación, y, además, introducir los parámetros del modelo, la red entrenada, las dimensiones de las imágenes, y la escala de los píxeles.

Además de las propiedades más importantes comentadas anteriormente, se definen también funciones que son importantes para la ejecución de la aplicación. Se establece una función *update(self)*, que es llamada cada 150ms para la actualización del hilo *ThreadCamera(camera)* creado en el componente principal, para obtener las imágenes de forma periódica y establecer un flujo de vídeo a tiempo real. Esta función, a su vez, necesita de otra, *getImage(self)*, que obtiene la imagen, la redimensiona, y le aplica una transformación necesaria antes de introducirla en el proceso de clasificación, devolviendo un array con las dos imágenes, la original y la transformada. Para esa transformación se utiliza una tercera función de la cámara, *trasformImage(self,img)*. En ella, se centra la imagen en un cuadrado, pues la captada es rectangular y la necesaria para introducir en la red debe ser cuadrada; se convierte a imagen de grises; se redimensiona al tamaño necesario para introducirla en la red (28x28); se le aplica un filtro gaussiano de 5x5 para reducir el ruido; y por último, se aplica un filtro de transformación por gradiente para lograr independencia de los niveles de intensidad particulares en las imágenes.

Finalmente, se crea una función para realizar la clasificación de los dígitos. En primer lugar se asegura que las dimensiones del *blob* de datos sea de 28x28. En el siguiente paso, se introduce a la red la imagen obtenida tras la transformación, aplicandole el factor de escala para que el intervalo de los píxeles esté entre 0 y 1, coincidiendo con el

aprendizaje. Posteriormente se ejecuta la red y se obtiene, como salida, una estructura que almacena, por un lado, la propiedad *'prob'* que se corresponde con un array que incluye las probabilidades de que la imagen introducida sea cada uno de los dígitos posibles, y, por otro, el tipo de datos que se almacena, en este caso *float32*. Posteriormente, de ese array de probabilidades, se escoge el dígito cuya probabilidad es mayor y se devuelve. Todo ello queda implementado con el siguiente código:

```
def classification(self, img):
    self.net.blobs['data'].reshape(1,1,28,28)
    self.net.blobs['data'].data[...]=img * 0.00390625
    output = self.net.forward()
    digito = output['prob'].argmax()
    return digito
```

Una vez establecida la lógica de la aplicación, se procede a desarrollar el interfaz gráfico que permite al usuario visualizar tanto las imágenes captadas y transformadas como el resultado de la clasificación.

3.1.2.2. Hilo de interfaz gŕafico

Para el aspecto gráfico de la aplicación, en el componente principal se inicializa un objeto llamado *window* mediante el constructor *Gui()*, al que posteriormente se le vincula la cámara mediante una función propia, *window.setCamera(camera)*. Por último, al tratarse de un componente gráfico, es necesario indicar que se muestre mediante *window.show()*. Al inicializar este objeto se crean todos los elementos gráficos que son necesarios y que se modificarán posteriormente para conseguir el resultado deseado.

Al igual que en el caso de la cámara, se establece un hilo que permite aligerar la ejecución de la aplicación mediante *ThreadGui(window)*, que establece el tiempo de actualización en 50ms. Debido al uso de este hilo, se crea en el objeto una función *update()* que, en este caso, se encarga de obtener las imágenes original y transformada mediante la función *getImage()* de la cámara, y adaptarlas para mostrarlas en las zonas de interfaz gráfico definidas para cada una de ellas. Además, llama a otra función propia, *lightON(out)*, que cambia el color del fondo del dígito que se haya clasificado, haciendo uso de la función de clasificación definida anteriormente en la cámara.

En la Figura 3.6 se puede observar el resultado gráfico de la aplicación. Al no tener detección, la ejecución de la clasificación es continua, por lo que, aunque no exista un dígito en la imagen, el componente decide constantemente un determinado dígito que considera correcto, encendiendo la bombilla adecuada.

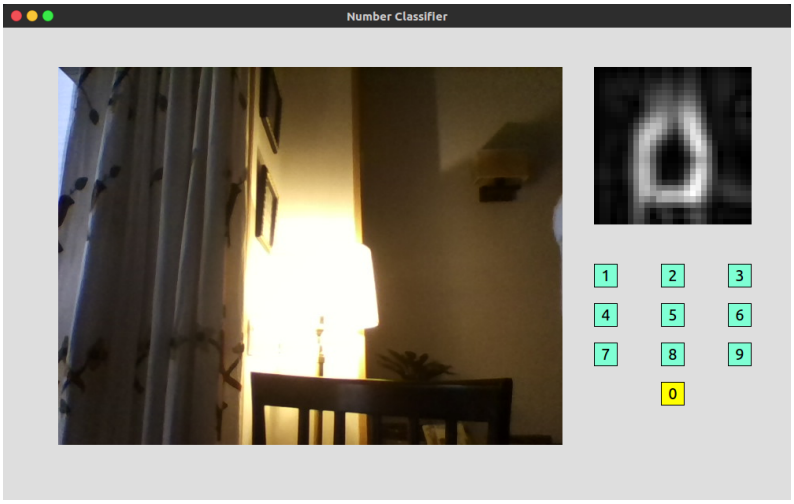


Figura 3.6: Captura de componente gráfico de la aplicación.

3.1.2.3. Ejecución del componente

El proceso de ejecución del componente se divide en dos pasos. Por un lado es necesaria la ejecución del servidor de imágenes, para lo que se utiliza el componente de JdeRobot. Por otro lado se debe lanzar el propio componente clasificador explicado.

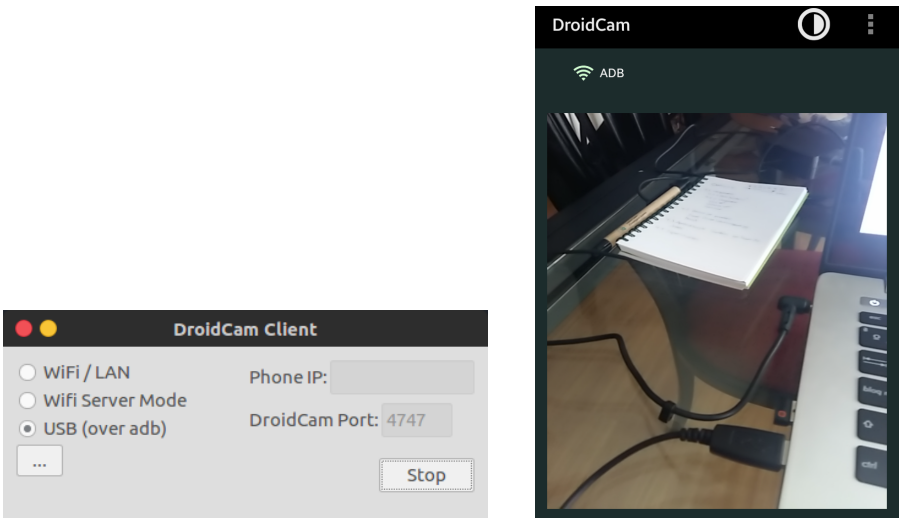
Para ejecutar el *Camera Server* se siguen las instrucciones que aporta la plataforma JdeRobot, utilizando el archivo de configuración que se facilita. Se utiliza el siguiente comando:

```
cameraserver cameraserver.cfg
```

En este trabajo, la propiedad de interés del archivo de configuración es *Camera.0.Uri*, que indica la fuente concreta de vídeo. Esta fuente puede ser (1) un archivo de vídeo almacenado, para el que se emplea la ruta del archivo en ese campo, (2) la webcam del propio ordenador, para el que se utiliza el valor 0, (3) otra cámara externa, para la que

se le indica el valor 1.

En la Sección 2.1.5 se comentó una aplicación que permite utilizar la camara de un *smartphone* Android como fuente de vídeo mediante una cámara externa. Para poder utilizar esta herramienta es necesario tener instalados el programa tanto en el dispositivo móvil a utilizar como en el propio ordenador, según se indica en la guía de la aplicación [2], y abrir la aplicación. Una vez abierta en ambos dispositivos, se debe conectar el USB del ordenador al móvil e indicar en la aplicación de escritorio que la conexión se hará vía USB. La razón del uso del USB y no de la conexión vía *WiFi* radica en la rapidez, siendo más adecuada para tiempo real. Una vez se han realizado las acciones anteriores se establece la conexión y se obtienen los resultados de la Figura 3.7 para el ordenador y el dispositivo.



(a) (b)

Figura 3.7: Capturas de DroidCam en: (a) Escritorio, (b) Dispositivo móvil

Al incluir esta aplicación, el esquema de las conexiones del componente se modifica, dando lugar a una nueva estructura mostrada en la Figura 3.8

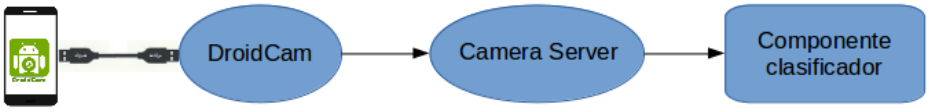


Figura 3.8: Esquema de conexión con DroidCam.

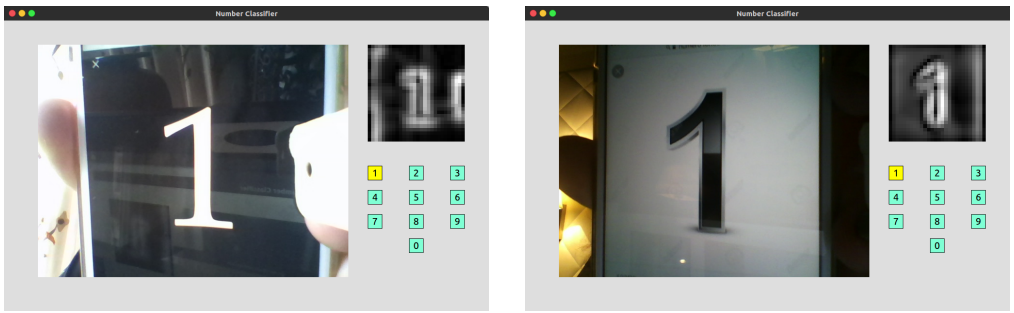
[2]https://www.dev47apps.com/droidcam/linuxx/

Tras tener en funcionamiento el servidor de imágenes se procede a la ejecución del componente clasificador, para ello se ejecuta el siguiente comando:

```
python numberclassifier.py --Ice.Config=numberclassifier.cfg
```

El componente Python contiene los procedimientos indicados en las secciones anteriores, la creación del GUI, la cámara y el lanzamiento de los hilos correspondiente a cada uno de ellos. En el fichero de configuración se tiene una propiedad que indica qué cámara utilizar, es importante que el nombre de esta cámara se corresponda con el indicado en el fichero de configuración del servidor, de esta manera se establece la comunicación entre ambos componentes.

Finalmente, tras la ejecución, obtenemos el resultado del componente mostrado en la Figura 3.9, donde se aprecia el funcionamiento del mismo para un número sencillo.



(a) (b)

Figura 3.9: Ejemplo del componente clasificador con: (a) Fondo negro, (b) Fondo blanco.

Tras conseguir la aplicación del clasificador, se ha evaluado la red obtenida mediante un banco de pruebas, que será explicado a continuación y se ha procedido a la mejora de la misma gracias a los diferentes resultados obtenidos.

3.2. Banco de pruebas

Para evaluar las redes neuronales que se desarrollan en el proyecto se ha elaborado un banco de pruebas que permite obtener las medidas de prestaciones explicadas en el Capítulo 2, a partir de una base de datos de test.

3.2.1. Obtención de datos de test

El primer paso para elaborar este banco de pruebas pasa por el desarrollo de un *script*, *testcaffenet.py*, que permite introducir una base de datos de test a la red neuronal y obtener la clasificación para cada elemento. Este *script* está dividido en tres partes claramente diferenciadas que permiten la obtención de los resultados finales y que serán detalladas a continuación.

Obtención de las imágenes

Las imágenes y las etiquetas que las identifican en una determinada clase están almacenadas en bases de datos de tipo *Lightning Memory-Mapped Database (lmdb)*. Este tipo de bases de datos requieren de un método específico para acceder a su contenido y poder manipular las imágenes que se almacenan en ellas, que queda definido a continuación.

```
lmdb_env = lmdb.open('.../test_lmdb')
lmdb_txn = lmdb_env.begin()
lmdb_cursor = lmdb_txn.cursor()
```

Tras este código se obtiene un puntero que apunta al comienzo de los datos en la base de datos y que permitirá recorrerla para obtener las imágenes y sus etiquetas.

Para procesar los datos obtenidos anteriormente utilizando la plataforma Caffe, será necesario crearse una estructura *Datum* de la propia plataforma. Esta estructura incluye, en cada iteración para recorrer la base de datos, la información de la instancia que se analiza. El siguiente código crea la estructura indicada y especifica la forma en que se recorre la base de datos, obteniendo por un lado los datos de la imagen en sí (*data*) y por otro sus etiquetas (*label*).

```
datum = caffe.proto.caffe_pb2.Datum()
...
for key, value in lmdb_cursor:
    datum.ParseFromString(value)
    label = datum.label
    data = caffe.io.datum_to_array(datum)
    ...
```

Finalmente, en la variable *data* se almacena la imagen que se utilizará posteriormente para realizar la clasificación, y en *label* la etiqueta correspondiente que se empleará para evaluar la capacidad de generalización de la red.

Clasificación de las imágenes

La tarea de clasificación se realizará de la misma manera que se especificó en la Sección 3.1.2.1, utilizando la misma función sobre cada uno de los *data* obtenidos, según se muestra en el código a continuación.

```
...
net_out = classification(data)
...
```

Una vez obtenido el dígito que la red interpreta, se procede a comparar éste con la etiqueta verdadera para evaluar las prestaciones de la red.

Evaluación de prestaciones

Para obtener resultados que puedan ser procesados por el banco de pruebas se creará un archivo de texto con la información necesaria. En este archivo se incluye una descripción del contenido y, para cada muestra procesada por la red, su número, la etiqueta real, la identificada por el clasificador y un booleano que indicará la coincidencia entre ambas. La obtención de este archivo de texto se logra gracias al siguiente código:

```
for key, value in lmdb_cursor:
    ...
    if label == net_out:
        conclusion = True
    else:
        conclusion = False
    testfile.write("Interacion " + str(loop) + ":")
    testfile.write(str(label) + " " + str(net_out) + " " )
    testfile.write(str(conclusion) + "\n")
    ...
```

Esta estructura permitirá un manejo más cómodo de los resultados por el banco de pruebas creado, además de una mejor comprensión para el usuario que lea el archivo.

3.2.2. Banco de pruebas manual

Las medidas de prestaciones que se obtienen con este banco de pruebas son las explicadas en la Sección 2.3: Matriz de confusión, *precision* y *recall*. De manera externa al banco de pruebas, y gracias a Caffe, se obtendrán también valores de *accuracy*, homólogo a la tasa de acierto, y *loss*, para cada una de las redes intermedias que se obtienen durante el entrenamiento hasta alcanzar la final, y la propia red final.

Para la elaboración de este banco de pruebas se ha optado por la herramienta de *Libre Office Calc*, que permite realizar diversas operaciones sobre hojas de cálculo gracias a múltiples fórmulas y funciones. Se han volcado los datos obtenidos con el *script* anterior estableciendo como separador el espacio y los dos puntos, obteniendo así distintas columnas, cada una de ellas con un determinado dato. De estas columnas, serán de interés la que contiene la etiqueta real, la clasificación realizada, y la conclusión final, acierto o fallo.

Una vez se dispone de los datos necesarios para la evaluación de prestaciones, separados y correctamente ordenados, se procederá a identificar el número de aciertos y fallos obtenidos. Esta acción es llevada a cabo a nivel global, para obtener los parámetros de tasa de acierto o *accuracy*, y a nivel de dígito, para obtener la matriz de confusión y con ella los valores de *precision* y *recall* para cada uno de los dígitos.

Tasa de acierto

Este valor es el más sencillo de obtener. Para calcular la tasa de acierto, independientemente del dígito que se trate, basta con contar el número de veces que se ha obtenido el valor *True* en la columna de conclusión y dividirlo entre el número de imágenes de test. De esta manera se obtiene el porcentaje de imágenes que se han clasificado de forma correcta, valor que se corresponde con una estimación de la tasa de acierto de la red.

Para calcular la tasa de acierto, el código empleado ha sido dividido en cuatro partes diferenciadas, obteniendo en cada una de ellas uno de los valores para calcular el resultado final, que serán expuestas a continuación.

- Para obtener el número de clasificaciones correctas:

```
CONTAR.SI('Sobel sin trasform'.E3:E20002;"True")
```

  Donde:

  - 'Sobel sin transform' es la hoja en la que se han volcado los resultados del archivo de texto.
  - E3:E20002 es la columna que contiene los datos de la conclusión.
  - "True" indica que se quiere contar el número de veces en esa columna que aparece ese valor

- Para obtener el número de clasificaciones incorrectas:

```
CONTAR.SI('Sobel sin trasform'.E3:E20002;"False")
```

  Es equivalente al anterior pero, en este caso, se cuenta el número de veces que se cometió un error en la clasificación.

- Para obtener el número de imágenes de evaluación totales será suficiente con realizar la suma de los correctos e incorrectos.

- Para calcular el porcentaje de acierto se realizará la división del número de aciertos entre el total y se multiplicará por 100 para obtener el porcentaje.

Matriz de confusión

Para elaborar la matriz de confusión se parte de la misma hoja de cálculo del apartado anterior. En este caso, para cada dígito real, se debe tener en cuenta tanto el número de veces que se clasifica correctamente, como el número de veces que se equivoca con cada uno de los dígitos restantes.

Se elabora una tabla en la que se enfrentan los dígitos reales, del 0 al 9, con las predicciones. En concreto, cada columna representa las veces que se presenta cada uno de los dígitos y, cada fila el número de veces que éste se predice.

El código de cada celda queda materializado de la siguiente manera:

```
CONTAR.SI.CONJUNTO(C3:C70002;"1";D3:D70002;"2")
```

Donde:

- C3:CC70002 se corresponde con la columna que contiene las etiquetas reales
- D3:D70002 se corresponde con la columna que contiene las etiquetas predichas.
- Los valores entre comillas, "1" y "2" se corresponden con el dígito que se quiera analizar, siendo el primer valor el real y el segundo el predicho. En este caso, se está contando el número de veces que se ha producido un 1 y se ha predicho, erróneamente, un 2.

De esta forma, cada vez que se prediga un dígito determinado, se sumará una unidad en la celda que se corresponda con el dígito real introducido en la red y la etiqueta ofrecida por la red. Una vez se dispone de esta matriz, obtener los valores de *precision* y *recall* resulta sencillo.

Precision

Para obtener el valor de *precision* para cada dígito, se divide el número de veces que se ha clasificado correctamente dicho dígito entre el número de veces totales que se predijo el mismo. Para ello se suman todos los valores por filas, obteniendo el número de predicciones de cada uno de los dígitos, y se divide cada valor de la diagonal, correspondiente con las clasificaciones correctas, entre el valor suma obtenido en la fila correspondiente.

Recall

Para este parámetro, se divide el número de clasificaciones correctas de cada dígito entre el número de veces que se produjo el mismo. En este caso, se sumarán los valores obtenidos por columnas, lo que da por resultado el número de veces que se introdujo a la red cada uno de los dígitos. Una vez obtenido ese valor, se divide el valor de la diagonal correspondiente, al igual que en el caso anterior, entre el valor obtenido para cada columna.

3.3. Efectos del aprendizaje en el rendimiento del clasificador

Existen numerosos factores que afectan a la robusted del diseño. Elementos como la base de datos, el número de neuronas empleadas, el número de capas o etapas en el entrenamiento [20], influyen en la robusted de la red.

En esta sección se tratará el efecto que la base de datos de entrenamiento/validación y el número de iteraciones tienen en las prestaciones de la red.

3.3.1. Aprendizaje con imágenes originales

La base de datos empleada en el primer ejemplo es excesivamente simple y, por lo tanto, no aporta la robustez necesaria para un problema de clasificación real. Por ello se estudiará la ampliación y modificación de la misma para obtener una red más robusta que permita la clasificación de imágenes reales de la manera más precisa posible.

Una particularidad de la base de datos de MNIST es que únicamente dispone de ejemplos con el fondo negro y el dígito en blanco. Esto limita bastante la funcionalidad de la aplicación, ya que se pretende clasificar dígitos independientemente del fondo sobre el que éstos se muestren. Para estudiar el efecto que tiene el cambio de fondo en las imágenes, se ha ampliado la base de datos incluyendo, para cada muestra, su negativo. La nueva base de datos se consigue gracias al *script create_neg_database.py*, que parte del tratamiento de bases de datos de tipo lmdb explicado en la Sección 3.2. Se debe abrir la base de datos con las imágenes originales y realizar la transformación deseada. Posteriormente, para almacenar las imágenes transformadas, se debe abrir una nueva base de datos de este tipo que permita escritura, según se muestra a continuación.

```
new_lmdb_env = lmdb.open('.../database_lmdb', map_size=int(1e12))
new_lmdb_txn = new_lmdb_env.begin(write=True)
new_lmdb_cursor = new_lmdb_txn.cursor()
new_datum = caffe.proto.caffe_pb2.Datum()
```

Se puede observar que el proceso es muy similar al explicado en la Sección 3.2, incluyendo dos parámetros que permitan la escritura en la base de datos. Posteriormente,

dentro del bucle, se debe almacenar la imagen original en la nueva base de datos, aplicar el negativo, y almacenar, también, el resultado de la transformación.

Para insertar imágenes en una nueva base de datos es necesario realizar dos acciones, la inserción en la base de datos y su actualización:

```
new_datum = caffe.io.array_to_datum(data,label)
keystr = '{:0>8d}'.format(item_id)
new_lmdb_txn.put( keystr, new_datum.SerializeToString() )
```

De esta manera se incluye en la posición *keystr* la imagen y la etiqueta deseada, a partir del puntero que señala las posiciones dentro de la base de datos. Posteriormente, para la actualización de la base de datos, se debe incluir un nuevo código que guarda los cambios realizados y actualiza la posición del puntero.

```
new_lmdb_txn.commit()
new_lmdb_txn = new_lmdb_env.begin(write=True)
```

Las anteriores líneas se incluyen dentro de un condicional que hará que únicamente se escriba en la base de datos cada cierto tiempo, ya que no es necesario realizar estas acciones en cada una de las inserciones realizadas, ahorrando carga computacional.

En la Figura 3.10 se muestra el negativo almacenado en la base de datos para cada dígito mostrado en la Figura 2.3. Estas imágenes han sido obtenidas con el *script dataread.py*, que lee las imágenes de la base de datos y crea un archivo para su visualización.

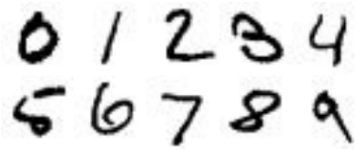


Figura 3.10: Negativo de muestras de la base de datos.

Tras ejecutar el *script* que aumenta la base de datos con la transformación, se obtiene una nueva base de datos de test con los negativos, la cual tendrá el doble de muestras que en el caso original, es decir 20000 ejemplos. Ésta es introducida en el banco de pruebas y

a partir de ella se obtienen valores de tasa de acierto.

En la Figura 3.11 se muestran los resultados obtenidos tras entrenar la red con el conjunto de entrenamiento de MNIST, tras la división entrenamiento/validación, y evaluada con las bases de datos de test MNIST y la aumentada con el negativo. Obsérvese que la red falla considerablemente al incluir las imágenes en negativo.

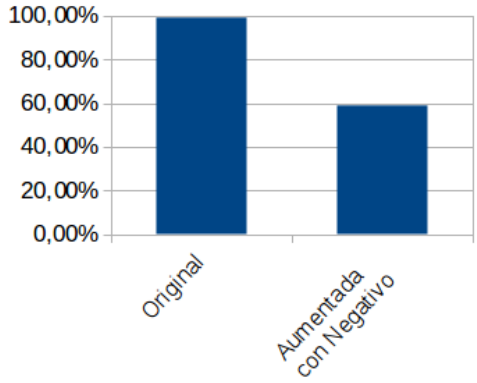


Figura 3.11: Porcentaje de acierto sobre red entrenada con MNIST y evaluada con base de datos original y con la ampliada con negativo.

3.3.2. Aprendizaje con imágenes de gradiente

Para mejorar la capacidad de generalización de la red, puesto que la aplicación no está enfocada a un único tipo de fondo, se opta por aplicar un filtro de gradiente que independice la representación del contenido de la imagen de los niveles de intensidad. Existen varios filtros de gradiente [21].

Para desarrollar la comparación entre los diferentes filtros se ha desarrollado un *script* similar al anterior, en el que se aplicaba el negativo, *create_edges_database.py*, que aplicará el filtro de borde seleccionado. Se parte de la base de datos ampliada con el negativo, por lo que no es necesario almacenar la imagen inicial en la base de datos. Es necesario aplicar el filtro también sobre los ejemplos de entrenamiento y validación, puesto que el objetivo es desarrollar una nueva red neuronal que interprete los bordes. Se obtiene, así, una base de datos de entrenamiento con 48000 muestras, otra de validación con 12000, y una última de test con 20000, a las que se les ha aplicado un determinado filtro de bordes. A continuación se explicarán los tres filtros evaluados en este proyecto: Canny, Laplaciano y Sobel.

Filtro de Canny

El algoritmo de Canny es un operador desarrollado por John F. Canny en 1986 que utiliza un algoritmo de múltiples etapas para detectar una amplia gama de bordes en imágenes [22]. Para ello utiliza el cálculo de variaciones, una técnica que encuentra la función que optimiza un funcional indicado. En este caso, la función óptima es definida por la suma de cuatro términos exponenciales, pero se puede aproximar por la primera derivada de una gaussiana. El resultado de aplicar este filtro es siempre una imagen binaria en la que los píxeles únicamente pueden tomar los valores 0 ó 1 (0 ó 255 dependiendo del rango).

Para aplicar este algoritmo en el código se debe implementar la función proporcionada por *openCV* [23]. En la base de datos de test se obtienen dos imágenes iguales de cada dígito, ya que el mismo filtro se está aplicando sobre el original y el negativo el mismo filtro.

Filtro Laplaciano

El laplaciano es un operador de segunda derivada que se utiliza con frecuencia en la detección de bordes [24], identificando un borde cuando se produce un cruce por cero en la segunda derivada. Este operador utiliza diferentes máscaras para obtener el borde, y en función de la máscara que se utilice se obtendrán distintos resultados.

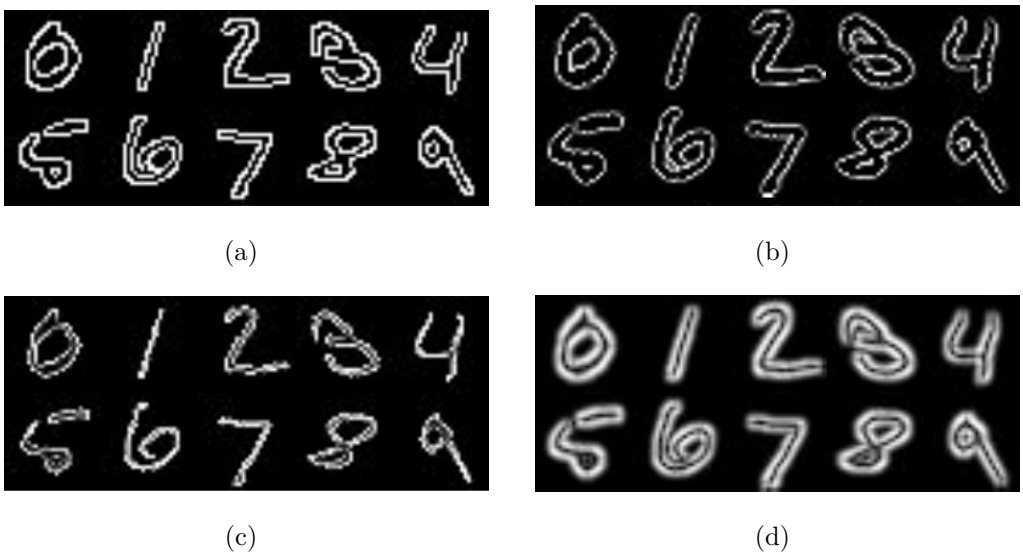
La aplicación del filtro es posible gracias a otra función de *openCV* [25]. Al aplicar este filtro sobre la imagen original y su negativo no se obtiene exactamente el mismo resultado. La diferencia entre ambos viene dada por el propio funcionamiento del filtro, que sitúa el borde de la imagen en la zona donde comienza o acaba el menor nivel de intensidad. De esta manera, al aplicarlo sobre la imagen original, con el fondo negro, se sitúa el borde en la zona "externa" del dígito, mientras que en la imagen del negativo lo localiza en la "interna". Esta diferencia podría perjudicar a la robustez de la red, pues dependiendo de la imagen la diferencia entre ambos podría ser suficiente como para crear confusión.

Filtro de Sobel

Este filtro está formado por dos máscaras que permiten obtener una aproximación a la derivada en las direcciones horizontal y vertical [24]. Para obtener la imagen de

bordes final será necesario sumar ambas soluciones, en valor absoluto, obteniendo la imagen de grises con los bordes.

Para aplicar este filtro, primero se debe aplicar cada una de las dos máscaras, horizontal y vertical, gracias a la función de *openCV* [26]. Una vez se tienen los bordes en ambas direcciones, se suman sus valores absolutos y se normaliza a valores entre 0 y 255, obteniendo una imagen de tipo *float* que deberá ser transformada a *uint8*. En este caso, y al igual que en el caso de Canny, se obtiene la misma imagen de bordes en ambas imágenes, pues el filtro no hace distinción entre bordes internos y externos.



(a) (b)

(c) (d)

Figura 3.12: Muestras de imágenes de bordes: (a) Canny, (b) Laplaciano sobre original, (c) Laplaciano sobre negativo, (d) Sobel.

La Figura 3.12 muestra la aplicación de las tres transformaciones por gradiente explicadas para cada dígito mostrado en la Figura 2.3, incluyendo las dos versiones del filtro Laplaciano, los bordes obtenidos para la original y su negativo.

Una vez obtenidas las diferentes bases de datos, con los filtros de bordes aplicados, se procede al entrenamiento de tres redes neuronales diferentes. Cada red se entrena con uno de los filtros, según se explicó en la Sección 3.1.1, modificando únicamente las bases de datos empleadas en entrenamiento y evaluación.

Tras entrenar las tres redes neuronales será posible comenzar con el test de las mismas, obteniendo las tasas de acierto representadas en la Figura 3.13. Para ello se ha empleado la base de datos de test ampliada con el negativo aplicando, para cada red, el filtro correspondiente.

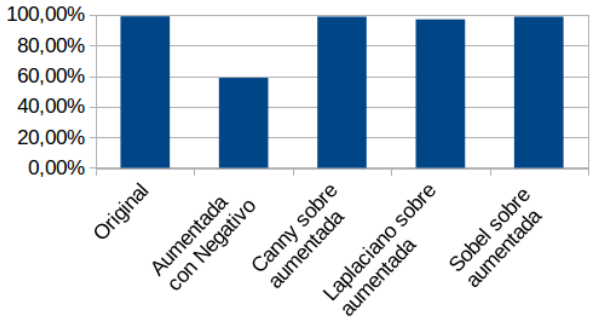


Figura 3.13: Comparación de tasa de acierto para la red entrenada con el conjunto original y evaluada con base de datos de test original y ampliado con negativo, y redes entrenadas con la base de datos con el filtro indicado y evaluada con la de test del mismo filtro.

En la Figura 3.13 se puede observar que el uso de imágenes de bordes mejora en gran medida el entrenamiento, haciendo la clasificación independiente de los niveles de intensidad particulares. Los filtros utilizados producen prácticamente el mismo resultado, siendo ligeramente peor el filtro laplaciano por la diferencia entre bordes internos y externos explicada anteriormente. Por todo ello, y por la preferencia de obtener imágenes en tono de grises que hagan más robusta la red, se elige el filtro de Sobel en la aplicación final.

3.3.3. Otras transformaciones. *Data Augmentation*

Al obtener una imagen con la cámara, ésta puede no estar perfectamente centrada, estar girada o a una escala diferente a la de la base de datos empleada en el entrenamiento. Además, puede incluir ruido introducido por la propia cámara. Esto hará que, si la red ha sido entrenada con imágenes sin ningún tipo de alteración, ésta no generalice bien.

A continuación se explicarán las distinas transformaciones que serán aplicadas para enriquecer la base de datos de entrenamiento/validación. Tambien se enriquece la base de datos de test, que es más realista con estas transformaciones, pues se parece más a

las imágenes reales a las que se enfrentará el clasificador. Todas estas transformaciones se realizan con ayuda de las funciones que proporciona *openCV*.

Rotación

La rotación consiste en el giro un determinado ángulo sobre un eje situado en el centro de la imagen. En concreto, en esta aplicación se ha optado por la rotación con un ángulo aleatorio en el rango [-20,20] grados.

Traslación

La traslación de una imagen consiste en desplazar la misma según un determinado vector definido por dos variables x e y, tomando como referencia el eje central.

Para la evaluación que es de interés en el proyecto, se ha establecido un rango de desplazamiento horizontal aleatorio de [-4,4] píxeles y uno vertical de [-4,2] píxeles, de tal manera que la imagen del dígito no quede recortada.

Escalado

El escalado de una imagen consiste en cambiar la escala de la misma estableciendo una proporción, manteniendo el centro de la imagen en el mismo punto.

Para integrar esta transformación en el estudio realizado se establece un parámetro de proporción aleatorio en el rango [0.5,1.5]. Los valores inferiores a 1 corresponden a una reducción del dígito, mientras que los superiores corresponden a un aumento del mismo.

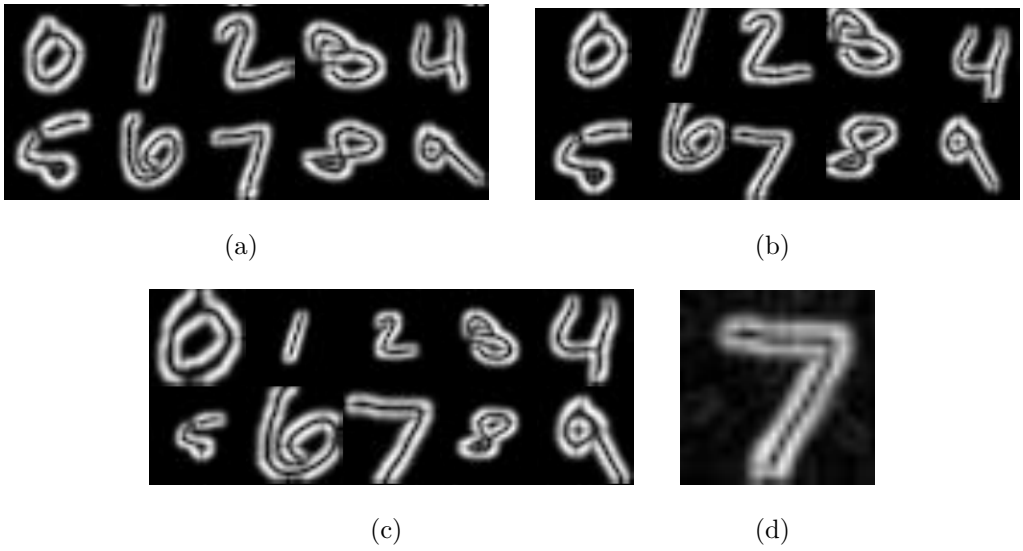
Tras aplicar el escalado, el resultado es una imagen donde las dimensiones del dígito han variado, aumentando o disminuyendo en función de la proporción. Para introducir las imágenes en la red y obtener resultados adecuados se debe adaptar el tamaño de las imágenes al necesario para la red (28x28). Si el tamaño de la imagen es mayor de 28x28 píxeles, la imagen, se recortará manteniendo el centro. Si, por el contrario, el tamaño es menor, se añadirá un borde del mismo nivel de intensidad que el fondo de la imagen hasta obtener el tamaño deseado.

Ruido aleatorio

El ruido aleatorio corresponde a una variación aleatoria de los niveles de intensidad. Existe ruido de distinta naturaleza que pueden estar producidos por diversas causas: por ejemplo ruido Gaussiano, ruido sal y pimienta, o ruido con distribución uniforme.

La aplicación pretende clasificar los dígitos mostrados ante una cámara en tiempo real, donde el ruido más presente es el ruido Gaussiano, por lo que será el empleado para el test. Se contaminará la imagen con ruido Gaussiano de media nula y varianza 0.02, utilizando, en este caso, una función que proporciona *skimage.util*.

La Figura 3.14 muestra la aplicación de las cuatro transformaciones explicadas para cada dígito mostrado en la Figura 2.3, a excepción del ruido, que se muestra sobre un único dígito para apreciar mejor su efecto.



(a) (b)

(c) (d)

Figura 3.14: Muestras de imágenes transformadas: (a) rotación, (b) traslación, (c) escalado, (d) contaminación con ruido aditivo.

3.3.4. Aprendizaje con bases de datos aumentadas

Para evaluar el efecto que tienen las transformaciones anteriores se genera una base de datos de test para cada transformación, a partir de la que proporciona MNIST de 10000 muestras, utilizando el *script create_transformation_database.py*. En estas nuevas bases de datos se incluyen 6 imágenes con la transformación aplicada para cada uno de los ejemplos y la imagen original. Por último, se aplica el filtro de Sobel a cada uno de

los ejemplos, obteniendo una base de datos de test con 70000 ejemplos para cada una de las transformaciones. Estas nuevas bases de datos serán presentadas a la misma red neuronal, diseñada para trabajar sobre imágenes con bordes de Sobel, para poder evaluar la robustez de la misma.

Además de las bases de datos creadas mediante la aplicación de una única transformación, se elabora una nueva base de datos que contiene una combinación de todas las transformaciones: escalado, traslación, rotación y contaminación con ruido, para obtener una evaluación más realista. En esta combinación, a la hora de aplicar la traslación, se tendrá en cuenta el factor de escala aplicado, de tal manera que si es mayor que 1, es decir, la imagen se ha ampliado, el rango de desplazamiento se ve reducido a la mitad en ambas direcciones. De esta manera, el dígito se mantendrá siempre dentro de la imagen, sin verse recortado por ningún lado.

En la Figura 3.15 se muestra la aplicación de la traslación para cada dígito mostrado en la Figura 2.3.

Figura 3.15: Muestra de dígitos con mezcla de transformaciones.

Una vez obtenidas las nuevas bases de datos, se incluirán en el banco de pruebas de la Sección 3.2 y se obtendrá la tasa de acierto, desglosada por dígitos y de manera global. Estos resultados quedan reflejados en la Figura 3.16.

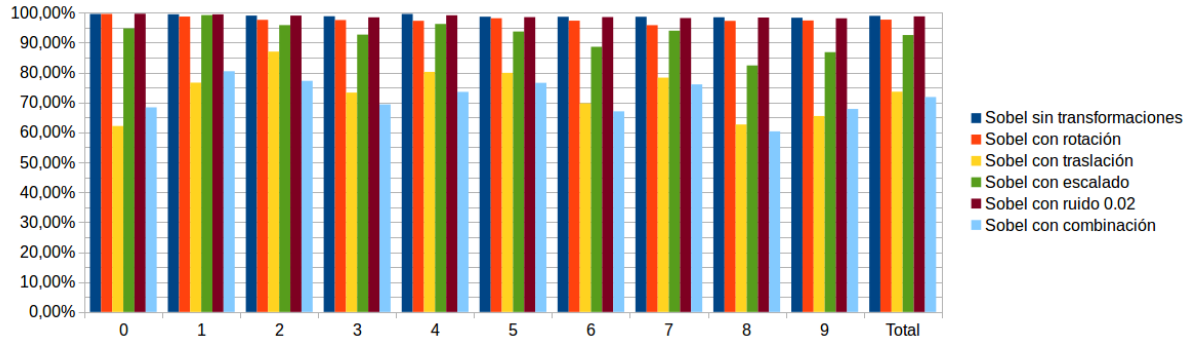


Figura 3.16: *Accuracy* para cada dígito y global en la red entrenada con imágenes de bordes Sobel evaluada con bases de datos transformadas.

Se observa que el hecho de introducir varias transformaciones sobre la imagen hace que la tasa de acierto en el test disminuya considerablemente, siendo especialmente sensible a la traslación, mientras que el ruido aleatorio apenas afecta, algo que concuerda con las imágenes mostradas. Estos resultados hacen ver que la red en un escenario real no reaccionaría de la manera que se desearía, existiendo una probabilidad de fallo del 0.3 si se consideran la combinación de las transformaciones.

Para desarrollar una red más robusta se elabora una nueva base de datos de entrenamiento/validación mediante la combinación de las transformaciones explicadas anteriormente. La nueva base de datos será una modificación de la utilizada en la red básica, presentada en la Sección 3.1.1. Para la evaluación de todas las redes creadas se empleará la base de datos de test que combina todas las transformaciones, con 70000 muestras.

En primer lugar, se calcula la matriz de confusión de la red entrenada únicamente con las imágenes de bordes de Sobel utilizando la base de datos de test ampliada con la combinación de transformaciones, que queda reflejada en la Tabla 3.2. De esta manera se podrán obtener valores de *precision* y *recall* y hacer una comparación adecuada de todas las redes.

		Real										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Predicción	**0**	4691	191	82	90	118	83	173	45	163	192	5828
	1	76	6391	119	65	213	22	209	212	246	142	7695
	2	188	98	5578	462	232	131	121	683	276	229	7998
	3	43	30	443	4904	270	201	48	275	243	311	6768
	4	158	415	291	145	5055	117	720	37	320	381	7639
	5	96	72	65	633	107	4780	234	112	323	243	6665
	6	472	234	86	49	130	216	4497	15	326	91	6116
	7	57	416	265	255	244	105	19	5473	235	488	7557
	8	138	80	131	139	139	129	182	84	4116	194	5332
	9	941	18	164	328	366	460	503	260	570	4792	8402
	Total	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	70000

Tabla 3.2: Matriz de confusión red 1-0 sobre base de datos de test ampliada con combinación de transformaciones.

Tras obtener la evaluación de la red básica, se procede a su modificación y posterior evaluación, variando el número de imágenes transformadas que se utilizan en su entrenamiento, así como la inclusión o no de la imagen original. Para esta tarea se parte de la base de datos de entrenamiento, modificada de la misma manera que se modificó la de test en la sección anterior, obteniendo 6 transformaciones y la imagen original. Tras evaluar los resultados se irá reduciendo el número de imágenes incluidas en la base de datos de entrenamiento, para evaluar el impacto y conseguir una base de datos más pequeña que proporcione buenos resultados disminuyendo la complejidad de cómputo.

Base de datos 1-6

Para elaborar esta base de datos se utilizan 6 imágenes transformadas y la imagen original, todas ellas con la aplicación de los filtros de Sobel. De esta forma se obtiene una base de datos de entrenamiento de 336000 muestras y una de validación de 84000 muestras.

Con estas bases de datos se entrena una nueva red y se calcula la matriz de confusión utilizando la base de datos de test ampliada con combinación de transformaciones,

mostrada en la Tabla 3.3.

		Real										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Predicción	**0**	6770	13	24	13	7	35	74	2	161	40	7139
	1	1	7868	26	15	13	15	25	43	11	12	8029
	2	13	16	6890	125	5	10	6	62	47	8	7182
	3	1	3	9	6567	0	66	0	3	9	8	6666
	4	18	3	53	13	6610	24	52	9	74	71	6927
	5	3	0	1	95	0	5820	16	2	17	5	5959
	6	25	10	7	3	10	123	6519	0	71	2	6770
	7	13	25	170	119	32	17	1	7030	43	108	7558
	8	4	7	31	54	7	67	9	8	6234	12	6433
	9	12	0	13	66	190	67	4	37	151	6797	7337
	Total	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	70000

Tabla 3.3: Matriz de confusión red 1-6 sobre base de datos de test ampliada con combinación de transformaciones.

Base de datos 1-1

En este caso se reducirá el número de imágenes transformadas para comprobar la importancia de las mismas en el aprendizaje. El objetivo es tratar de reducir el número de muestras en la base de datos de entrenamiento y validación para disminuir la carga computacional manteniendo las prestaciones.

Para lograr el objetivo se incluirá en la base de datos de entrenamiento y de validación una única imagen transformada y la imagen original, obteniendo un total de 96000 muestras en la base de datos de entrenamiento y 24000 en la de validación.

Tras obtener las bases de datos se entrenará una nueva red de la que se obtendrá de nuevo la matriz de confusión utilizando la base de datos de test ampliada con combinación de transformaciones, representada en la Tabla 3.4.

		Real										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Predicción	**0**	6637	6	29	8	9	11	36	4	56	39	6835
	1	3	7821	29	9	16	4	18	35	5	16	7956
	2	13	10	6747	45	21	2	8	107	37	12	7002
	3	8	16	117	6703	2	97	4	35	66	42	7090
	4	2	4	45	10	6576	9	41	33	38	164	6922
	5	26	4	27	152	18	5989	80	28	84	91	6499
	6	116	31	10	2	66	51	6474	0	82	10	6842
	7	19	30	135	58	29	12	0	6873	27	89	7272
	8	23	21	68	70	35	54	41	20	6345	66	6743
	9	13	2	17	13	102	15	4	61	78	6534	6839
	Total	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	70000

Tabla 3.4: Matriz de confusión red 1-1 sobre base de datos de test ampliada con combinación de transformaciones.

Base de datos 0-6

Esta reducción consiste en mantener las 6 transformaciones de la imagen en cada muestra, pero no incluir la original. De esta manera se podrá establecer una conclusión sobre la importancia de la imagen original en el aprendizaje.

En esta ocasión se tendrá una base de datos de entrenamiento con 288000 muestras y una de validación con 72000. Al igual que en los casos anteriores se entrenará una nueva red y se obtendrá su matriz de confusión utilizando la base de datos de test ampliada con combinación de transformaciones, representada en la Tabla 3.5.

		Real										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Predicción	**0**	6639	0	23	8	5	12	16	1	41	16	6761
	1	4	7853	12	8	14	1	11	62	2	13	7980
	2	14	18	7022	78	31	6	13	133	58	6	7379
	3	1	15	29	6797	1	69	4	29	32	27	7004
	4	12	5	21	1	6675	3	24	24	40	149	6954
	5	19	3	4	90	7	6065	57	7	91	50	6393
	6	117	23	9	1	19	46	6545	0	62	7	6829
	7	16	13	58	43	12	6	0	6859	24	65	7096
	8	30	13	36	34	15	27	36	10	6399	30	6630
	9	8	2	10	10	95	9	0	71	69	6700	6974
	Total	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	70000

Tabla 3.5: Matriz de confusión red 0-6 sobre base de datos de test ampliada con combinación de transformaciones.

Base de datos 0-1

Finalmente, puesto que los resultados de la reducción de muestras en el entrenamiento resultan bastante satisfactorios, se reduce el número de imágenes transformadas y no se incluye la imagen original, obteniendo una base de datos de entrenamiento de 48000 muestras y de validación de 12000.

Con esta nueva base de datos se entrena una nueva red y se obtiene su matriz de confusión utilizando la base de datos de test ampliada con combinación de transformaciones, reflejada en la Tabla 3.6.

		Real										
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**Total**
Predicción	**0**	6728	4	31	10	11	18	63	6	66	41	6978
	1	2	7854	32	8	26	7	27	72	9	20	8057
	2	10	22	6873	79	27	11	15	145	59	13	7254
	3	5	15	51	6661	6	79	5	38	30	27	6917
	4	1	2	39	6	6348	2	25	16	31	65	6535
	5	15	1	7	137	6	5964	64	11	61	43	6309
	6	45	16	12	3	38	61	6462	0	60	8	6705
	7	14	18	93	55	33	10	0	6833	25	86	7167
	8	26	12	73	88	62	56	40	12	6391	52	6812
	9	14	1	13	23	317	36	5	63	86	6708	7266
	Total	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	70000

Tabla 3.6: Matriz de confusión red 0-1 sobre base de datos de test ampliada con combinación de transformaciones.

Una vez obtenidas las matrices de confusión de cada red neuronal entrenada, se pueden establecer algunas conclusiones a simple vista. En primer lugar, es clara la mejora al entrenar introduciendo alguna imagen ruidosa: si se observan los valores de la diagonal, correspondiente con los dígitos correctamente clasificados, estos valores son superiores al introducir ruido en el entrenamiento. Además, entrenar introduciendo un mayor número de imágenes ruidosas, aparentemente, no aporta gran información a la red, siendo los resultados muy similares en las redes 1-6 y 1-1. Finalmente, introducir la imagen original en el entrenamiento tampoco aporta información fundamental, pues los resultados obtenidos con la red 1-1 y la red 0-1 son muy similares, al igual que ocurre con las redes 1-6 y 0-6.

Para establecer conclusiones más firmes se calculan el *precision* y *recall* de cada una de las redes. Estos resultados quedan reflejados en la Figura 3.17 que confirman las conclusiones alcanzadas con anterioridad: (1) Existe una clara mejora al introducir imágenes ruidosas en el entrenamiento; (2) una única imagen ruidosa aporta suficiente información; y (3) la inclusión de la imagen original no aporta gran información para el entrenamiento.

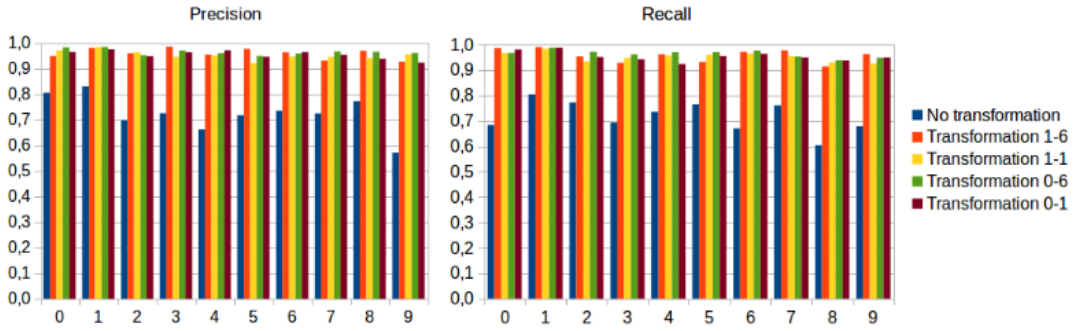


Figura 3.17: Resultados de *Precision* y *Recall* para la red entrenada con imágenes de bordes y las redes entrenadas con transformaciones (1-6,1-1,0-6,0-1), sobre base de datos de test ampliada con combinación de transformaciones

Tras mejorar la red en cuanto a términos de precisión con el cambio en las bases de datos, se analizará la posibilidad de reducir el número de iteraciones para disminuir la carga de cómputo en el entrenamiento de la red.

3.3.5. Número de iteraciones

Hasta ahora se había fijado el número de iteraciones en el entrenamiento de la red en un valor de 10000, siendo la iteración, según lo explicado en la Sección 2.1.2, el paso de un *batch*. Sin embargo, este número de iteraciones no tiene por qué ser el idóneo para la aplicación.

Durante el entrenamiento de una red neuronal el aprendizaje es progresivo. De esta manera es previsible que la red obtenida en la iteración *n+1* sea mejor que la obtenida en la iteración *n*. Existe un momento durante el entrenamiento de la red en el que la mejora entre iteraciones consecutivas es prácticamente nula, pudiéndose producir, incluso, un deterioro en las prestaciones de la red. Este deterioro es conocido como sobreaprendizaje, producido por la excesiva focalización en las muestras proporcionadas en el entrenamiento, empeorando la generalización.

Como se comentó en la Sección 3.1.1, Caffe permite obtener durante el entrenamiento un archivo *log* donde se plasman los datos de *accuracy* calculados cada cierto número de iteraciones. Estos resultados han sido recogidos para cada una de las redes explicadas en

la sección anterior, obteniendo una comparativa, mostrada en la Figura 3.18, que permite seleccionar la red más adecuada para la aplicación.

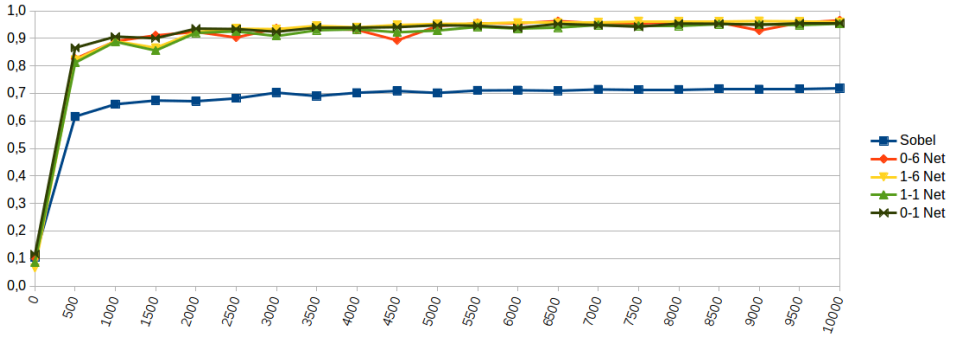


Figura 3.18: Valores de *Accuracy* en redes intermedias para cada una de las redes entrenadas.

En la Figura 3.18 se puede comprobar que, además de la mejora mencionada anteriormente al introducir imágenes ruidosas en el entrenamiento, la estabilidad de la red se alcanza mucho antes de las 10000 iteraciones marcadas.

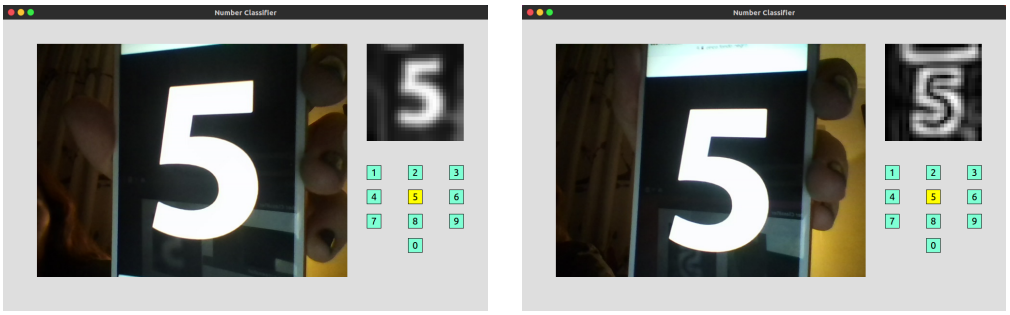
Por todo lo mencionado anteriormente, se decide escoger la red entrenada con la base de datos 0-1, parando el entrenamiento en la iteración 5000. De esta manera, se disminuye consideráblemente la carga computacional, ya que se reduce más de la mitad el número de iteraciones de entrenamiento de la red y se tiene un bajo número de muestras de entrenamiento y validación, manteniendo una buena capacidad de generalización.

3.4. Experimentos

Tras el análisis realizado anteriormente se ha obtenido una red neuronal más robusta que la básica, que permite una mejor clasificación en tiempo real de los dígitos mostrados a la cámara. Con todo lo estudiado en puntos anteriores se realiza una comparación entre la primera red básica que se desarrolló y la red escogida tras el estudio de los efectos del aprendizaje, realizado en la Sección 3.3.

En primer lugar se prueba la aplicación con una imagen sintética, donde el dígito se obtuvo con el ordenador y el fondo es negro, tal y como se entrenó la primera red

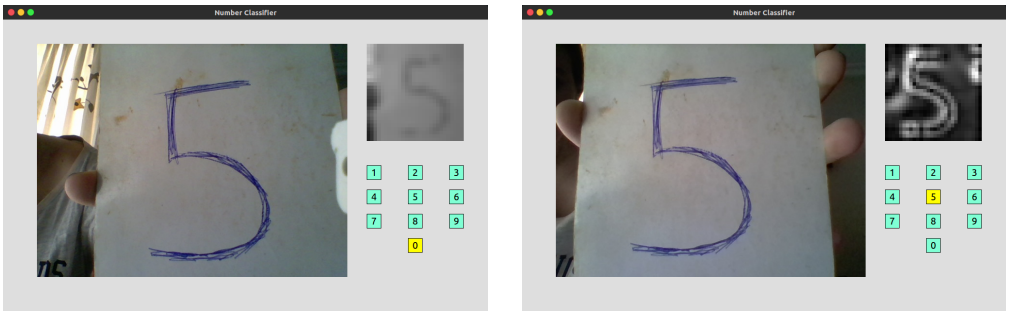
desarrollada. Los resultados de este experimento quedan reflejados en la Figura 3.19.



(a) (b)

Figura 3.19: Evaluación de la aplicación con dígito sintético: (a) red básica, (b) red robusta.

Tras evaluar un ejemplo sintético se pone a prueba la aplicación con una imagen más complicada, variando el fondo y escribiéndolo a mano. En la Figura 3.20, se comprueba cómo un dígito que no lograba ser identificado con la red básica, sí lo hace con la red más robusta obtenida tras el estudio.



(a) (b)

Figura 3.20: Evaluación de la aplicación con dígito manuscrito sobre fondo blanco: (a) red básica, (b) red robusta.

Para realizar los experimentos anteriores se ha tomado como red básica la presentada en la Sección 3.1.1, en cuyo entrenamiento únicamente se disponía de imágenes con fondo negro y el dígito aparece en blanco. Como era de esperar, al mostrar a la cámara un dígito con el fondo blanco la red hace una clasificación errónea. En el caso de la red robusta, se ha seleccionado la red que se concluyó tras la Sección 3.3, entrenada con imágenes transformadas a las que se les ha aplicado el filtro de bordes Sobel. Esta red permite independizar la clasificación del nivel de intensidad del fondo y que la imagen obtenida

por la cámara no necesite ser tan perfecta. Con esta nueva red la aplicación sí consigue clasificar correctamente el dígito mostrado a la cámara.

La aplicación desarrollada es una muestra sencilla de la clasificación de imágenes realizada mediante técnicas de aprendizaje profundo. Este ejemplo abre una nueva puerta para abarcar nuevos problemas de clasificación más complejos, con bases de datos más completas, y otros estímulos a clasificar, que permitan solucionar problemas de interés real para el ser humano.

Capítulo 4

Detección con Aprendizaje Profundo

En este capítulo se realiza una primera aproximación al problema de la detección mediante el aprendizaje profundo utilizando la plataforma Caffe. Para ello se ha utilizado la rama *Single Shot MultiBox Detector (SSD)* [27] de la misma, que, además de permitir el entrenamiento de modelos para la detección de diferentes objetos en función de la base de datos utilizada, ofrece varios modelos que ya han sido entrenados para facilitar esta tarea.

Antes de comenzar en el campo de la detección es importante explicar la diferencia que exite entre esta tarea y la clasificación. En la clasificación, se obtiene como salida una de las clases posibles para la red entrenada. En esta tarea no es posible obtener un resultado nulo, ya que siempre se asociará la imagen presentada a la red con una de las clases. En la detección, por el contrario, se obtiene como salida la presencia o no de un determinado objeto en una imagen, pudiendo obtener un resultado en caso de que no se detecte el objeto . Esta tarea es más compleja que la clasificación, ya que se desconoce el tamaño y posición del objeto a detectar en la imagen.

Para abarcar la tarea de la detección existen numerosos métodos que se diferencian entre sí en la información que consideran relevante para realizar la detección y la forma que tienen de analizar esta información. Así, por ejemplo, existen los métodos basados en apariencia, que usa imágenes de ejemplo (llamadas plantillas o prototipos) de los objetos para realizar la detección. Dentro de este bloque se encuentran, entre otros, el emparejamiento de bordes, la búsqueda “divide y vencerás”, y el emparejamiento por escala de grises. Otro bloque es el de los métodos basados en características, en los que

una búsqueda es usada para encontrar emparejamientos factibles entre las características del objeto y las características de la imagen. Dentro de este bloque se encuentran, por ejemplo, los árboles de interpretación, consistencia de pose, y *Scale-Invariant Feature Transform* (SIFT).

4.1. Detección con técnica SSD

El método utilizado, SSD, está diseñado para detectar objetos en imágenes utilizando una única red neuronal profunda. El enfoque SSD se basa en una red convolucional *feed-forward* que produce una colección de tamaño fijo de cajas delimitadoras (*bounding boxes*) y puntuaciones para la presencia de instancias de objetos en esas cajas, seguido de un paso de supresión no máxima para producir las detecciones finales. En el momento de la predicción, la red genera puntuaciones para la presencia de cada categoría de objeto en cada caja delimitadora predeterminada y produce ajustes en la caja para que coincida mejor con la forma del objeto. Además, la red combina predicciones de múltiples mapas de características con diferentes resoluciones para manejar de forma natural objetos de varios tamaños. Las primeras capas de red se basan en una arquitectura estándar utilizada para la clasificación de imágenes, similar a la explicada en el Capítulo 3. Posteriormente se añade una estructura auxiliar para producir las detecciones, teniendo en cuenta una serie de características:

- **Mapas de características de múltiples escalas para detección.** Se añaden capas de características convolucionales al final de la red base, cuyo tamaño disminuye progresivamente, permitiendo detecciones a múltiples escalas. El modelo convolucional para la predicción de las detecciones es diferente para cada capa característica.

- **Predictores convolucionales para la detección.** Cada capa de característica añadida puede producir un conjunto fijo de predicciones de detección usando un conjunto de filtros convolucionales, que son indicados en la parte superior de la arquitectura de red SSD.

- **Cajas por defecto y relaciones de aspecto.** Se asocia un conjunto de cajas delimitadoras predeterminadas con cada celda de mapa de características, para

varios mapas de características en la parte superior de la red. Las cajas por defecto anidan el mapa de características de una manera convolucional, de modo que la posición de cada caja con respecto a su celda correspondiente es fija. En cada celda de mapa de características se predicen los desplazamientos relativos a las formas de cajas predeterminadas en la celda, así como las puntuaciones por clase que indican la presencia de una instancia de clase en cada una de esas cajas.

En la Figura 4.1 se muestra un resumen del funcionamiento de este modelo. SSD sólo necesita una imagen de entrada y cajas de verdad para cada objeto durante el entrenamiento. De forma convolucional, se evalua un conjunto pequeño de cajas predeterminadas de diferentes relaciones de aspecto en cada ubicación, en varios mapas de características con diferentes escalas (por ejemplo, 8×8 en la Figura (b) y 4×4 en la Figura (c)). Para cada caja predeterminada se predice tanto los desplazamientos de forma como las confianzas para todas las categorías de objetos $(c_1, c_2, ..., c_p)$.

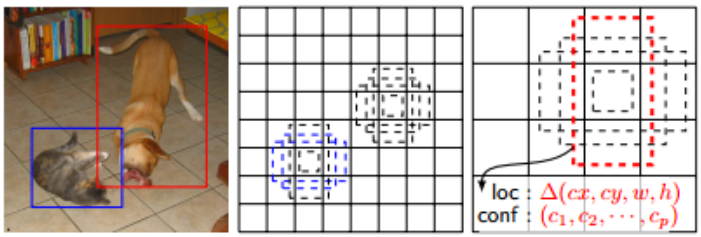


Figura 4.1: Modelo SSD. Imagen obtenida de [13]

Durante el entrenamiento es necesario determinar qué casillas por defecto corresponden a una detección de verdad, entrenando la red en consecuencia. Se comienza haciendo coincidir cada caja de verdad con la caja predeterminada con la mejor superposición de *Jaccard*, un estadístico utilizado para comparar la similitud y diversidad de conjuntos de muestras. Esto simplifica el problema de aprendizaje, permitiendo a la red predecir puntuaciones altas para múltiples cajas por defecto superpuestas, en lugar de requerir que elija sólo la que tenga superposición máxima. Tras el paso de la coincidencia, la mayoría de las cajas predeterminadas son negativas, especialmente cuando el número de cajas predeterminadas posibles es grande, introduciendo un desequilibrio significativo entre los ejemplos de entrenamiento positivos y negativos. Por ello es necesario realizar una minería negativa dura, de tal manera que, en lugar de usar todos los ejemplos negativos, se

clasifican usando la pérdida de confianza más alta para cada caja predeterminada y se seleccionan los más altos. Esto hace que la relación entre los negativos y los positivos sea como mucho 3:1, conduciendo a una optimización más rápida y un entrenamiento más estable.

Por último, para hacer que el modelo sea más robusto a los tamaños y formas de los objetos de entrada, cada imagen de entrenamiento se muestreará aleatoriamente.

4.2. SSD en Caffe

Una vez entendido el funcionamiento de las redes SSD, cuya información ha sido obtenida de [13], se pasará al uso de algunos ejemplos proporcionados por la propia rama de la plataforma Caffe. El motivo de usar modelos ya entrenados viene dado por la alta capacidad de cómputo para entrenar este tipo de redes, y al tiempo que implica el entrenamiento. En concreto se analizarán algunas imágenes con dos de las redes SSD que se proporcionan:

- **VOC0712-SSD300x300.** Este modelo ha sido entrenado con las bases de datos *Pascal VOC* de los años 2007 [16] y 2012 [17], explicadas en la Sección 2.2.3. Se utilizan imágenes de 300x300 píxeles y la red formada en la iteración 120000.
- **COCO-SSD300x300.** Este modelo ha sido entrenado con las bases de datos COCO, explicadas en la Sección 2.2.2. Se utilizan imágenes de 300x300 píxeles y la red formada en la iteración 400000.

Estos modelos, proporcionados por la rama SSD de Caffe, pueden ser descargados de su repositorio *Git Hub* [1] y utilizados para desarrollar la aplicación que sea de interés.

Al obtener los modelos se encuentra una carpeta con una serie de archivos que permiten comprobar la estructura de la red, el solucionador o el propio modelo entrenado, entre otros. La aplicación de interés se centra en el uso de tres de los archivos proporcionados:

- ***labelmap_voc.prototxt***, contiene la estructura de las diferentes etiquetas que pueden ser asignadas

[1] https://github.com/weiliu89/caffe/tree/ssd

- ***deploy.prototxt***, contiene la estructura del modelo entrenado
- ***model_iter_niter.caffemodel***, contiene los pesos del modelo, *model*, entrenado hasta la iteración *niter*

Una vez se dispone del modelo entrenado se desarrolla un pequeño *script* en Python, *image_detection.py*, que permite introducir una determinada imagen, realizar la detección sobre ella con el modelo escogido, y obtener una nueva imagen donde se marcan las detecciones realizadas según un determinado criterio. Para ello, se ha utilizado como guía el ejemplo proporcionado en el mismo *Git Hub* de la rama[2].

Para realizar la detección con el *script* mencionado, lo primero que se debe hacer es indicar la red SSD que se empleará. Para ello se utilizan unas líneas similares a las empleadas en la Sección 3.1.2.1, con alguna variación para adaptarla al problema de la detección, según se muestra a continuación:

```
labelmap_file = '.../labelmap_voc.prototxt'
file = open(labelmap_file, 'r')
self.labelmap = caffe_pb2.LabelMap()
text_format.Merge(str(file.read()), self.labelmap)

model_def = '.../deploy.prototxt'
model_weights = '.../VGG_VOC0712_SSD_300x300_iter_120000.caffemodel'

self.net = caffe.Net(model_def,  # defines the structure of the model
                     model_weights, # contains the trained weights
                     caffe.TEST)    # use test mode
```

Por otro lado, cabe destacar la función *detection(self,img)*, que realiza diferentes pasos para obtener las detecciones, que serán explicados a continuación.

- En primer lugar se realiza una **transformación de la imagen** para adaptarla a la entrada utilizada en el entrenamiento de la red. Para ello se emplean una serie de funciones que proporciona Caffe mediante la clase *Transformer*.

[2]https://github.com/weiliu89/caffe/blob/ssd/examples/ssd_detect.ipynb

- Una vez se obtiene la imagen adaptada a la entrada de la red, se debe **introducir esta imagen** de la misma manera que se realizó en la Sección 3.1.2.1, utilizando para ello la siguiente línea:

```
self.net.blobs['data'].data[...] = transformed_image
```

- Tras introducir la imagen se procede a **realizar la detección**. Para ello, se recogen las detecciones con la función que proporciona Caffe y la salida se parsea en diferentes vectores que contienen: las etiquetas, confianzas, y coordenadas de las cajas delimitadoras de forma ordenada. Este proceso se materializa en el *script* mediante el siguiente código.

```
# Forward pass.
detections = self.net.forward()['detection_out']
# Parse the outputs.
det_label = detections[0,0,:,1]
det_conf = detections[0,0,:,2]
det_xmin = detections[0,0,:,3]
det_ymin = detections[0,0,:,4]
det_xmax = detections[0,0,:,5]
det_ymax = detections[0,0,:,6]
```

- Una vez se obtienen los vectores con los parámetros de interés de las detecciones realizadas, es posible **implementar un filtro** que permita obtener las detecciones que se ajustan a las necesidades de la aplicación. Es muy común realizar este filtrado para obtener aquellas detecciones que superan un determinado umbral de confianza, dotando de mayor exactitud a la aplicación, así como por el objeto que sea de interés. En concreto, el siguiente fragmento de código realiza el filtrado para aquellas detecciones cuya confidencia supera el valor de 0.6.

```
top_indices = []
# Get detections with confidence higher than 0.6.
for i in range(0, len(det_conf)):
    if (det_conf[i] >= 0.6):
        top_indices.append(i)

top_conf = det_conf[top_indices]
top_label_indices = det_label[top_indices].tolist()
top_labels = self.get_labelname(self.labelmap, top_label_indices)
top_xmin = det_xmin[top_indices]
top_ymin = det_ymin[top_indices]
top_xmax = det_xmax[top_indices]
top_ymax = det_ymax[top_indices]
```

De esta manera se obtienen únicamente los vectores con los parámetros de aquellas detecciones que cumplen los requisitos. La función *get_labelname(self,labelmap, labels)* está definida en el propio *script* y realiza el paso de las etiquetas numéricas a los nombres correspondientes.

- Por último, se realizan los pasos necesarios para **incluir en la propia imagen el resultado de la detección**. En concreto, se dibujarán las cajas delimitadoras de los objetos detectados, así como la etiqueta considerada.

Una vez entendido el funcionamiento de este *script* se han realizado una serie de pruebas con imágenes obtenidas de Google para comparar el funcionamiento de los dos modelos mencionados, VOC0712-SSD300x300 y COCO-SSD300x300. En la Figura 4.2 se puede comprobar cómo, para diferentes imágenes introducidas en ambos modelos, los resultados obtenidos son diferentes.

(a) (b)

(c) (d)

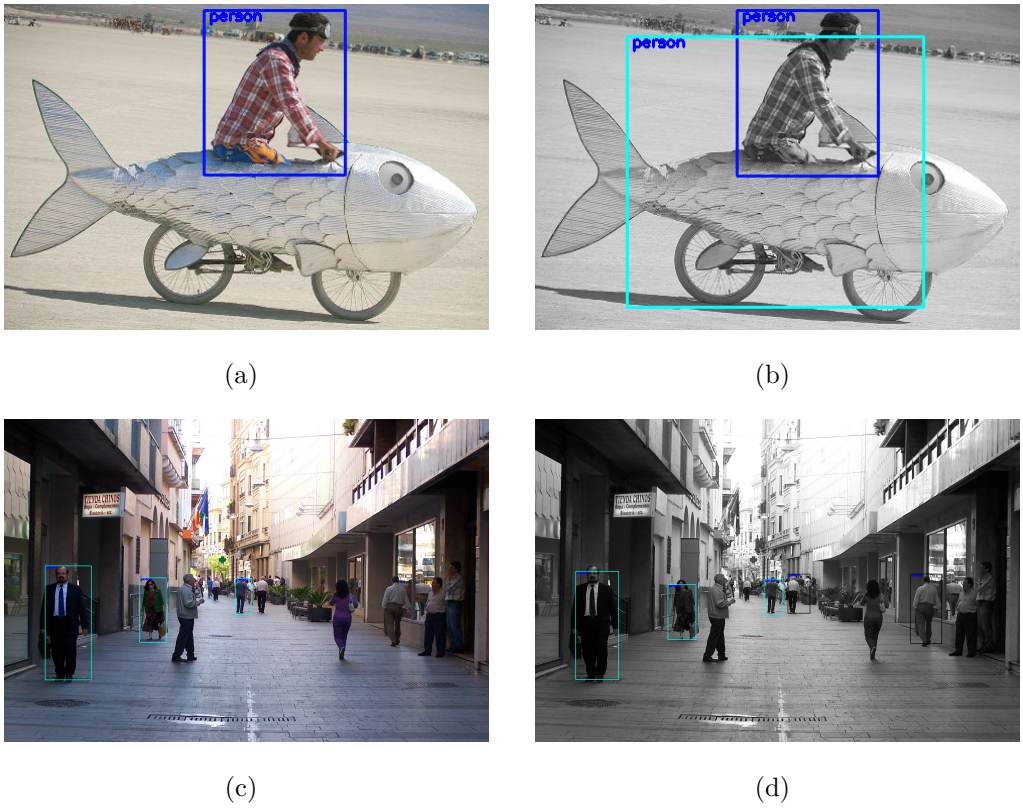
(e) (f)

Figura 4.2: Detección en distintas imágenes: (a) imagen sencilla con COCO, (b) imagen sencilla utilizando VOC, (c) imagen compleja utilizando COCO, (d) Imagen compleja utilizando VOC, (e) señal de tráfico utilizando COCO, (f) señal de tráfico utilizando VOC

La tendencia general sería que ambos modelos detectasen los mismos objetos en las mismas imágenes, siempre que el objeto haya sido incluido en el entrenamiento. Sin embargo se comprueba que ésto no es así. En una imagen bastante sencilla (a) y (b), el modelo de *Pascal VOC* es capaz de detectar a la persona, mientras que el modelo COCO no la detecta. Por el contrario, en un escenario un poco más complicado (c) y (d), el modelo VOC no detecta uno de los coches de la escena, mientras que el modelo COCO realiza la detección de todos los coches. Por último, en las imágenes (e) y (f), la señal

que aparece en primer plano queda identificada por COCO, pues entre sus etiquetas se incluyen este tipo de objetos, pero no por VOC, pues no incluyen etiquetas de este estilo.

Para tratar de obtener algún resultado concluyente se introdujo en la red una serie de imágenes de grises. Puesto que la red está entrenada con imágenes de color RGB será necesario replicar la propia imagen en las tres componentes para hacer coincidir las dimensiones de las imágenes y realizar la detección. En la Figura 4.3 se muestra cómo, con el modelo COCO (a) y (b), se comete un error al utilizar la imagen de grises que no fue cometido con la imagen RGB, equivocando la bicicleta con una persona. En el caso del modelo VOC, Figura 4.3 (c) y (d), se puede comprobar cómo se detecta un mayor número de personas al utilizar la imagen de grises que al emplear la imagen RGB.



(a) (b)

(c) (d)

Figura 4.3: Detección en distintas imágenes: (a) imagen RGB utilizando COCO, (b) imagen de grises utilizando COCO, (c) imagen RGB utilizando VOC, (d) imagen de grises utilizando VOC,

Se puede afirmar que el problema de detección con los ejemplos proporcionados por Caffe se resuelve de manera bastante satisfactoria. Al introducir a la red imágenes escogi-

das de forma totalmente aleatoria se observan buenos resultados en la mayoría de casos, pues se detecta correctamente un gran número de estímulos. Sin embargo, sobre qué red resulta mejor para la detección de un determinado estímulo no es posible establecer ninguna conclusión firme, pues es necesario un estudio más profundo.

Respecto a la importancia de la componente de color en la red entrenada, ocurre algo similar que en la comparación de ambas redes. Al introducir imágenes de grises en las redes entrenadas con color no existe ningún resultado que marque una evidencia sobre cómo de importante es esta componente en la detección. Es por esto que será necesario realizar un estudio más profundo de los modelos utilizados, con bases de datos de evaluación significativas, para obtener conclusiones sólidas sobre el enfoque SSD de Caffe.

Capítulo 5

Conclusiones

En este capítulo se exponen las conclusiones alcanzadas con el desarrollo del proyecto, las aportaciones principales, así como posibles líneas para continuar el trabajo realizado.

5.1. Conclusiones

El desarrollo de este trabajo ha permitido llegar una serie de conclusiones en el ámbito de aprendizaje profundo mediante el empleo de redes neuronales entrenadas con Caffe. Son expuestas a continuación, divididas según los sub-objetivos que se establecieron en la Sección 1.2.

Estudio de la plataforma Caffe.

La plataforma Caffe posee un funcionamiento muy adecuado para el ámbito de la visión artificial gracias a un entrenamiento rápido y sencillo. También posee un gran número de ejemplos que hacen más sencillo el trabajo con la misma y permite abarcar problemas de clasificación y detección.

Desarrollo de componente JdeRobot clasificador de dígitos.

El componente Python programado resulta bastante robusto gracias a la integración de la mejor red obtenida tras realizar diferentes pruebas con Caffe. Este componente permite la clasificación de dígitos mostrados a una cámara RGB en tiempo real, obteniendo buenos resultados independientemente de la procedencia de los mismos.

Desarrollo de un banco de pruebas.

Se ha construido un banco de pruebas gracias a un conjunto sencillo de hojas de cálculo que implementan las fórmulas necesarias. Este banco ha sido utilizado como herramienta para obtener las medidas de prestaciones necesarias para comparar las distintas redes neuronales.

Estudio y mejora de redes neuronales para la clasificación de dígitos.

Este sub-objetivo ha sido el bloque central del trabajo, por lo que el número de conclusiones alcanzadas sobre el mismo es superior a los demás. Tras alcanzar el sub-objetivo se ha concluido que:

- El entrenamiento con imágenes de borde amplía el número aciertos en la clasificación, pues independiza la representación del contenido en la imagen de los niveles de intensidad.
- Al realizar una serie de transformaciones en las imágenes y considerarlas en el entrenamiento de la red las prestaciones de la red mejoran, permitiendo obtener una red más robusta.
- Las prestaciones obtenidas al incluir un alto número de imágenes transformadas frente a la que se obtiene disminuyendo ese número de imágenes es similar, por lo que no es necesario incluir un gran número de estas imágenes en el entrenamiento.
- Sobre la inclusión de la imagen limpia en el entrenamiento, siendo ésta la imagen sin ninguna transformación, los resultados avalan que no es necesario incluirla para obtener mejores prestaciones.

Primera aproximación a la detección con Caffe.

Los resultados obtenidos en este ámbito hacen ver que el problema de detección es muy amplio. Según la base de datos empleada en el entrenamiento, la estructura de la red y el estímulo que se desee detectar, los resultados de la detección y las prestaciones de la red varían. La plataforma Caffe proporciona varias herramientas que permiten implementar redes neuronales para la detección SSD.

5.2. Líneas futuras

Para continuar con la investigación abordada en este trabajo se pueden seguir varias vías que permitan obtener resultados interesantes en el campo del aprendizaje profundo.

- En la tarea de clasificación, es posible extender lo aprendido en un ejemplo sencillo, la clasificación de dígitos con redes neuronales gracias al conjunto de datos MNIST, a un ejemplo más complejo como puede ser la clasificación de signos o de señales de tráfico.

- Los resultados obtenidos en el campo de la detección dejan ver por delante una gran línea de investigación antes de obtener una red robusta para la detección de estímulos interesantes.

- Es posible realizar un estudio de los mismos problemas tratados en este trabajo con herramientas diferentes, como TensorFlow. De esta manera, es posible contrastar distintas redes para la solución de un mismo problema, permitiendo la elección de aquella más adecuada a cada problema.

- Es posible realizar el estudio de redes desarrolladas con Caffe para estímulos no visuales, que permitan solucionar problemas de distinta naturaleza.

- En relación al banco de pruebas, sería interesante integrar las redes desarrolladas con Caffe en un banco automático, como el desarrollado por David Pascual[1], que agilice la obtención de parámetros de evaluación.

- Sería interesante la elaboración de una aplicación que permitiese a un coche circular de forma autónoma, detectando los estímulos que recibe de la carretera y clasificándolos para actuar en consecuencia. En esta línea de investigación se encuentra el proyecto Udacity, que proporciona una base de datos con varias horas de conducción para facilitar el desarrollo de una red con Aprendizaje Profundo que ayude en la percepción visual necesaria para la conducción autónoma con aprendizaje profundo.

[1] https://github.com/RoboticsURJC-students/2016-tfg-david-pascual

Bibliografía

[1] Anyu. "RNA – Redes Neuronales Artificiales", *Bitácoras de un Ingeniero*. http://andrealezcano.blogspot.com.es/2011/04/rna-redes-neuronales-artificiales.html, 2011. [Accedido 25 de Junio de 2017].

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] M.Eng. Dewi Suryani, S.Kom. "CONVOLUTIONAL NEURAL NETWORK", *Binus University - School of Computer Science*. http://socs.binus.ac.id/2017/02/27/convolutional-neural-network/, 2017. [Accedido 20 de Junio de 2017].

[4] D. Wang, A. Khosla, R. Gargeya, H. Irshad, and A. H. Beck. Deep Learning for Identifying Metastatic Breast Cancer. *ArXiv e-prints*, June 2016.

[5] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. *ArXiv e-prints*, May 2015.

[6] Otavio Good. "How Google Translate squeezes deep learning onto a phone", *Google Research Blog*. https://research.googleblog.com/2015/07/how-google-translate-squeezes-deep.html, 2015. [Accedido 25 de Junio de 2017].

[7] A. Owens, P. Isola, J. McDermott, A. Torralba, E. H. Adelson, and W. T. Freeman. Visually Indicated Sounds. *ArXiv e-prints*, December 2015.

[8] Clayton Mellina Rob Hess and Friends. "Introducing: Flickr PARK or BIRD", *Flickr*. http://code.flickr.net/2014/10/20/introducing-flickr-park-or-bird/, 2014. [Accedido 13 de Junio de 2017].

[9] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P.Dollár. Microsoft COCO: Common Objects in Context. *ArXiv e-prints*, May 2014.

[10] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[11] Tom Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27(8):861 – 874, 2006. ROC Analysis in Pattern Recognition.

[12] B. Kolo. *Binary and Multiclass Classification*. Weatherford Press, 2011.

[13] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: Single Shot MultiBox Detector. *ArXiv e-prints*, December 2015.

[14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[15] Andreas Veit, Tomas Matera, Lukas Neumann, Jiri Matas, and Serge Belongie. Coco-text: Dataset and benchmark for text detection and recognition in natural images. In *arXiv preprint arXiv:1601.07140*, 2016.

[16] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results. http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html.

[17] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[18] L.L. Pullum, B.J. Taylor, and M.A. Darrah. *Guidance for the Verification and Validation of Neural Networks*. Emerging Technologies. Wiley, 2007.

[19] C. Sammut and G.I. Webb. *Encyclopedia of Machine Learning*. Encyclopedia of Machine Learning. Springer US, 2011.

[20] R.F. López and J.M.F. Fernández. *Las Redes Neuronales Artificiales*. Metodología y Análisis de Datos en Ciencias Sociales. Netbiblo, 2008.

[21] José Jaime Esqueda Elizondo and Luis Enrique Palafox Maestre. *Fundamentos para el procesamiento de imágenes*. Uabc.

[22] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986.

[23] OpenCV. "Canny Edge Detection", *OpenCV Python Tutorials*. `http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html`, 2017. [Accedido 14 de Junio de 2017].

[24] R.C. Gonzalez and R.E. Woods. *Digital Image Processing*. Pearson/Prentice Hall, 2008.

[25] OpenCV. "Laplace Operator", *OpenCV Tutorials*. `http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html`, 2017. [Accedido 14 de Junio de 2017].

[26] OpenCV. "Sobel Derivatives", *OpenCV Tutorials*. `http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html`, 2017. [Accedido 18 de Junio de 2017].

[27] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *ECCV*, 2016.