



ESCUELA TECNICA 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

Tutora: Inmaculada Mora Jiménez

Co-tutor: José María Cañas Plaza

Curso académico 2016/2017



©2017 Nuria Oyaga de Frutos

Esta obra estÃ¡ distribuida bajo la licencia de
“Reconocimiento-CompartirIgual 4.0 Internacional (CC BY-SA 4.0)”

de Creative Commons.

Para ver una copia de esta licencia, visite
<http://creativecommons.org/licenses/by-sa/4.0/> o envíe
una carta a Creative Commons, 171 Second Street, Suite 300,
San Francisco, California 94105, USA.

Agradecimientos

Resumen

Índice general

1. Introducción	1
1.1. Contexto y motivación	1
1.2. Objetivos	1
1.3. Metodología	1
1.4. Estructura de la memoria	1
2. Infraestructura	2
2.1. Software	2
2.1.1. JdeRobot	2
2.1.2. Caffe	4
2.1.3. DroidCam	8
2.2. Bases de datos	9
2.2.1. MNIST	9
2.2.2. COCO	10
2.2.3. VOC	12
2.3. Evaluación de prestaciones	14
2.3.1. Matriz de confusión	14
2.3.2. <i>Precision</i>	15
2.3.3. <i>Recall</i>	16
3. Clasificación con Aprendizaje Profundo	17
3.1. Clasificador de dígitos	17
3.1.1. Red básica	18
3.1.1.1. Definición de la red	19
3.1.1.2. Definición del solucionador	25
3.1.1.3. Ejecución de la red	26
3.1.2. Componente Python	28
3.1.2.1. Cámara	29
3.1.2.2. GUI	31
3.1.2.3. Ejecución	32

3.2.	Banco de pruebas	34
3.2.1.	Obtención de datos de test	34
3.2.2.	Banco de pruebas manual	37
3.3.	Efectos del aprendizaje	40
3.3.1.	Modificación de bases de datos	40
3.3.1.1.	Imágenes de bordes	40
3.3.1.2.	Imágenes ruidosas en test	47
3.3.1.3.	Imágenes ruidosas en entrenamiento	51
3.3.2.	Número de iteraciones	57
3.4.	Experimentos	58
4.	Detección	61
5.	Conclusiones	62

Índice de figuras

2.1. Estructura y funcionamiento básico de red en Caffe. Figura obtenida de [13]	4
2.2. Función de activación <i>relu</i> .	6
2.3. Estructura básica de anotaciones en COCO.	11
2.4. Estructura de instancias de objetos en COCO.	12
2.5. Ejemplos de imágenes en VOC. Imagen tomada de [5]	13
2.6. Estructura de las clases en Visual Objects Classes (VOC)2007. Imagen tomada de [5]	14
2.7. Ejemplo sencillo de matriz de confusión. Imagen tomada de [9]	15
3.1. Muestras de base de datos MNIST original.	19
3.2. Red básica LeNet MNIST.	24
3.3. Ejecución de entrenamiento de red LeNet MNIST.	27
3.4. Fin de entrenamiento de red LeNet MNIST.	27
3.5. Archivos log.	28
3.6. Captura de componente gráfico de la aplicación.	32
3.7. Capturas de DroidCam en los diferentes dispositivos	33
3.8. Captura del componente clasificador.	34
3.9. Muestras de base de datos con negativo.	42
3.10. Porcentaje de acierto de base de datos original y ampliada con negativo.	42
3.11. Muestra de dígitos con filtro Canny.	44
3.12. Muestra de dígitos con filtro Laplaciano.	44
3.13. Muestra de dígitos con filtro de Sobel.	45
3.14. Comparación de tasa de acierto con diferentes filtros.	46
3.15. Captura del componente clasificador con filtro Sobel.	47
3.16. Muestra de dígitos rotados.	48
3.17. Muestra de dígitos trasladados.	48
3.18. Muestra de dígitos escalados.	49
3.19. Muestra de dígitos con ruido.	49
3.20. Muestra de dígitos con mezcla de transformaciones.	50
3.21. Evaluación de la red con bases de datos transformadas.	50

3.22. Resultados de parámetros de evaluación: (a) <i>Precision</i> , (b) <i>Recall</i>	57
3.23. Red básica LeNet MNIST.	58
3.24. Evaluación de la aplicación con imagen de fondo negro.	59
3.25. Evaluación de la aplicación con diferentes redes.	59

Índice de tablas

3.1. Estructura de conjuntos de datos.	18
3.2. Matriz de confusión red 1-0.	52
3.3. Matriz de confusión red 1-6.	53
3.4. Matriz de confusión red 1-1.	54
3.5. Matriz de confusión red	55
3.6. Matriz de confusión red	56

Acrónimos

COCO Common Objects in Context.

JSON JavaScript Object Notation.

MNIST Modified National Institute of Standards and Technology.

NIST National Institute of Standards and Technology.

ReLU Rectified Linear Unit.

RLE Run-Length Encoding.

VOC Visual Objects Classes.

Capítulo 1

Introducción

1.1. Contexto y motivación

1.2. Objetivos

1.3. Metodología

1.4. Estructura de la memoria

Imagen Estatica Movimiento

Clasificacion

Deteccion

Deep Learning

Tecnicas existentes: Caffe

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 del comportamiento de la propia red y, así, emplear la más adecuada. Por último, serán expuestas 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. JdeRobot

JdeRobot¹ es una plataforma de software libre que facilita la tarea de los desarrolladores en el campo de la robótica, visión por computador y otras relacionadas, siendo este su principal fin.

Está escrito en su mayoría en el lenguaje C ++ y proporciona un entorno de programación basado en componentes distribuidas, de tal manera que una aplicación está formada por una colección de varios componentes asincrónos y concurrentes. Esta estructura per-

¹<http://jderobot.org>

mite la ejecución de los distintos componentes en diferentes equipos, estableciendo una conexión entre ellos mediante el middleware de comunicaciones ICE. Además, se obtiene 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.

A pesar de que 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, su uso no es la verdadera finalidad del proyecto, ya que únicamente se centrará en la utilización de uno de sus componentes para facilitar la obtención de las imágenes.

Camera Server

Se trata de un componente que permite servir a un número determinado de cámaras, ya sean reales o simuladas, a partir 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. Caffe

Caffe [7] 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, escrito 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 mayor 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.

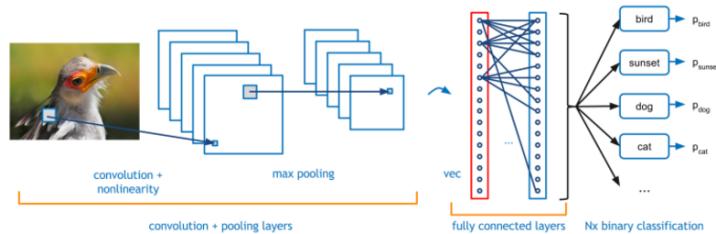


Figura 2.1: Estructura y funcionamiento básico de red en Caffe. Figura obtenida de [13]

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 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 *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 *bias*.

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 sobre ajuste. 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.

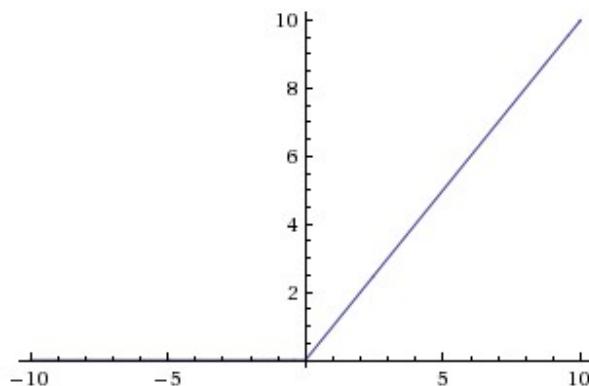


Figura 2.2: Función de activación *relu*.

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 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 $l_n \in [0, 1, 2, \dots, K - 1]$, 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 explicación 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. DroidCam

DroidCam² 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

²<https://www.dev47apps.com/>

tener instalada la misma aplicación.

Entre sus características principales destacan:

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

2.2. Bases de datos

2.2.1. MNIST

La base de datos *Modified National Institute of Standards and Technology (MNIST)*³ 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.

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.

³<http://yann.lecun.com/exdb/mnist/>

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)*⁴ es un gran conjunto de datos de imágenes diseñado para la detección de objetos, segmentación y generación de subtítulos [16]. 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

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 2.3.

⁴<http://mscoco.org/>

```
{  
    "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 2.3: 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 2.4. 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 etiquetar 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 2.4: Estructura de instancias de objetos en COCO.

2.2.3. VOC

El objetivo del desafío de *Visual Objects Classes (VOC)* [5] 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 2.5, formando el conjunto de datos *VOC2007* [3]. 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, aproximadamente, 12.000 objetos diferentes, por lo que, en total, este conjunto dispone de unas 10000 imágenes en las que se representan unos 24000 objetos. En el año 2012 se modifica este conjunto, aumentando a 11530 el número de imágenes con representación de 27450 objetos diferentes [4].

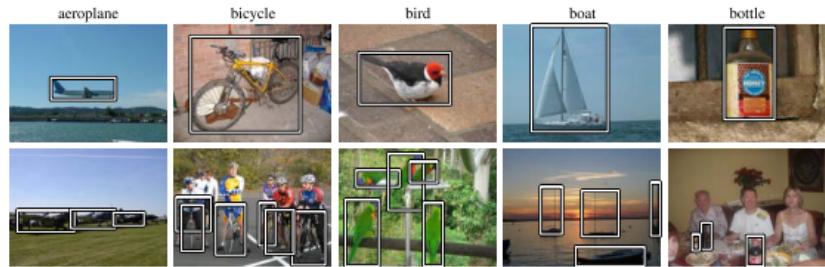


Figura 2.5: Ejemplos de imágenes en VOC. Imagen tomada de [5]

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 están clasificados en 20 clases diferentes. En la Figura 2.6 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.

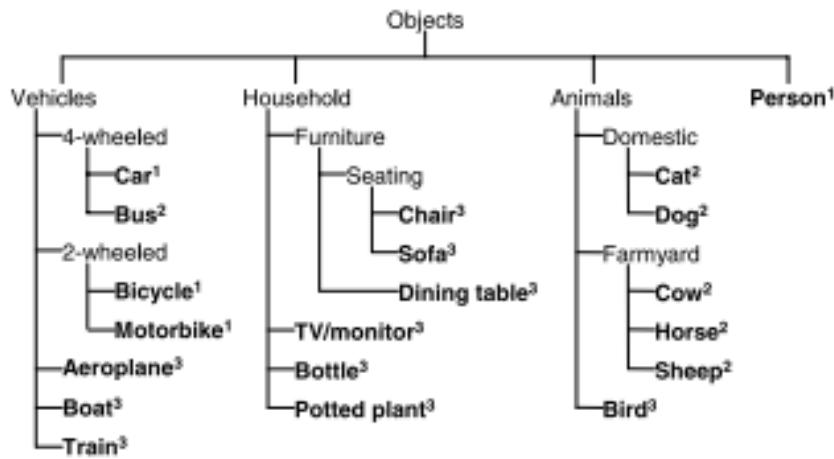


Figura 2.6: Estructura de las clases en Visual Objects Classes (VOC)2007. Imagen tomada de [5]

. El año de inclusión de cada clase en el desafío está indicado por superíndices: 2005¹, 2006², 2007³.

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* [12].

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 (Kohavi y Provost, 1998) contiene información sobre las clasificaciones reales y predichas hechas por un sistema de clasificación. El rendimiento de estos sistemas se evalúa comúnmente utilizando los datos de la matriz [9]. En la Figura 2.7 se muestra un ejemplo sencillo de la construcción de esta matriz donde:

- **a** es el número de predicciones correctas de que una instancia es negativa
- **b** es el número de predicciones incorrectas de que una instancia es positiva

- **c** es el número de predicciones incorrectas de una instancia negativa
- **d** es el número de predicciones correctas de que una instancia es positiva.

		Predicted	
		Negative	Positive
Actual	Negative	a	b
	Positive	c	d

Figura 2.7: Ejemplo sencillo de matriz de confusión. Imagen tomada de [9]

Además de ayudar con el cálculo de *precision* y *recall*, es importante mirar la matriz de confusión para analizar sus resultados, ya que también proporciona información importante sobre dónde el clasificador funciona mal [6]. Para que una matriz proporcione información sobre el correcto funcionamiento de un clasificador se obtendrán valores altos en la diagonal, y lo más pequeños posible en el resto de celdas de la misma.

2.3.2. *Precision*

Se trata de la proporción de los casos positivos predichos que fueron correctos [9]. Para obtener este valor en el ejemplo sencillo anteriormente explicado se utiliza la fórmula:

$$P = \frac{d}{b + d}$$

Donde P es el valor de *precision* y b y d los mismo valores explicados en la sección anterior.

En un caso más complejo en el que la clasificación no sea binaria, se deben de tener en cuenta todas las clases, por ello la fórmula queda expresada de la siguiente forma [6]:

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

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*

Se trata de la proporción de casos positivos que fueron correctamente identificados [9]. Para el ejemplo sencillo se calcula mediante la siguiente expresión:

$$R = \frac{d}{c + d}$$

Donde R es el valor de *recall* y c y d los mismo valores explicados anteriormente.

Al igual que en el caso de la sección anterior, en caso de tener una clasificación multiclase se deberás tener en cuenta todas ellas, utilizando para ello la expresión [6]:

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

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 una plataforma 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, que ha sido mejorado gracias a un amplio estudio sobre las variantes posibles aplicadas a las redes entrenadas, utilizando la plataforma Caffe.

3.1. Clasificador de dígitos

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

3.1.1. Red básica

La red que se empleará, 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 entrará en detalle a continuación.

La base de datos explicada, MNIST, proporciona dos conjuntos de datos, uno de entrenamiento y otro de test, pero, a diferencia de otros conjuntos no dispone de una de validación, utilizada para evaluar el modelo durante el entrenamiento. Por ello, el primer paso que se derá 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, *createvalidationdatabase.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.

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 el dividir la base de datos para no alterar la naturaleza de la base de datos original. Para mantener esta proporción se calculará el porcentaje sobre el total de cada dígito y no sobre el total del conjunto de datos.

En la Figura 3.1 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 el resto de bases de datos explicadas.

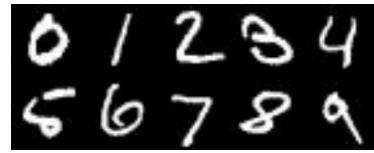


Figura 3.1: Muestras de base de datos MNIST original.

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 editarán para adaptar la red al problema que se abarque. 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 emplearán, la propia estructura de la red y la forma en la que se analizarán 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 las perceptrones multicapa convencionales. Con Caffe la estructura habitual de la red LeNet se ve ligeramente modificada, pues se utiliza 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 utilizarán 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 preprocessamiento 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 reales. 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 utilizarán dos capas de datos que difieren en la fase en la que se utilizan los datos, entrenamiento o evaluación 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 obtendrán 20 salidas en la primera de ellas, en la segunda, sin embargo, se obtendrán 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 *bias* 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, será alimentada por la capa de convolución anterior y alimentará a la siguiente en caso de que la haya. En caso de ser la última de las capas de agrupamiento sus salidas serán la entrada de las capas completamente conectadas cuya estructura se explicará más adelante en esta sección y que fueron detalladas en la Sección 2.1.2. Se definen en ella 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. A continuación se muestra un ejemplo de cómo se define esta capa en el documento que se está tratando.

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

Como se mencionó anteriormente, tras estas capas se establecen las dos capas completamente conectadas, *InnerProduct* 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 deberá de tener 10 salidas.

Las capas completamente conectadas están separadas entre sí por una capa de activación, en este caso linear, 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"
}
```

Esta capa 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. Estas capas, se deben añadir como una capa más a continuación de las anteriormente explicadas y se definirá 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 3.2 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 siguiente comando:

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

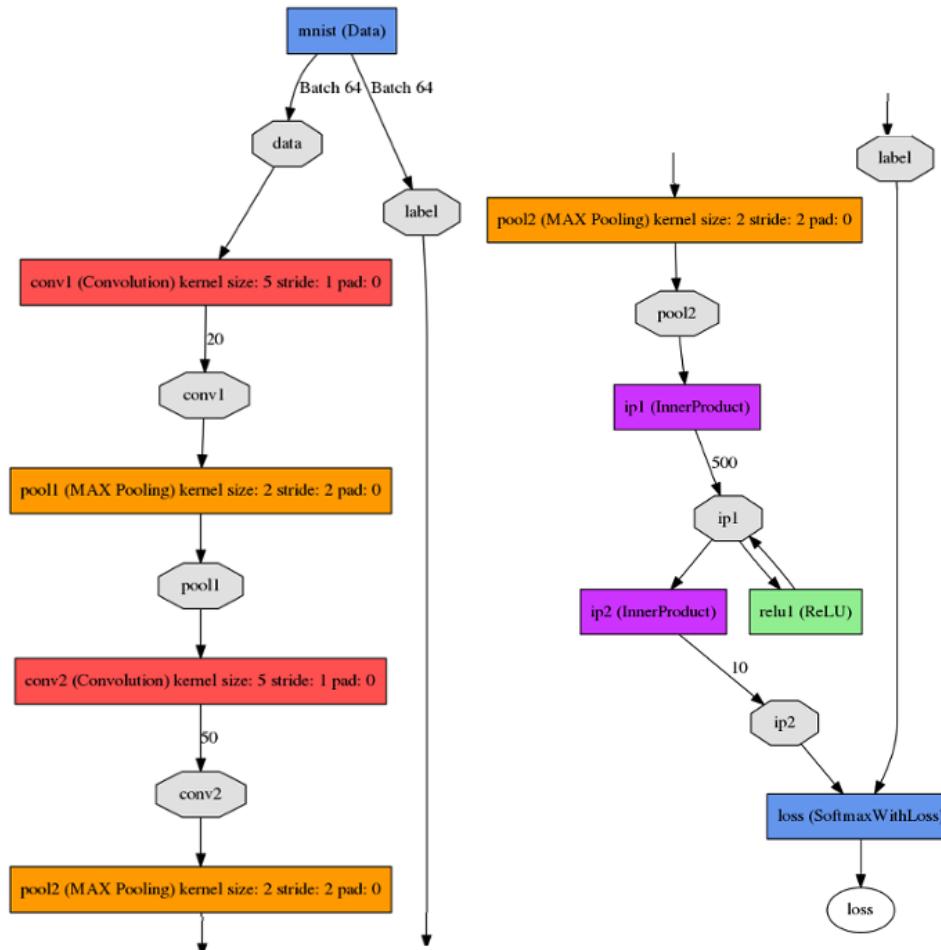


Figura 3.2: Red básica LeNet MNIST.

3.1.1.2. Definición del solucionador

Para esta tarea se va a utilizar el archivo de Caffe *lenet_solver.prototxt*. En este documento se definen parámetros como la estructura de red que se utilizará, definida en el apartado anterior, y el número de iteraciones que se ejecutarán 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 manejarán en este proyecto, como la evaluación de la red o las redes intermedias que se guardarán.

```
# 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 fijar 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 realizará 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 cuántas iteraciones se creará un archivo con la red intermedia correspondiente, y, finalmente, *snapshot_prefix*, que marcará la ruta en la que se almacenarán los archivos. La forma en la que se almacenan los archivos se corresponden con la ruta indicada hasta el último “/”, siendo lo posterior el nombre deseado que será completado con el número de iteración correspondiente. En el ejemplo mostrado, el archivo será almacenado en la ruta *”examples/mnist/”* y el nombre de la red final será *”lenet_iter_10000”*.

Tras definir el solucionador se procederá 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 el solucionador, se deben ejecutar los siguientes comandos:

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

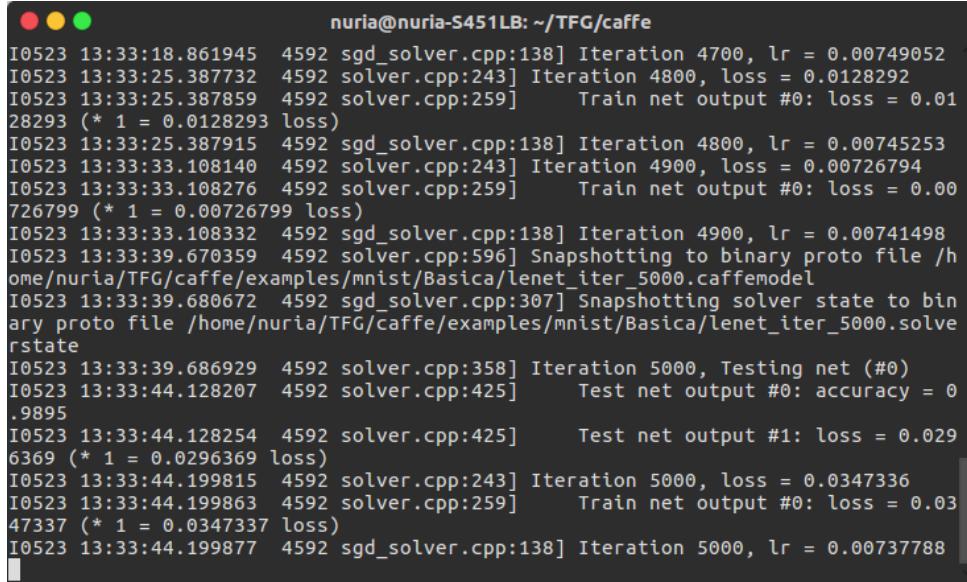
El archivo que se ejecuta contiene información sobre qué solucionador se debe implementar y el modo de ejecución, de la forma que se muestra a continuación.

```
#!/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.

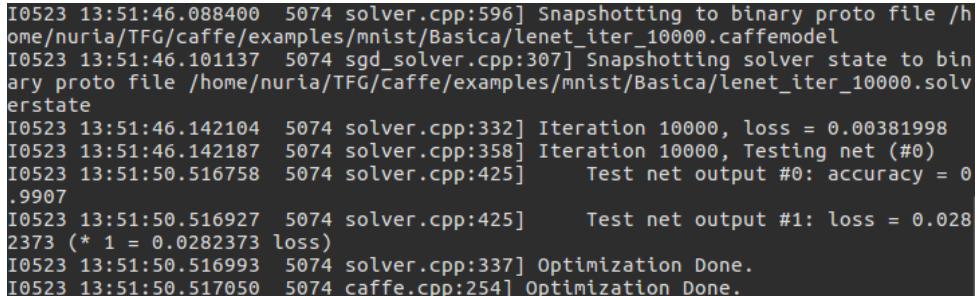
```
2>&1 | tee /home/nuria/TFG/logs/RedBasica.log $@
```

En la Figura 3.3 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.



```
nuria@nuria-S451LB: ~/TFG/caffe
I0523 13:33:18.861945 4592 sgd_solver.cpp:138] Iteration 4700, lr = 0.00749052
I0523 13:33:25.387732 4592 solver.cpp:243] Iteration 4800, loss = 0.0128292
I0523 13:33:25.387859 4592 solver.cpp:259] Train net output #0: loss = 0.01
28293 (* 1 = 0.0128293 loss)
I0523 13:33:25.387915 4592 sgd_solver.cpp:138] Iteration 4800, lr = 0.00745253
I0523 13:33:33.108140 4592 solver.cpp:243] Iteration 4900, loss = 0.00726794
I0523 13:33:33.108276 4592 solver.cpp:259] Train net output #0: loss = 0.00
726799 (* 1 = 0.00726799 loss)
I0523 13:33:33.108332 4592 sgd_solver.cpp:138] Iteration 4900, lr = 0.00741498
I0523 13:33:39.670359 4592 solver.cpp:596] Snapshotting to binary proto file /h
ome/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_5000.caffemodel
I0523 13:33:39.680672 4592 sgd_solver.cpp:307] Snapshotting solver state to bin
ary proto file /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_5000.solve
rstate
I0523 13:33:39.686929 4592 solver.cpp:358] Iteration 5000, Testing net (#0)
I0523 13:33:44.128207 4592 solver.cpp:425] Test net output #0: accuracy = 0
.9895
I0523 13:33:44.128254 4592 solver.cpp:425] Test net output #1: loss = 0.029
6369 (* 1 = 0.0296369 loss)
I0523 13:33:44.199815 4592 solver.cpp:243] Iteration 5000, loss = 0.0347336
I0523 13:33:44.199863 4592 solver.cpp:259] Train net output #0: loss = 0.03
47337 (* 1 = 0.0347337 loss)
I0523 13:33:44.199877 4592 sgd_solver.cpp:138] Iteration 5000, lr = 0.00737788
```

Figura 3.3: Ejecución de entrenamiento de red LeNet MNIST.



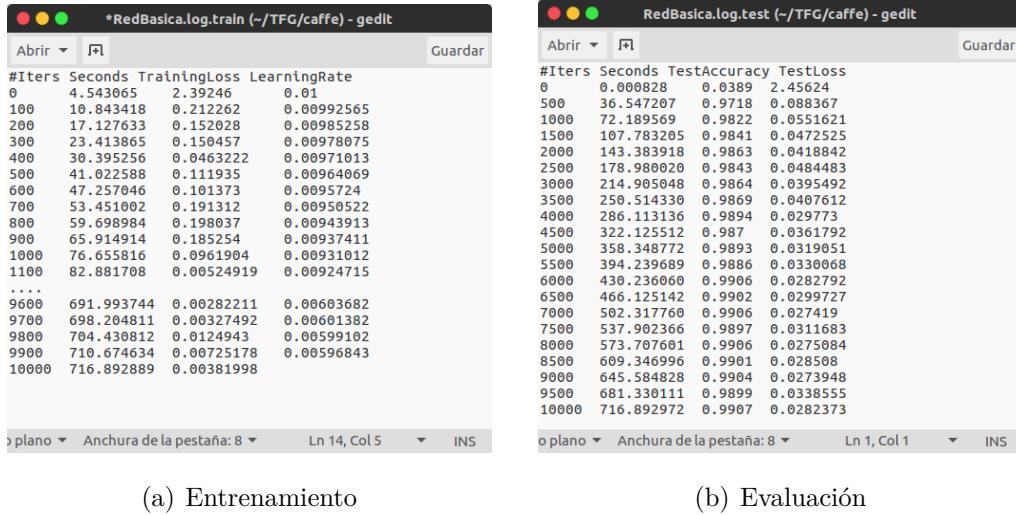
```
I0523 13:51:46.088400 5074 solver.cpp:596] Snapshotting to binary proto file /h
ome/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_10000.caffemodel
I0523 13:51:46.101137 5074 sgd_solver.cpp:307] Snapshotting solver state to bin
ary proto file /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_10000.solv
erstate
I0523 13:51:46.142104 5074 solver.cpp:332] Iteration 10000, loss = 0.00381998
I0523 13:51:46.142187 5074 solver.cpp:358] Iteration 10000, Testing net (#0)
I0523 13:51:50.516758 5074 solver.cpp:425] Test net output #0: accuracy = 0
.9907
I0523 13:51:50.516927 5074 solver.cpp:425] Test net output #1: loss = 0.028
2373 (* 1 = 0.0282373 loss)
I0523 13:51:50.516993 5074 solver.cpp:337] Optimization Done.
I0523 13:51:50.517050 5074 caffe.cpp:254] Optimization Done.
```

Figura 3.4: Fin de entrenamiento de red LeNet MNIST.

Tras terminar el entrenamiento, mostrado en la Figura 3.4, 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, queda almacenados en el archivo *log* generado, y serán 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/extrá*. En la Figura 3.5 se muestra el aspecto de estos archivos desglosados.



#Iters	Seconds	TrainingLoss	LearningRate
0	4.543065	2.39246	0.01
100	18.843418	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
11000	82.881708	0.00524919	0.00924715
...			
9600	691.993744	0.00282211	0.00603682
9700	698.204811	0.00327492	0.00601382
9800	764.430812	0.0124943	0.00599102
9900	710.674634	0.00725178	0.00596843
10000	716.892889	0.00381998	

#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	250.514338	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

(a) Entrenamiento

(b) Evaluación

Figura 3.5: Archivos log.

3.1.2. Componente Python

Se ha desarrollado 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, que se especificará en un archivo de configuración, en tiempo real, encendiendo una bombilla que se corresponde con el número obtenido.

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 encargará 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 encargará 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 podrá ser encontrado en [?].

3.1.2.1. Cámara

El hilo fundamental de la aplicación, que se encargará de la lógica de la misma mediante la adquisición de la imagen y su posterior procesamiento, estará 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 indicará en la ejecución de la aplicación. La línea que indica la cámara en este archivo es la siguiente:

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

Esta propiedad estará enlazada con el componente *Camera Server* de JdeRobot que nos proporciona un servidor de imágenes mediante la cámara.

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 = '/home/nuria/TFG/caffe/examples/mnist/lenet.prototxt'  
pretrained_file = '/home/nuria/TFG/caffe/examples/mnist/Basica/  
                    /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 utilizará. En primer lugar, se indica cuál será el modelo empleado para la clasificación. Este modelo es un archivo proporcionado 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 exactamente iguales a la estructura de la red entrenada para que no se produzcan errores. En segundo lugar, se indica la red entrenada que se utilizará 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, en primer lugar, 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 serán importantes para la ejecución de la aplicación. Se establece una función *update(self)*, que será llamada cada 150ms para la actualización del hilo *ThreadCamera(camera)*, creado en el componente principal, para obtener las imágenes de forma periódica y poder 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, original y 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), y por último, se le aplica un filtro gaussiano de 5x5 para reducir el ruido.

Finalmente, se crea la siguiente función para realizar la clasificación de los dígitos:

```
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
```

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, pues eso fue lo que se indicó en 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, es decir, la clasificación realizada, y se devuelve.

Una vez establecida la lógica de la aplicación, con las funciones explicadas anteriormente, 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. GUI

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

Al igual que en el caso de la cámara, se establecerá un hilo que permita 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 poder mostrarlas en las etiquetas 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 considerará correcto, encendiendo la bombilla adecuada.

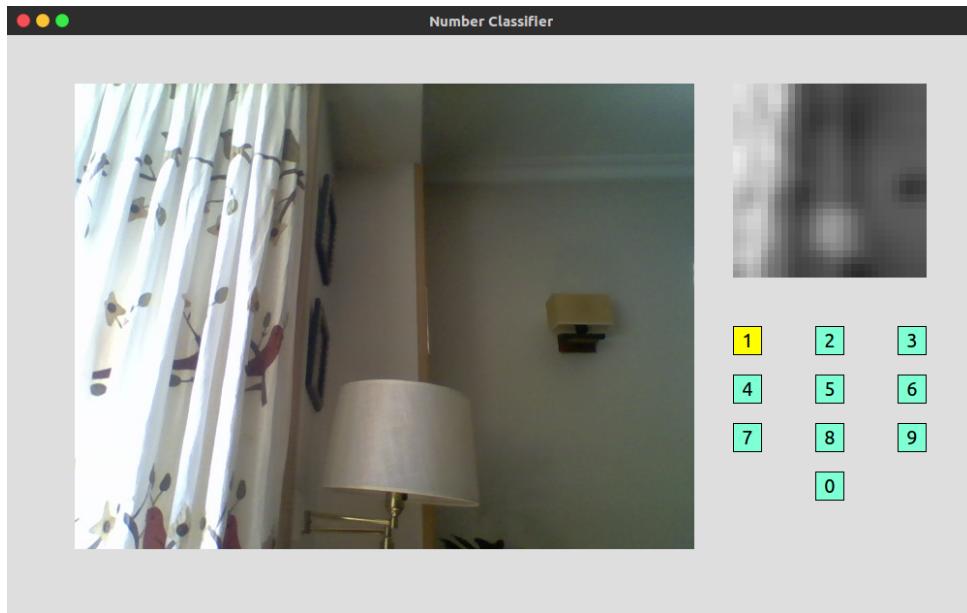


Figura 3.6: Captura de componente gráfico de la aplicación.

3.1.2.3. Ejecución

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

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

```
cameraserver cameraserver.cfg
```

En el desarrollo de este trabajo, la propiedad de interés del archivo de configuración es *CameraSrv.Camera.0.Uri*, que se centra en indicar la fuente de vídeo. Esta fuente puede ser un archivo de vídeo almacenado, para el que se empleará la ruta del archivo en ese campo, la webcam del propio ordenador, para el que se utilizará el valor 0, u otra cámara externa, para la que se le indicará el valor 1.

En la Sección 2.1.3 se comentó una aplicación que permitía utilizar la cámara 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 ¹, 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 estable la conexión y se obtienen los resultados de la Figura 3.7 para el ordenador y el dispositivo.

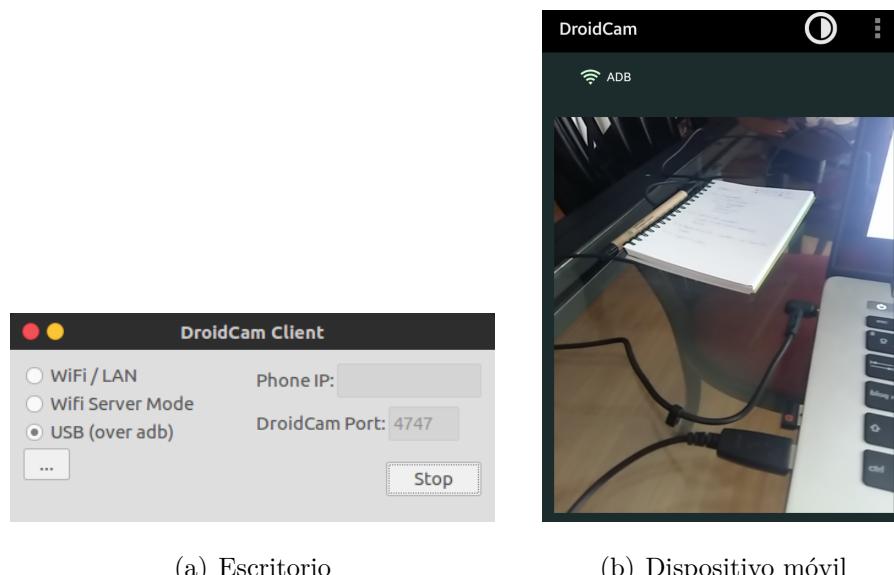


Figura 3.7: Capturas de DroidCam en los diferentes dispositivos

Tras tener en funcionamiento el servidor de imágenes se debe proceder a la ejecución del componente clasificador, para ello se ejecutará 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.

¹<https://www.dev47apps.com/droidcam/linuxx/>

Finalmente, tras la ejecución, obtenemos el resultado del componente mostrado en la Figura 3.8, donde se aprecia el funcionamiento del mismo para un número sencillo y perfectamente definido según el entrenamiento de la red, es decir, fondo negro y número blanco.

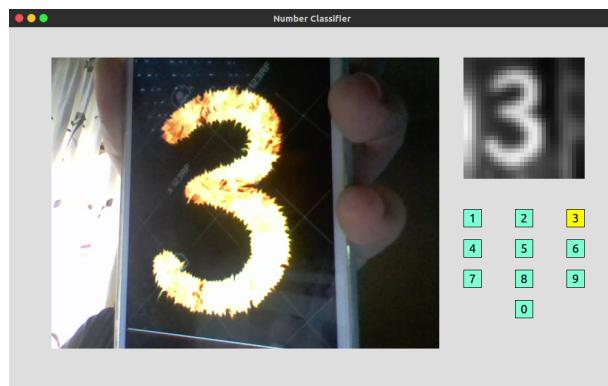


Figura 3.8: Captura del componente clasificador.

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 la evaluación de las diferentes redes neuronales que se desarrollarán en el proyecto se elabora un banco de pruebas que permite obtener los parámetros de evaluación explicados en el Capítulo 2, siendo necesario, previamente, la obtención de datos de clasificación sobre una determinada base de datos de test.

3.2.1. Obtención de datos de test

El primer paso para la elaboración de 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 deseada y obtener la clasificación para cada uno de los elementos existentes en

la misma. Este script está dividido en tres partes claramente diferenciadas que permite la obtención de los resultados finales y que serán detalladas a continuación.

Obtención de las imágenes

Las imágenes y sus correspondientes etiquetas están almacenadas en bases de datos de tipo *lmdb*. Este tipo de bases de datos requiere de un método específico para poder acceder al contenido de las mismas y poder manipular las imágenes que se almacenan en ellas, que queda definido a continuación.

```
lmdb_env = lmdb.open('/home/nuria/TFG/lmdb_test/test_lmdb')
lmdb_txn = lmdb_env.begin()
lmdb_cursor = lmdb_txn.cursor()
```

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

Para poder procesar los datos obtenidos anteriormente utilizando la plataforma Caffe, será necesario crearse una estructura *Datum* de la propia plataforma que incluirá, en cada iteración para recorrer la base de datos, la información de la instancia que se analiza. El siguiente código, crea la estructura indicada e indica 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, las etiquetas de las mismas (*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 hacer las comparaciones.

Clasificación de las imágenes

La tarea de clasificación se realizará exactamente 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.

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

Una vez se ha conseguido obtener el dígito que la red interpreta, se procede a las comparaciones para poder obtener datos más cómodos para la evaluación.

Comparación de datos

La tarea de comparación de los datos obtenidos por la red con los reales almacenados en la base de datos es bastante sencilla.

Se creará archivo de texto que incluirá una breve descripción del contenido y, para cada iteración, el número de iteración, la etiqueta real, la identificada por la red y un booleano que indicará si ambas etiquetas coinciden o no, todo ello separado por espacios, según se muestra a continuación.

```
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 datos por el banco de pruebas creado, además de un fácil entendimiento para el usuario que lea el archivo de lo que se está mostrando en él.

Una vez se ha obtenido el archivo con los datos necesarios para obtener valores que

ilustren sobre la robustez de la red, se procede a abarcar la manera en que se procesarán los mismos, obteniendo el banco de pruebas.

3.2.2. Banco de pruebas manual

Los datos de evaluación que se obtienen con este banco de pruebas son los explicados 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 uno de las redes intermedias que se obtienen durante el entrenamiento de la 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 formadas 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 de fallos tanto a nivel global, para obtener los parámetros de tasa de acierto o *accuracy*, como 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 lógico y 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 que se han utilizado. De esta manera se obtiene el porcentaje de imágenes que se han clasificado de forma correcta, valor que se corresponde con la tasa de acierto de la red.

Para realizar esta operación, el código empleado ha sido dividido en cuatro partes

diferenciadas:

- 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 se debe de tener en cuenta, para cada dígito real, 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 posibles en el mismo rango. En concreto, cada columna representa las veces que se introduce una imagen de cada uno de los dígitos y, cada fila, el número de veces que se predice uno de los dígitos.

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 corresponde con el dígito en cuestión 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 uno en la celda que se corresponda con el dígito real introducido en la red y la etiqueta resultante de la predicción.

Una vez se ha obtenido esta matriz, obtener los valores de *precision* y *recall* resulta bastante 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 debe dividir 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 dará por resultado el número de veces que se introdujo a la red cada uno de los dígitos. Una vez obtenido ese valor, se debe dividir 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

Existen numerosos factores que afectan a la robusted de la red en el proceso de entrenamiento de la misma. Elementos como la base de datos, el número de neuronas empleadas, el número de capas o las etapas que se realizan en el entrenamiento [8], hacen que la red tenga una mayor robusted, mejorando la aplicación deseada.

En esta sección se tratará el efecto en el aprendizaje de dos de los factores que se pueden manipular para adaptar la robusted de la red a la aplicación que se vaya a tratar, estos elementos son el cambio en las bases de datos de entrenamiento y validación, y la disminución del número de iteraciones.

3.3.1. Modificación de bases de datos

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

3.3.1.1. Imágenes de bordes

El primer problema que se encuentra en esta base de datos es que únicamente se dispone de muestras con el fondo negro y el dígito en blanco. Ésto limita bastante la funcionalidad de la aplicación, ya que se pretende clasificar cualquier dígito, independientemente del fondo sobre el que se muestre. Para estudiar el efecto que tiene el cambio de fondo en las imágenes en la red neuronal desarrollada se ha elaborado una base de datos de evaluación ampliada, incluyendo, para cada muestra, su negativo.

La obtención de la base de datos se consigue gracias al script *create_neg_database.py*. El proceso llevado a cabo en este script 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 utilizar los datos de la imagen obtenidos para realizar la transformación

deseada. Posteriormente, para almacenar las imágenes transformadas, se debe abrir una nueva base de datos de este tipo, que permita escritura. Esta apertura se realiza con las siguientes líneas:

```
new_lmdb_env = lmdb.open('/home/nuria/TFG/lmdb_test/test_edgesCanny_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 explicado en la misma sección mencionada, se debe almacenar la imagen original en la nueva base de datos, aplicar el negativo a la imagen, realizando la resta de 255 y los valores de la misma, y almacenar, también, la transformación.

Para insertar imágenes en una nueva base de datos es necesario realizar dos acciones, la inserción en la base de datos y la actualización de la misma. Esta inserción, al finalizar la interpretación de los datos de cada muestra de la base de datos original, se realiza mediante las siguientes líneas:

```
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 deben 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)
```

Estas 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 todas las inserciones realizadas, ahorrando carga computacional.

En la Figura 3.9 se muestra el negativo almacenado en la base de datos para cada dígito mostrado en la Figura 3.1. Estas imágenes han sido obtenidas con el script *data-read.py*, que lee las imágenes de la base de datos y crea un archivo para su visualización.

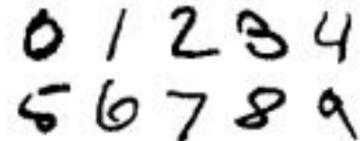


Figura 3.9: Muestras de base de datos con negativo.

Tras ejecutar este script se obtiene la base de datos de test con los negativos, la cual tendrá el doble de muestras que en el caso original, es decir 20000. Ésta es introducida en el banco de pruebas explicado y se obtienen valores de tasa de acierto.

En la Figura 3.10 se muestran los resultados obtenidos, donde se puede observar que la red falla considerablemente al incluir las imágenes en negativo. Se obtiene un porcentaje de acierto cercano al 60 %, lo que se corresponde, en su práctica totalidad, a la clasificación correcta de las imágenes originales.

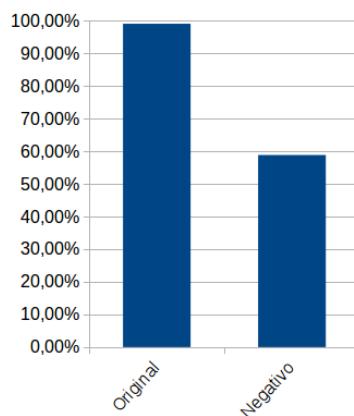


Figura 3.10: Porcentaje de acierto de base de datos original y ampliada con negativo.

Para solucionar el problema que acarrea el tener esta gran diferencia únicamente modificando el fondo de la imagen, y puesto que la aplicación no está enfocada a un único tipo de fondo, se opta por aplicar un filtro de bordes que independice la imagen del fondo. Existen varios filtros de bordes que es posible aplicar para solucionar el problema [1].

Para desarrollar la comparación entre los diferentes filtros posibles 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 de la que se parte en la base de datos. Será necesario aplicar el filtro, también sobre las bases de datos 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 que han sido 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 [2]. 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* según [10]. En la Figura 3.11 se muestra la aplicación de este filtro para cada dígito mostrado en la Figura 3.1. En la base de datos de test, se van a obtener dos imágenes iguales de cada dígito ya que se está aplicando sobre el original y el negativo el mismo filtro, que por su propio funcionamiento, detecta los mismos bordes en ambos.



Figura 3.11: Muestra de dígitos con filtro Canny.

Filtro Laplaciano

El laplaciano es un operador de segunda derivada que se utiliza con frecuencia en la detección de bordes [17]. Su fundamento se encuentra en la identificación de un borde cuando se produce un cruce por cero en la segunda derivada obtenida. Este operador posee dos filtros diferentes, uno positivo con el que se obtienen los bordes externos, este es el utilizado en este proyecto, y uno negativo que obtiene los bordes internos de la misma [14]. El resultado final será una imagen en escala de grises correspondiente con los bordes de interés.

La aplicación del filtro es posible gracias a otra función de *openCV*, según [11]. En la Figura 3.12, se muestra la aplicación de este filtro para cada dígito mostrado en la Figura 3.1 y a su negativo. La diferencia entre ambos resultados se debe a que en la imagen original, como se comentó anteriormente, se están obteniendo los bordes externos gracias al operador positivo, pero al aplicarlo en el negativo de la imagen, los correspondientes bordes externos son ahora los internos de la imagen original. 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.



(a) Original

(b) Negativo

Figura 3.12: Muestra de dígitos con filtro Laplaciano.

Filtro de Sobel

Este filtro está formado por dos máscaras de derivadas que permiten obtener los bordes en una determinada dirección, horizontal y vertical [15]. 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*². Una vez se tienen los bordes en ambas direcciones se suman ambos en valor absoluto y se normaliza a valores entre 0 y 255, obteniendo una imagen de tipo *float* que deberá ser transformada a *uint8*. En la Figura 3.13 se muestra la aplicación de este filtro para cada dígito mostrado en la Figura 3.1. 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.



Figura 3.13: Muestra de dígitos con filtro de Sobel.

Una vez obtenidas las diferentes bases de datos con los filtros de bordes aplicados se procede al entrenamiento de tres redes neuronales diferentes, una con cada uno de los filtros, según se explicó en la Sección 3.1.1. En concreto, únicamente se modificarán las distintas bases de datos empleadas en entrenamiento y evaluación por la obtenida para cada caso.

Tras obtener las tres redes neuronales entrenadas será posible comenzar con el test de las mismas según lo explicado en la Sección 3.2, obteniendo la tasa de acierto para cada uno de ellos, que quedan representados en la Figura 3.14. Para ello se ha empleado la base de datos de test ampliada con el negativo aplicando, para cada red, el filtro correspondiente.

²http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html

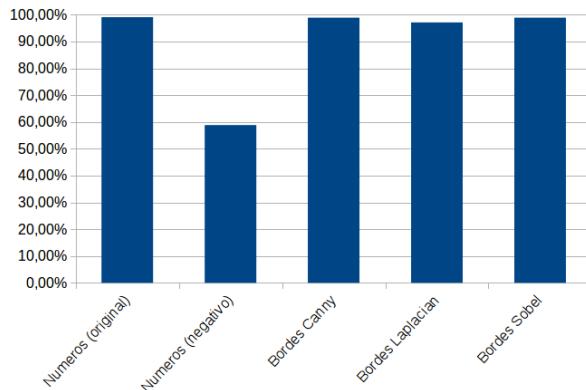


Figura 3.14: Comparación de tasa de acierto con diferentes filtros.

En esta gráfica se puede observar que el uso de imágenes de bordes mejora en gran medida el entrenamiento con un determinado fondo, haciendo la clasificación independiente del mismo. Dentro de los distintos filtros utilizados, todos ellos 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 la preferencia de obtener imágenes en tono de grises, que hagan más robusta la red, se opta por elegir el filtro de Sobel en la aplicación.

Para obtener resultados coherentes es fundamental que el preprocesado aplicado a la base de datos en el entrenamiento sea igualmente aplicado a las imágenes antes de introducirlas en la red para la clasificación. Para ello se modificará la función que transforma la imagen obtenida por la cámara en el componente, explicada en la Sección 3.1.2.1, incluyendo el filtrado de bordes tras eliminar el ruido. En la Figura 3.15 se puede observar cómo se muestra la imagen final en el componente, aplicando el filtrado indicado.

Una vez se ha obtenido una clasificación independiente del fondo de la imagen escogida, dotando a la red de una mayor robustez, se analizarán algunas transformaciones típicas que se pueden dar frecuentemente en un problema real, es decir, se estudiará el efecto de no tener imágenes perfectas sino imágenes ruidosas.

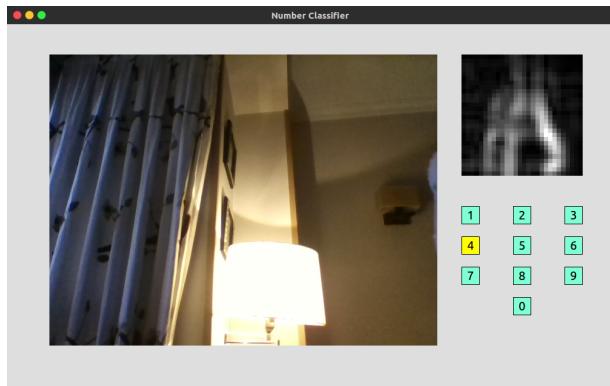


Figura 3.15: Captura del componente clasificador con filtro Sobel.

3.3.1.2. Imágenes ruidosas en test

Al obtener una imagen con la cámara esta puede no estar perfectamente centrada, recta o con un tamaño igual al de la base de datos empleada en el entrenamiento, además de poder incluir ruido introducido por la propia cámara. Esto hará que, al introducir una de estas imágenes en la red, entrenada con imágenes sin ningún tipo de alteración, no se obtenga la precisión deseada.

Para la evaluación del ruido sobre la red neuronal, entendiendo por ruido cualquier posible alteración de la imagen, se elaborará una base de datos de test a partir de la que proporciona MNIST de 10000 muestras, incluyendo en ella 6 imágenes con la transformación aplicada y la imagen original para cada muestra, aplicando, por último el filtro de Sobel a cada una. De esta manera se obtiene una base de datos de test con 70000 muestras de bordes ruidosas.

Tras aplicar las diferentes transformaciones sobre la base de datos de test se obtendrán diversas bases de datos modificadas que serán introducidas sobre la misma red neuronal, la desarrollada anteriormente con bordes de Sobel, para poder evaluar la robustez de la misma.

A continuación se explicarán las distintas transformaciones realizadas, todas ellas con ayuda de las funciones que proporciona *openCV* y cómo afecta cada una a los parámetros que permiten evaluar la red.

Rotación

La rotación consiste en el giro sobre un eje situado en el centro de la imagen, un determinado ángulo establecido por el desarrollador.

En concreto, en esta aplicación se ha optado por la rotación con un ángulo aleatorio dentro del rango [-20,20] grados. En la Figura 3.16 se muestra la aplicación de la rotación para cada dígito mostrado en la Figura 3.1.



Figura 3.16: Muestra de dígitos rotados.

Traslación

La traslación de una imagen consiste en desplazar la misma en una determinada dirección y sentido marcados 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] y uno vertical de [-4,2], de tal manera que la imagen del dígito no quede recortada. En la Figura 3.17 se muestra la aplicación de la traslación para cada dígito mostrado en la Figura 3.1.



Figura 3.17: Muestra de dígitos trasladados.

Escalado

El escalado de una imagen consiste en cambiar el tamaño 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].

Tras aplicar el escalado, el resultado es una imagen cuyas dimensiones 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 mismas al necesario para la red (28x28) sin deformar la imagen. Para ello, si el tamaño de la imagen es mayor, se recortará la misma manteniendo el centro, si por el contrario, el tamaño es menor, se añadirá un borde del mismo color que el fondo de la imagen hasta obtener el tamaño deseado.

En la Figura 3.18 se muestra la aplicación de la traslación para cada dígito mostrado en la Figura 3.1.



Figura 3.18: Muestra de dígitos escalados.

Ruido

El ruido de una imagen es una variación aleatoria de la información de brillo o color en la misma. Existen diferentes tipos de ruido con naturalezas distintas que pueden estar producidos por diversas causas como por ejemplo ruido Gaussiano, ruido *Salt&Pepper* o ruido uniforme.

La aplicación pretende clasificar los dígitos mostrados a una cámara en tiempo real, en donde el ruido más presente es el ruido Gaussiano, por lo que será el empleado para el test. Se aplicará un ruido Gaussiano con una varianza de 0.02, utilizando, en este caso, una función que proporciona *skimage.util*. En la Figura 3.19 se muestra la aplicación del ruido para cada dígito mostrado en la Figura 3.1.



Figura 3.19: Muestra de dígitos con ruido.

Además de las bases de datos creadas con la transformación única, se elabora una nueva base de datos que contiene una combinación de todas las transformaciones explicadas: escalado, traslación, rotación y 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 3.20 se muestra la aplicación de la traslación para cada dígito mostrado en la Figura 3.1.



Figura 3.20: Muestra de dígitos con mezcla de transformaciones.

Una vez se han obtenido las bases de datos que permitan realizar el estudio sobre la robustez de la red, se incluirán en el banco de pruebas explicado en 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 3.21.

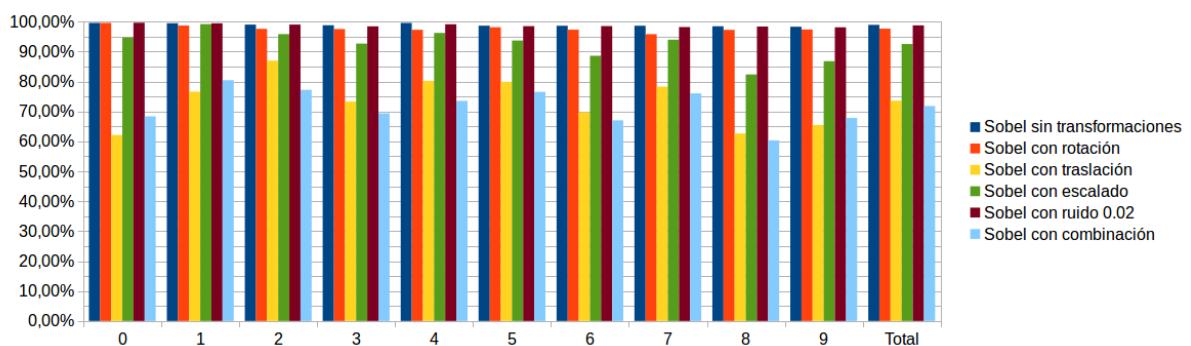


Figura 3.21: Evaluación de la red con bases de datos transformadas.

Se puede observar que el introducir varias transformaciones sobre la imagen hace que la tasa de acierto disminuya considerablemente, siendo especialmente sensible a la traslación, mientras que el ruido apenas afecta, algo que concuerda con las imágenes

mostradas. Estos resultados hacen ver que la red, frente a un escenario real en el que las imágenes no son perfectas, no reaccionaría de la manera que se desearía, existiendo una probabilidad de fallo del 0.3. Por ello, se continuará el estudio elaborando bases de datos de entrenamiento y validación que incluyan estas transformaciones y permitan mejorar la tasa de acierto de la aplicación.

3.3.1.3. Imágenes ruidosas en entrenamiento

Para conseguir desarrollar una red más robusta, según lo analizado las secciones anteriores, se elaborarán nuevas bases de datos de entrenamiento y validación mediante la combinación de las transformaciones explicadas anteriormente. Las nuevas bases de datos serán modificaciones de la utilizada en la red básica, explicada en la Sección 3.1.1. Para la evaluación de todas las redes creadas se empleará la base de datos de test creada anteriormente, en la que se combinan 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, que queda reflejada en la Tabla 3.2. De esta manera se podrán obtener valores de *precision* y *recall* de la misma 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.

Tras obtener la evaluación de la red basica, se procede a la modificación de la misma y su posterior evaluación, variando el número de imágenes transformadas que se utilizan, así como la inclusión o no de la imagen original. Para esta tarea, se partirá 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, para evaluar el impacto y conseguir una base de datos 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 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, 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.

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 la máxima precisión posible.

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, 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.

Base de datos 0-6

La siguiente 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 la mejora del 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, respresentada en la Tabla ??.

		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 .

Base de datos 0-1

Finalmente, visto que los resultados de la reducción de muestras explicadas anteriormente resultan bastante satisfactorios, se reduce el número de imágenes transformadas utilizadas 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, 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 .

Una vez se han obtenido las matrices de confusión de cada una de las redes neuronales entrenadas, se pueden establecer algunas conclusiones a simple vista. En primer lugar, es clara la mejora al entrenar introduciendo alguna imagen ruidosa, ya que, si se observan los valores de la diagonal, correspondiente con los dígitos correctamente clasificados, éstos son superiores al introducir el ruido en el entrenamiento. Además, entrenar introduciendo un mayor número de imágenes de ruido, 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 poder establecer conclusiones más firmes, se calculan el *precision* y *recall* de cada una de las redes, con ayuda de la matriz de confusión obtenida según lo explicado en la Sección 3.2. Estos resultados quedan reflejados en la Figura 3.22.

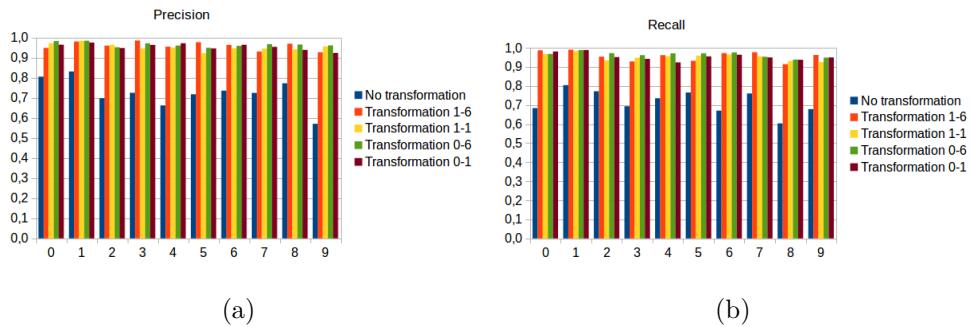


Figura 3.22: Resultados de parámetros de evaluación: (a) *Precision*, (b) *Recall*

Los resultados obtenidos confirman las conclusiones que se alcanzaron con anterioridad. Existe una clara mejora al introducir imágenes ruidosas en el entrenamiento, una única imagen ruidosa aporta suficiente información y la inclusión de la imagen original no aporta gran información para el entrenamiento.

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.

3.3.2. Número de iteraciones

Hasta ahora, se había fijado el número de iteraciones que se realizan 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 por un *batch*. Sin embargo, este número de iteraciones no tiene por qué ser el más idóneo para la aplicación.

Durante el entrenamiento de una red neuronal el aprendizaje es progresivo, de esta manera, la red obtenida en la iteración $n+1$ es mejor que la obtenida en la iteración n . Ésto se cumple hasta un cierto punto. 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 la red. Este deterioro de la red 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* en el que 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 3.23, que permita seleccionar la red más adecuada para la aplicación.

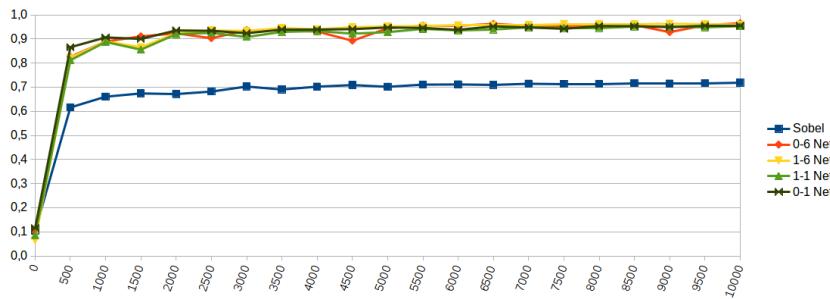


Figura 3.23: Red básica LeNet MNIST.

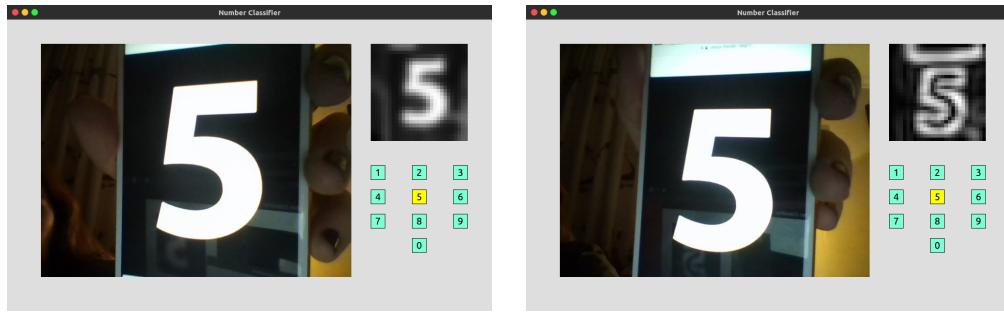
En esta figura se puede comprobar que, además de la mejora mencionada anteriormente al introducir imágenes de ruido en el entrenamiento, la estabilidad de la red se alcanza mucho antes de las 10000 iteraciones marcadas.

Por todo lo mencionado anteriormente, se decide escoger la red entrenada con la base de datos 0-1 parando en la iteración 5000, obteniendo buenos resultados en cuanto a la precisión de la red y disminuyendo considerablemente la carga computacional, pues se reduce más de la mitad el número de iteraciones de entrenamiento de la red y se mantiene el número de muestras de las bases de datos de entrenamiento y validación, aunque estas hayan sido modificadas.

3.4. Experimentos

Tras todo el análisis realizado anteriormente se ha obtenido una red neuronal más robusta 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 a la que se le han aplicado las variaciones precisas.

En primer lugar se prueba la aplicación con un dígito cuyo fondo es negro, tal y como se entrenó en la primera red desarrollada. Los resultados de este experimento quedan reflejados en la Figura 3.24.

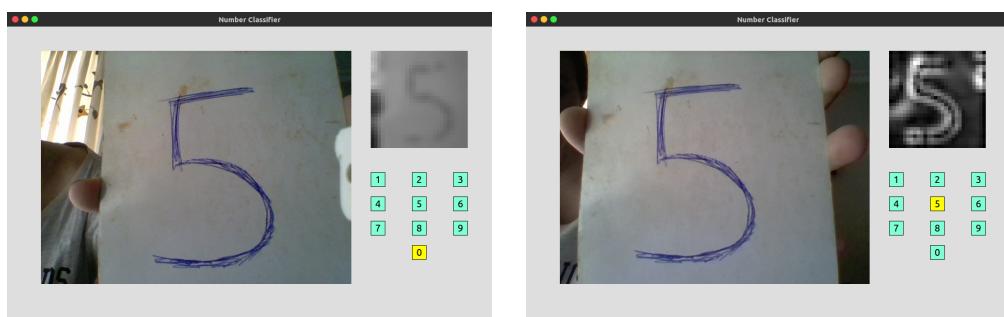


(a) Red básica

(b) Red robusta

Figura 3.24: Evaluación de la aplicación con imagen de fondo negro.

Tras evaluar un ejemplo completamente limpio se pone a prueba la aplicación con una imagen más complicada para la misma, variando su fondo y no siendo tan perfecto. Esta prueba queda reflejada en la Figura 3.25, donde se expone un ejemplo con el que se comprueba cómo un dígito que con la red básica que se desarrolló en primer lugar no lograba ser identificado, pues no cumplía con todas las características que la red exigía para ser más precisa, sí es posible su clasificación tras mejorar la red entrenada según lo estudiado en los puntos anteriores.



(a) Red básica

(b) Red robusta

Figura 3.25: Evaluación de la aplicación con diferentes redes.

Para realizar los experimentos anteriores se ha tomado como “Red básica”, la explicada en la Sección 3.1.1, en cuyo entrenamiento únicamente se disponía de imágenes con fondo

CAPÍTULO 3. CLASIFICACIÓN

negro y el dígito en blanco. Como era de esperar, y se demostró en la Sección 3.3.1.1, 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 3.3.1, una red entrenada con imágenes transformadas a las que se les ha aplicado el filtro de bordes Sobel iterando un total de 5000 veces, permitiendo independizar la imagen del fondo y que la imagen obtenida por la cámara no sea tan perfecta. Con esta nueva red la aplicación sí consigue clasificar correctamente el mismo dígito mostrado a la cámara.

La aplicación desarrollada es una muestra sencilla de herramienta para la clasificación de imágenes con 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, que permitan solucionar problemas de interés para el ser humano.

Capítulo 4

Detección

Capítulo 5

Conclusiones

Bibliografía

- [1] *Fundamentos para el procesamiento de imágenes.* Uabc.
- [2] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986.
- [3] 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>.
- [4] 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>.
- [5] 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.
- [6] K. Ganesan. *Text Mining, Analytics & More.* <http://text-analytics101.rxnlp.com/2014/10/computing-precision-and-recall-for.html>, 2014. [Accedido 25 de Mayo de 2017].
- [7] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.
- [8] R. López and J. Fernández. *Las Redes Neuronales Artificiales.* Metodología y Análisis de Datos en Ciencias Sociales. Netbiblo, 2008.
- [9] U. of Regina. *Text Mining, Analytics & More.* http://www2.cs.uregina.ca/~dbd/cs831/notes/confusion_matrix/confusion_matrix.html, 2008.
- [10] OpenCV-Doxygen. *Canny Edge Detection.* http://docs.opencv.org/trunk/dad22/tutorial_py_canny.html, 2017. [Accedido 14 de Junio de 2017].

BIBLIOGRAFÍA

- [11] OpenCV-Sphinx. *Laplace Operator.* http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html, 2017. [Accedido 14 de Junio de 2017].
- [12] L. Pullum, B. Taylor, and M. Darrah. *Guidance for the Verification and Validation of Neural Networks.* Emerging Technologies. Wiley, 2007.
- [13] C. M. Rob Hess and Friends. *Introducing: Flickr PARK or BIRD.* <http://code.flickr.net/2014/10/20/introducing-flickr-park-or-bird/>, 2014. [Accedido 13 de Junio de 2017].
- [14] Tutorialspoint. *Laplacian Operator.* https://www.tutorialspoint.com/dip/laplacian_operator.htm, 2017. [Accedido 7 de Junio de 2017].
- [15] Tutorialspoint. *Sobel Operator.* https://www.tutorialspoint.com/dip/sobel_operator.htm, 2017. [Accedido 7 de Junio de 2017].
- [16] A. Veit, T. Matera, L. Neumann, J. Matas, and S. Belongie. Coco-text: Dataset and benchmark for text detection and recognition in natural images. In *arXiv preprint arXiv:1601.07140*, 2016.
- [17] X. Wang. Laplacian operator-based edge detectors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(5):886–890, May 2007.