



ESCUELA TECNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Grado en Ingeniería en
Sistemas Audiovisuales y Multimedia

Trabajo Fin de Grado

Deep-learning para detección de vehículos

Autor: Nuria Oyaga de Frutos

Tutores: José María Cañas Plaza, Inmaculada Mora Jiménez

Curso académico 2016/2017



©2017 Nuria Oyaga de Frutos

Esta obra está 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.

Índice general

1. Infraestructura	1
1.1. Software	1
1.1.1. JdeRobot	1
1.1.2. Caffe	3
1.1.3. DroidCam	7
1.2. Bases de datos	7
1.2.1. MNIST	7
1.2.2. COCO	8
1.3. Métricas	10
2. Clasificación con Deep Learning	11
2.1. Clasificador de dígitos	11
2.1.1. Red básica	11
Definición de la red	12
Definición del solucionador	17
Ejecución de la red	18
2.1.2. Componente Python	20
Cámara	21
GUI	23
Ejecución	24
2.2. Banco de pruebas	27
2.2.1. Obtención de datos de test	27
2.2.2. Banco de pruebas manual: Excel	29
2.3. Efectos del aprendizaje	32
2.3.1. Bases de datos	33
2.3.2. Número de neuronas	37
2.3.3. Dropout	37
2.4. Experimentos	37

Índice de figuras

1.1. Estructura y funcionamiento básico de red en Caffe	3
1.2. Función de activación ReLu	5
1.3. Estructura de instancias de objetos	9
1.4. Estructura básica de anotaciones	10
2.1. Red básica LeNet MNIST	17
2.2. Ejecución de entrenamiento de red LeNet MNIST	19
2.3. Fin de entrenamiento de red LeNet MNIST	19
2.4. Curva de aprendizaje Red Básica	20
2.5. Captura de componente gráfico de la aplicación	24
2.6. Captura de DroidCam en el ordenador	25
2.7. Captura de DroidCam en el dispositivo móvil	25
2.8. Captura del componente clasificador	26
2.9. Imagen original y su correspondiente negativo	35
2.10. Porcentaje de acierto de base de datos original y ampliada con negativo . .	35
2.11. Muestra de imagen con filtro Canny	37

Capítulo 1

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 las diferentes métricas empleadas para evaluar el impacto del aprendizaje en las redes neuronales y que permitirán escoger la red más adecuada para el problema.

Software

JdeRobot

JdeRobot ¹ es una plataforma de software libre que facilita la tarea de los desarrolladores del 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 asincrónicos 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 ... 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, no es la verdadera finalidad del proyecto su uso, por lo 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 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

Caffe

Caffe [4] es un framework de deep learning 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 múltiples ejemplos y modelos ya entrenados, lo que ofrece mayor agilidad a la hora de empezar a entender el funcionamiento del aprendizaje profundo, 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 ??, utilizando un entrenamiento por lotes. En concreto, su estructura y funcionamiento básico queda explicado en la Figura 1.1.

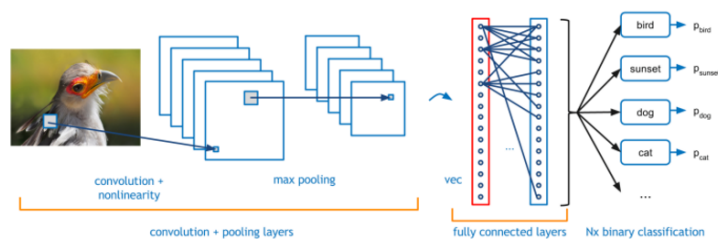


Figura 1.1: Estructura y funcionamiento básico de red en Caffe

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 etiquetas 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 pueden provenir de diferentes vías como 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 test, 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

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 tomando el máximo, el promedio, u otras operaciones dentro de las regiones, siendo su finalidad la reducción del muestreo. En esta capa se pueden especificar 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 *bolb* 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, evistando así el sobre ajuste. Se debe indicar el ration 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 *bolb* inferior y producen uno superior del mismo tamaño. Existen varias capas con este funcionamiento en la plataforma, en concreto se empleará la ReLu.

ReLu

Utiliza la función $y = \max(0, x)$ cuya gráfica se define en la Figura 1.2

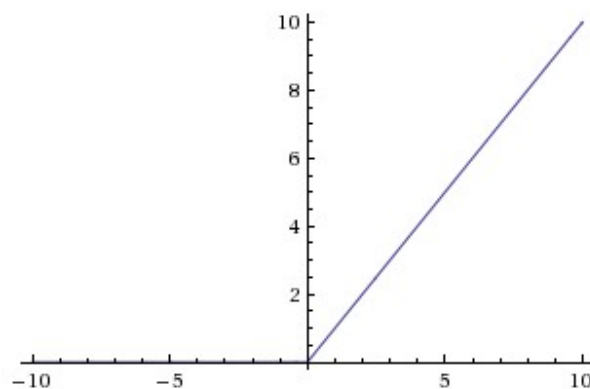


Figura 1.2: Función de activación ReLu

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

Se calcula como:

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

Siendo N el número total de muestras y \hat{p} las probabilidades de cada etiqueta para cada muestra.

Accuracy

Se calcula como:

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

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

Por último, además de las capas y parámetros definidos anteriormente, Caffe, permite el desarrollo de un *solver* en el que se podrán ajustar parámetros como el número de iteraciones totales que se ejecutarán, el de test que se van a realizar, cada cuantas iteraciones se realizarán esos test, así como 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. 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:

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

En cuanto al número de iteraciones que se establecerán de test, se debe cumplir la siguiente igualdad:

$$\text{Iteraciones test} = \frac{\text{Muestras test}}{\text{Tamaño lote de test}}$$

DroidCam

DroidCam ² es una aplicación que permite convertir un dispositivo móvil en una cámara web, estableciendo una conexión mediante WiFi/LAN, modo servidor wifi, o USB. Esta aplicación es muy usada para establecer videoconferencias a través de plataformas como Skype o Google+, entre otras aplicaciones. En este trabajo será usada para obtener el flujo de vídeo desde un dispositivo distinto a la webcam del ordenador, haciendo más sencillo el manejo del mismo.

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

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

Bases de datos

MNIST

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.

²<https://www.dev47apps.com/>

³<http://yann.lecun.com/exdb/mnist/>

Se trata de un subconjunto de una más grande, NIST, en la que las imágenes originales en blanco y negro (NIV) 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. 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.

COCO

Microsoft COCO ⁴ es un gran conjunto de datos de imágenes diseñado para la detección de objetos, segmentación y generación de subtítulos[7]. Algunas 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

⁴<http://mscoco.org/>

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 JSON y comparten estructura de datos establecida en la Figura 1.4.

Para la detección son de interés las anotaciones de instancias de objetos, cuya estructura se muestra en la Figura 1.3. 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 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 1.3: Estructura de instancias de objetos

```
{
  "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 1.4: Estructura básica de anotaciones

Métricas

Existen multitud de métricas para 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* [6]

Capítulo 2

Clasificación con Deep Learning

En este capítulo se expondrá el trabajo realizado para el entendimiento del problema de clasificación, mediante la elaboración de un componente en Python que permite la clasificación de dígitos del 0 al 9 en tiempo real, y la realización de un amplio estudio sobre las variantes posibles aplicadas a las redes entrenadas, utilizando la plataforma Caffe.

Clasificador de dígitos

Se ha desarrollado un componente en Python para la clasificación de dígitos entre 0 y 9 en tiempo real, siendo necesario, previamente, un entendimiento de una primera red básica, utilizada por el mismo para la tarea. En esta sección se explicará el procedimiento seguido para el entendimiento de la red y el desarrollo del propio componente.

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 1.2.1. Para realizar el entrenamiento de la 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.

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, define 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.

```
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 y, finalmente, dos capas totalmente conectadas similares a las perceptrones multi-capas convencionales. En el ejemplo de Caffe, la estructura habitual de la red LeNet se ve ligeramente modificada, ya que en lugar de emplear una función de activación sigmoidal se utiliza una lineal.

Tras la definición del nombre se definen dos capas de datos, una de ellas correspondiente a los datos de entrenamiento de la red y, la otra, correspondiente a los datos que se utilizarán para realizar el test durante el entrenamiento para obtener datos de *accuracy* y *loss*.

```
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
  }
}
```



```
}  
}
```

Es importante que los parámetros de transformación, en este caso un factor de escala que establece el rango de la imagen en $[0,1]$, sean los mismos en ambas fases, pues si se evaluase la red con una transformación de la imagen distinta a la aplicada en el entrenamiento los resultados obtenidos no serían reales.

Se utilizará, por tanto, dos capas de datos que difieren en la fase en la que se utilizarán los datos, entrenamiento o evaluación de la red, el tamaño del lote, siendo 64 muestras para el entrenamiento y 100 para el test, y la ruta de la que se cogen los datos.

El conjunto de datos proporcionado por MNIST no contiene una base de datos de validación, por lo que se debe dividir la base de datos de entrenamiento en dos partes, una de ellas se utilizará para el entrenamiento y la otra para la validación, incluyendola en la capa de datos de test. 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.

Dígito	Total	80 %	20 %
0	5923	4738	1185
1	6742	5393	1349
2	5958	4767	1191
3	6131	4905	1226
4	5842	4674	1168
5	5421	4337	1084
6	5918	4734	1184
7	6265	5012	1253
8	5851	4681	1170
9	5949	4759	1190
Total	60000	48000	12000

Tabla 2.1: División base de datos de entrenamiento

En este proceso es muy importante respetar las proporciones existentes en cada dígito para no alterar la naturaleza de la base de datos original.

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 dos veces.

```
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"}
  }
}
```

En la capa de convolución, explicada en la Sección 1.1.2, se define que el tamaño del filtro será de 5x5 y que se obtendrán 20 salidas, en la segunda capa de convolución, 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.

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

```
        kernel_size: 2
        stride: 2
    }
}
```

La capa de agrupamiento, también explicada en la Sección 1.1.2, será alimentada por la capa de convolución anterior y alimentará a la siguiente en caso de que la haya. 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.

Tras estas capas, se establecen dos capas completamente conectadas, *InnerProduct*, separadas por la capa de activación, *ReLu*, ambas explicadas en la Sección 1.1.2.

```
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"}
  }
}
```

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

Las capas completamente conectadas se definen con, 500 salidas la primera de ellas, y tantas como clases se tengan en la segunda, en el caso concreto que se trata serán 10,

correspondientes con los dígitos del 0 al 9.

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. Para ello, en la Sección 1.1.2, se explicaron varias capas de *Loss*, que serán empleadas en esta red para su evaluación, se deberán explicitar en este documento.

```
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 siguiente documento.

En la Figura 2.1 se puede observar un esquema de la estructura definida en este apartado, los valores de interés y cada una de las entradas y salidas de las capas. Para obtener esta figura, se ha ejecutado un código proporcionado por la propia plataforma, que, mediante el archivo que define la estructura, explicado anteriormente, dibuja la red. Para ello se debe ejecutar el siguiente comando:

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

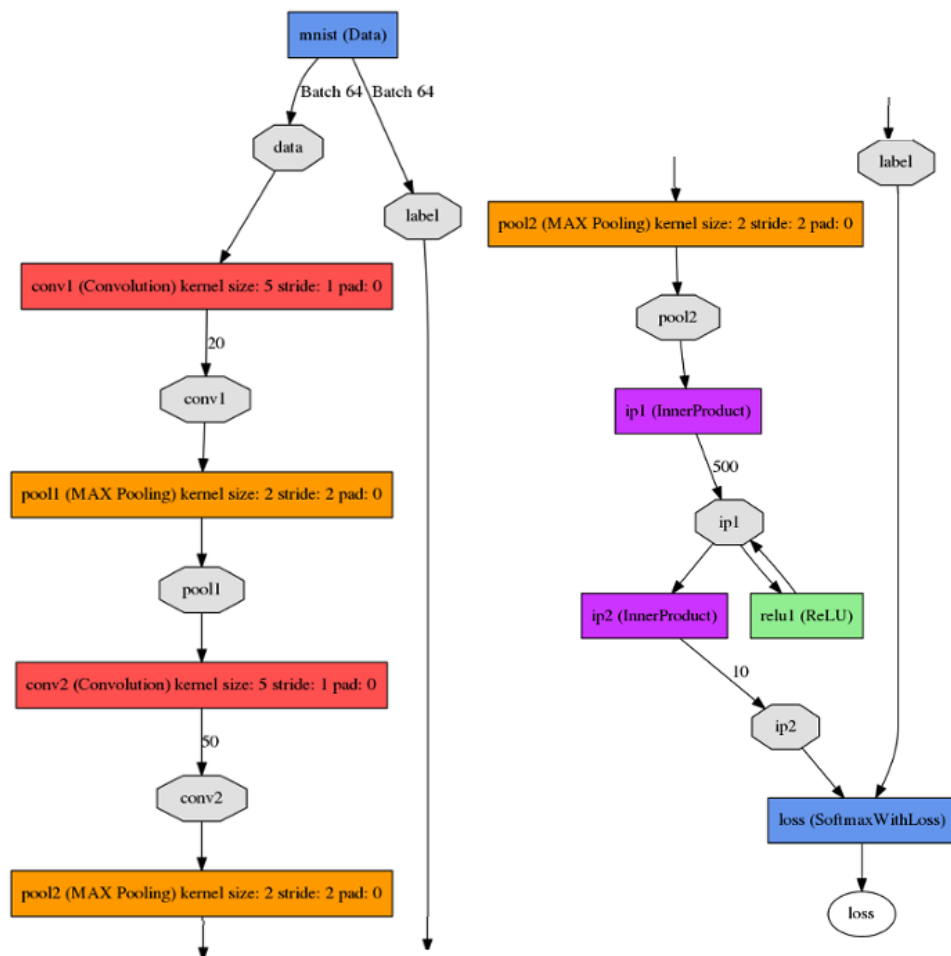


Figura 2.1: Red básica LeNet MNIST

Definición del solucionador

Para esta tarea se va a utilizar el archivo de Caffe *lenet_solver.prototxt*, que permitirá manejar parámetros del propio entrenamiento de la red.

Se definen en él 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 1.1.2. 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
```

Ejecución de la red

Una vez se han definido los parámetros adecuados para la red que se quiera entrenar, se ejecutarán los siguientes comandos, que comenzará con el entrenamiento de la red:

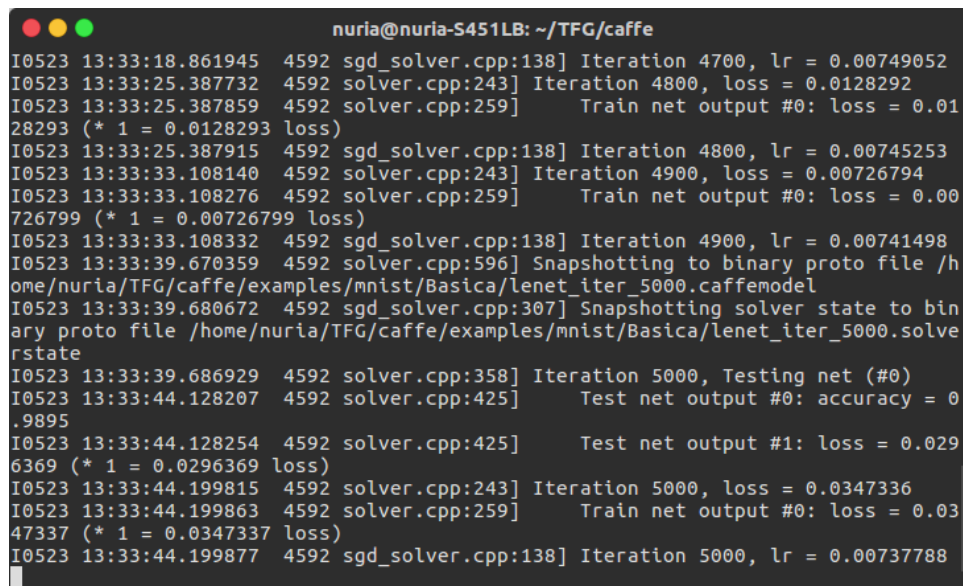
```
cd $CAFFE_ROOT
./examples/mnist/train_lenet.sh
```

El archivo que se ejecuta contiene información sobre qué solucionador se debe implementar y el modo de ejecución. Además es posible añadirle una línea que guardará un archivo con información de *log* del proceso de entrenamiento.

```
#!/usr/bin/env sh

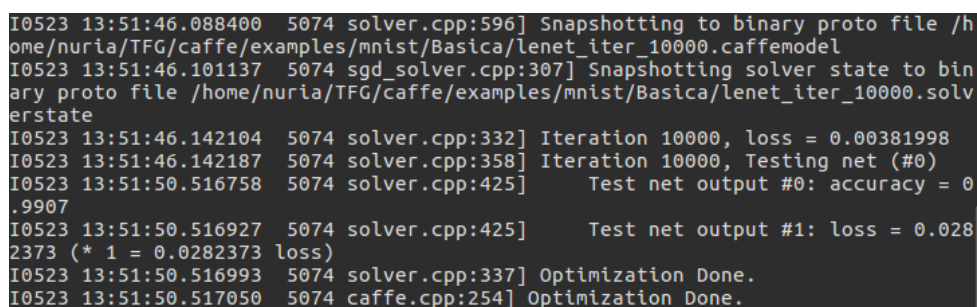
set -e

./build/tools/caffe train
--solver=examples/mnist/lenet_solver_validation.prototxt
2>&1 | tee /home/nuria/TFG/logs/RedBasica.log $@
```



```
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 = 0.0128293 (* 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 = 0.00726799 (* 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 /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_5000.caffemodel
I0523 13:33:39.680672 4592 sgd_solver.cpp:307] Snapshotting solver state to binary proto file /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_5000.solverstate
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 = 0.0296369 (* 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 = 0.0347337 (* 1 = 0.0347337 loss)
I0523 13:33:44.199877 4592 sgd_solver.cpp:138] Iteration 5000, lr = 0.00737788
```

Figura 2.2: Ejecución de entrenamiento de red LeNet MNIST



```
I0523 13:51:46.088400 5074 solver.cpp:596] Snapshotting to binary proto file /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_10000.caffemodel
I0523 13:51:46.101137 5074 sgd_solver.cpp:307] Snapshotting solver state to binary proto file /home/nuria/TFG/caffe/examples/mnist/Basica/lenet_iter_10000.solverstate
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 = 0.0282373 (* 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 2.3: Fin de entrenamiento de red LeNet MNIST

Tras terminar el entrenamiento, mostrado en la Figura 2.3, se obtiene el archivo con la red neuronal entrenada, almacenado según la ruta que se indicó en el solucionador, que podrá ser utilizada en la herramienta que sea de interés.

El archivo *log* generado podrá ser visualizado mediante la ejecución del script *plot_learning_curve.py*¹ de la siguiente forma:

```
python plot_learning_curve.py /home/nuria/TFG/logs/RedBasica.log
RedBasicaLog.png
```

En la Figura 2.4 se puede observar el resultado final de la curva de aprendizaje, con valores de precisión y pérdidas, según lo explicado en la Sección 1.1.2, para la fase de evaluación, y en entrenamiento únicamente de pérdidas.

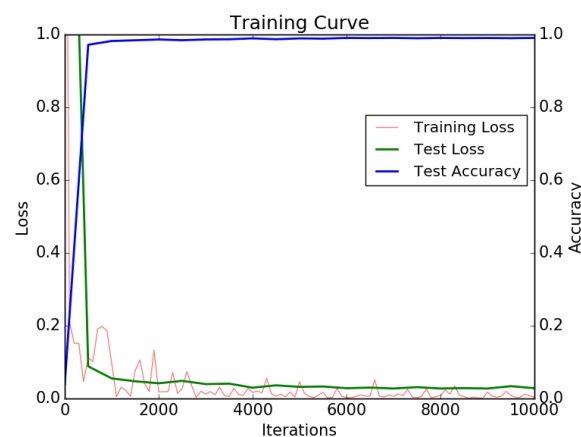


Figura 2.4: Curva de aprendizaje Red Básica

Componente Python

Se ha desarrollado un componente escrito en Python que, mediante la ayuda del *Camera Server* de JdeRobot, comentado en la Sección 1.1.1, y la red explicada en la Sección 2.1.1, 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

¹https://github.com/adilmoujahid/deeplearning-cats-dogs-tutorial/blob/master/code/plot_learning_curve.py

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

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)
```

Mediante estas líneas 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 Caffe de manera homóloga al *lenet.train.test.prototxt*, con la excepción de que la capa de datos no recurre a archivos almacenados sino que utiliza imágenes que serán insertadas en la ejecución de la red. El resto de datos deben ser

exáctamente iguales a la estructura de la red entrenada para que no se produzcan errores. En segundo lugar, se indica la red entrenada que se utilizará en la ejecución, el archivo obtenido al finalizar el entrenamiento según se indicó en la Sección 2.1.1. 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 ya que, en el componente principal, se crea el hilo *ThreadCamera(camera)* 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 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 una 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 *bolb* de datos sea de 28x28, valores establecidos para las imágenes. 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, se pasa a desarrollar el interfaz gráfico que permite al usuario visualizar, tanto las imágenes captadas y transformadas, como el resultado de la clasificación.

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 2.5 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.

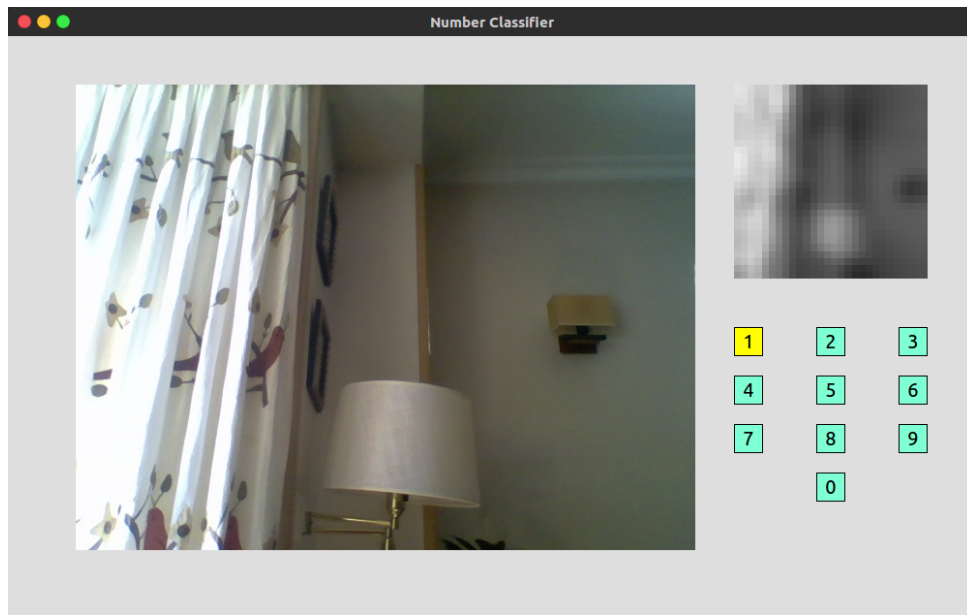


Figura 2.5: Captura de componente gráfico de la aplicación

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 emplearía 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 indicará el valor 1.

En la Sección 1.1.3 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 instalada el programa tanto en el dispositivo móvil a utilizar como en el propio ordenador, según se indica en las guía de la aplicación², 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. Se opta por la conexión USB, pues es más rápida y, por tanto, más adecuada a tiempo real. Una vez se han realizado las acciones anteriores se establece la conexión y se obtienen los resultados de las Figuras 2.6, en el ordenador, y 2.7, en el dispositivo.

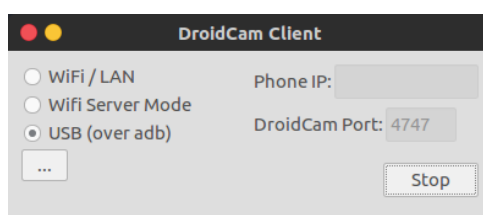


Figura 2.6: Captura de DroidCam en el ordenador

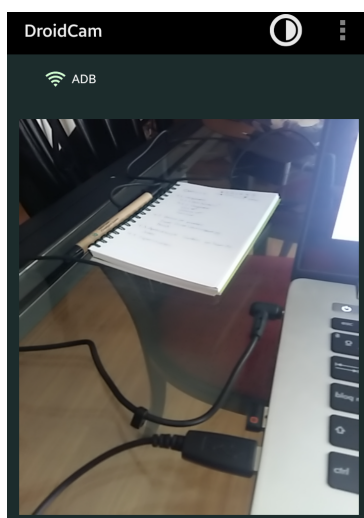


Figura 2.7: Captura de DroidCam
en el dispositivo móvil

Tras tener en funcionamiento el servidor de imágenes se debe proceder a la ejecución del componente clasificador. Para ello se ejecutará el siguiente comando:

²<https://www.dev47apps.com/droidcam/linuxx/>

```
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. En el fichero de configuración se tiene una propiedad que indica qué cámara utilizar. Es importante que el nombre de esta cámara se corresponda con el indicado en el fichero de configuración del servidor, de esta manera se establece la comunicación entre ambos componentes.

Finalmente, tras la ejecución, obtenemos el resultado del componente mostrado en la Figura 2.8, 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.

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.

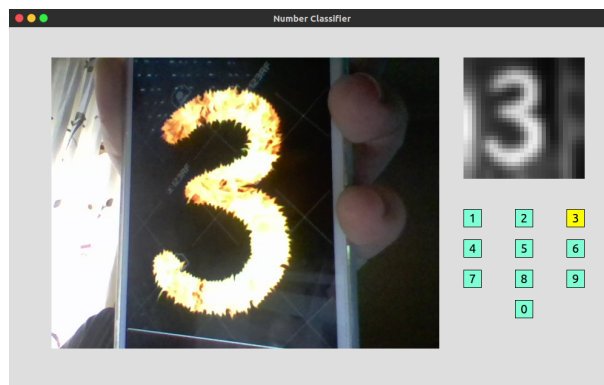


Figura 2.8: Captura del componente clasificador

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 las métricas explicadas en el Capítulo 1, siendo necesario, previamente, la obtención de datos de clasificación sobre una determinada base de datos de evaluación.

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 evaluación a la red neuronal deseada y obtener la clasificación para cada uno de los elementos de esa base de datos. Este script está dividido en tres partes claramente diferenciadas que permite la obtención de los resultados.

Obtención de las imágenes

Las imágenes y sus correspondientes etiquetas estan 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.

```
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 misma y que permitirá recorrerla para obtener las imágenes y etiquetas.

Para poder procesar los datos obtenidos mediante Caffe, será necesario crearse una estructura *Datum* de Caffe, que incluirá, en cada iteración para recorrer la base de datos, la información de la instancia de la base de datos. 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 2.1.2, 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 ha optado por la creación de un archivo de texto que incluirá una pequeña descripción de cómo interpretar el contenido y, para cada iteración, el número de

iteración, las etiquetas reales, las identificadas por la red y un booleano que indicará si ambas etiquetas coinciden o no, todo ello separado por espacios.

```
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")
    loop = loop + 1
    ...
```

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.

Banco de pruebas manual: Excel

Las métricas que se obtienen con este banco de pruebas son las explicadas en la Sección 1.3: 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 *precision*, 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, homóloga a Excel, 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. De estas columnas formadas serán de interés la que contiene la etiqueta real, la clasificación realizada, y la conclusión final, acierto o fallo.

Tras obtener los datos de interés separados y correctamente ordenados se procederá a identificar el número de aciertos y de fallos tanto a nivel global, para obtener los parámetros de precisión total 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.

Precisión total

Este valor es el más lógico y sencillo de obtener. Para calcular la precisión total, es decir, 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 evaluación que se han utilizado para la misma. De esta manera se obtiene el porcentaje de imágenes que se han clasificado de forma correcta, valor que se corresponde con la precisión 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 transform'.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 transform'.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: Para ello se realizará la división del número de acierto entre el total y multiplicarlo por 100 para obtener el porcentaje.

Matriz de confusión

Tanto para este apartado, como para los dos posteriores, se ha utilizado la información proporcionada por [3].

Para elaborar la matriz de confusión se parte de la misma hoja de cálculo con las columnas de etiqueta real, clasificada y conclusión 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. 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.

Efectos del aprendizaje

Existen numerosos factores que afectan al nivel de 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 [5], hacen que la red tenga una mayor, mejorando la aplicación deseada.

En esta sección se tratará el efecto en el aprendizaje de tres 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, el incremento del número de neuronas y el uso o no del *Dropout*.

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.

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 2.2. 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 2.2, 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 la propia imagen, 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.

Para realizar la inserción, al finalizar la interpretación de los datos de cada muestra de la base de datos original, se deben incluir las siguientes líneas:

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

De esta manera se incluye en la posición *keyststr*, 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 2.9 se muestra uno de los dígitos y su correspondiente negativo almacenado en la base de datos. Estas imágenes han sido obtenidas con el script *dataread.py*, que lee las imágenes de la base de datos y crea un archivo para su visualización.



Figura 2.9: Imagen original y su correspondiente negativo

Tras ejecutar este script se obtiene la base de datos de evaluación 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 precisión global.

En la Figura 2.10 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.

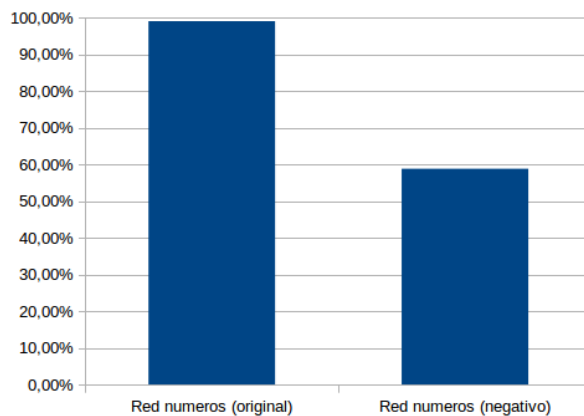


Figura 2.10: Porcentaje de acierto de base de datos original y ampliada con negativo

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, todos ellos se basan en la localización en la imagen de aquellas zonas en las que la diferencia de valor entre un píxel y el contiguo supera un determinado umbral [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*. En este caso, se parte de la base de datos ampliada con el negativo, no siendo necesario almacenar la imagen de la que se parte en la base de datos, y sustituyendo el negativo por el filtro adecuado en la transformación. Además, será necesario aplicar el mismo sobre la base de datos de entrenamiento, 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 y, otra de evaluación, con 20000 muestras, a las que se les ha aplicado un determinado filtro.

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 que proporciona *openCV* para obtener los bordes según este algoritmo ³.

```
data_filter = cv2.Canny(data[0],100,200) #shape (28,28)
data_filter = data_filter[np.newaxis, :, :] #dimensions (1,28,28)
```

³http://docs.opencv.org/trunk/da/d22/tutorial_py_canny.html

En la Figura 2.11 se puede observar una muestra de las que se obtienen en la base de datos final, en la que se van a obtener dos imágenes iguales de cada dígito. Este hecho se debe a 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 2.11: Muestra de imagen con filtro Canny

Filtro Laplaciano

Filtro de Sobel

Número de neuronas

Dropout

Experimentos

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] 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].
- [4] 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.
- [5] R. López and J. Fernández. *Las Redes Neuronales Artificiales*. Metodología y Análisis de Datos en Ciencias Sociales. Netbiblo, 2008.
- [6] L. Pullum, B. Taylor, and M. Darrah. *Guidance for the Verification and Validation of Neural Networks*. Emerging Technologies. Wiley, 2007.
- [7] 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.