



## 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

<b>1. Infraestructura</b>	<b>1</b>
1.1. Software . . . . .	1
1.1.1. JdeRobot . . . . .	1
1.1.2. Caffe . . . . .	3
1.2. Bases de datos . . . . .	7
1.2.1. MNIST . . . . .	7
1.2.2. COCO . . . . .	8
<b>2. Clasificación con Deep Learning</b>	<b>10</b>
2.1. Clasificador de dígitos . . . . .	10
2.1.1. Red básica . . . . .	10
2.1.2. Componente Python . . . . .	19

# Índice de figuras

1.1. Estructura y funcionamiento básico de red en Caffe . . . . .	3
1.2. Función de activación ReLu . . . . .	5
1.3. Estructura básica de anotaciones . . . . .	9
1.4. Estructura de instancias de objetos . . . . .	9
2.1. Red básica LeNet MNIST . . . . .	15
2.2. Ejecución de entrenamiento de red LeNet MNIST . . . . .	17
2.3. Fin de entrenamiento de red LeNet MNIST . . . . .	18
2.4. Curva de aprendizaje Red Básica . . . . .	18

# 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.

## Software

### JdeRobot

JdeRobot <sup>1</sup> 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 asincrónos y concurrentes. Esta estructura permite la ejecución de los distintos componentes en diferentes equipos, estableciendo una conexión entre ellos mediante el middleware de comunicaciones ICE. Además, se obtiene

---

<sup>1</sup><http://jderobot.org>

gran flexibiidad 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 <sup>2</sup> 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, donde se .

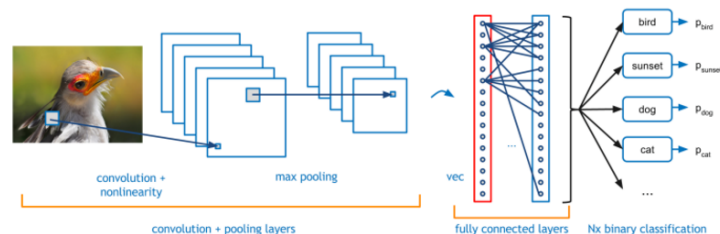


Figura 1.1: Estructura y funcionamiento básico de red en Caffe

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.

---

<sup>2</sup><http://caffe.berkeleyvision.org/>

### 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

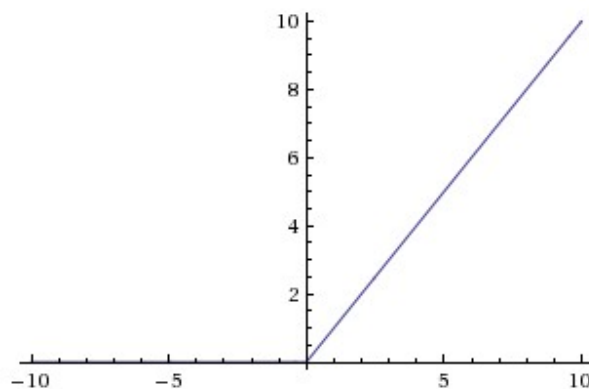


Figura 1.2: Función de activación ReLu

## 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:

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

# Bases de datos

## MNIST

MNIST <sup>3</sup> 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, 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.

---

<sup>3</sup><http://yann.lecun.com/exdb/mnist/>

### COCO

Microsoft COCO <sup>4</sup> es un gran conjunto de datos de imágenes diseñado para la detección de objetos, segmentación y generación de subtítulos. 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 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 1.3.

Para la detección son de interés las anotaciones de instancias de objetos, cuya estructura se muestra en la Figura 1.4. 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 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.

---

<sup>4</sup><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 1.3: Estructura básica de anotaciones

```
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 1.4: Estructura de instancias de objetos

## 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 el Capítulo 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 multicapa 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 sigmoideal 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.

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 el Capítulo 1, se define que el tamaño del filtro será de  $5 \times 5$  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 el Capítulo 1, 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 el Capítulo 1.

```
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 el Capítulo 1, 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.

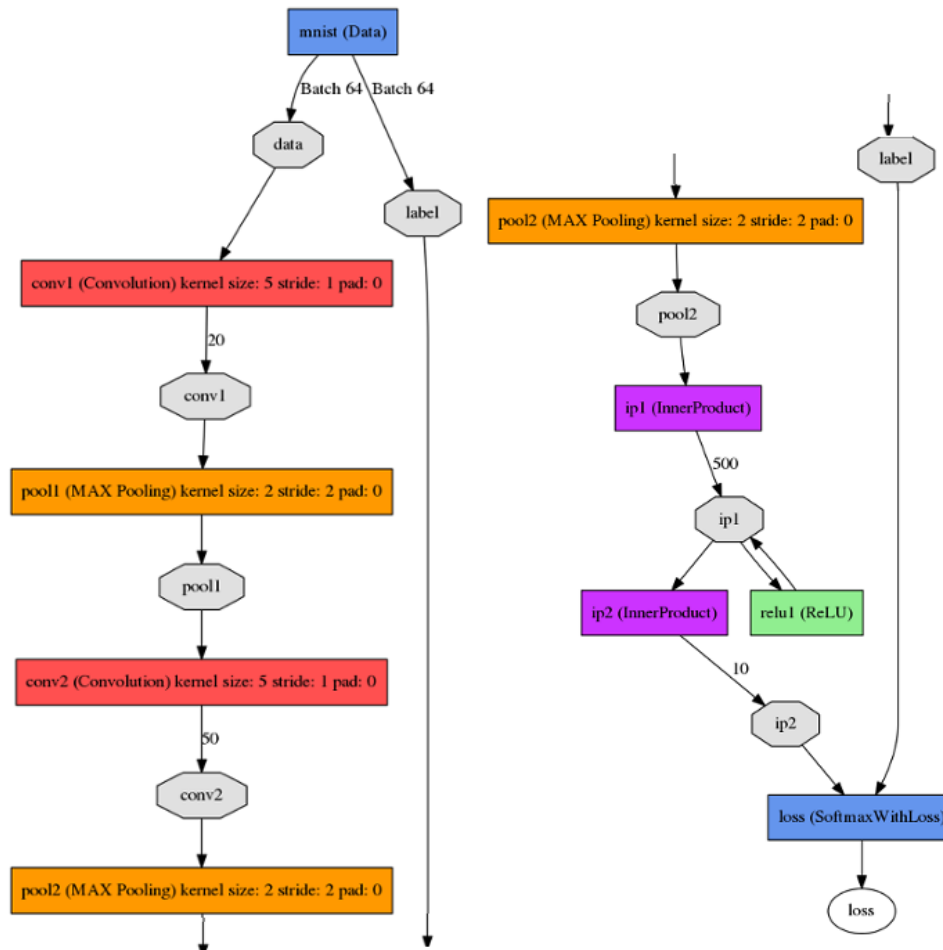


Figura 2.1: Red básica LeNet MNIST

Para obtener la Figura 2.1, 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>
```

### 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 el Capítulo 1. 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:

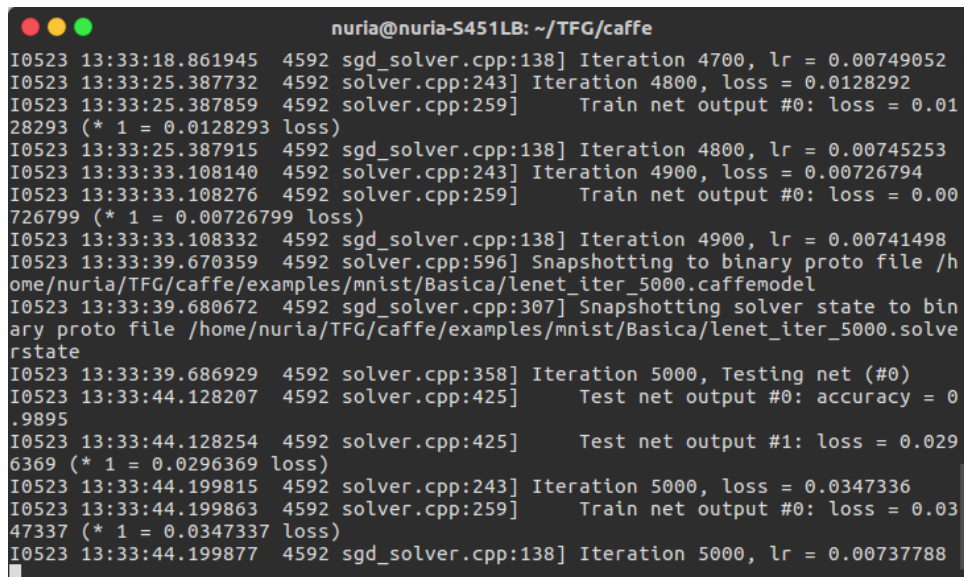
```
cd $CAFFE_ROOT
./examples/mnist/train_lenet.sh
```

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.solvestate
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*<sup>1</sup> 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 el Capítulo 1, para la fase de evaluación, y en entrenamiento únicamente de pérdidas.

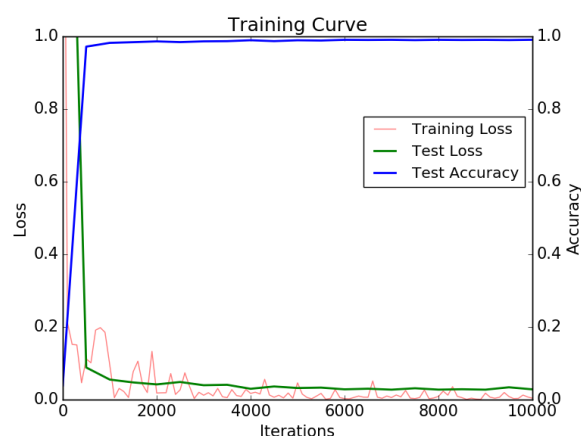


Figura 2.4: Curva de aprendizaje Red Básica

<sup>1</sup>[https://github.com/adilmoujahid/deeplearning-cats-dogs-tutorial/blob/master/code/plot\\_learning\\_curve.py](https://github.com/adilmoujahid/deeplearning-cats-dogs-tutorial/blob/master/code/plot_learning_curve.py)

## Componente Python