ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN INGENIERÍA EN
SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

DEEP LEARNING EN SENSORES RGBD

Autor: David Butragueño Palomar
Tutor:

Curso académico 2016/2017

Resumen

Agradecimientos

Índice general

Índice de figuras

Capítulo 1

Introducción

Capítulo 2

Infraestructura

2.1. Caffe

Caffe es un framework de aprendizaje profundo desarrollado por Berkeley AI Research (BAIR) y por contribuyentes de la comunidad. Fue creado por Yangqing Jia durante su doctorado en UC Berkeley. Caffe está publicado bajo la licencia BSD 2-Clause.

2.1.1. Línea de comandos

Caffe dispone de una interfaz de línea de comandos llamada *cmdcaffe* la cual es la herramienta utilizada por Caffe para el entrenamiento del modelo y el diagnóstico del mismo. Los principales comandos que se pueden ejecutar son:

- **Entrenamiento:** Con el comando *caffe train* es posible aprender modelos desde cero,
- **Test:** El comando *caffe test* puntúa los modelos ejecutándolos en la fase de test e informa de la salida de la red como su puntuación. En primer lugar se informa la puntuación por lotes de datos de entrada y finalmente el promedio general.
- **Comparación:** El comando *caffe time* compara el modelo capa a capa. Esto es util para comprobar el rendimiento del sistema y medir los tiempos de ejecución relativos a los modelos.

2.1.2. Python

La interfaz de Python *pycaffe* contiene el módulo de Caffe y sus propios scripts en la ruta caffe/python. Con el comando *import caffe* se importará esta interfaz, pudiendo así cargar diferentes modelos de Caffe, manejar instrucciones de entrada/salida, visualizar redes y numerosas funcionalidades más. Todos los datos y parámetros se encuentran disponibles tanto para lectura como para escritura. Algunas de las tareas que se pueden realizar con está interfaz son:

- **caffe.Net** es la interfaz central para cargar, configurar y ejecutar modelos.

- **caffe.Classifier y caffe.Detector** proporcionan interfaces para tareas de clasificación y detección.
- **caffe.SGDSolver** se trata de la interfaz de resolución.
- **caffe.io** maneja funciones de entrada/salida con preprocesamiento.
- **caffe.draw** visualiza las arquitecturas de red.

2.1.3. Capas de una red neuronal

Para crear un modelo de Caffe es necesario definir la arquitectura del mismo utilizando para ello un archivo de definición de buffer de protocolo (prototxt).

Capas de datos

Los datos entran en Caffe a través de las capas de datos las cuáles se encuentran en la parte inferior de las redes. Estos datos pueden provenir de bases de datos (LevelDB o LMDB), directamente de la memoria, o, cuando la eficiencia no es crítica, desde archivos en disco en formato HDF5 o formatos de imagen comunes.
Tareas comunes de preprocesamiento de los datos de entrada, tales como escalado o reflejo, están disponibles especificando *TransformationParameters* por algunas de las capas. Los tipos de capas "bias", "scalez çrop"pueden ser útiles para el preprocesamiento de la entrada cuando la opción *TransformationParameters* no está disponible.

- **Image Data:** Lee imágenes sin procesar.
- **Database:** Lee los datos de LevelLDB o LMDB.
- **HDF5 Input:** Lee los datos en formato HDF5 permitiendo que estos tengan dimensiones arbitrarias.
- **HDF5 Output:** Escribe datos en formato HDF5
- **Input:** Normalmente utilizada para redes que se están implementando.
- **Window Data:**
- **Memory Data:** Lee archivos directamente desde memoria.
- **Dummy Data:** Utilizado para datos estáticos.

Capas de visión

Las capas de visión, generalmente toman imágenes como datos de entradas y generan otras imágenes como salida aunque también pueden tomar datos de otros tipos y dimensiones. Una imagen puede tener un canal ($c = 1$) si se trata de una imagen en escala de grises o 3 canales ($c = 3$) si se trata de una imagen RGB. Pero en este contexto, las características distintivas para el tratamiento de las imágenes de entrada serán la altura y la anchura de las mismas. La mayoría

de las capas de visión trabajan aplicando una operación particular sobre alguna región de la entrada para producir una región correspondiente a la salida.

- **Convolution Layer:** Convoluciona la imagen de entrada con un conjunto de filtros.
- **Pooling Layer:** Realiza *pooling* de los datos de entrada utilizando para ello funciones de máximo, media o estocásticas.
- **Spatial Pyramid Pooling (SPP)**
- **Crop**
- **Deconvolution Layer:** Realiza uno convolución transpuesta.
- **Im2Col**

Capas recurrentes

- **Recurrent**
- **RRNN**
- **Long-Short Term Memory (LSTM)**

Capas comunes

- **Inner Product:** Capa totalmente conectada
- **Dropout**
- **Embed**

Capas de pérdida

Estas capas de pérdida conducen al aprendizaje comparando la salida obtenida con el valor de la entrada asignado así un coste para minimizarla.

- **Multinomial Logistic Loss**
- **Infogain Loss**
- **Softmax with Loss**
- **Sum-of-Squares / Euclidean**
- **Hinge / Margin**
- **Sigmoid Cross-Entropy Loss**
- **Accuracy / Top-k layer**
- **Contrastive Loss**

Capas de normalización

- **Local Response Normalization (LRN):** Normaliza regiones locales de los datos de entrada.
- **Mean Variance Normalization (MVN):** Realiza una normalización de contraste / normalización de instancia.
- **Batch Normalization:** Realiza normalizaciones sobre pequeños lotes de datos de entrada.

Capas de activación

En general, estas capas son operados que toman un dato de la salida de la capa anterior y generan datos con las mismas dimensiones.

- **ReLU / Rectified-Linear and Leaky-ReLU:** Se trata de una función lineal, rectilínea con pendiente uniforme.

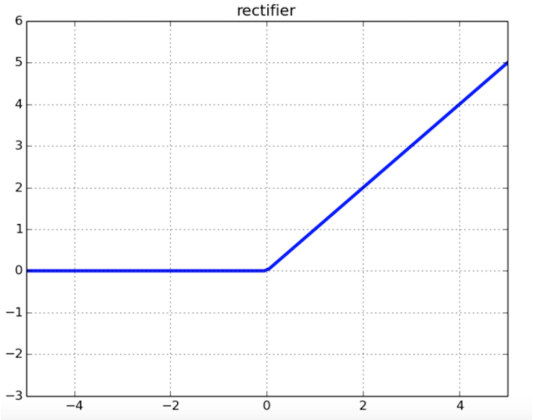

Figura 2.1: Función de activación ReLu

Se puede definir a partir de la siguiente ecuación:

$$f(x) = \begin{cases} 0 \text{ si } x < 0 \\ \text{x si } x \geq 0 \end{cases} \tag{2.1}$$

- **PReLU**
- **ELU**
- **Sigmoid**

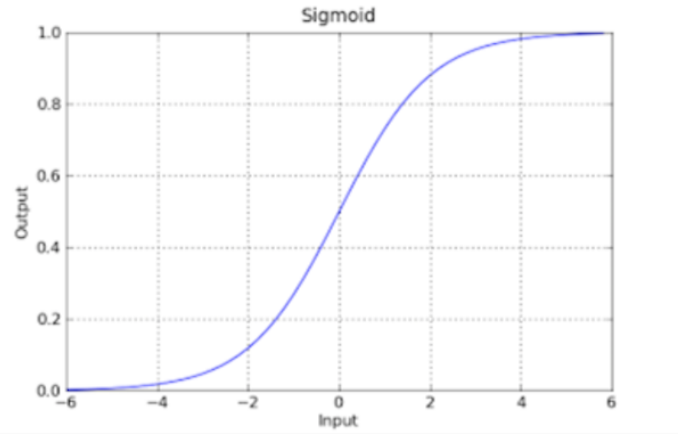

Figura 2.2: Función de activación Sigmoide

Se puede definir a partir de la siguiente ecuación:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **TanH**
- **Absolute Value**
- **Power**

$$f(x) = (shift + scale * x)^{power} \tag{2.2}$$

- **Exp**

$$f(x) = base^{(shift + scale * x)} \tag{2.3}$$

- **Log**

$$f(x) = log(x) \tag{2.4}$$

- **BNLL**

$$f(x) = log(1 + exp(x)) \tag{2.5}$$

- **Threshold**: Realiza la función de paso en el umbral definido por el usuario.
- **Bias**
- **Scale**

2.2. JdeRobot

Se trata de un framework cuyo objetivo es desarrollar aplicaciones en robótica y visión por computadora. También tiene actuación en domótica y en escenarios con sensores, accionadores y software inteligente. Ha sido desarrollado para ayudar en la programación de este software inteligente. Está escrito principalmente utilizando el lenguaje C++ proporcionando un entorno de programación en el que el programa de aplicación está compuesto por una colección de varios componentes asíncronos concurrentes. Estos componentes pueden ejecutarse en diferentes equipos y están conectados mediante el middleware de comunicaciones ICE. Los componentes pueden estar escritos en C++, Python, Java y todos ellos ellos interactúan a través de interfaces ICE explícitas.

JdeRobot simplifica el acceso a dispositivos hardware desde el programa de control. Obtener mediciones de sensores es tan simple como llamar a una función local y ordenar comandos de motor tan fácil como llamar a otra función local. La plataforma adjunta esas llamadas a invocaciones remotas sobre los componentes conectados al sensor o los dispositivos de accionamiento. También, pueden conectarse a sensores y activadores reales o simulados, tanto a nivel local como remoto utilizando para ello la red. Esas funciones construyen la API para la capa de abstracción del hardware. La aplicación robótica obtiene las lecturas del sensor y ordena los comandos del actuador usando esa API para desplegar su comportamiento. Se han desarrollado varios drivers para soportar diferentes sensores, activadores y simuladores. Los robots y sensores actualmente soportados son:

- **Sensores RGBD:** Kinect and Kinect2 de Microsoft, Asus Xtion
- **Robots con ruedas:** Kobuki (TurtleBot) de Yujin Robot y Pioneer de MobileRobotics Inc.
- **ArDrone quadrotor de Parrot**
- **Escáneres laser:** LMS de SICK, URG de Hokuyo y RPLidar
- **Simulador Gazebo**
- **Cámaras Firewire, cámaras USB, archivos de vídeo (mpeg, avi), cámaras IP (como Axis)**

JdeRobot incluye varias herramientas de programación de robots y bibliotecas. En primer lugar, teleespectadores y teleoperadores para varios robots y sus sensores y motores. En segundo lugar, un componente de calibración de cámara y una herramienta de tunning para filtros de color. En tercer lugar, una herramienta llamada VisualHFSM para la programación del comportamiento del robot utilizando la jerarquía Finite State Machines. Además, también proporciona una biblioteca para desarrollar controladores difusos y otra para la geometría proyectiva y el procesamiento de la visión por computadora.

Cada componente puede tener su propia interfaz gráfica de usuario o ninguna en absoluto. Actualmente, las bibliotecas GTK y Qt son compatibles,

incluyéndose varios ejemplos de OpenGL para gráficos 3D con ambas bibliotecas.

JdeRobot es un software de código abierto con licencia como GPL y LGPL. También utiliza software de terceros como el simulador Gazebo, ROS, OpenGL, GTK, Qt, Player, Stage, GSL, OpenCV, PCL, Eigen u Ogre.

2.2.1. CameraServer y CameraView

CameraServer y CameraView es una configuración disponible para probar Jderobot.

En primer lugar se tiene el archivo cameraserver.cfg, el cual tiene el siguiente código:

```
# client/server mode
# rpc=1 ; request=0
CameraSrv.DefaultMode=1
CameraSrv.TopicManager=IceStorm/TopicManager:default -t 5000 -p 10000

#General Config
CameraSrv.Endpoints=default -h 0.0.0.0 -p 9999
CameraSrv.NCameras=1
CameraSrv.Camera.0.Name=cameraA
#0 corresponds to /dev/video0, 1 to /dev/video1, and so on...
CameraSrv.Camera.0.Uri=0
CameraSrv.Camera.0.FramerateN=25
CameraSrv.Camera.0.FramerateD=1
CameraSrv.Camera.0.Format=RGB8
CameraSrv.Camera.0.ImageWidth=640
CameraSrv.Camera.0.ImageHeight=480

# If you want a mirror image, set to 1
CameraSrv.Camera.0.Mirror=1


NamingService.Enabled=0
NamingService.Proxy=NamingServiceJdeRobot:default -h 0.0.0.0 -p 10000
```

Se pueden observar parámetros como por ejemplo el End Point en el que espera recibir peticiones, textitCameraSrv.Endpoints=default -h 0.0.0.0 -p 9999, el número de frames por segundo, textitCameraSrv.Camera.0.FramerateN=25, el formato de la imagen, textitCameraSrv.Camera.0.Format=RGB8 y el ancho y largo de la imagen que se representará, textitCameraSrv.Camera.0.ImageWidth=640 y CameraSrv.Camera.0.ImageHeight=480.

Posteriormente, se tiene el archivo cameraview.cfg, el cual tendrá en siguiente formato:

```
Cameraview.Camera.Proxy=cameraA:default -h 0.0.0.0 -p 9999
Cameraview.Camera.Format=RGB8
```

Para la ejecución de esta configuración, habrá que arrancar en primer lugar cameraserver en un terminal.

```
Cameraview.Camera.Proxy=cameraA:default -h 0.0.0.0 -p 9999
Cameraview.Camera.Format=RGB8
```

Para la ejecución de esta configuración, habrá que arrancar en primer lugar cameraserver en un terminal.

```
./cameraserver –Ice.Config=cameraserver.cfg
```

A continuación, en otro terminal, ejecutar cameraview.

```
./cameraview –Ice.Config=cameraview.cfg
```

2.3. LMDB

Lightning Memory-Mapped Database (LMDB) es una biblioteca de software que proporciona una base de datos de clave-valor de alto rendimiento. LMDB almacena los pares de datos de forma arbitraria utilizando arrays de bytes.

2.3.1. LMDB con Python

Es posible trabajar fácilmente con bases de datos LMDB utilizando Python. En primer lugar, es necesario importar la librería de LMDB.

```
import lmdb
```

Posteriormente, se pueden utilizar los siguientes comandos.

```
lmdb_env = lmdb.open('Ruta donde se encuentra la base de datos')
lmdb_txn = lmdb_env.begin()
lmdb_cursor = lmdb_txn.cursor()
datum = caffe.proto.caffe_pb2.Datum()
```

De esta manera tendremos a disposición los datos del archivo LMDB en formato Datum. Datum es una clase que almacena datos y opcionalmente etiquetas. Estos datos que guarda se representan con un array de 3 dimensiones: altura, anchura y canal.

Posteriormente, se podrá diferenciar entre la propia imagen y su etiqueta. Para ello, se crearán 2 variables, una *data* y *label*, de la siguiente forma:

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

Así, la variable *data* será un array de 3 dimensiones con los niveles de intensidad de cada uno de los píxeles de cada imagen de la base de datos y la variable *label* será un entero comprendido entre el 0 y el 9.

Capítulo 3

Clasificación

3.1. Red Neuronal

3.1.1. Estructura de la red

En esta sección se explicará la estructura de la red que utilizaremos para la clasificación de los dígitos manuscritos que nos facilita la base de datos MNIST.

En primer lugar, se especificará el nombre de la red, en este caso "not"

```
name: "LeNet"
```

Posteriormente, es necesario leer los datos de la base de datos LMDB. Para ello, se definirán 2 capas de datos, una para tomar los datos de entrenamiento y otra para los de test.

A continuación se muestra la capa de datos para las imágenes de entrenamiento:

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

Esta capa, con nombre "'mnist", tomará lo datos de un fichero con extensión LMDB y con nombre "mnist_train_lmdb".
El factor de escala que actúa sobre las imágenes de entrada debe estar en el rango [0,1) teniendo en este caso un valor de 0.00390625. Es muy importante que los factores de transformación tenga el mismo valor en la capa de entrenamiento y en la de test, ya que si no es así se pueden generar resultados incorrectos.
Finalmente, se utilizará un tamaño de lote de 64 imágenes, siendo éste de 100 para la capa de test.

A continuación, se puede observar la estructura básica de las redes neuronales convolucionales (CNN), combinando capas de convolución con capas de pooling, finalizando con una capa totalmente conectada. En este caso, se utilizarán 2 capas de convolución y otras 2 de pooling.

En primer lugar, la capa de convolución, recibe como entrada la imagen que se obtiene en la capa de datos, y aplica sobre ella un filtro o *kernel*. Este proceso transforma la matriz de píxeles de la imagen original en otra matriz de características. Esta nueva imagen es esencialmente la original pero cada píxel tiene mucha más información ya que contiene información de la región en la que se encuentra el píxel, no sólo del píxel aislado.

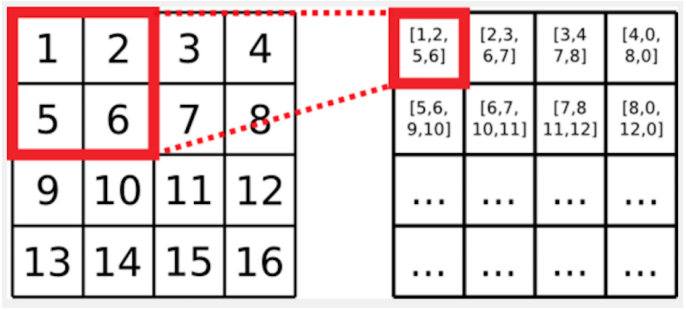

Figura 3.1: Ejemplo convolución con tamaño del núcleo 2x2

A continuación, se muestra la primera capa de convolución de la red:

```
layer {
  name: çonv1"
  type: Çonvolution"
  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: çonstant"
    }
  }
  bottom: "data"
  top: çonv1"
}
```

Esta capa tomará los datos de la capa de entrenamiento y los transformará utilizando un núcleo de convolución de 5X5, produciendo 20 salidas. Para inicializar los pesos se utilizará el algoritmo Xavier, el cuál determina automáticamente la escala de inicialización basándose en el número de entradas y salidas de cada neurona. Adicionalmente, el sesgo se inicializará como una constante, siendo esta por defecto 0.

La capa de pooling se coloca generalmente después de la capa de convolución. Su utilidad principal radica en la reducción espacial de la imagen de entrada. Para ello, se divide el mapa de características obtenido anteriormente en un conjunto de bloques de m x n. A continuación, se aplica una función de agrupación para cada uno de los bloques. Tras este proceso, se obtendrá una matriz de características más pequeño. Dentro de las funciones de agrupación, destacan max pooling, el cual elige el valor más alto dentro del bloque y pooling promedio (average) el cual toma como respuesta de bloque el valor promedio de las respuestas del bloque.

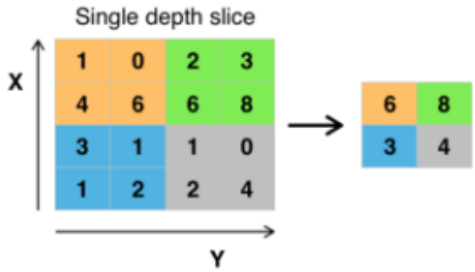

Figura 3.2: Pooling con función de agrupación del máximo

A continuación, se muestra la primera capa de pooling de la red:

```
layer {
  name: "pool1"
  type: "Pooling"
  pooling_param {
    kernel_size: 2
    stride: 2
    pool: MAX    }
  bottom: çonv1"
  top: "pool1"
}
```

Esta capa, tomará los datos de la capa de convolución anterior y utilizará bloques de 2x2 para dividir la imagen entrante. Para que no haya solapamiento entre bloques contiguos utilizará el parámetro *stride = 2*. Finalmente, como función de agrupación utilizará el máximo, escogiendo el píxel con mayor nivel de intensidad de cada bloque.

Posteriormente, aparecerán 2 capas totalmente conectadas separadas por la función de activación ReLu. La primera capa totalmente conectada tendrá 500 salidas, mientras que la segunda tendrá 10, las cuales hacen referencia al número total de salidas que tendrá la red neuronal, una para cada dígito entre el 0 y el 9.

A continuación se muestra la primera capa totalmente conectada.

```
layer {
  name: ïp1”
  type: ÏnnerProduct”
  param { lr_mult: 1 }
  param { lr_mult: 2 }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: ”xavier”
    }
    bias_filler {
      type: çonstant”
    }
  }
bottom: ”pool2”
top: ïp1”
}
```

A continuación se muestra la capa de activación, la cual utiliza la función ReLu explicada en el apartado 2.1.3.

```
layer {
  name: relu1”
  type: ReLU”
  bottom: ïp1”
  top: ïp1”
}
```

```
layer {
  name: ”loss”
  type: ”SoftmaxWithLoss”
  bottom: ïp2”
  bottom: ”label”
}
```

En la siguiente imagen se observa la estructura de la red explicada anteriormente:

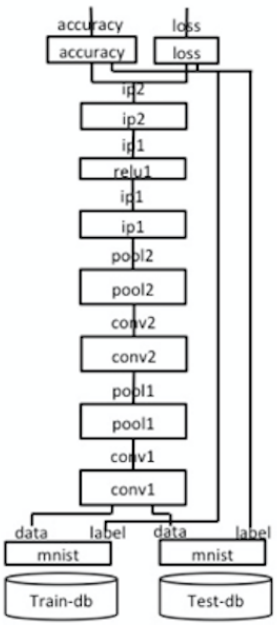

Figura 3.3: Estructura red neuronal

3.1.2. Definición del solucionador

El solucionador es el responsable de la optimización del modelo. Se define en un archivo con extensión *.prototxt*, en este caso, *lenet_solver.prototxt*.

Este solucionador calcula la precisión del modelo usando el conjunto de validación cada 100 iteraciones. El proceso de optimización tendrá una duración de máximo 10000 iteraciones y tomará una instantánea del modelo entrenado cada 500 iteraciones.

En esta configuración, se empezará con una tasa de aprendizaje de 0.01 (*base_lr = 0.01*), la cual irá cayendo con un factor de 10000 (*gamma = 0.0001*).

El solucionador *lenet_solver.prototxt* tendrá el siguiente aspecto. Se puede observar un comentario antes de cada sentencia indicando su funcionalidad:

```
# The train/test net protocol buffer definition
net: .exªamples/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: ïnv”
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: .exªamples/mnist/lenet”
# solver mode: CPU or GPU
solver_mode: GPU
```

3.1.3. Entrenamiento de la red

Tras la definición de la red y el solucionador, para entrenar el modelo hay que ejecutar, en la ruta donde se encuentre el directorio de caffe, el siguiente comando.

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

El script *train_lenet.sh* hará una llamada al solucionador que se quiera ejecutar.

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


./build/tools/caffe train –solver=examples/mnist/lenet_solver.prototxt
```

Una vez entrenada la red, se generarán 2 ficheros:

- lenet_iter_10000.caffemodel: Es un binario que contiene el estado actual de los pesos para cada capa de la red.
- lenet_iter_10000.solverstate: Es un binario que contiene la información necesaria para continuar el entrenamiento del modelo desde donde se detuvo por última vez.

3.2. Base de datos MNIST

La base de datos MNIST (Modified National Institute of Standards and Technology database) es una gran base de datos de dígitos escritos a mano que se utiliza comúnmente para el entrenamiento de sistemas de procesamiento de imágenes.

Esta base de datos está compuesta por 60000 imágenes destinadas al entrenamiento y 10000 imágenes de prueba. Las imágenes están representadas en escala de grises con 256 niveles de intensidad y con un tamaño normalizado de 28x28 píxeles. Cada imagen está formada por un dígito escrito a mano comprendido entre el 0 y el 9, centrado y con el fondo negro, es decir, con un nivel de intensidad igual a 0.

3.2.1. Sección de datos

Para entender mejor con que datos se va a trabajar, es necesario visualizar algún ejemplo de la base de datos MNIST. Con este fin, se utilizará el formato PGM (Portable Graymap Format) para representar las imágenes. Para conseguir una imagen .pgm hay que crear un documento que tenga el siguiente formato:

- En la primera línea escribir el código **P2** para identificar que lo que se quiere conseguir es una imagen con extensión pgm.
- En la segunda línea el ancho y el alto de la imagen, en nuestro caso **28 28**.
- En la tercera línea el número de grises entre el blanco y el negro, en nuestro caso **255**.
- Por último, escribir la intensidad de cada uno de los píxeles de la imagen perfectamente alineados.

La siguiente imagen muestra un ejemplo de una imagen en formato PGM.

```
P2
28 28    # Width and Height.
255       # Numbers of grey between black and white.

0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255 247 127   0   0   0   0
0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0
0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82  82  56  39   0   0   0   0   0
0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253 253 207   2   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201  78   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figura 3.4: Fichero PGM

Para crear este archivo se utilizará lo indicado en la sección 2.3, utilizando la variable *data* comentada anteriormente.

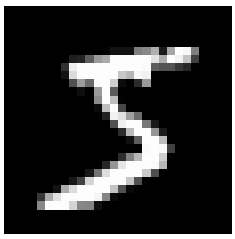

Figura 3.5: Imagen base de datos MNIST

3.2.2. Sección de etiquetas

Una vez se conoce el aspecto que tienen las imágenes de la base de datos, es importante saber como están distribuidas sus 60000 muestras. Para ello, se utilizará la variable *label* comentada en el apartado 2.3.

```
label = datum.label
```

Realizando un bucle que recorra toda la base de datos y creando un contador para cada uno de los dígitos, se podrá conocer la distribución de las imágenes dentro de la base de datos MNIST.

```
for key, value in lmdb_cursor:
    datum.ParseFromString(value)
    label = datum.label
    data = caffe.io.datum_to_array(datum)
    if label == 0:
        Zero = Zero + 1
    elif label == 1:
        One = One + 1
    elif label == 2:
        Two = Two + 1
    elif label == 3:
        Three = Three + 1
    elif label == 4:
        Four = Four + 1
    elif label == 5:
        Five = Five + 1
    elif label == 6:
        Six = Six + 1
    elif label == 7:
        Seven = Seven + 1
    elif label == 8:
        Eight = Eight + 1
    elif label == 9:
        Nine = Nine + 1
```

Tras la ejecución de estos comandos, se obtienen los siguientes resultados:

Dígito	Número de imágenes
Cero	5923
Uno	6742
Dos	5958
Tres	6131
Cuatro	5842
Cinco	5421
Seis	5918
Siete	6265
Ocho	5851
Nueve	5949

3.3. Componente Python

Uno de los objetivos principales es utilizar una cámara para clasificar o detectar las imágenes que esta recoge. Para ello, se ha utilizado un GUI (Graphical User Interface) y la herramienta CameraServer proporcionada por JdeRobot explicada en el apartado 2.2.1.

3.3.1. GUI

La interfaz gráfica creada, utiliza PyQt4, el cual es uno de los bindings más populares de Python. Este GUI mostrará 3 pantallas:

- La ventana principal mostrará la imagen que captura la cámara.
- Otra pantalla mostrará la imagen transformada.
- Otra pantalla mostrará 10 etiquetas que harán referencia a cada uno de los números comprendidos entre el 0 y el 9, las cuales cambiarán de color en función del dígito que capte la cámara utilizada.

En primer lugar, se definirán las diferentes ventanas que compondrán el GUI. Para cada una de ellas se escogerán parámetros como la posición, textitmove, el tamaño, textitresize, o el estilo de la ventana, textitsetStyleSheetset.

```
lab0=QtGui.QLabel(self)
lab0.resize(30,30)
lab0.move(835,450)
lab0.setText('0')
lab0.setAlignment(QtCore.Qt.AlignCenter)
lab0.setStyleSheet("background-color: #7FFFD4; color: #000; font-size:
20px; border: 1px solid black;")
self.numbers.append(lab0)
```

Finalmente, se definirá la función textitlightON, la cual se encargará de cambiar el fondo de una de las 10 etiquetas en función del dígito que se identifique.

```
def lightON(self,out): #This function turn on the light for the network output
    for number in self.numbers:
        number.setStyleSheet("background-color: #7FFFD4; color: #000;
font-size: 20px; border: 1px solid black;")
        self.numbers[out].setStyleSheet("background-color: #FFFF00; color:
#000; font-size: 20px; border: 1px solid black;")
```

3.3.2. Camara

En primer lugar, se definirán las variables *model_file* y *petrained_file*, las cuales harán referencia a la red utilizada para el entrenamiento y al fichero .caffemodel que contiene los pesos de cada capa de la red tras el entrenamiento. Posteriormente, se utilizará la función de caffe Classifier.

```
self.net = caffe.Classifier(model_file, pretrained_file, image_dims=(28, 28),
raw_scale=255)
```

Tras esto, se definirán funciones para el tratamiento de la imagen de entrada. La función *getImage* tomará la imagen captada por la cámara y la transformará para la red. Para ello, hará una llamada a la función *transformImage*, la cual redimensionará la imagen de entrada a un tamaño de 28x28 píxeles y la convertirá en una imagen en escala de grises, para adaptarla al formato de las imágenes utilizadas para el entrenamiento de la red. Tras esto, se podrá aplicar un filtro, ya sea Canny, Sobel o Laplaciano.

Finalmente, se definirá la función classification, la cual utilizará la variable net definida anteriormente para clasificar el dígito que se esta mostrando.

3.3.3. Ejecución

Para su ejecución, en primer lugar, habrá que arrancar la herramienta CameraServer explicada en el apartado 2.2.1.

Tras esto, habrá que lanzar el script *numberclassifier.py*, el cual iniciará 2 hilos o *thread*, uno para la cámara y otro para el GUI.

```
t1 = ThreadCamera(camera)
t1.start()

t2 = ThreadGui(window)
t2.start()
```

Para sincronizarlo con CameraServer habrá que crear un fichero de configuración, *numberclassifier.cfg*, para que apunte al EndPoint en el que está escuchando CameraServer. Este fichero de configuración tendrá la siguiente apariencia:

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

Finalmente, el script *numberclassifier.py* se lanzará utilizando el siguiente comando.

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

3.4. Pruebas

Capítulo 4

Bibliografía

Edgar Nelson Sánchez Camperos y Alma Yolanda Alanís García: Redes Neuronales: Conceptos fundamentales y aplicaciones a control automático"