



# 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

<b>1. Introducción</b>	<b>1</b>
<b>2. Infraestructura</b>	<b>2</b>
2.1. Caffe . . . . .	2
2.1.1. Línea de comandos . . . . .	2
2.1.2. Python . . . . .	3
2.1.3. Capas de una red neuronal . . . . .	3
2.2. JdeRobot . . . . .	8
2.2.1. CameraServer y CameraView . . . . .	9
2.3. Bases de datos . . . . .	11
2.3.1. VOC . . . . .	11
2.3.2. COCO . . . . .	12
<b>3. Clasificación</b>	<b>15</b>
3.1. Red Neuronal . . . . .	15
3.1.1. Estructura de la red . . . . .	15
3.1.2. Definición del solucionador . . . . .	21
3.1.3. Entrenamiento de la red . . . . .	22
3.2. Base de datos MNIST . . . . .	23
3.2.1. Acceso a datos y etiquetas . . . . .	23
3.2.2. Sección de datos . . . . .	24
3.2.3. Sección de etiquetas . . . . .	26
3.3. Componente Python . . . . .	27
3.3.1. GUI . . . . .	27
3.3.2. Camara . . . . .	28
3.3.3. Ejecución . . . . .	29
3.4. Pruebas . . . . .	29
3.4.1. Filtro Canny . . . . .	30
3.4.2. Filtro Sobel . . . . .	30
3.4.3. Filtro Laplaciano . . . . .	31
3.4.4. Casos de pruebas . . . . .	31

<b>4. Detección</b>	<b>34</b>
4.1. Red Neuronal . . . . .	34
4.1.1. Estructura de la red . . . . .	34
4.2. Jaccard index . . . . .	34
4.3. Componente Python . . . . .	34
4.4. Medidor de calidad . . . . .	34
<b>5. Bibliografía</b>	<b>35</b>

# Índice de figuras

2.1. Función de activación ReLu . . . . .	6
2.2. Función de activación Sigmoide . . . . .	7
2.3. Estructura básica de anotaciones en COCO . . . . .	12
2.4. Estructura básica de anotaciones en COCO . . . . .	13
2.5. Estructura básica de anotaciones en COCO . . . . .	14
2.6. Estructura básica de anotaciones en COCO . . . . .	14
3.1. Ejemplo convolución con tamaño del núcleo 2x2 . . . . .	17
3.2. Pooling con función de agrupación del máximo . . . . .	18
3.3. Estructura red neuronal . . . . .	21
3.4. Fichero PGM . . . . .	25
3.5. Imagen base de datos MNIST . . . . .	25
3.6. Máscaras de un filtro Laplaciano . . . . .	31
3.7. Resultados tras varias transformaciones en las imágenes . . . .	33

# 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 útil 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 esta 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 crop" 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 LevelDB 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 una 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.

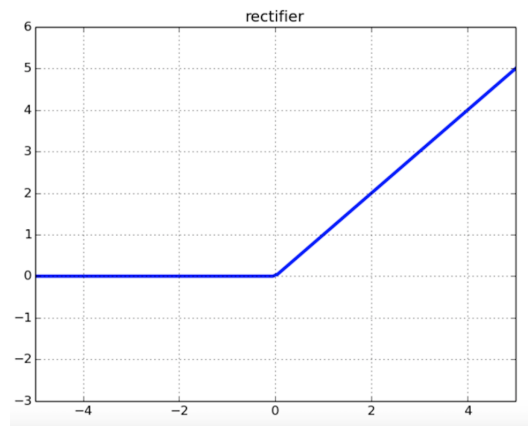


Figura 2.1: Función de activación ReLu

Se puede definir a partir de la siguiente ecuación:

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

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

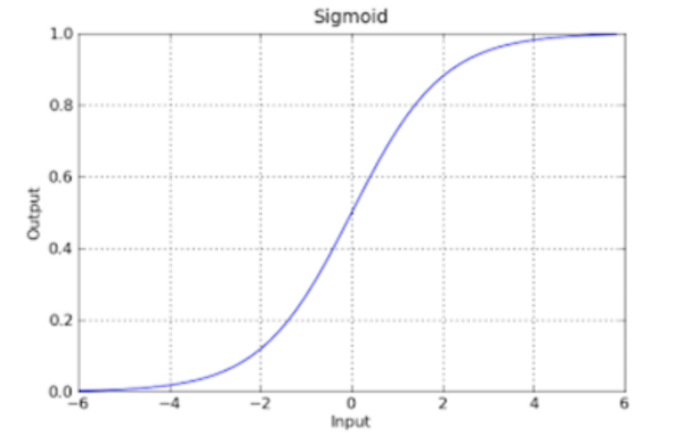


Figura 2.2: Función de activación Sigmoide

Se puede definir a partir de la siguiente ecuación:

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

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

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

- **Exp**

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

- **Log**

$$f(x) = \log(x) \quad (2.4)$$

- **BNLL**

$$f(x) = \log(1 + \exp(x)) \quad (2.5)$$

- **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 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 tuning 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 camerasetup en un terminal.

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

A continuación, en otro terminal, ejecutar cameraview.

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

## 2.3. Bases de datos

### 2.3.1. VOC

Esta base de datos contiene 11,530 imágenes de entrenamiento y validación que representan 27,450 objetos diferentes distribuidos en 20 clases. Los datos de entrenamiento proporcionados consisten en un conjunto de imágenes; cada imagen tiene un archivo de anotación que proporciona un cuadro delimitador y una etiqueta de clase de objeto para cada objeto en una de las veinte clases presentes en la imagen. Por lo tanto, la tarea de detección consiste en predecir el cuadro delimitador y la etiqueta de cada objeto de las veinte clases objetivo en la imagen de prueba. La siguiente imagen muestra un ejemplo de esta base de datos:

Es importante saber cuántas imágenes aparecen en cada uno de los 20 objetos para saber si la base de datos está bien escalada, o si, por el contrario, muchos objetos aparecen con más frecuencia que otros. En la base de datos VOC2012, las distribuciones de imágenes y objetos por clase son aproximadamente iguales en todos los conjuntos de entrenamiento / validación y prueba, y se pueden encontrar aquí. Específicamente, la distribución de objetos para la tarea de detección se muestra en la siguiente imagen.

	train		val		trainval		test	
	Images	Objects	Images	Objects	Images	Objects	Images	Objects
<b>Aeroplane</b>	327	432	343	433	670	865	-	-
<b>Bicycle</b>	268	353	284	358	552	711	-	-
<b>Bird</b>	395	560	370	559	765	1119	-	-
<b>Boat</b>	260	426	248	424	508	850	-	-
<b>Bottle</b>	365	629	341	630	706	1259	-	-
<b>Bus</b>	213	292	208	301	421	593	-	-
<b>Car</b>	590	1013	571	1004	1161	2017	-	-
<b>Cat</b>	539	605	541	612	1080	1217	-	-
<b>Chair</b>	566	1178	553	1176	1119	2354	-	-
<b>Cow</b>	151	290	152	298	303	588	-	-
<b>Diningtable</b>	269	304	269	305	538	609	-	-
<b>Dog</b>	632	756	654	759	1286	1515	-	-
<b>Horse</b>	237	350	245	360	482	710	-	-
<b>Motorbike</b>	265	357	261	356	526	713	-	-
<b>Person</b>	1994	4194	2093	4372	4087	8566	-	-
<b>Pottedplant</b>	269	484	258	489	527	973	-	-
<b>Sheep</b>	171	400	154	413	325	813	-	-
<b>Sofa</b>	257	281	250	285	507	566	-	-
<b>Train</b>	273	313	271	315	544	628	-	-
<b>Tvmonitor</b>	290	392	285	392	575	784	-	-
<b>Total</b>	5717	13609	5823	13841	11540	27450	-	-

Figura 2.3: Estructura básica de anotaciones en COCO

### 2.3.2. COCO

COCO es un conjunto de datos creado para la detección y segmentación de objetos y para generación de subtítulos a gran escala. Algunas de las características de esta base de datos son:

- Más de 300.000 imágenes
- 1.5 millones de instancias de objetos
- 80 categorías de objetos

Actualmente, COCO utiliza 3 tipos de anotaciones: instancias de objetos, puntos claves de objetos y leyendas de imágenes. Las anotaciones se almacenan usando el formato de archivo JSON. Todas las anotaciones comparten la estructura de datos básica definida a continuación:

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

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

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

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

Figura 2.4: Estructura básica de anotaciones en COCO

Para la tarea de detección, es de especial interés la anotación usando instancias de objetos. Cada anotación de instancia contiene una serie de campos, incluida la identificación de categoría y la máscara de segmentación del objeto. El formato de segmentación depende de si la instancia representa un solo objeto (`iscrowd = 0` en cuyo caso se usan polígonos) o una colección de objetos (`iscrowd = 1` en cuyo caso se usa RLE). Tenga en cuenta que un solo objeto (`iscrowd = 0`) puede requerir múltiples polígonos, por ejemplo, si está ocluido. Las anotaciones de multitudes (`iscrowd = 1`) se utilizan para etiquetar grandes grupos de objetos (por ejemplo, una multitud de personas). Además, se proporciona un cuadro delimitador para cada objeto (las coordenadas del cuadro se miden desde la esquina superior izquierda de la imagen y están indexadas en 0). Finalmente, el campo de categorías de la estructura de anotación almacena la asignación de los nombres de categoría a categoría y supercategoría.

```

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

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

```

Figura 2.5: Estructura básica de anotaciones en COCO

En referencia a los datos, el conjunto de datos MS COCO se divide en dos partes aproximadamente iguales. La primera mitad del conjunto de datos se lanzó en 2014, la segunda mitad se lanzará en 2015. La versión 2014 contiene 82,783 entrenamientos, 40,504 validaciones y 40,775 imágenes de prueba (aproximadamente 1/2 tren, 1/4 val y 1/4 prueba). Hay casi 270k personas segmentadas y un total de 886k instancias de objetos segmentados en los datos train + val 2014 solo. La versión acumulada de 2015 contendrá un total de 165,482 imágenes de trenes, 81,208 val y 81,434 de prueba.

La distribución de los objetos en esta base de datos se puede obtener desde su sitio web. En la sección Explorar, es posible elegir y combinar cada uno de los objetos y observar cuántas imágenes aparecen. La distribución de cada uno de los objetos en el conjunto de entrenamiento / validación se muestra en la siguiente imagen:

	images		images		images		images
person	66808	cat	4298	wine glass	2643	dinning table	12338
backpack	5756	dog	4562	cup	9579	toilet	3502
umbrella	4142	horse	3069	fork	3710	tv	4768
handbag	7133	sheep	1594	knife	4507	laptop	3707
tie	3955	cow	2055	spoon	3682	mouse	1964
suitcase	2507	elephant	2232	bowl	7425	remote	3221
bicycle	3401	bear	1009	banana	2346	keyboard	2221
car	12786	zebra	2001	apple	1662	cell phone	5017
motorcycle	3661	giraffe	2647	sandwich	2463	microwave	1601
airplane	3083	frisbee	2268	orange	1784	oven	2992
bus	4141	skis	3202	broccoli	2010	toaster	225
train	3745	snowboard	1703	carrot	1764	sink	4865
truck	6377	sports ball	4431	hot dog	1273	refrigerator	2461
boat	3146	kite	2352	pizza	3319	book	5562
traffic light	4330	baseball bat	2603	donut	1585	clock	4863
fire hydrant	1797	baseball glove	2729	cake	3049	vase	3730
stop sign	1803	skateboard	3603	chair	13354	scissors	975
parking meter	742	surfboard	3635	couch	4618	teddy bear	2234
bench	5805	tennis racket	3561	potted plant	4624	hair drier	198
bird	3362	bottle	8880	bed	3831	toothbrush	1041

Figura 2.6: Estructura básica de anotaciones en COCO

# 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 los 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 tengan 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.

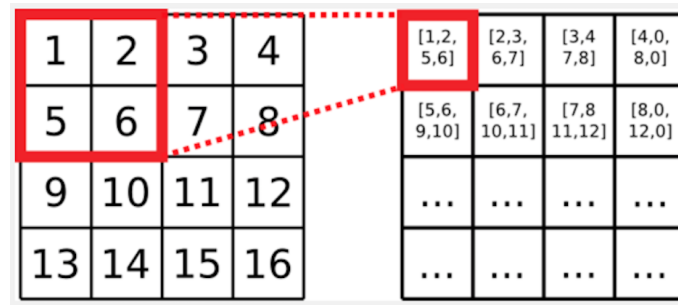


Figura 3.1: Ejemplo convolución con tamaño del núcleo 2x2

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

```

layer {
  name: "conv1"
  type: "Convolution"
  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"
    }
  }
  bottom: "data"
  top: "conv1"
}

```

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  $m \times n$ . A continuación, se aplica una función de agrupación para cada uno de los bloques. Tras este proceso, se obtendrá una matriz de características más pequeña. 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.

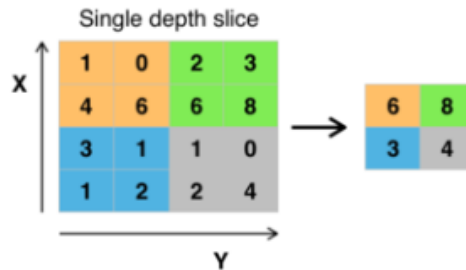


Figura 3.2: Pooling con función de agrupación del máximo

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: "conv1"
  top: "pool1"
}
```

Esta capa, tomará los datos de la capa de convolución anterior y utilizará bloques de  $2 \times 2$  para dividir la imagen entrante. Para que no haya solapa-

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

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: "ip1"  
  top: "ip1"  
}
```

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

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

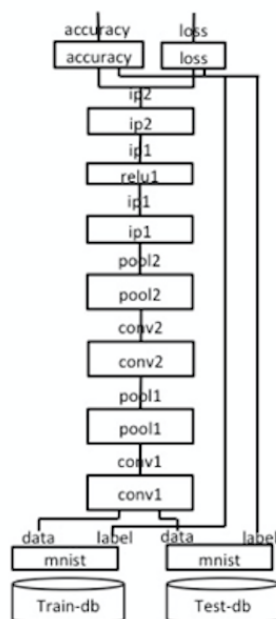


Figura 3.3: Estructura red neuronal

### 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: "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: 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. Acceso a datos y etiquetas

La base de datos utilizada tendrá una extensión `.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.

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.

### 3.2.2. 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.

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

Figura 3.4: Fichero PGM

3

25



### 3.2.3. 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 refLMDB.

```
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 2.3.2.

#### 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, textit-move, el tamaño, textitresize, o el estilo de la ventana, textitsetStyleSheet.

```
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; co-
lor: #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 está mostrando.

### 3.3.3. Ejecución

Para su ejecución, en primer lugar, habrá que arrancar la herramienta `CameraServer` explicada en el apartado 2.3.2.

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

Para la realización de las pruebas se van a utilizar varias bases de datos con el objetivo de medir la capacidad de clasificación que tiene la red neuronal entrenada.

Todas estas bases de datos se van a crear a partir de la utilizada para el entrenamiento, es decir, la base de datos MNIST, pero utilizando diferentes tipos de filtros y funciones de translación, rotación y escalado entre otras, para observar cómo actúa la red frente a estas situaciones.

Es posible la creación de las bases de datos con comandos Python. Como se comentó en el apartado 3.2.1, se puede acceder a la imagen, data, y a su etiqueta asociada, label. A partir de aquí, operaremos con diferentes funciones de OpenCV sobre la variable data, transformando cada una de las imágenes de la base de datos.

### 3.4.1. Filtro Canny

El algoritmo Canny consiste, principalmente, en 3 grandes pasos:

- Obtención del gradiente: en este paso se calcula la magnitud y orientación del vector gradiente en cada píxel.
- Supresión no máxima: en este paso se consigue el adelgazamiento del ancho de los bordes obtenidos con el gradiente hasta lograr que estos tengan de un píxel de ancho.
- Histéresis del umbral: aquí se aplica una función de histéresis basada en 2 umbrales. Con este proceso se pretende reducir la posibilidad de aparición de contornos falsos.

OpenCV aplica todo esto en una función, `cv2.Canny()`. El primer argumento es la imagen de entrada. El segundo y tercer argumento son el umbral mínimo y el umbral máximo respectivamente. El cuarto argumento hace referencia al tamaño del núcleo encargado de buscar gradientes de la imagen, el cual por defecto es 3. Finalmente, el último argumento es `L2gradient` que especifica la ecuación para encontrar la magnitud del gradiente. Por defecto este parámetro tiene valor `false`, haciendo referencia a la ecuación  $Edge\_Gradient = -G_x - + -G_y$ .

### 3.4.2. Filtro Sobel

El filtro Sobel es uno de los más utilizados para el procesamiento de imágenes, en especial para la detección de bordes. Se trata de un operador diferencial discreto que calcula una aproximación al gradiente de la función

de intensidad de una imagen. Para cada punto de la imagen a procesar, aplicando este filtro se obtiene tanto el vector gradiente correspondiente como la norma de ese vector.

En OpenCV, el filtro sobel se simula utilizando la función `cv2.Sobel()`

### 3.4.3. Filtro Laplaciano

El filtro Laplaciano es una medida 2-D isotrópica de la segunda derivada espacial de una imagen. Aplicar este filtro sobre una imagen destaca las regiones donde hay cambios de bruscos de intensidad y por lo tanto es un filtro muy utilizado en la detección de bordes.

The diagram illustrates the decomposition of a 3x3 grid into a 3x3 grid of 1x1 cells and a 3x3 grid of 1x1 cells. The top 3x3 grid is composed of 1x1 cells, and the bottom 3x3 grid is also composed of 1x1 cells. The values in the cells are as follows:

1	1	1
1	-8	1
1	1	1

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 3.6: Máscaras de un filtro Laplaciano

OpenCV aplica un filtro Laplaciano aplicando la función, `cv2.Laplacian()`

### 3.4.4. Casos de pruebas

Para la realización de las pruebas, en primer lugar, se han creado 6 bases de datos con el objetivo de entrenar la red neuronal existente. Estas, se crearán a partir de la base de datos MNIST, es decir, constarán de 60000 imágenes para el entrenamiento, pero se aplicarán sobre ellas varias funciones de filtrado comentadas anteriormente. Se crearon una utilizando el filtro

Canny, otra utilizando Sobel y otras 4 más usando el filtro Laplaciano variando el tamaño del núcleo [3,5,7,9].

Posteriormente, se crearon 5 bases de datos para realizar el test de las redes neuronales entrenadas anteriormente. Con el objetivo de testear la precisión de la red, se han añadido a cada una de ellas el negativo de cada una de las imágenes, siendo finalmente 20000 las imágenes totales de cada una de las bases de datos. De esta manera, tendremos las siguientes bases de datos:

- Base de datos con 10000 imágenes filtradas utilizando el filtro Canny y el negativo de cada una de ellas.
- Base de datos con 10000 imágenes filtradas utilizando el algoritmo de Sobel y el negativo de cada una de ellas.
- Base de datos con 10000 imágenes filtradas utilizando un filtro Laplaciano con tamaño de núcleo de 3 y el negativo de cada una de ellas.
- Base de datos con 10000 imágenes filtradas utilizando un filtro Laplaciano con tamaño de núcleo de 5 y el negativo de cada una de ellas.
- Base de datos con 10000 imágenes filtradas utilizando un filtro Laplaciano con tamaño de núcleo de 7 y el negativo de cada una de ellas.
- Base de datos con 10000 imágenes filtradas utilizando un filtro Laplaciano con tamaño de núcleo de 9 y el negativo de cada una de ellas.

Tras la realización del test, el resultado es el siguiente:

Hasta ahora, las redes neuronales se han testado con imágenes centradas y siempre del mismo tamaño. Para comprobar la precisión de la red ante imágenes de entrada de diferentes tamaños y posiciones, se han creado 6 nuevas bases de datos, cada una de ellas con 60000 imágenes. Todas ellas, utilizan el filtro Sobel sobre cada una de las imágenes y posteriormente se realizan varias transformaciones sobre cada imagen, dando lugar a que, dentro de la base de datos, existan los siguientes tipos de imágenes.

- Imágenes rotadas
- Imágenes en diferentes posiciones
- Imágenes de mayor o menor tamaño

- Imágenes con ruido gaussiano de varianza 0.2
- Imágenes con ruido gaussiano de varianza 0.5
- Imágenes utilizando todas las transformaciones anteriores.

Después de realizar el test, los resultados serían los siguientes:

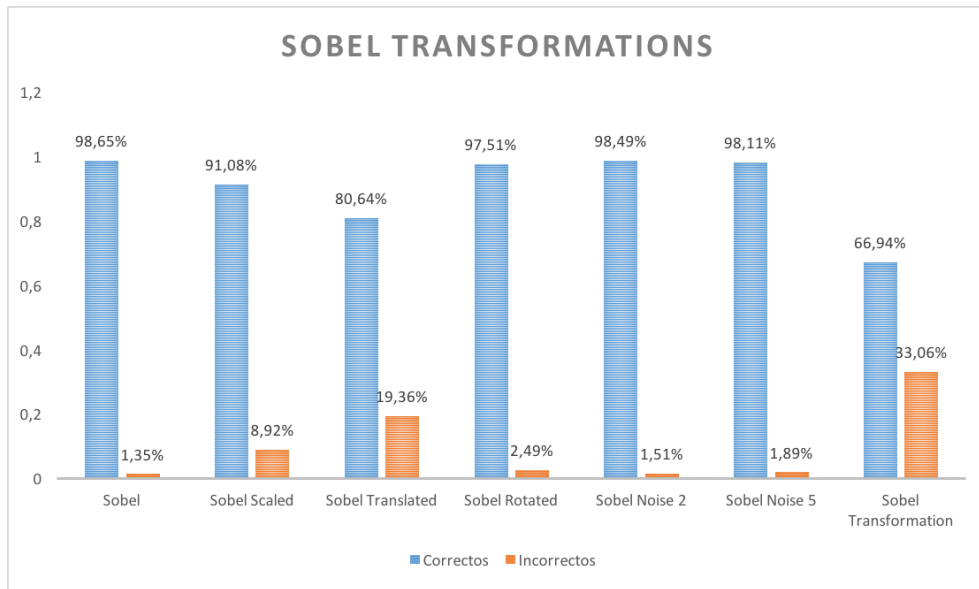


Figura 3.7: Resultados tras varias transformaciones en las imágenes



# Capítulo 4

## Detección

### 4.1. Red Neuronal

#### 4.1.1. Estructura de la red

### 4.2. Jaccard index

### 4.3. Componente Python

### 4.4. Medidor de calidad

## Capítulo 5

## Bibliografía

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