SUPERIOR TECHNICAL SCHOOL OF TELECOMMUNICATION ENGINEERING

Media studies and
Audiovisual Systems engineering

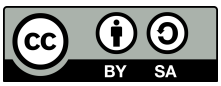
Bachelor's Degree Final Project

Vehicle Detection using Deep Learning

Author: David Pascual Hernández

Tutors: José María Cañas Plaza, Inmaculada Mora Jiménez

Academic year 2016/2017



Contents

List of Figures

Chapter 1

Framework

This chapter serves as a way to introduce the tools that have been employed during the development of this project. All of them are **open-source**. The transparency provided by the open-source platforms is a major advantage, because the software can be joined together and adapted to our specific applications, which are mainly written in **Python** [1].

1.1 Keras

As stated by **Keras** documentation [1]: "Keras is a high-level **neural network library**, written in Python and capable of running on top of either TensorFlow or Theano". TensorFlow and Theano are open-source libraries for numerical computation optimized for GPU and CPU that Keras treats as its *backends*. In this project, Keras is running on top of **Theano** [2] optimized for CPU, but it's quite easy to switch from one backend to another.

In the following sections, the main elements that make up a neural network built with Keras are going to be analyzed, starting with the ***model* object**, its core component.

1.1.1 Models

Every neural network in Keras is defined as a ***model***. For those models which can be built as a stack of *layers* 1.1.2, Keras provides the ***.Sequential()* object**. An example of a sequential model built with Keras can be seen in the following chapter in the figure

[1] `https://www.python.org/`

[2] `http://deeplearning.net/software/theano/index.html`

2.2. It is also possible to build more complex models with multiple outputs and shared layers using the **Keras functional API**.

Sequential models have several methods, and the following ones are essential for the learning process:

.compile() It configures the learning process. It's main arguments are:

- ***loss***: name of the **cost function** employed to check the difference between the predicted label and the real one. In this project, the **categorical cross-entropy**, also known as log loss, has been used. It returns the cross-entropy between an approximating distribution q and a true distribution p [20] and it's defined as:

$$H(p, q) = -\Sigma_x p(x) \log(q(x)) \tag{1.1}$$

  Other loss functions such as mean squared error (MSE), mean absolute error and hinge are also provided by Keras.

- ***optimizer***: name of the optimizer that will update the weights values during training in order to minimize the loss function. The chosen algorithm for this task is **ADADELTA** [22]. This optimizer is an extension of the **gradient descent** optimization method that has the particularity of adapting the learning rate during training with no need of manual tuning. According to the paper in which it is defined [22], it follows the next algorithm:

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ

Require: Initial parameter x_1

1: Initialize accumulation variables $E[g^2]_0 = 0, E[\Delta x^2]_0 = 0$

2: **for** $t = 1 : T$ **do** ▷ Loop over # of updates

3: Compute gradient: g_t

4: Accumulate gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2$

5: Compute update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$

6: Accumulate updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1 - \rho) \Delta x_t^2$

7: Apply update: $x_{t+1} = x_t + \Delta x_t$

8: **end for**

Other optimization methods such as Adagrad, Adamax and Adam are also available.

- ***metrics***: name of the parameters that must be evaluated during training and testing. The only metric that is going to be calculated with Keras through this project, besides the result of the loss function which is automatically computed, is **accuracy**. It is defined as the proportion of examples for which the model produces the correct output [4]. Other metrics are calculated with **Scikit-learn** 1.6 library, in order to obtain a evaluation of the model that is independent from Keras.

.fit() It trains the model. The following arguments are required:

- ***x***, ***y***: training samples and labels. They must be defined as **Numpy arrays**[3].
- ***batch_size***: number of samples that are evaluated before updating the weights. It defaults to 32.
- ***epochs***: number of iterations over the whole dataset that are going to be executed. It defaults to 10.
- ***callbacks***: list of callbacks 1.1.3 that are going to be applied during training. It defaults to *None*.
- ***validation_split* or *validation_data***: On one hand, *validation_split* defines the fraction of the training data that has to be used as held-out validation data. On the other hand, *validation_data* is a tuple containing the samples and labels of a validation dataset provided by the user. They are mutually exclusive.
- ***shuffle***: a boolean that determines whether to shuffle training data or not.

.evaluate() It takes a set of samples and labels and evaluates the **model performance**, returning a list of the metrics previously defined.

.predict() It takes a sample and returns the label predicted by the model.

.save() It stores the model into a **HDF5 file** 1.5, which will contain the weights, architecture and training configuration of the model.

.load_model() It loads a model from a **HDF5 file**.

[3] `http://www.numpy.org/`

1.1.2 Layers

As it has been said before, the models are usually built as a **stack of layers**. These layers are added to the model using the ***.add()* method**, inside of which the kind of layer is declared and its particular parameters are set. Several kinds of layers are available, but only the ones that have been used in this project are going to be described.

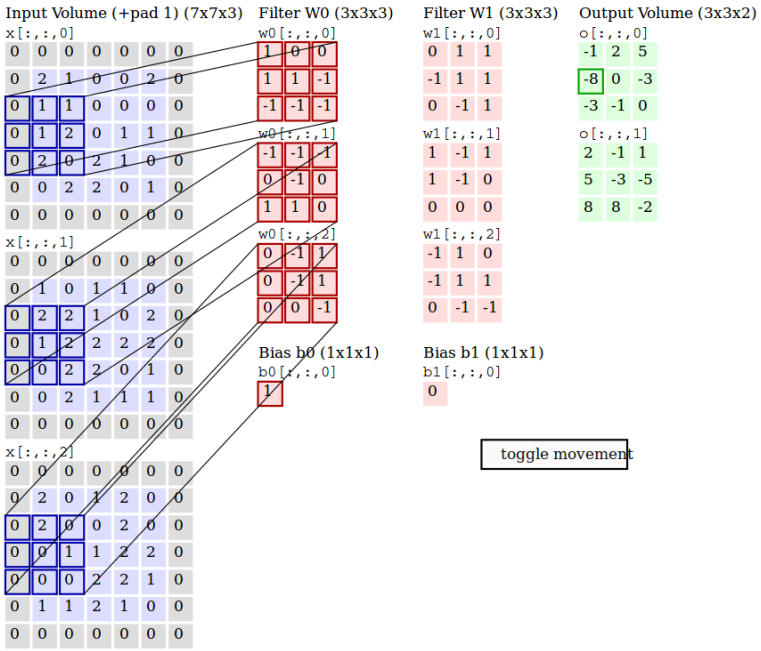
Convolutional layer This particular layer is the one that turns the neural network into a **convolutional neural network (CNN)**. It is formed by a fixed number of **filters/kernels** with a fixed size. These filters are convolved along the input image, generating each one a **feature or activation map** which will tell us to what extent the feature learned by that particular filter is present in the input image [10]. It's important to note that the *depth* of the filter will be equal to the number of channels of the input, which implies that each filter will generate just one activation map, instead of generating one for each channel.

Keras provides different kinds of convolutional layers depending on the input dimensions: *Conv1D*, *Conv2D* and *Conv3D*. These are the main arguments required by Keras to define a convolutional layer:

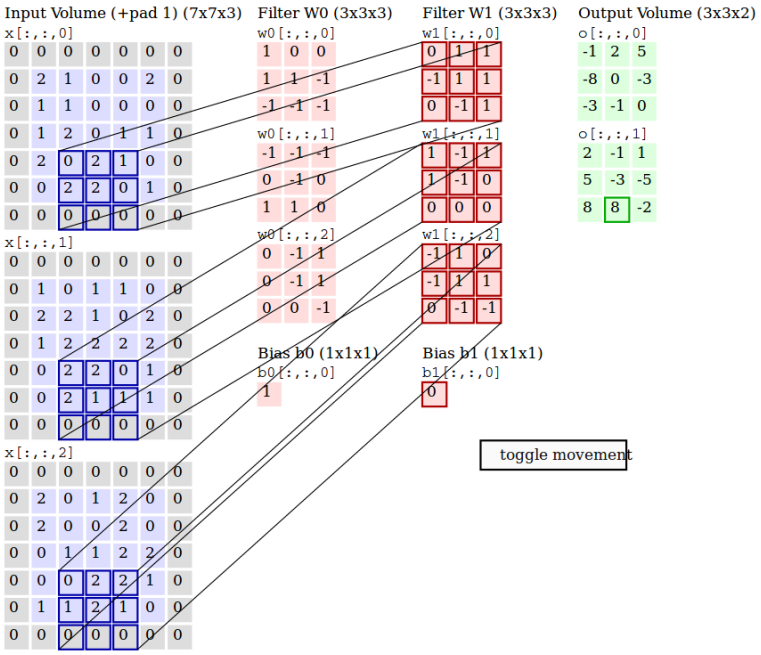
- ***filters***: number of filters.
- ***kernel_size***: width and height of the filters.
- ***strides***: how many pixels the filter must be shifted before applying the next convolution. Output size depends on this parameter. It defaults to 1.
- ***padding***: it can be *valid* or *same*. If *valid* mode is set, no padding is applied, resulting in a reduced output. However, if *same* mode is set, the input will be padded with zeros in order to produce an output that preserves the input size. It defaults to *valid*.

Figure 1.1 shows how the convolutional layers work.

Pooling layer It shifts a window of a certain size along the input image applying an operation (mean or maximum) that will return a ***downsampled* version** of it, reducing the computational cost and avoiding over-fitting [15]. Figure 1.2 shows how the pooling operation is applied.



(a)



(b)

Figure 1.1: In the figure 1.1a, the filter w_0 (3x3x3) is convolved with the input image (5x5x3). As padding is set to 1 pixel around the input and the stride is equal to 2 pixels, the operation will return a 3x3 activation map. The same procedure is followed in the figure 1.1b with the filter w_1. It generates another 3x3 activation map, ending up with a 3x3x2 output (depth = number of filters = 2). These images have been extracted from [10]

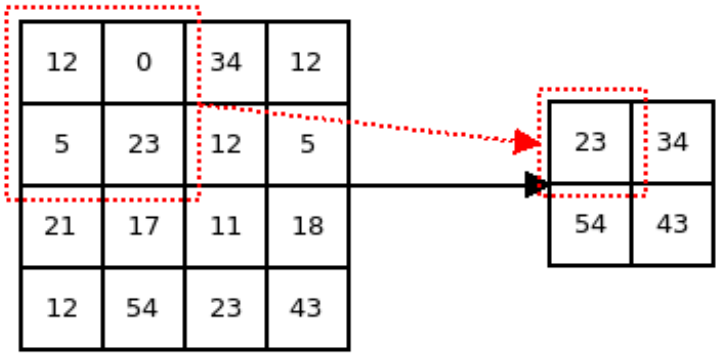


Figure 1.2: Example of a max. pooling operation

Depending on the dimensions of the input and the operation applied, Keras provides several pooling layers: *MaxPooling1D*, *MaxPooling2D*, *MaxPooling3D*, *AveragePooling1D*... The main arguments required by Keras to define these layers are:

- ***pool_size***: size of the window that is shifted along the input. It can also be interpreted as the factor by which the input is going to be downsampled.
- ***strides***: how many pixels the window must be shifted before applying the next operation.

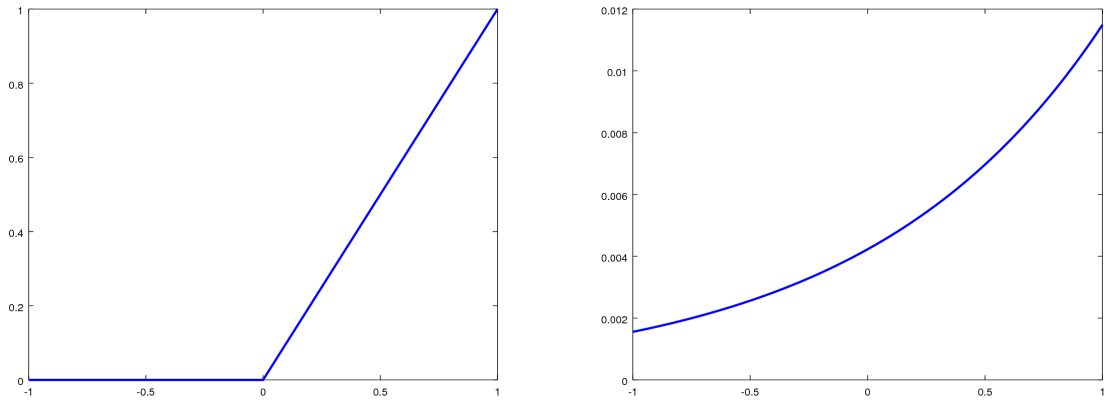
Dense layer Fully-connected layers in Keras are defined as *Dense layers*. In a **fully-connected layer**, every neuron is connected to every activation (output) of the previous one [10]. The main argument of this layer is:

- ***units***: number of neurons.

Activation layers In Keras models, activations can be declared as a layer itself, or as an argument within the *.add()* method of the previous layer. Keras provides several **activation functions**, such as sigmoid, linear, ReLU and softmax. The only argument that must be provided to activation layers is the name of the desired activation function. These are the ones that have been used during the development of this project:

- **ReLU (Rectified Linear Unit)**: This activation function introduces **non-linearity** right after each convolutional layer, allowing the CNN to learn more complex features. It's defined as:

$$g(z) = \max(0, z) \tag{1.2}$$



(a) ReLU activation function

(b) Softmax activation function

Figure 1.3

- **Softmax** This activation function is very useful when is placed after the **output layer** of classification tasks. It takes a vector of real values z and returns a new vector of real values in the range [0,1]. The N elements of the output vector can be considered **probabilities** because the softmax function ensures that they sum up to 1. It is defined as follows:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\Sigma_j \exp(z_j)} \quad \text{for } j = 1, ..., N \tag{1.3}$$

These equations have been extracted from [4]. Figure 1.3 shows these activation functions plotted in the interval [-1,1].

Flatten layer It *flattens* the input. For instance, it converts the activation maps returned by the convolutional layers into a **vector of neurons** before being connected to a dense layer. It takes no arguments.

Dropout layer It's considered a **regularization layer**, because its main purpose is to avoid over-fitting. Dropout [18] is a technique that randomly ***switches-off*** a fraction of hidden units during training, both forward and backward propagation. Another point of view of how dropout works can be seen in figure 1.4.

This layer, as other regularization layers (i.e. GaussianNoise layer), is only active during training. It's main argument is:

- ***rate***: fraction of units that must be dropped.

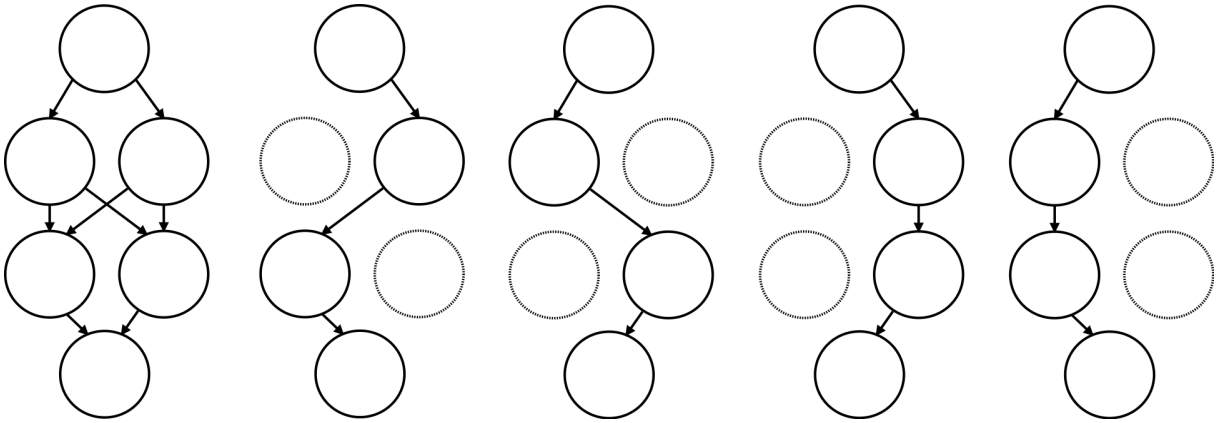


Figure 1.4: Dropout can be understood as a technique that "trains an ensemble consisting of all subnetworks that can be structured by removing non-output units from an underlying base network" [4].

1.1.3 Callbacks

As defined by Keras documentation [1], **callbacks** are a set of functions which are applied at given stages while the model is being trained. They can be used to take a look at the state of the model during training. The built-in callbacks that have been used for this project are:

- ***.History()***: it is automatically applied to every Keras model and is returned by the *.fit()* method. It evaluates the declared metrics with the validation set after each epoch and saves the results.

- ***.EarlyStopping()***: it monitors the value of a given metric and forces the model to stop training when that metric has stopped improving. It has a ***patience*** argument which determines how many epochs in a row without improving must be tolerated before the model quits training. Setting up an appropriate **stopping criteria** may prevent the model from over-fitting.

- ***.ModelCheckpoint()***: it saves the model and its weights after each epoch. It can be configured to overwrite the model only if a certain metric has improved with respect to the previous best result, saving the best *version* of it.

Additionally, Keras provides the *Callback* base class that can be used to build **user-defined callbacks**.

1.1.4 Image Preprocessing

Image preprocessing is a key factor in every computer vision application. Specifically, in machine learning, besides adapting the image and extracting features before the training that can improve the model performance (i.e. edge extraction), it can be used to avoid **over-fitting** through data augmentation. **Data augmentation** [21] consists in taking the samples that the dataset already contains and applying transformations to them, generating new samples that may be closer to real world and, in any case, enlarging the dataset with new data.

This functionality is included in Keras thanks to the ***.ImageDataGenerator()* method**. It returns a batch generator which randomly applies the desired **transformations** to random samples of the dataset provided by the user. Built-in transformations like rotation, shifting and zooming, are passed as arguments to the aforementioned method. Additionally, it's possible to build a user-defined function and pass it as an argument as well. The dataset, and the batch size are defined through the ***.flow()* method**. During training, the generator will loop until the number of samples per epoch and the number of epochs set by the user are satisfied.

1.1.5 Utils

Keras include a module for multiple supplementary tasks called ***Utils***. The most important functionality for the project provided by this module is the ***.HDF5Matrix()* method**. It reads the **HDF5 datasets** 1.5, which are going to be used as inputs to the neural networks.

1.2 JdeRobot

JdeRobot [7] is an open source middleware for robotics and computer vision. It has been designed to simplify the software development within these fields. It's mostly written in C++ language and it's structured like a collection of components (tools and drivers) that communicate to each other through **ICE interfaces** [4]. It is also compatible with **ROS** [5], which allows the interoperation of ROS nodes and JdeRobot components. This

[4] `https://zeroc.com/products/ice`

[5] `http://www.ros.org/`

flexibility makes it very useful for our application. Its ***cameraserver* driver** is going to be employed to capture images from different video sources.

1.2.1 *cameraserver*

According to JdeRobot documentation [7], this driver can serve both real cameras and video files. It communicates with other components thanks to the ***Camera* interface**.

In order to use ***cameraserver***, its configuration file has to be properly set. These are the parameters that must be specified:

- The **network address** where the server is going to be listening.
- Parameters related with the **video stream**: URI, frame rate, image size and format.

1.3 DroidCam

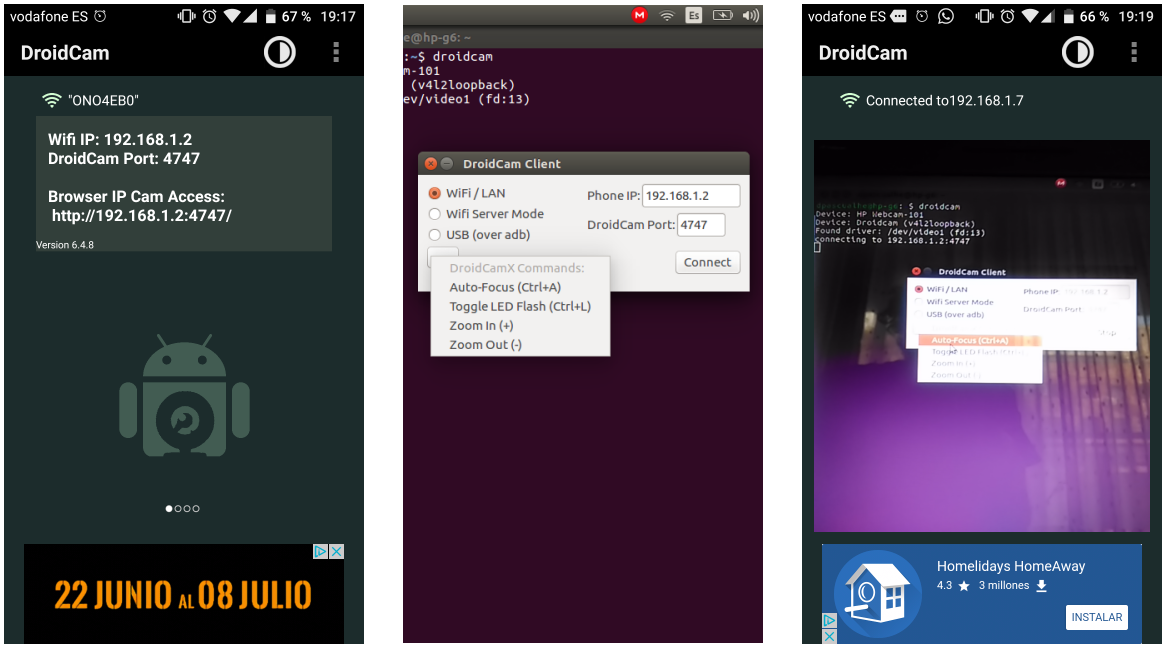
On one hand, **DroidCam** [3] is an application for Android which serves the images captured with a **smartphone camera**. On the other hand, it is **a client for Linux** which receives the video stream served by Android and makes it accessible for the computer as a **v4l2** [6] **device driver**. The Linux client can be connected to the phone camera over a **USB** cable or a **WiFi network** and allows the user to control camera flash, auto-focus and zoom. DroidCam provides the address (IP and port) at which the Linux client must be listening to receive the images. In the figure 1.5 can be seen how it works. Additionally, the Android application provides a **URL** that can be used to access the video stream from any browser.

1.4 MNIST

MNIST (Modified National Institute of Standards and Technology database) [11] is a database of **handwritten digits** formed by a training set, which contains 60000 samples, and a test set, containing 10000 samples. It's a *remixed* and reduced version of the original **NIST datasets** [7]. MNIST is a well-known **benchmark** for all kinds of machine learning algorithms.

[6] `https://www.linuxtv.org/wiki/index.php/Main_Page`

[7] `https://www.nist.gov/srd/nist-special-database-19`



(a) (b) (c)

Figure 1.5: First, the Android APP is opened. It shows the address where the video is served (figure 1.5a). Then, the adress is set in the Linux client (figure 1.5b). Finally, when the *Connect* button is pressed, the connection is established (figure 1.5c)

As may be seen in figure 1.6, each sample of the MNIST database is a 28x28 pixels **grayscale image** that contains a **size-normalized** and **centered** digit. While it may be useful for testing machine learning algorithms, it's not enough to train a model that aims to solve a **real-world task**, because the images are almost noiseless and share similar orientation, position, size and intensity levels. The dataset must be **augmented** to face this problem 2.3.

1.5 HDF5

During the development of this project, **huge amounts of data** have been processed. That's why an efficient way of reading and saving this data has been an important point. Keras employs the **HDF5 file format** to save models and read datasets.

According to HDF5 (Hierarchical Data Format) documentation [19], it is designed for high volumes of data with complex relationships. While relational databases employ tables to store data, HDF5 supports **n-dimensional datasets** and each element in the dataset may be as complex as needed.

Figure 1.6: Samples extracted from the MNIST database

In order to deal with HDF5 files, the **h5py** [8] library for Python has been employed.

1.6 Scikit-learn

Scikit-learn [12] is a **machine learning library** that includes a wide variety of algorithms for clustering, regression and classification. It can be used during the whole machine learning process: preprocessing, training, model selection and evaluation.

Scikit-learn functions have been used to evaluate the neural networks developed with Keras. Using a tool that is **independent from Keras** enables the comparison of the results achieved by different neural network libraries (e.g. Keras and Caffe). These are the **metrics** employed in this project (equations obtained from [16]):

- **Precision**: ability of the classifier not to label as positive a sample that is negative.

$$\text{precision} = \frac{true_{positives}}{true_{positives} + false_{positives}} \tag{1.4}$$

- **Recall**: ability of the classifier to find all the positive samples.

$$\text{recall} = \frac{true_{positives}}{true_{positives} + false_{negatives}} \tag{1.5}$$

[8] `http://www.h5py.org/`

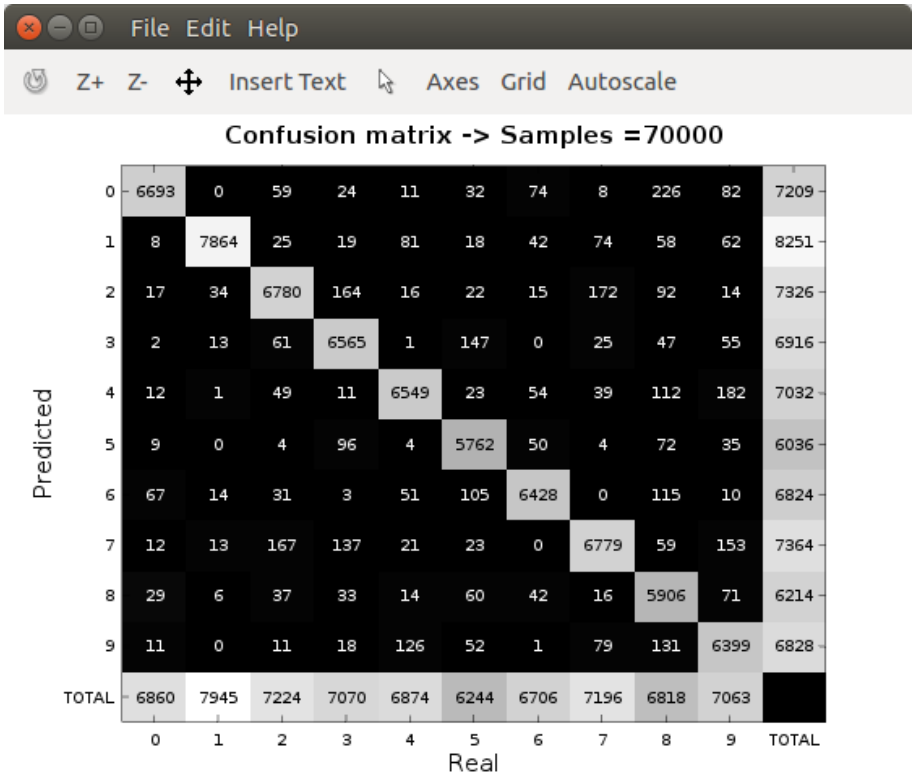


Figure 1.7: Example of a confusion matrix visualization using Octave

- **Confusion matrix**: a matrix where the element i, j represents the number of samples that belongs to the group i but has been classified as belonging to group j. True predictions can be found in the diagonal of the matrix, where $i = j$. An example of a confusion matrix constructed with Scikit-learn and displayed with Octave 1.7 can be found in figure 1.7.

Besides the metrics that have just been mentioned, **accuracy** and **log loss** have also been used and they're defined as in section 1.1.1.

1.7 Octave

GNU Octave [9] is a scientific programming language compatible with **Matlab**. It provides powerful tools for **plotting**, which have been used to visualize the data collected with Scikit-learn about the performance of the models. An example of Octave usage can be seen in the figure 1.7.

Chapter 2

Digit classifier

Taking advantage of the **convolutional neural networks (CNN)** impressive performance in classification tasks, we have built a **real-time digit classifier** which captures images from any video stream, applies the necessary preprocessing and displays the predicted digit.

These are the major challenges that have been overcome during the development of the application:

- Understanding **how the CNNs are built** with Keras.
- Developing a **component** for the application.
- Creating a **dataset** with images that resemble the ones found in the real world.
- Building a **benchmark** to evaluate the performance of the CNNs.
- Finding the **optimal CNN model**: architecture and learning process.

2.1 Understanding the Keras model

The starting point for the digit classifier is a **Keras example** that can be accessed from its GitHub [1]. In that example, a CNN is trained and tested with the **MNIST dataset** 1.4. That code has been used to take the first steps in this project and an adapted version of it [2] is going to be analyzed in the following sections.

[1] `https://git.io/vH0qw`
[2] `https://git.io/vH0qK`

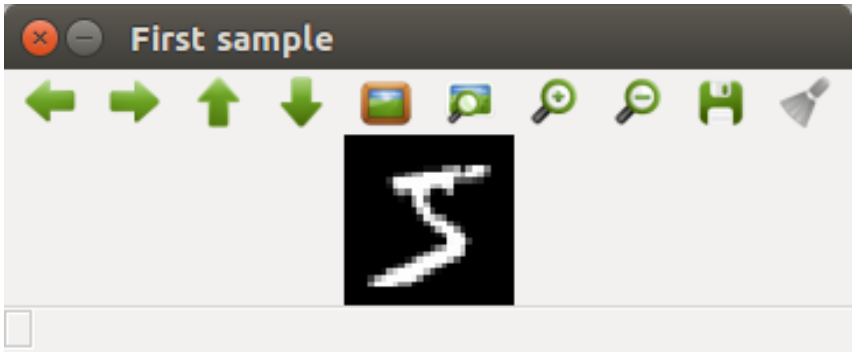


Figure 2.1: First sample of the MNIST database

2.1.1 Preparing data

First of all, the input data has to be **loaded** and **adapted**. Keras library contains a module named ***datasets*** from which a variety of databases can be imported, including MNIST 1.4. The MNIST database can be loaded calling the ***mnist.load_data()*** **method**. It returns, as **Numpy arrays**, the images and labels from both training and test datasets: ***x_train***, ***y_train*** and ***x_test***, ***y_test***, respectively.

```
'''
Loading and shaping data in a way that it can work as input of our model
'''
# MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

cv2.imshow('First sample',x_train[0])
cv2.waitKey(5000)
cv2.destroyWindow('First sample')

print ('Original input images data shape: ', x_train.shape)
```

In the figure 2.1, the **first sample** of the MNIST training dataset, as displayed by the code above, can be seen. That code also prints the shape of the dataset:

```
Original input images data shape:  (60000, 28, 28)
```

This means that the training dataset includes 60000 images, each one containing 28x28 pixels. In order to feed a Keras model 1.1.1, the **number of channels** of the samples have to be explicitly declared as well, so the dataset must be **reshaped**. In this case, the

samples are **grayscale images**, which implies that the number of channels is equal to 1 (e.g. if they had been RGB images, the number of channels would be equal to 3). As data is now stored in Numpy arrays, it can be reshaped using the ***.reshape()* method**. The order in which dimensions must be declared depends on the ***.image_dim_ordering()* parameter** of the backend.

```
if backend.image_dim_ordering() == 'th':
# reshapes 3D data provided (nb_samples, width, height) into 4D
# (nb_samples, nb_features, width, height)
   x_train = x_train.reshape (x_train.shape[0], 1, img_rows, img_cols)
   x_test = x_test.reshape (x_test.shape[0], 1, img_rows, img_cols)
   input_shape = (1,img_rows,img_cols)
   print ('Input images data reshaped: ', (x_train.shape))
   print ('------------------------------------------------------------')
else:
# reshapes 3D data provided (nb_samples, width, height) into 4D
# (nb_samples, nb_features, width, height)
   x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
   x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
   input_shape = (img_rows, img_cols, 1)
   print ('Input images data reshaped: ', (x_train.shape))
   print ('------------------------------------------------------------')
```

In this case, the input data gets reshaped as follows:

```
Input images data reshaped:  (60000, 28, 28, 1)
```

The last step to get input images ready is to **convert data type** from *uint8* to *float32* and **normalize pixel values** to [0,1] range:

```
# converts the input data to 32bit floats and normalize it to [0,1]
print('Input images type: ',x_train.dtype)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
print('New input images type: ',x_train.dtype)
print ('------------------------------------------------------------')
x_train /= 255
```

```
x_test /= 255
```

Which outputs:

```
Input images type:  uint8
New input images type:  float32
```

Finally, the **labels** have to be reshaped from an array in which each element is an integer in the range [0, 9] to an array in which each element is **an array of probabilities**. E.g., if the element of the original array is 2, in the reshaped array it will be [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]. This conversion is achieved using the Keras built-in method ***np_utils.to_categorical()***.

```
print ('First 10 class labels: ', (y_train[:10]))
print ('Original class label data shape: ', (y_train.shape))
# converts class vector (integers from 0 to nb_classes) to class matrix
# (nb_samples, nb_classes)
y_train = np_utils.to_categorical(y_train, nb_classes)
y_test = np_utils.to_categorical(y_test, nb_classes)
print ('Class label data reshaped: ', (y_train.shape))
print ('----------------------------------------------------------------')
```

This code prints:

```
First 10 class labels:  [5 0 4 1 9 2 1 3 1 4]
Original class label data shape:  (60000,)
Class label data reshaped:  (60000, 10)
```

2.1.2 Model architecture

Once the data is ready, it's time to define the architecture of the CNN. In this example, a **sequential model** 1.1.1 is enough for solving the classification task and it is declared as follows:

```
# defines the model architecture, in this case, sequential
model = Sequential()
```

The next step is to add input, hidden and output layers. The core layers of a CNN, as treated by Keras, have been already defined in section 1.1.2.

```
'''
Adding layers to our model
'''
# convolutional layer
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1],
        border_mode='valid', input_shape=input_shape,
        activation='relu'))
# convolutional layer
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1],
        activation='relu'))
# pooling layer
model.add(MaxPooling2D(pool_size=pool_size))
# dropout layer
model.add(Dropout(0.25))

# flattening the weights (making them 1D) to enter fully connected layer
model.add(Flatten())
# fully connected layer
model.add(Dense(128, activation='relu'))
# dropout layer to prevent overfitting
model.add(Dropout(0.5))
# output layer
model.add(Dense(nb_classes, activation='softmax'))
```

As defined by the code above, the model is formed by the following layers:

- A **2D convolutional layer** with 32 filters whose size is 3x3x1.
  - We have to provide the argument ***input_shape*** in the first layer of our model, in this case, 28x28x1.
  - As *valid* mode is set, **no padding** is applied and the output dimension will be reduced.
  - **ReLU activation function** 1.2 introduces non-linearity into the network.

  - This layer outputs 32 activation maps with size 26x26.

- Another **convolutional layer** with the **same arguments**: 32 filters, no padding and ReLU as activation function.

  - Increasing the number of convolutional layers allows the CNN to learn **more complex features**.
  - As the **depth** of its input is 32 (one channel per activation map), the size of the filters will be 3x3x32.
  - This layer outputs 32 activation maps with size 24x24.

- A **2D MaxPooling layer** with a ***pool_size*** of 2x2.

  - This layer outputs the 32 activation maps generated by the previous layer, but ***downsampled*** by a factor of 2, resulting in maps with size 12x12.

- A **dropout layer** to prevent **over-fitting**.

  - The fraction of random units that are going to be ***swicthed-off*** is 0.25.
  - This layer preserves the size and the shape of its input.

- A **flatten layer** that turns the matrices of weights that it receives at its input into a vector that can be fed to the fully-connected layer.

- A fully-connected or **dense layer**.

  - This layer contains 128 neurons that will output an array of 128 values.
  - Once more, the **ReLU activation function** is applied.

- A **dropout layer** with a 0.5 fraction.

- Finally, the **output layer** is another **dense layer** which contains as many neurons as classes, in this example, 10.

  - In order to output a **probability distribution** of the predicted classes, the activation function will be **softmax** 1.3.

The resulting architecture and the shape of the data as it goes through every layer can be seen in the figure 2.2.

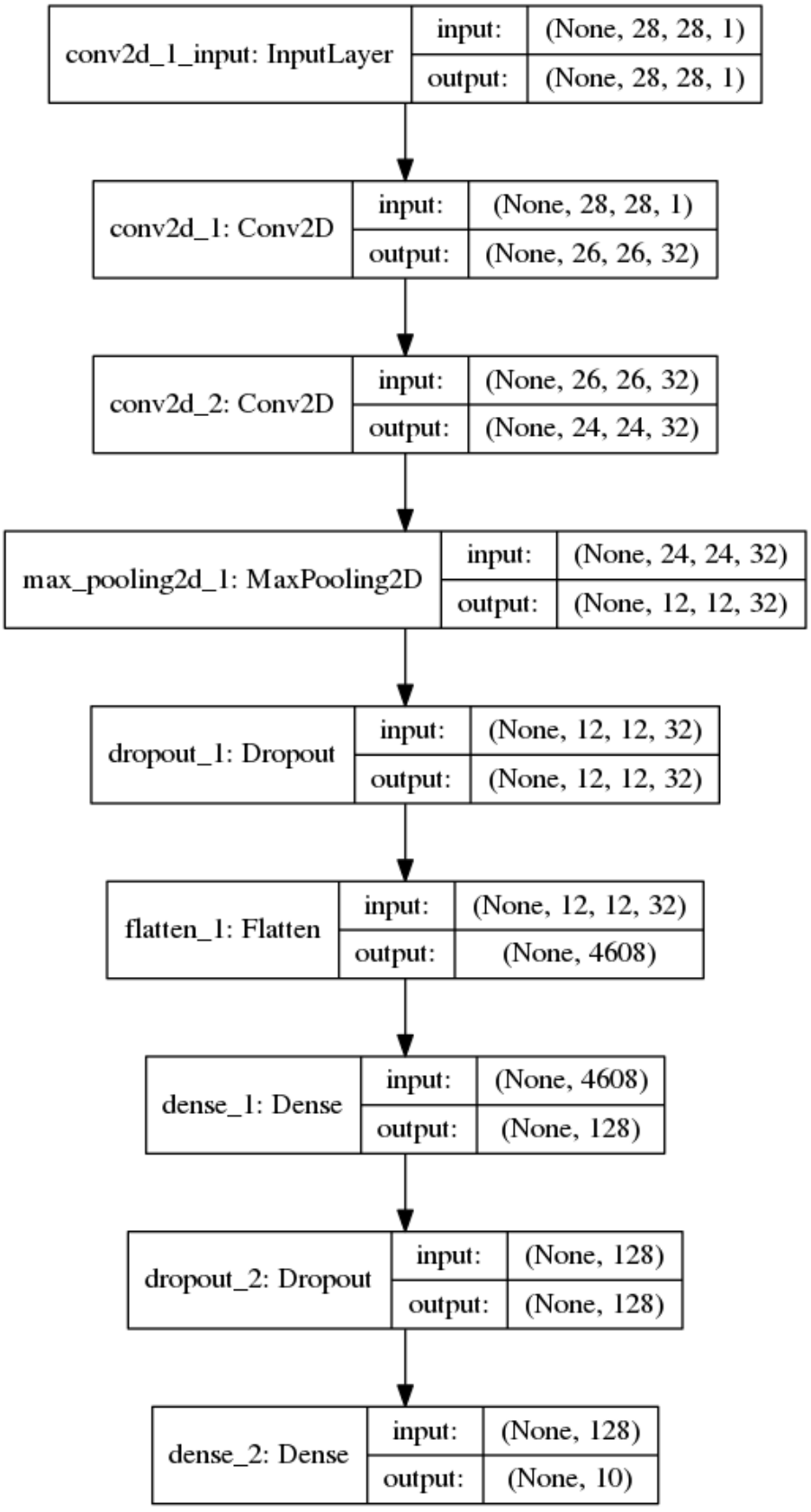


Figure 2.2: Diagram of a Keras sequential model

2.1.3 Compiling the model

After declaring the model and defining its architecture, the **learning process** must be compiled. The arguments required to set this process are defined in the section 1.1.1.

```
'''
Compiling the model
'''
model.compile(loss='categorical_crossentropy', optimizer='adadelta',
              metrics=['accuracy'])
```

For this example, the loss function that is computed after every batch is the **categorical cross-entropy** 1.1 and the optimizer that updates the weights of the CNN in order to minimize that loss function is **ADADELTA** 1. Additionally, the **accuracy** is also computed to monitor **the CNN performance** during training.

2.1.4 Training the model

The CNN is trained thanks to the ***.fit()* method**, which has been already described in the section 1.1.1.

```
'''
Training the model
'''
model.fit(x_train, y_train, batch_size=batch_size, nb_epoch=nb_epoch,
          verbose=1, validation_data=(x_test, y_test))
```

The model that has been built is trained for 12 **epochs** and the **batch size**, i.e., the number of samples that pass through the CNN before updating the weights, is 128. The test dataset is used here as **validation data** for which the log loss and the accuracy is computed after every epoch just for **monitoring purposes**. During training time, Keras prints the results after every batch and epoch as follows:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/12
128/60000 [.....................] - ETA: 350s - loss: 2.3223 - acc: 0.1016
256/60000 [.....................] - ETA: 312s - loss: 2.3073 - acc: 0.1094
384/60000 [.....................] - ETA: 287s - loss: 2.2927 - acc: 0.1458
```

```
512/60000 [..................] - ETA: 275s - loss: 2.2812 - acc: 0.1504
640/60000 [..................] - ETA: 275s - loss: 2.2643 - acc: 0.1734
768/60000 [..................] - ETA: 270s - loss: 2.2513 - acc: 0.1875
896/60000 [..................] - ETA: 265s - loss: 2.2355 - acc: 0.2054
1024/60000 [..................] - ETA: 260s - loss: 2.2132 - acc: 0.2178
...
59648/60000 [==================>.] - ETA: 2s - loss: 0.0455 - acc: 0.9871
59776/60000 [==================>.] - ETA: 1s - loss: 0.0455 - acc: 0.9871
59904/60000 [==================>.] - ETA: 0s - loss: 0.0455 - acc: 0.9871
60000/60000 [====================] - 407s - loss: 0.0455 - acc: 0.9871 -
val_loss: 0.0306 - val_acc: 0.9891
```

2.1.5 Testing the model

Once the model is trained, we can store its weights, architecture and learning configuration in a **HDF5 file** 1.5. In order to see the performance of the CNN, the ***.evaluate()* method** takes the test dataset and computes the log loss and the accuracy. It returns both of them in an array.

```
'''
Saving the model architecture and weights
'''
model.save('MNIST_net.h5')


'''
Testing the model
'''
score = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

These are the results obtained with this example (*Test score* refers to loss):

```
Test score: 0.0306129640532
Test accuracy: 0.9891
```

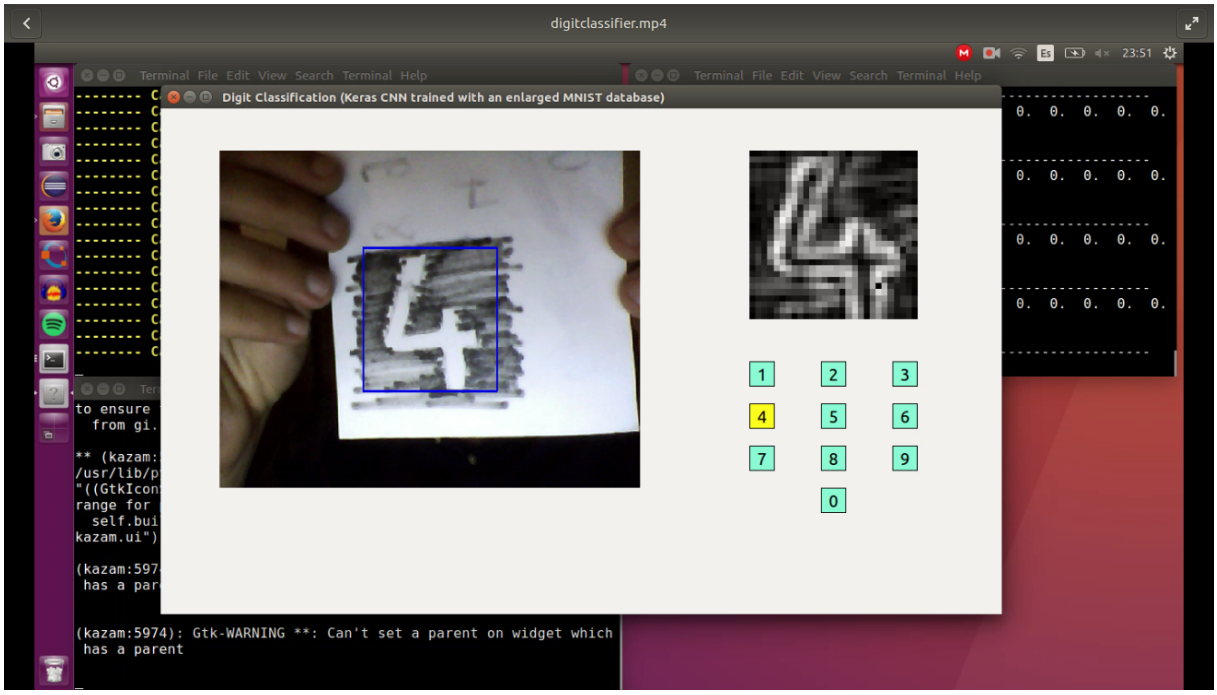


Figure 2.3: Example of *digitclassifier.py* execution

2.2 *digitclassifier.py*

Once the neural network has been trained and the resultant model is saved, the next milestone is to integrate it into a program that must be able to **get images** from a video stream and **display the predictions** obtained from them. That program is ***digitclassifier.py*** [3] and it is based on Nuria Oyaga code [4]. *digitclassifier.py* depends on two classes: ***Camera*** and ***GUI***. Figure 2.3 shows the program running.

Camera class

***Camera* class** [5] is responsible for getting the images, transforming them into a suitable input for the Keras model and returning the classification results.

- **Acquisition**. The images are served by the JdeRobot 1.2 component ***cameraserver*** 1.2.1. Depending on how its **configuration file** (*cameraserver.cfg*) has been set, the images can come from different kinds of video streams. During the development of this application, the digit classifier component has been tested with **webcams**,

[3] https://git.io/vH0qi
[4] https://git.io/vH0qD
[5] https://git.io/vH0qS

video files and **smartphone cameras**. Connection with webcams and video files is straightforward: the **URI property** in the configuration file must be changed to the number of device or the path of the video, respectively.

```
#0 corresponds to /dev/video0, 1 to /dev/video1, and so on...
#CameraSrv.Camera.0.Uri=1                              # webcam
CameraSrv.Camera.0.Uri=/home/dpascualhe/video.mp4      # video file
```

In order to establish a connection with smartphone cameras, the **DroidCam** 1.3 application for Android has been used. As this application turns the video stream provided by the smartphone into a **v4l2** [6] **device driver**, the video stream will be listed as another webcam and the number of device must be set in the *cameraserver* configuration file.

Besides that, the **address** at which the video is being served by the *cameraserver* component is provided to the *Camera* class thanks to another configuration file: ***digitclassifier.cfg*** [7].

- **Pre-processing**. As the images can be captured with **different devices**, the *digitclassifier.py* component must apply some pre-processing that mitigates the differences between video streams and that makes the images **suitable for the Keras model**. The following **transformations** are applied before classification:

  1. Images are **cropped** into 80x80 pixels images. The **ROI** from which cropped images are extracted is **draw over the original image**, making it easier to aim at digits with the camera.
  2. Color doesn't provide any useful information about digits and MNIST database is formed by **grayscale images**. That's why the images captured with the component must be converted into grayscale images as well.
  3. A **Gaussian filtering** is applied in order to **reduce image noise**. When using this operator, the **kernel size** and the **standard deviation** σ in x and y should be specified [6]. In this case, the kernel size will be 5x5 and the standard deviation is automatically calculated depending on that size. The 2D

[6] https://www.linuxtv.org/wiki/index.php/Main_Page
[7] https://git.io/vH0zi

Gaussian filter [17] is given by:

$$G(x, y) = \exp(-\frac{x^2 + y^2}{2\sigma^2}) \tag{2.1}$$

4. After reducing noise, the image is **resized** to fit the Keras **model input**. The new size is 28x28 pixels, like MNIST samples.

5. Last step is **edge detection**. Working with edges instead of original images allows the application to deal with **different light and color conditions**. The chosen algorithm for this task is **Sobel filter** 2.2. This will be deeply discussed in the section 2.3.1.

The following code applies the transformations mentioned above:

```
def trasformImage(self, im):
    ''' Transforms the image into a 28x28 pixel grayscale image and
    applies a sobel filter (both x and y directions).
    '''
    im_crop = im [140:340, 220:420]
    im_gray = cv2.cvtColor(im_crop, cv2.COLOR_BGR2GRAY)
    im_blur = cv2.GaussianBlur(im_gray, (5, 5), 0) # Noise reduction.

    im_res = cv2.resize(im_blur, (28, 28))

    # Edge extraction.
    im_sobel_x = cv2.Sobel(im_res, cv2.CV_32F, 1, 0, ksize=5)
    im_sobel_y = cv2.Sobel(im_res, cv2.CV_32F, 0, 1, ksize=5)
    im_edges = cv2.add(abs(im_sobel_x), abs(im_sobel_y))
    im_edges = cv2.normalize(im_edges, None, 0, 255, cv2.NORM_MINMAX)
    im_edges = np.uint8(im_edges)

    return im_edges
```

- **Classification**. Before entering the CNN, the images are **reshaped** as mentioned in the section 2.1.1. *Camera* class calls Keras ***.predict()* method** 1.1.1 to get the predicted digit. Finally, the case in which prediction doesn't return a clear answer is handled.

```
def classification(self, im):
''' Adapts image shape depending on Keras backend (TensorFlow
or Theano) and returns a prediction.
'''
if backend.image_dim_ordering() == 'th':
   im = im.reshape(1, 1, im.shape[0], im.shape[1])
else:
   im = im.reshape(1, im.shape[0], im.shape[1], 1)

dgt = np.where(self.model.predict(im) == 1)
print("Keras CNN prediction: ", self.model.predict(im))
print("Prediction index: ", dgt)
print("------------------------------------------------------------")
if dgt[1].size == 1:
   self.digito = dgt
else:
   self.digito = (([0]), (["none"]))

return self.digito[1][0]
```

2.2.1 *GUI* class

***GUI* class** [8] displays the **original image**, the **processed image** and the result of the **classification**, as it can be seen in figure 2.3. It has been built employing the **pyQt package** [9] [10]. It is based in Nuria Oyaga code [11], but it has been updated from Qt4 to Qt5 thanks to the information provided by its documentation [14].

2.2.2 Threads

In order to capture images and update the GUI **concurrently**, the ***threading* module** [13], provided by Python, has been employed. From this module, a subclass of the

[8] https://git.io/vH0YK
[9] https://pypi.python.org/pypi/PyQt4
[10] https://pypi.python.org/pypi/PyQt4
[11] https://git.io/vH0mx

***Thread* object** is created. In this new subclass, *__init__()* and *.run()* methods are overriden. The *.run()* method is responsible for calling a process which must **update the thread**. For example, the *.update()* method of the *Camera* class, which reads a new image from the video stream each time it is invoked, is called within the *.run()* method of the ***ThreadCamera* class**. Besides that, in the *.run()* method, the **cycle time** is adjusted.

```
import time
import threading
from datetime import datetime

t_cycle = 150 # ms

class ThreadCamera(threading.Thread):

def __init__(self, cam):
   ''' Threading class for Camera. '''
   self.cam = cam
   threading.Thread.__init__(self)

def run(self):
   ''' Updates the thread. '''
   while(True):
      start_time = datetime.now()
      self.cam.update()
      end_time = datetime.now()

      dt = end_time - start_time
      dtms = ((dt.days * 24 * 60 * 60 + dt.seconds) * 1000
              + dt.microseconds / 1000.0)

      if(dtms < t_cycle):
         time.sleep((t_cycle - dtms) / 1000.0);
```

This code, as well as the one corresponding to the ***ThreadGUI* class**, can be accessed

in GitHub [12] [13].

2.2.3 Main program

All of these elements are joined together in ***digitclassifier.py***. *Camera* and *GUI* classes and their threads are initialized and the ***.start()*** methods of the *Thread* objects are invoked.

```
if __name__ == '__main__':

    cam = Camera()
    app = QtWidgets.QApplication(sys.argv)
    window = GUI()
    window.setCamera(cam)
    window.show()

    # Threading camera
    t_cam = ThreadCamera(cam)
    t_cam.start()

    # Threading GUI
    t_gui = ThreadGUI(window)
    t_gui.start()

    sys.exit(app.exec_())
```

In order to execute the program:

1. *cameraserver* must be launched with its configuration file as an argument in a terminal.

   ```
   dpascualhe@hp-g6:~$ cameraserver cameraserver.cfg
   ```

2. In another terminal, *digitclassifier.py* must be launched with its configuration file as well.

[12] https://git.io/vHOlY
[13] https://git.io/vHOlW

```
dpascualhe@hp-g6:~$ python digitclassifier.py digitclassifier.cfg
```

An example of usage of the digit classifier component can be seen in figure 2.3.

2.3 Datasets

The digit classifier component is possible thanks to the data provided by the **MNIST database** of handwritten digits [11]. As it has been mentioned before 1.4, this database contains 60000 samples for training and 10000 samples for testing. Nevertheless, these large datasets may not be enough for our application, because they represent an ***ideal* situation**.

2.3.1 Edge detection

The first problem with MNIST database is that the grayscale images that it contains share similar intensity levels: **a white digit over a black background**. In real world, the digits can be found written in several colors over different backgrounds and the datasets must resemble every possible combination. In order to achieve that generalization, an edge detection has been applied to the datasets. The resultant images are less dependent from the light and color conditions of the original ones, forcing the neural network to focus in **the shape of the digits** to classify them.

According to the study carried out by Nuria Oyaga [14], the edge detection algorithm that leads to better results is the **Sobel filter**. This operator approximates the **gradient** of an image function [17], convolving the image with the following **kernels** to detect horizontal and vertical edges, respectively:

$$h_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.2}$$

The absolute values of the resulting images, x and y, are then added, obtaining the edge image.

[14] http://jderobot.org/Noyaga-tfg#Testing_Neural_Network

2.3.2 Data augmentation

The second problem that has been detected with MNIST is that the images are **noiseless** and the digits are always centered with a scale and a rotation angle that are almost **invariant**. However, the digit classifier has to deal with noisy images that can be randomly scaled, translated and/or rotated. In order to get a database with images that look like the ones that our application is going to work with, the MNIST database must be augmented.

Two alternatives have been considered to solve this problem: **real-time data augmentation** provided by Keras and **generating our own database**.

Real-time data augmentation with Keras

Thanks to Keras ***.ImageDataGenerator()* method** 1.1.4, the MNIST dataset can be augmented in **real-time** during training. In order to cover most of the real cases, random **rotation**, **translation** and **zooming** were applied to generate new samples. In addition to that, a **Sobel filtering** was also applied through a **user-defined function**. The samples generated by the following code [15] can be seen in the figure 2.4.

```
   if mode == "full":
      datagen = imkeras.ImageDataGenerator(
         zoom_range=0.2, rotation_range=20, width_shift_range=0.2,
         height_shift_range=0.2, fill_mode='constant', cval=0,
         preprocessing_function=self.sobelEdges)
   elif mode == "edges":
      datagen = imkeras.ImageDataGenerator(
         preprocessing_function=self.sobelEdges)

generator = datagen.flow(x, y, batch\_size=batch\_size)
```

Besides these transformations, it's also necessary to simulate the **noise** that will be present in real images. Keras generator doesn't support the addition of noise. For this purpose, Keras includes noise layers such as the **GaussianNoise layer**, which adds Gaussian noise with a standard deviation distribution defined by the user. It's important to note that Keras treat noise layers as regularization methods that are only active during

[15] https://git.io/vH0qz

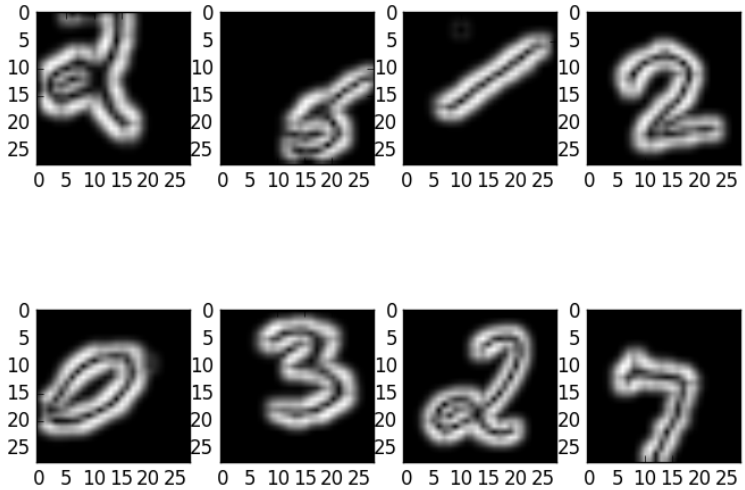


Figure 2.4: Samples generated with Keras from MNIST database

training time to avoid over-fitting. In order to add noise to the generated samples, a GaussianNoise layer was established as the **input layer** of the model.

Handmade augmented datasets

The alternative to real-time data augmentation with Keras is building **our own datasets** applying the previously mentioned transformations to the images. My mate Nuria Oyaga has build 5 new databases with two sets each one: training and validation [16]. These are the new databases:

- **Sobel**: MNIST database after applying the Sobel filter to every image. 48000 samples for traning and 12000 samples for validation.
- **0-1**: Same size than Sobel database. One transformed image per every Sobel database image. Sobel database images are replaced by the the transformed ones. 48000 samples for traning and 12000 samples for validation.
- **1-1**: Double size than Sobel database. One transformed image per every Sobel database image. Both Sobel database images and the transformed images are

[16] `http://jderobot.org/Noyaga-tfg#Comparing_Neural_Network`

included in the 1-1 database. 96000 samples for training and 24000 samples for validation.

- **0-6**: Six times the size of Sobel database. Six transformed images per every Sobel database image. Sobel database images are replaced by the transformed ones. 288000 samples for training and 72000 samples for validation.

- **1-6**: Seven times the size of Sobel database. Six transformed images per every Sobel database image. Both Sobel database images and the transformed images are included in the 1-6 database. 336000 samples for training and 84000 samples for validation.

Additionally, the test dataset of the MNIST database (10000 samples) has been converted into a **1-6 test dataset** (70000 samples).

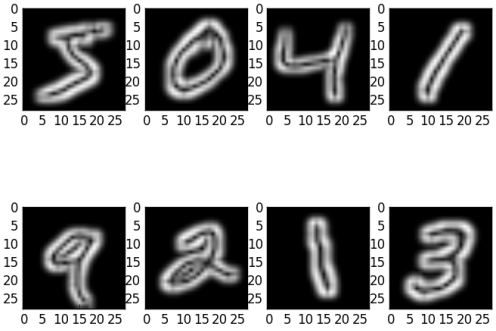
In the figure 2.5, the first samples of every handmade dataset can be seen.

From LMDB to HDF5

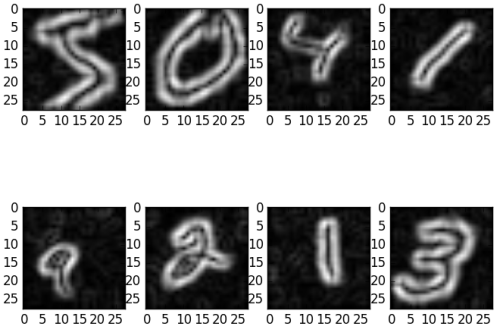
These databases were initially built to feed a **Caffe** [8] neural network. That's why they were saved as **LMDB files** [17]. In order to make it easier to feed the Keras network, the LMDB databases have been converted into **HDF5 files** 1.5. For this conversion, the script ***datasetconversion.py*** [18] was written.

- **Reading the LMDB database**. The LMDB library for Python [19] was employed to open the database, initialize a cursor and iterate over each key-value pair in the database. In addition, **Google's Protocol Buffers** [20], a.k.a. Protobuf, was used to parse the data that was being obtained from the database. "With protocol buffers, you write a ***.proto* description** of the data structure you wish to store. From that, the **protocol buffer compiler** creates a class that implements automatic encoding and parsing of the protocol buffer data with an efficient binary format" [5]. Here can be seen the *.proto* file that defines the data structure used by Caffe to store the MNIST database, as obtained from [2]:

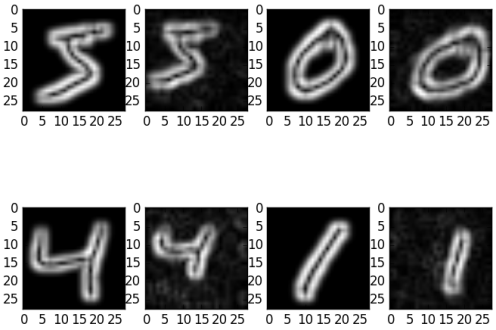
[17] `http://www.lmdb.tech/doc/`
[18] `https://git.io/vHWTe`
[19] `https://lmdb.readthedocs.io/en/release/#`
[20] `https://developers.google.com/protocol-buffers/`



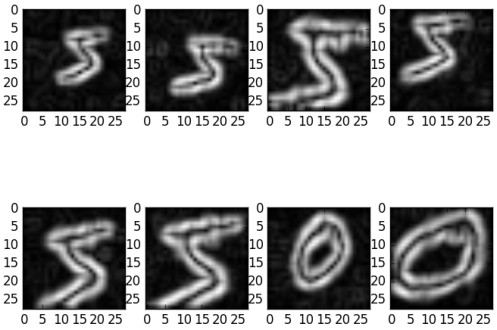
(a) Sobel dataset



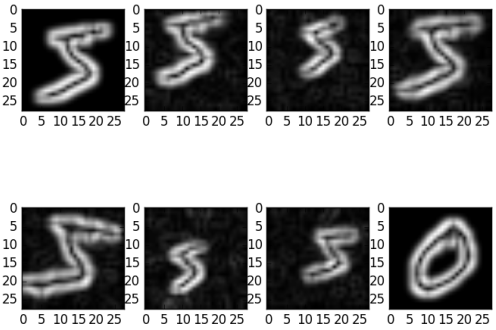
(b) 0-1 dataset



(c) 1-1 dataset



(d) 0-6 dataset



(e) 1-6 dataset

Figure 2.5

```
package datum;
message Datum {
   optional int32 channels = 1;
   optional int32 height = 2;
   optional int32 width = 3;
   // the actual image data, in bytes
   optional bytes data = 4;
   optional int32 label = 5;
   // Optionally, the datum could also hold float data.
   repeated float float_data = 6;
   // If true data contains an encoded image that need to be decoded
   optional bool encoded = 7 [default = false];
}
```

Thanks to the *.proto* file, the compiler generates a Python module that contains the **Datum class**. Datum class provides the ***.ParseFromString()* method**, which is employed to parse the image data. Here is the resulting code:

```
# We initialize the cursor that we're going to use to access every
# element in the dataset.
lmdb_env = lmdb.open(sys.argv[1])
lmdb_txn = lmdb_env.begin()
lmdb_cursor = lmdb_txn.cursor()

x = []
y = []
nb_samples = 0

# Datum class deals with Google's protobuf data.
datum = datum.Datum()

if __name__ == '__main__':
# We extract the samples and its class one by one.
for key, value in lmdb_cursor:
   datum.ParseFromString(value)
```

```
    label = np.array(datum.label)
    data = np.array(bytearray(datum.data))
    im = data.reshape(datum.width, datum.height,
                  datum.channels).astype("uint8")

    x.append(im)
    y.append(label)
    nb_samples += 1

    print("Extracted samples: " + str(nb_samples) + "\n")

x = np.asarray(x)
y = np.asarray(y)
```

- **Writing the HDF5 files**. After extracting the data, it was stored in a HDF5 file. Thanks to the **h5py library** [21] for Python, a HDF5 file with two *datasets* (label and data) was created.

```
f = h5py.File("../../Datasets/" + sys.argv[2] + ".h5", "w")

# We store images.
x_dset = f.create_dataset("data", (nb_samples, datum.width,
                        datum.height, datum.channels), dtype="f")
x_dset[:] = x

# We store labels.
y_dset = f.create_dataset("labels", (nb_samples,), dtype="i")
y_dset[:] = y
f.close()
```

Conclusions

After coding and testing both implementations for augmenting the database, it has been decided to go for the **handmade datasets**. While real-time data augmentation is really

[21] http://www.h5py.org/

useful to avoid storing all the data that is needed for training, it makes it harder to take a look into what is being fed to the network and reproduce results. Also, in this particular case, we are interested in **compare the performance** of neural networks built with different libraries, so they must be trained with the same datasets.

2.4 Benchmark

2.5 Tuning the classifier

Bibliography

[1] F. Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[2] C. Choy. Reading protobuf db in python, 2015. `https://chrischoy.github.io/research/reading-protobuf-db-in-python/`.

[3] DEV47APPS. Dev47Apps, 2010-2016. `http://www.dev47apps.com/`.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[5] Google developers. Protocol buffer basics: Python, 2017. `https://developers.google.com/protocol-buffers/docs/pythontutorial`.

[6] Itseez. *The OpenCV Reference Manual*, 2.4.9.0 edition, April 2014.

[7] JdeRobot developers. Jderobot - robotics and computer vision technology that rocks and matters!, 2017. `http://jderobot.org/Main_Page`.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[9] S. H. John W. Eaton, David Bateman and R. Wehbring. *GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations*. 2015.

[10] A. Karpathy. Cs231n convolutional neural networks for visual recognition, 2017. `http://cs231n.github.io/convolutional-networks/`.

[11] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[13] Python Software Foundation. 16.2. threading — higher-level threading interface — python 2.7.13 documentation, 1990-2017. `https://docs.python.org/2.7/library/threading.html`.

[14] Riverbank Computing Limited. Pyqt5 reference guide, 2015. `http://pyqt.sourceforge.net/Docs/PyQt5/`.

[15] D. Scherer, A. Müller, and S. Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. *Artificial Neural Networks–ICANN 2010*, pages 92–101, 2010.

[16] Scikit-learn developers. Documentation scikit-learn: machine learning in python — scikit-learn 0.18.1 documentation, 2010 - 2016. `http://scikit-learn.org/stable/documentation.html`.

[17] M. Sonka, V. Hlavac, and R. Boyle. *Image processing, analysis, and machine vision*. Cengage Learning, 2014.

[18] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 2014.

[19] The HDF Group. Hierarchical data format, version 5, 1997-2017. `http://www.hdfgroup.org/HDF5/`.

[20] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[21] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell. Understanding data augmentation for classification: when to warp? *CoRR*, abs/1609.08764, 2016.

[22] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.