Universidad
Rey Juan Carlos

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

List of Tables

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 JdeRobot

JdeRobot [3] 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** [2]. It is also compatible with **ROS** [3], which allows the interoperation of ROS nodes and JdeRobot components. This flexibility makes it very useful for our application. Its ***cameraserver* driver** is going to be employed to capture images from different video sources.

1.1.1 *cameraserver*

According to JdeRobot documentation [3], 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:

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

[2] https://zeroc.com/products/ice

[3] http://www.ros.org/

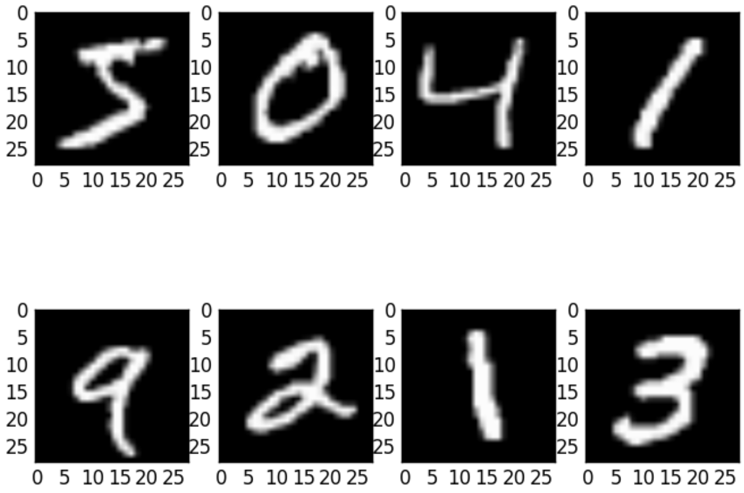


Figure 1.1: Samples extracted from the MNIST database

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

1.2 MNIST

MNIST (Modified National Institute of Standards and Technology database) [6] 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** [4]. MNIST is a well-known **benchmark** for all kinds of machine learning algorithms.

As may be seen in figure 1.1, 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 2.1.1.

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

1.3 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** [5] 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.

1.3.1 Models

Every neural network in Keras is defined as a ***model***. For those models which can be built as a stack of *layers* 1.3.2, Keras provides the ***.Sequential()* object**. An example of a sequential model built with Keras can be seen in figure 1.2. 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 [13] 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** [15]. This optimizer is an extension of the **gradient**

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

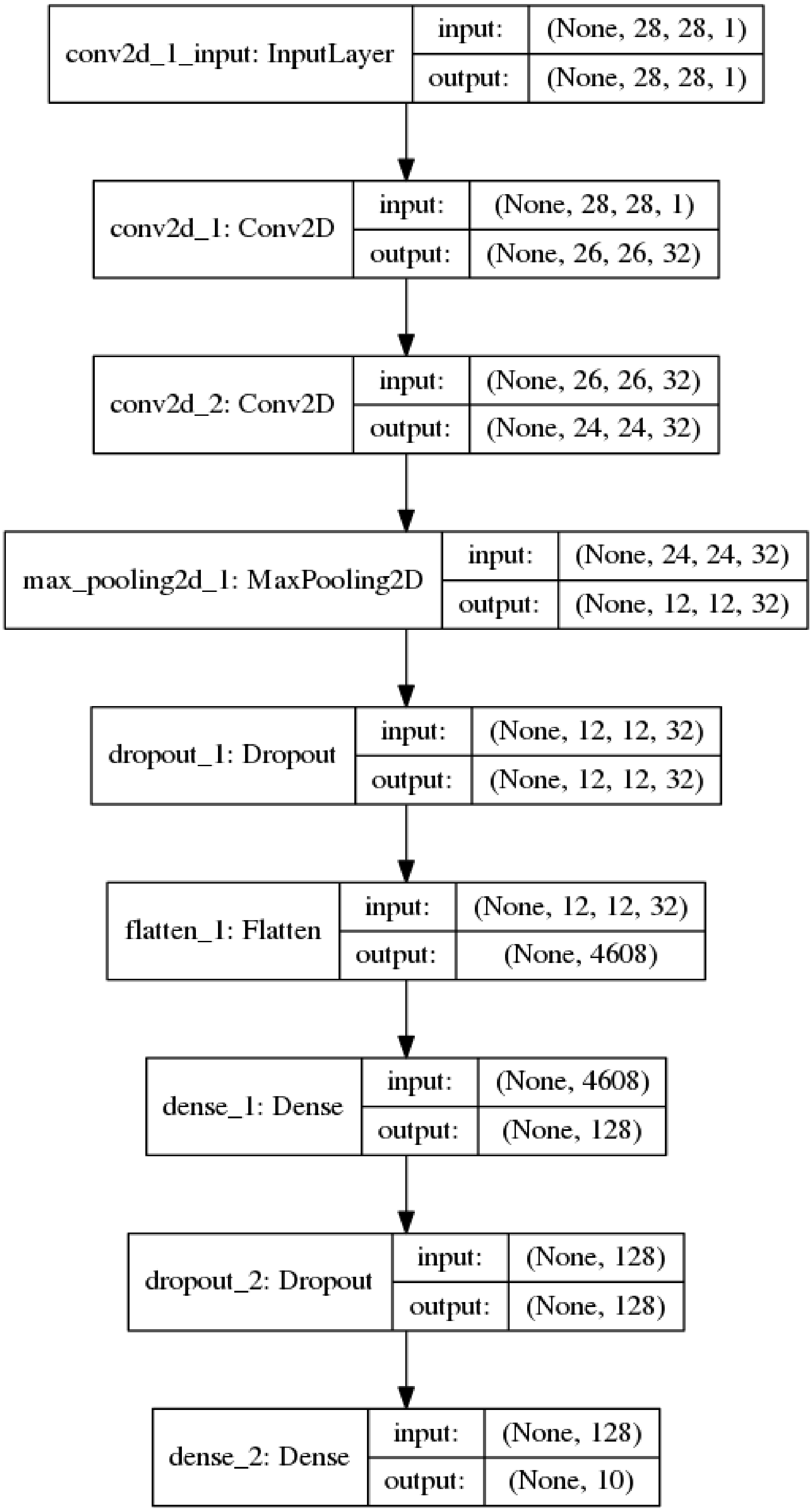


Figure 1.2: Diagram of a Keras sequential model

descent optimization method that has the particularity of adapting the learning rate during training with no need of manual tuning. It follows the next algorithm [15]:

Algorithm 1 Computing ADADELTA update at time t

```
Require: Decay rate $\rho$, Constant $\epsilon$
Require: Initial parameter $x_1$
Initialize accumulation variables $E[g^2]_0 = 0, E[\Delta x^2]_0 = 0$
for $t = 1 : T$ do                                    ▷ Loop over # of updates
    Compute gradient: $g_t$
    Accumulate gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1 - \rho) g_t^2$
    Compute update: $\Delta x_t = -\frac{\mathrm{RMS}[\Delta x]_{t-1}}{\mathrm{RMS}[g]_t} g_t$
    Accumulate updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1 - \rho) \Delta x_t^2$
    Apply update: $x_{t+1} = x_t + \Delta x_t$
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 [2]. Other metrics are calculated with **Scikit-learn** 1.5 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**[6].
- ***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.

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

- ***callbacks***: list of callbacks 1.3.3 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** 1.4, which will contain the weights, architecture and training configuration of the model.

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

1.3.2 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 [5]. 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 1.3 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 [8]. Figure 1.4 shows how the pooling operation is applied.

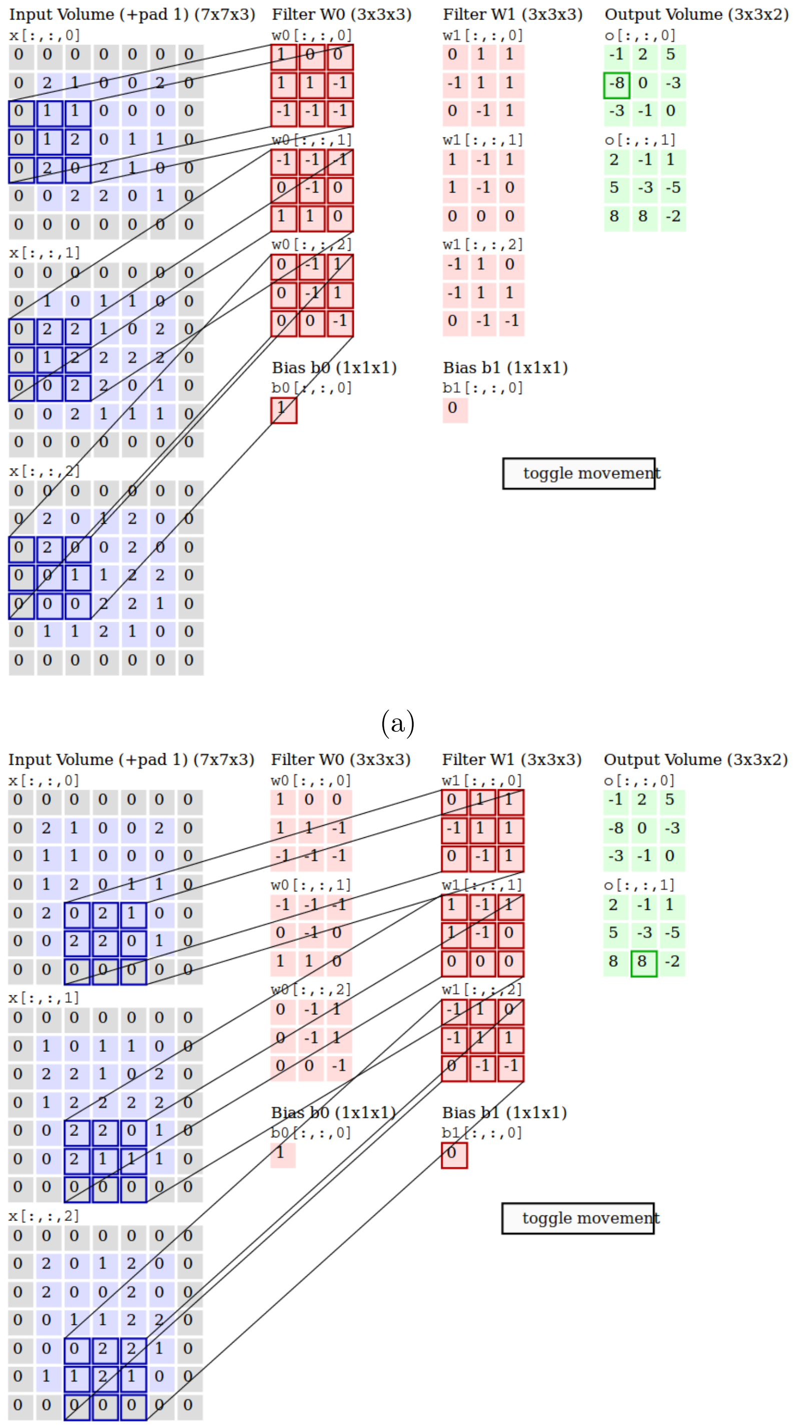
Depending on the dimensions of the input and the operation applied, Keras provides several pooling layers: *MaxPooling1D*, *MaxPooling2D*, *MaxPooling3D*, *AveragePooling1D*... The main arguments required by Keras to define these layers are:

- ***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 [5]. 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



(a)

(b)

Figure 1.3: In the figure 1.3a, 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 1.3b 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 [5]

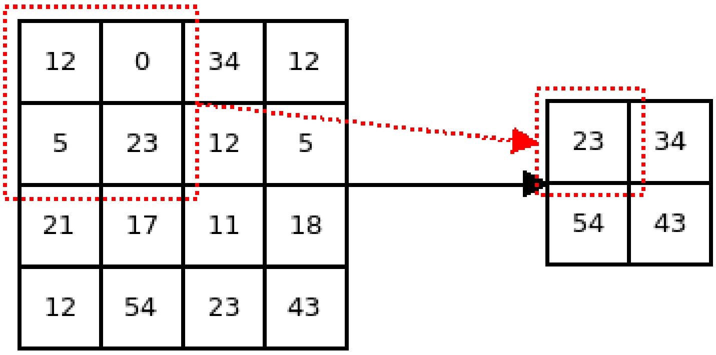


Figure 1.4: Example of a max. pooling operation

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:

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

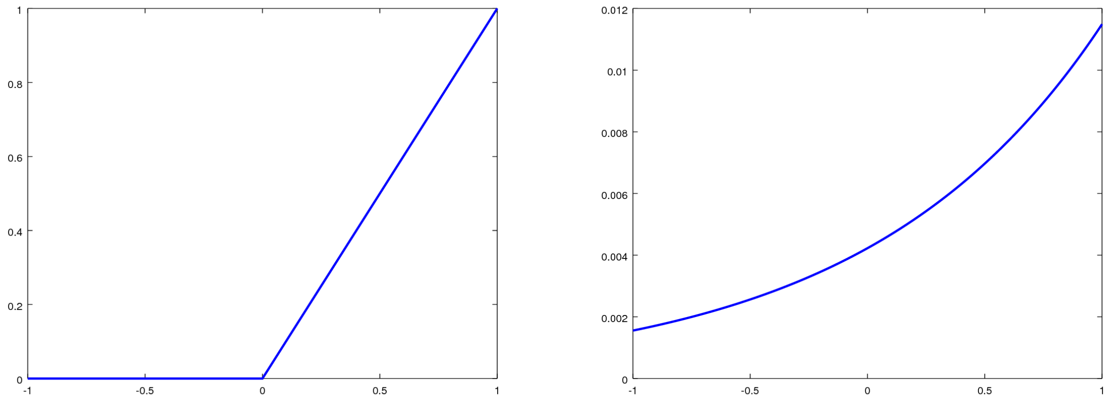
- **Softmax** This activation function is very useful when is placed after the **output layer** of classification tasks. It takes a vector of real values z and returns a new vector of real values in the range [0,1]. The N elements of the output vector can be considered **probabilities** because the softmax 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 [2]. Figure 1.5 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 [11] 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 1.6.



(a) ReLU activation function

(b) Softmax activation function

Figure 1.5

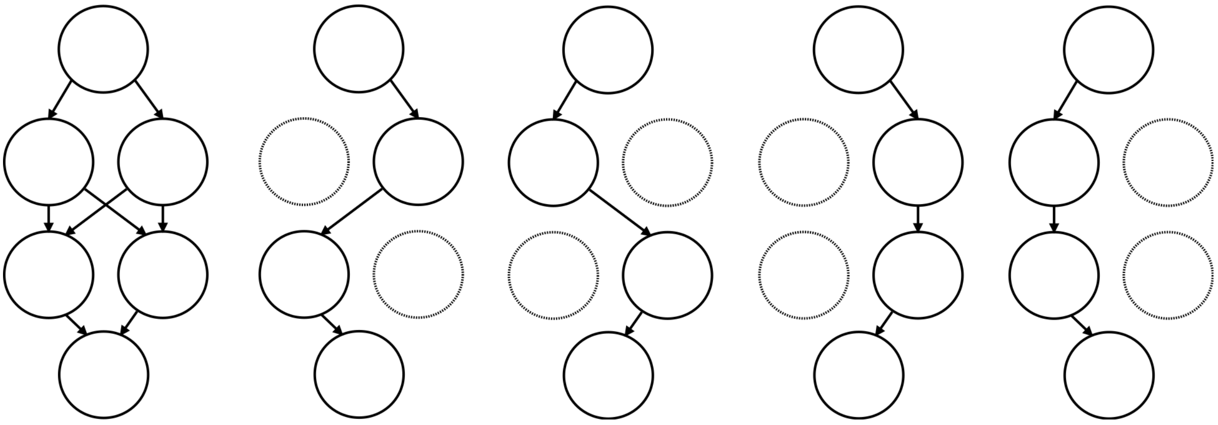


Figure 1.6: Dropout can be understood as a technique that ”trains an ensemble consisting of all subnetworks that can be structured by removing non-output units from an underlying base network” [2].

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.

1.3.3 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**.

1.3.4 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** [14] 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.

1.3.5 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** 1.4, which are going to be used as inputs to the neural networks.

1.4 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 [12], 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.

In order to deal with HDF5 files, the **h5py** [7] library for Python has been employed.

1.5 Scikit-learn

Scikit-learn [7] 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

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

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 [9]):

- **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.

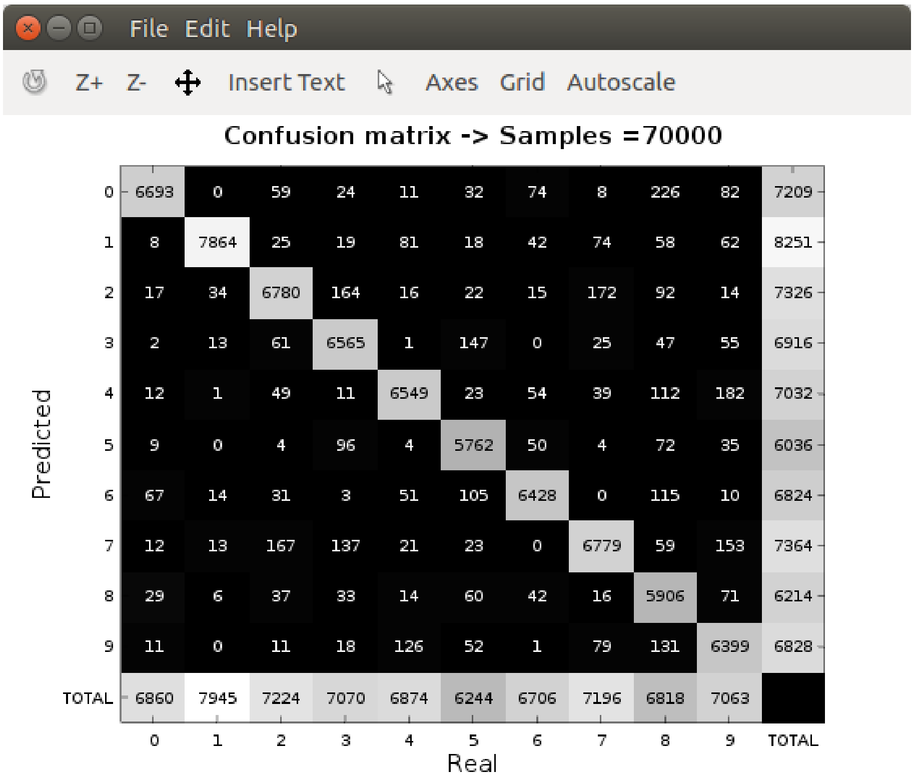
$$\text{recall} = \frac{true_{positives}}{true_{positives} + false_{negatives}} \tag{1.5}$$

- **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 1.6 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 1.3.1.

1.6 Octave

GNU Octave [4] 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.



Confusion matrix -> Samples =70000

	0	1	2	3	4	5	6	7	8	9	TOTAL
0	6693	0	59	24	11	32	74	8	226	82	7209
1	8	7864	25	19	81	18	42	74	58	62	8251
2	17	34	6780	164	16	22	15	172	92	14	7326
3	2	13	61	6565	1	147	0	25	47	55	6916
4	12	1	49	11	6549	23	54	39	112	182	7032
5	9	0	4	96	4	5762	50	4	72	35	6036
6	67	14	31	3	51	105	6428	0	115	10	6824
7	12	13	167	137	21	23	0	6779	59	153	7364
8	29	6	37	33	14	60	42	16	5906	71	6214
9	11	0	11	18	126	52	1	79	131	6399	6828
TOTAL	6860	7945	7224	7070	6874	6244	6706	7196	6818	7063	

Figure 1.7: Example of a confusion matrix visualization using Octave

Chapter 2

Development

2.1 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:

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

2.1.1 Datasets

MNIST database of handwritten digits is the starting point for the digit classifier. As it has been mentioned before 1.2, 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.

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 dataset. The resultant images are less dependent from the intensity values 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 [1], the edge detection algorithm that leads to better results is the Sobel filter. This operator approximates the gradient of an image function [10], convolving the image with the following kernels to detect horizontal and vertical edges, respectively:

$$h_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.1}$$

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

Data augmentation with Keras

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. In this section, the first one is going to be described.

Thanks to Keras *.ImageDataGenerator()* method, 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

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

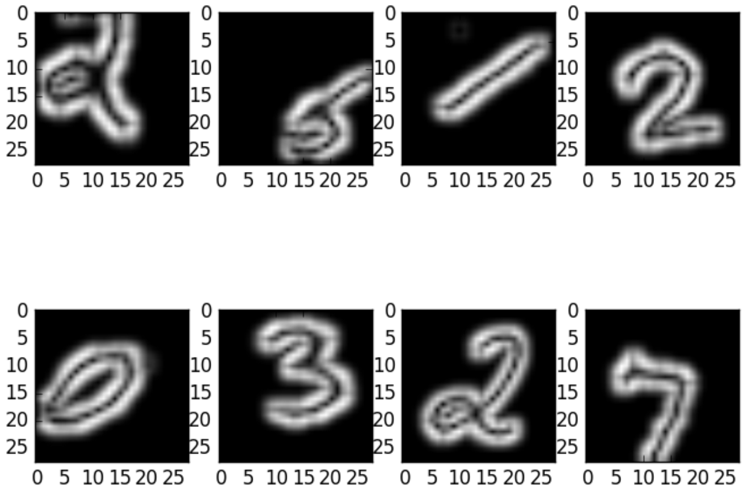


Figure 2.1: Samples generated with Keras from MNIST database

that, a Sobel filtering was also applied through a user-defined function. The samples generated by the following code can be seen in the figure 2.1.

```
    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 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

2.1.2 Classifier

2.1.3 Benchmark

2.1.4 Tuning the classifier

2.1.5 *digitclassifier.py*

Bibliography

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

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

[3] JdeRobot developers. JdeRobot - Robotics and Computer Vision Technology that rocks and matters! `http://jderobot.org/Main_Page`.

[4] 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.

[5] A. Karpathy. CS231n Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/convolutional-networks/`.

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

[7] 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.

[8] 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.

[9] Scikit-learn developers. Documentation scikit-learn: machine learning in Python — scikit-learn 0.18.1 documentation. `http://scikit-learn.org/stable/documentation.html`.

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

[11] 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.

[12] The HDF Group. Hierarchical Data Format, version 5, 1997-2017. `http://www.hdfgroup.org/HDF5/`.

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

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

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