ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

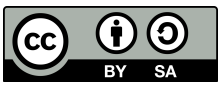
Study of Convolutional Neural Networks using Keras Framework

Autor: David Pascual Hernández

Tutor: José María Cañas Plaza

Cotutor: Inmaculada Mora Jiménez

Curso académico 2016/2017



Agradecimientos

Este trabajo de fin grado supone para mí la culminación de una dura etapa de aprendizaje de la que, sin duda alguna, he salido reforzado. Reforzado no sólo por los conocimientos adquiridos, si no por las herramientas brindadas para adquirir aquellos que están por llegar. Hasta hace poco desconocía la existencia de materias que a día de hoy me apasionan profundamente. Me siento privilegiado por tener la oportunidad de ser espectador de los cambios que estas materias están produciendo en el mundo, y me emociona trabajar para intentar ser partícipe de ellos. Estos descubrimientos son resultado directo del buen hacer de muchos de los docentes que he conocido estos últimos años. En especial, me gustaría dar las gracias a Inmaculada y Jose María. Su guía incondicional en el desarrollo de este trabajo, lo ha convertido en una experiencia fructífera y gratificante. También quiero dar las gracias a Nuria, compañera de fatigas en estos últimos meses. Gracias a vosotros puedo presumir de estar aprendiendo con los mejores.

Esta aventura ha estado llena de retos y obstáculos, y no tengo ningún reparo en reconocer que jamás habría sido capaz de superar muchos de ellos sin ayuda. En este sentido, quiero dar las gracias a todos los compañeros en los que me he apoyado para superarlos. En particular, agradezco enormemente a Marco y Abel haber compartido conmigo su amistad. Larga vida al *metal*.

Más alla de lo académico, tengo la fortuna de haber disfrutado de la mejor compañía que pueda imaginar. Por su excelente conversación, su ingenio, su paciencia y su sapiencia, me faltan horas en el día para dar las gracias a Andrés y Alejandra. Os quiero.

Los últimos años, han sido una convulsión de viajes, proyectos, satisfacciones y decepciones. Es difícil digerir estos cambios cuando no encuentras la forma de comunicarte y de hacer comprender a los demás aquello que te aflige. Por suerte, entre esa maraña de cambios, me crucé con Carolina. Gracias por tu bondad y tu forma de ver el mundo. Te quiero.

Por último, doy las gracias a mi familia, y especialmente a mis padres, Virginia y David. Todo lo que he aprendido en la vida puedo dividirlo en dos grandes grupos: aquello que me habéis enseñado vosotros y aquello que me habéis enseñado a aprender.

Resumen

El *reconocimiento de objetos* en imágenes ha sido un problema recurrente en la historia de la *visión artificial*. Gracias a la inclusión de algoritmos basados en aprendizaje máquina y, más recientemente, de las técnicas de *aprendizaje profundo*, se han enfrentado con éxito problemas como el reconocimiento de señales de tráfico o la videovigilancia. En concreto, las *redes neuronales convolucionales* se han convertido en la punta de lanza de este tipo de algoritmos en los últimos años. Su eficacia en la resolución de problemas como los anteriormente mencionados resulta indiscutible en muchos casos, lo que poco a poco está favoreciendo su uso en aplicaciones comerciales. A pesar de ello, siguen siendo acusadas de actuar como una caja negra o *black box*, ya que por lo general su aprendizaje es un proceso opaco y difícil de interpretar.

Por todo ello, este trabajo de fin grado tiene como metas el *estudio detallado* de las redes neuronales convolucionales y su *aplicación* en el abordaje de un determinado problema. En este sentido, se desarrollará un *clasificador de dígitos manuscritos en tiempo real*. Para entrenar e implementar las redes neuronales convoluciones, se empleará la plataforma *Keras*. El proyecto comienza con el análisis de una red neuronal convolucional de ejemplo proporcionada por dicha plataforma. Posteriormente, se procede al desarrollo del *componente* clasificador de dígitos. Este componente adquiere imágenes desde una fuente de vídeo, las clasifica gracias a una red neuronal de Keras y muestra el resultado en una interfaz gráfica. Además, se ha conformado un *banco de pruebas* en el que se incluyen bases de datos para alimentar las redes neuronales convolucionales y herramientas para calcular y visualizar parámetros de evaluación. Por último, gracias a las herramientas del banco de pruebas, se discutirán los *efectos que produce el aprendizaje sobre el desempeño* de distintas redes neuronales y aquella que mejores resultados arroje será integrada en el componente clasificador de dígitos para lograr una mayor robustez.

Los resultados obtenidos ponen de manifiesto el gran potencial de las redes neuronales convoluciones y proyectan algo de luz sobre su aprendizaje, dejando abierta la puerta a su empleo en la resolución de problemas más complejos.

Summary

Object recognition has been a recurring problem in the history of *computer vision*. Thanks to the inclusion of machine learning and, recently, *deep learning* algorithms, issues like traffic sign recognition and video surveillance have been successfully addressed. In particular, *convolutional neural networks* have become the spearhead of this kind of algorithms in the last few years. In many cases, their effectiveness solving issues like the ones mentioned before can't be denied, which has enabled their usage in commercial applications. Nevertheless, convolutional neural networks keep being accused of acting like *black boxes*, because their learning process is usually very opaque and hard to interpret.

For all of these reasons, this final degree project aims to serve as a *detailed study* of convolutional neural networks and their *implementation* for solving a certain problem. In this case, a *real-time handwritten digits classifier* will be developed. These objectives will be faced employing the *Keras* platform. The project starts with the analysis of a convolutional neural network example provided by the aforementioned library. Then, the digit classifier *component* is discussed. This component acquires images from a video source, classifies them thanks to a neural network built with Keras and displays the result in a graphical user interface. Besides that, a *test bench* has been developed. It is formed by datasets that will feed the convolutional neural networks and tools for computing and visualizing evaluation parameters. Finally, thanks to the tools created for the test bench, the effects of the learning process in the performance of different neural networks will be discussed, and the one that achieves better results will be integrated within the digit classifier component to accomplish a greater robustness.

The results obtained reveal the great potential of the convolutional neural networks and cast some light on how they are able to learn what they learn, opening the door for their usage in the settlement of more complex problems.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 1, 3

ANN Artificial Neural Network. 2

CNN Convolutional Neural Network. 1, 3, 4, 8, 9, 18, 21, 22, 25, 26, 29, 43, 47, 48, 51–54

GUI Graphical User Interface. 18, 26, 30

HDF5 Hierarchichal Data Format version 5. 8, 13, 15, 26, 37, 40

LMDB Lightning Memory-Mapped Database. 37, 40

ML Machine Learning. 1–3

MNIST Modified National Institute of Standards and Technology. 18, 19, 28, 33–37, 39

ReLU Rectified Linear Unit. 9, 11, 22, 23, 51

ROI Region of Interest. 28

Chapter 1

Introduction

1.1 Context and motivation

Since the term *Artificial Intelligence (AI)* was coined at the Dartmouth Conference in 1956 [1] until nowadays, this field of computer science has developed at a great pace. During this time, its contributions to robotics and computer vision have brought machines that can solve certain tasks as well as humans and, in some cases, even surpass human performance. In order to understand the context in which this project has been developed, the fields and subfields that have led to the birth of *Convolutional Neural Networks (CNNs)* are going to be defined, trying to clarify the differences between them and how they are related to each other.

Artificial intelligence. "It is the subfield of computer science devoted to developing programs that enable computers to display behavior that can (broadly) be characterized as *intelligent*" [2]. In this definition, intelligent refers to the ability of perceiving the environment and acting consequently, trying to maximize the chances of achieving a certain goal [3].

Machine learning. According to a quote attributed to Arthur Samuel, it is the "field of study that gives computers the *ability to learn* without being *explicitly programmed*"[4]. Given this definition, it can be asserted that Machine Learning (ML) is a subfield of AI, because computers that have the ability to learn will exhibit an intelligent behaviour, but displaying an intelligent behaviour doesn't necessarily mean to learn. For instance, *Deep Blue* chess-playing system can be considered intelligent as it achieves a *human comparable performance*, but instead

of actually learning to play, it was hard-coded with a function that evaluated the board positions [5]. Machine learning algorithms can be divided in two main groups:

- **Supervised learning**. Given a training set of N example input–output pairs $(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$, where each y_j was generated by an unknown function $y = f(x)$, supervised learning algorithms try to find a function h that approximates the true function f[3]. The example inputs and their outputs are usually called samples and labels, respectively.
- **Unsupervised learning**. In this case, example inputs x_j are provided without their corresponding outputs y_j. The goal of these algorithms is to find a function that describes some underlying structure in data[6].

The main challenge of ML is *generalization*, that is, "the ability to perform well on previously unobserved inputs"[5]. By contrast, if the algorithm learns to generate correct outputs for already seen inputs, but not for unseen ones, it is said to *overfit*. In ML, the example inputs are usually *vectors of features* extracted from data. These example inputs, and their corresponding outputs in the case of supervised learning, are treated as *training data*. As the training process is *iterative*, some examples are usually kept as *validation data*, which are used to evaluate the algorithm performance during training. Validation is usually employed to stop training when a certain criteria is met, avoiding overfitting. When the training process finishes, *test data*, formed by unseen samples, are fed to the algorithm to evaluate its performance.

Artificial neural networks. They are a computational approach that tries to model the way a *biological neural network* solves problems[5]. As it can be seen in Figure 1.1, they're formed by layers of interconnected *neural units*.

In Artificial Neural Networks (ANNs), each neural unit sums the weighted input signals and apply an *activation function* that can be linear or non-linear (see Figure 1.2). The result of this operation is transferred to the neurons of the next layer. During training, weights are updated based on a *learning rule* that will try to minimize the difference between the current output and the desired one. As ANNs are able to learn from experience, they're classified within the ML field. It is important to clarify that if an ANN employs only linear activation functions, it will only be able to solve linearly separable problems. The usage of non-linear activation

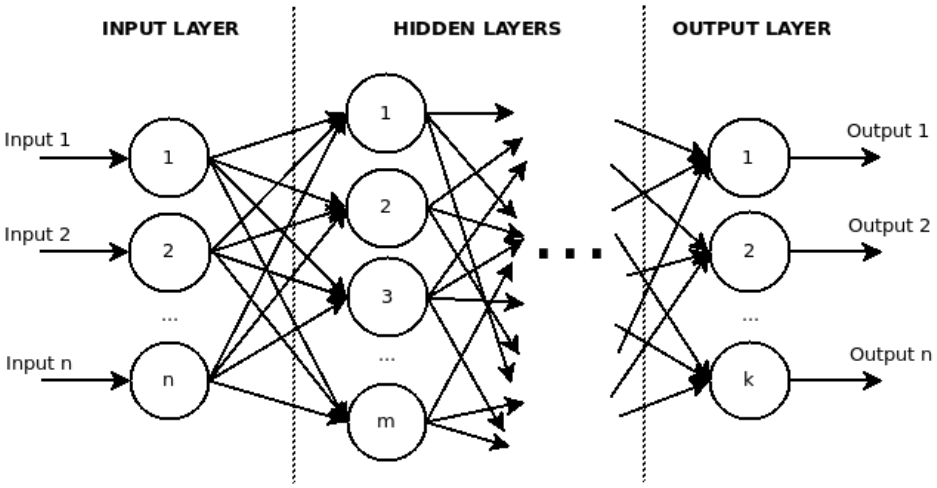


Figure 1.1: Basic architecture of an artificial neural network.

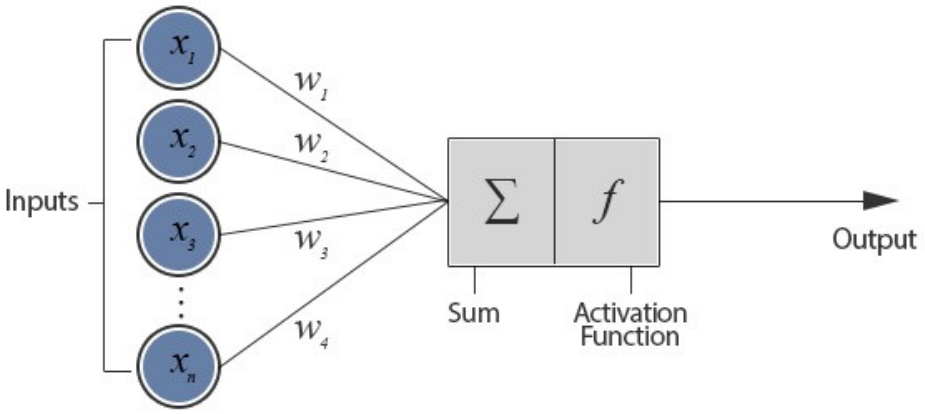


Figure 1.2: A neural unit (source [7]).

functions allows the settlement of more complex tasks. Examples for both cases are shown in Figure1.3.

Deep learning. It is a branch of ML which is based on algorithms that share the following properties [8]:

- *Multiple layers* of *non-linear* processing units.
- The supervised or unsupervised *hierarchical learning* of feature representations in each layer.

Although the term *deep learning* is not explicitly linked to ANNs, in practice we could talk about deep learning as a subset of neural networks algorithms that share the properties mentioned above. With deep learning algorithms, it is no longer

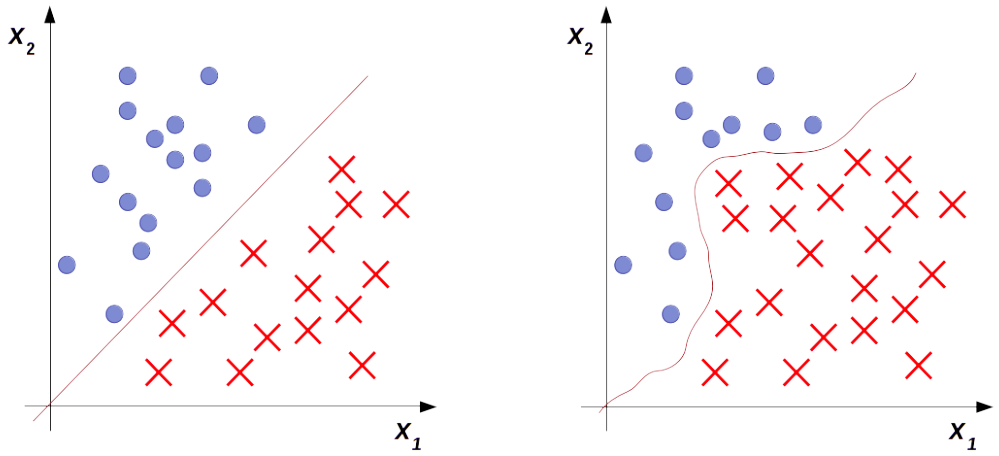


Figure 1.3: Example of linearly and non-linearly separable problems.

necessary to extract a vector of features to represent the input data, as these algorithms have the ability of learning, not only how to generate a correct output, but also how to represent data in a hierarchical way.

Convolutional neural networks. They are "neural networks that use *convolution* in place of general matrix multiplication in at least one of their layers" [5]. CNNs are deep learning algorithms which are specifically designed to process data that have a *grid-like topology*, like images and audio. In a few years, they have become one of the most promising subfields of ML, outperforming the results achieved by the previous algorithms in the most popular benchmarks for tasks like object classification, object detection and natural language processing. The details about how CNNs work will be the deeply discussed in the following chapters.

Computer vision. It is a field of computer science which "aims to build *autonomous systems* which could perform some of the tasks that the *human visual system* can perform" [9]. In order to build autonomous systems, computer vision applications have to deal with image acquisition, processing and analysis. Computer vision has always been closely related to AI and ML, and in the last years, the integration of deep learning algorithms (e.g. CNNs) in computer vision applications has led to major advances in the field.

There are multiple *motivations* behind this project. On the one hand, I am very

passionate about *computer vision*, because of its great implications in everyday life. For instance, it is involved in medical imaging, surveillance, augmented reality, automatic inspection in manufacturing, self-driving cars... and the list goes on and on. On the other hand, applications that have just been mentioned, have benefited from the inclusion of *deep learning* in computer vision. It is really exciting to see how many researchers are currently working in the field. Of course, this is not a coincidence. The great results achieved with these new algorithms has boosted the growth of AI in the last years and has normalized its usage in *commercial applications*. However, everything has its ups and downs. There are lots of people worried about the so-called *black box problem* in deep learning [10], i.e. how machines are actually learning to do what they do. In this work, besides developing a real-world application to show how powerful CNNs are, we're going to try to cast some light into the aforementioned *black box*.

1.2 Objectives

The ultimate objective of this project is to fully understand CNNs in order to integrate them in a computer vision application that must be able to solve a real-world task, specifically a *real-time handwritten digits classifier*. This main objective has been divided in the following sub-objectives:

- Accomplishing a *deep understanding* of how a basic CNN works, analyzing its main layers, the learning process and the particularities of building this kind of networks with *Keras*, a neural network library for Python.

- Building a *component* which integrates a CNN to classify images of handwritten digits in real time.

- Developing a *test bench* which allows the comparison of the performance achieved by the CNNs. This test bench must provide the input data and the tools required to visualize and evaluate the results.

- Studying the *effects of the learning process in CNNs performance* when they are trained with different datasets, architectures and regularization methods.

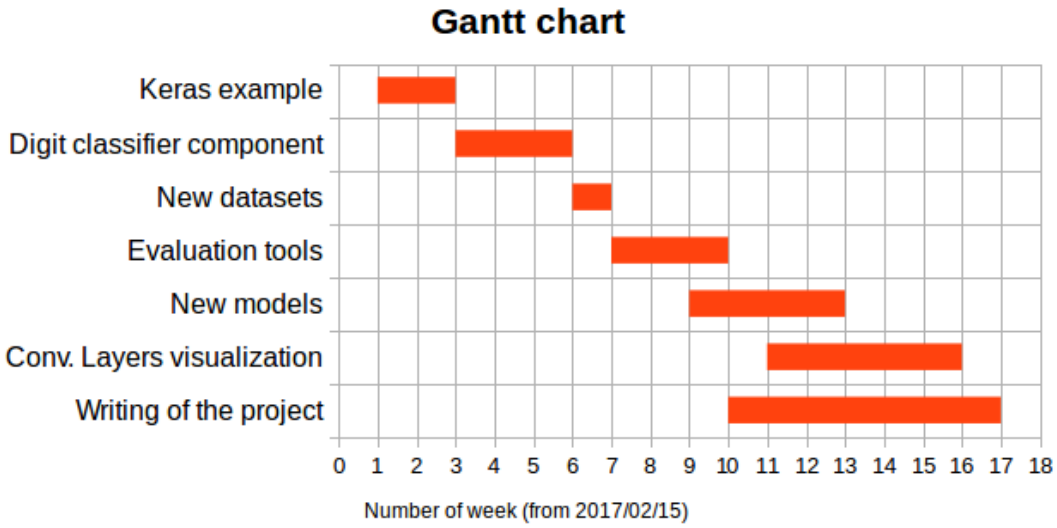


Figure 1.4: Gantt chart.

1.3 Methodology

The development of this project has been weekly followed by the tutors. In the *weekly meetings* the work done in the previous week was discussed and new milestones were set for the following one. This methodology has allowed a continuous feedback, which has led to a better understanding of the topic. Besides that, thanks to the weekly meetings the workload has been constant during these months.

Additionally, the following tools have been employed to keep track of the project progress:

- **GitHub**. All the code written in this project is available in GitHub and has been frequently updated. In the following link, the main repository can be accessed: `https://github.com/RoboticsURJC-students/2016-tfg-david-pascual`

- **MediaWiki**. It has been used as a logbook of the progress of this project. It can be accessed in the following link: `http://jderobot.org/Dpascual-tfg`

In Figure 1.4, a *Gantt chart* with the number of weeks dedicated to every task in the project can be seen.

1.4 Project structure

For ease of reading, the structure followed in the writing of this document and how the chapters are related to each other are explained in this section.

Chapter 1. Introduction. This chapter starts with a brief introduction to the main fields in which CNNs are based and the motivations behind the project. Then, the objectives, the methodology followed to meet these objectives and the structure of the project are presented.

Chapter 2. Framework. The third-party software that has supported the development of this project is described here. It's specially significant the section about Keras library, its main functionalities and the theory behind its core layers and learning process.

Chapter 3. Digit classifier. In this chapter, an example of a CNN built with Keras for handwritten digits classification is analyzed. Then, the internals of the developed component, in which the CNN is integrated, are explained.

Chapter 4. Test bench. The datasets that will be used for training new models and the tools that will be employed to evaluate and visualize the results are described in this chapter.

Chapter 5. Evaluation. Taking as a starting point the CNN that was analyzed in the *Digit classifier* chapter, new models will be created. The performance of these new models will be evaluated with the tools developed in the *Test bench* chapter.

Chapter 6. Conclusions. Finally, the conclusions reached during the development of this project will be summarized and possible further works will be proposed.

Chapter 2

Infrastructure

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 third-party software can be easily integrated in our specific applications, which are mainly written in *Python*[1]. These applications will reach a high-quality performance faster than if they would have been written from scratch.

2.1 Keras framework

As stated by *Keras* documentation [11]: "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. Keras treats them as its *backends*. The main version used in this project is Keras 1.2.2, although it has been recently updated to 2.0.4 version because of its backward compatibility. In this work, Keras has run on top of *Theano*[2], optimized for CPU and has been employed to train and implement several CNNs.

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

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

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

2.1.1 Models

Every neural network in Keras is defined as a *model*. For those models which can be built as a stack of *layers* (see Section 2.1.2), Keras provides the *.Sequential() object*. In the following chapter, an example of a sequential model built with Keras is shown in Figure 3.2. 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*. Its main arguments are:

- ***loss***: name of the *cost function* that measures the difference between predicted and real labels. In this project, the *categorical cross-entropy*, also known as log loss, has been used. This function returns the cross-entropy between an approximating distribution q and a true distribution p [12] and it's defined as:

$$H(p, q) = -\Sigma_x p(x) \log(q(x)) \tag{2.1}$$

  Other loss functions such as mean squared error (MSE) and mean absolute error 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*. This optimizer is an extension of the *gradient descent* optimization method that has the particularity of adapting automatically the learning rate during training [13]. Other optimization methods such as Adagrad, Adamax and Adam are also available.

- ***metrics***: name of the functions that must be computed during training and testing for performance evaluation. *Accuracy* is the only function which will be evaluated with Keras through this project, besides the loss function, which is automatically computed. It is defined as the proportion of examples for which the model produces the correct output [5]. Other measurements about the performance of the model are obtained with the *Scikit-learn* library (see Section 2.3).

.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 (see Section 2.1.3) that are going to be applied during training. It defaults to *None*.
- ***validation_split* or *validation_data***: in Keras, there are two alternatives to provide a validation dataset. On one hand, it is possible to pass the validation data as a Numpy array to the *validation_data* argument. On the other hand, a fraction of the training samples can be set as validation data through the *validation_split* argument. *validation_data* and *validation_split* arguments are mutually exclusive, so just one of them can be used.
- ***shuffle***: a boolean that determines whether to shuffle training data or not. If data are not shuffled during training, samples belonging to the same class can be presented consecutively. In that case, the model will be forced to learn the features of a certain class. When the model starts to see samples of the next class, it fits to the new data and forgets the previously learned feature. If data are sorted by classes, this process goes on and on leading to a worse performance.

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

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

.save() It stores the model into a *Hierarchichal Data Format version 5 (HDF5) file* (see Section 2.2), 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/`

2.1.2 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 *CNN*. It is formed by a certain amount of *filters/kernels* with a fixed size. These filters are convolved with the input volume, 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 that volume [14]. In our case, the input volume will be a three dimensional matrix defined by its width (number of columns), height (number of rows) and depth (number of channels). It's important to note that the number of channels 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. It defaults to 1.
- ***padding***: it can be *valid* or *same*. If *valid* mode is set, no padding is applied, resulting in an output that will be smaller than the input. 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 2.1 shows a detailed representation of the operation performed by convolutional layers. In Figure 2.1(a), the filter w_0 (3x3x3) is convolved with the input volume (5x5x3). In this example, padding is set to 1 and stride is equal 2, so the operation will return a 3x3 activation map. The same procedure is followed in Figure 2.1(b) with the filter w_1. It generates another 3x3 activation map, ending up

with a 3x3x2 output. To clarify this process, let's see how to obtain the first element in the second row of the activation map returned by the filter w_0, i.e. the element within the green square in Figure 2.1(a).

1. An element-wise or *Hadamard product* is performed between the filter w_0 and the corresponding region (*receptive field*) of the input volume x.

$$z_1 = x[:,:,0]_{r.field} \odot w_0[:,:,0] = \begin{bmatrix} 0 \cdot 1 & 1 \cdot 0 & 1 \cdot 0 \\ 0 \cdot 1 & 1 \cdot 1 & 2 \cdot -1 \\ 0 \cdot -1 & 2 \cdot -1 & 0 \cdot -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -2 \\ 0 & -2 & 0 \end{bmatrix} \quad (2.2)$$

$$z_2 = x[:,:,1]_{r.field} \odot w_0[:,:,1] = \begin{bmatrix} 0 \cdot -1 & 2 \cdot -1 & 2 \cdot -1 \\ 0 \cdot 0 & 1 \cdot -1 & 2 \cdot 0 \\ 0 \cdot 1 & 0 \cdot 1 & 2 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & -2 & -2 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.3)$$

$$z_3 = x[:,:,2]_{r.field} \odot w_0[:,:,2] = \begin{bmatrix} 0 \cdot 0 & -1 \cdot 2 & 1 \cdot 0 \\ 0 \cdot 0 & -1 \cdot 0 & 1 \cdot 1 \\ 0 \cdot 0 & 0 \cdot 0 & -1 \cdot 0 \end{bmatrix} = \begin{bmatrix} 0 & -2 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad (2.4)$$

2. Then, all elements of the three resulting matrices are added together.

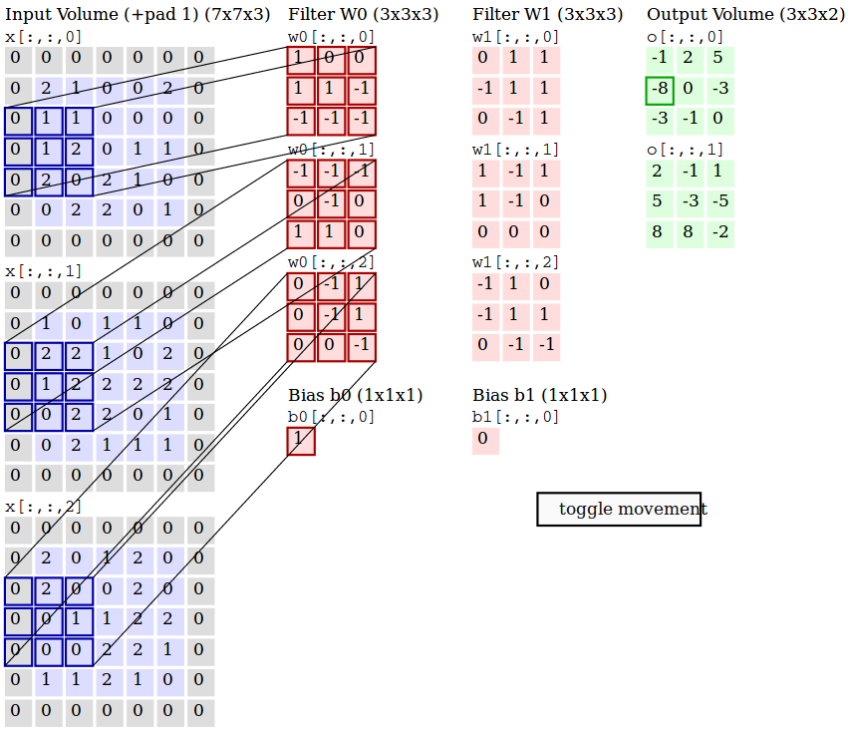
$$\Sigma z_1 + \Sigma z_2 + \Sigma z_3 = -3 - 5 - 1 = -9 \quad (2.5)$$

3. Finally, the *bias* (b_0) is added to the scalar returned by the previous addition. Bias allows to *shift* the input of the activation function away from the origin, like the constant in a linear function. For instance, if the receptive field is filled with zeros, the bias gives the opportunity of generating a non-zero output. Activation functions will be discussed later in this section.

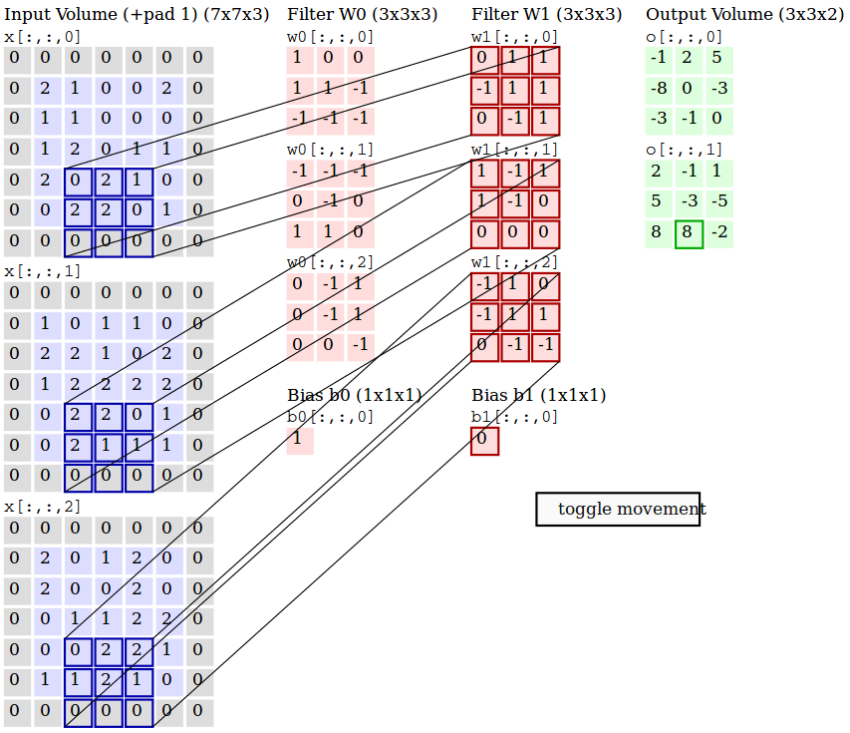
$$o[:,:,0] = -9 + b_0 = -8 \quad (2.6)$$

4. To compute the value of the next element of the activation map, the same operation is applied to the next receptive field which, according to the stride that has been set, must be 2 pixels away from the previous one.

A more intuitive and general representation of how a convolutional layer works is shown in Figure 2.2.



(a)



(b)

Figure 2.1: Example of operation of a convolutional layer (source [14]): (a) filter w_0; (b) filter w_1.

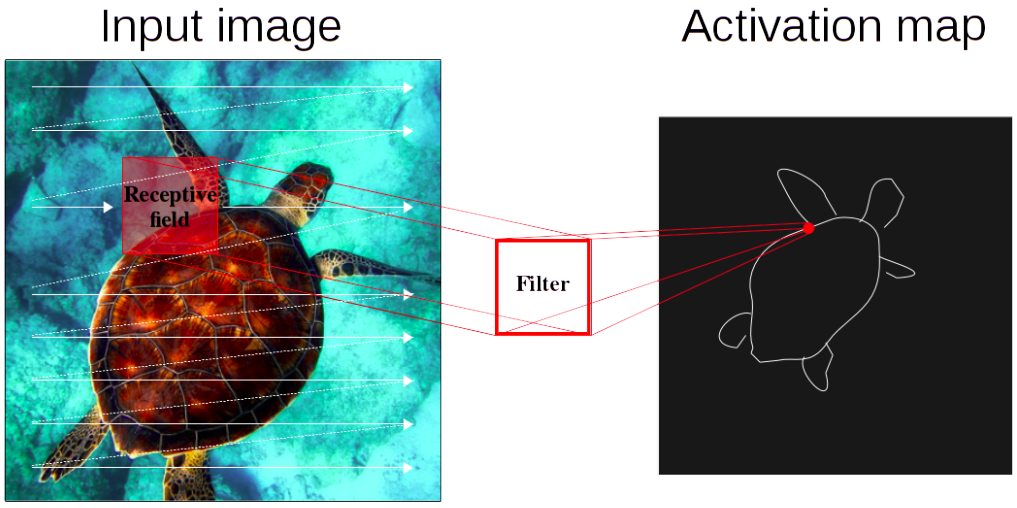


Figure 2.2: Intuitive representation of a convolutional layer.

Pooling layer. It shifts a window of a certain size along the input volume applying an operation (mean or maximum) that will return a *downsampled version* of it, reducing the computational cost and avoiding overfitting [15]. Figure 2.3 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.
- ***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 (i.e. output) of the previous one [14]. The main argument of this layer is:

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

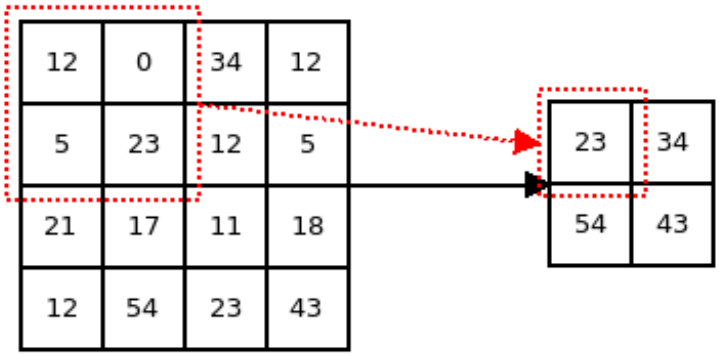


Figure 2.3: Example of a max. pooling operation.

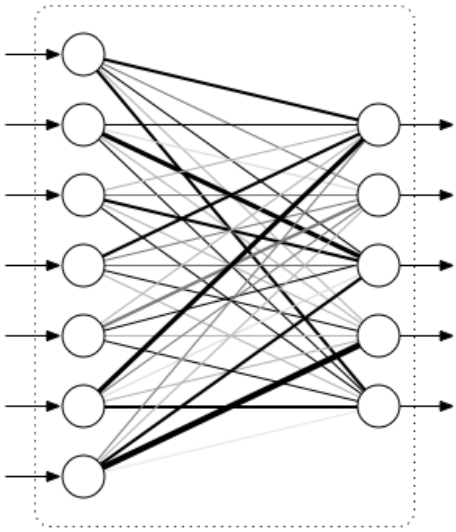


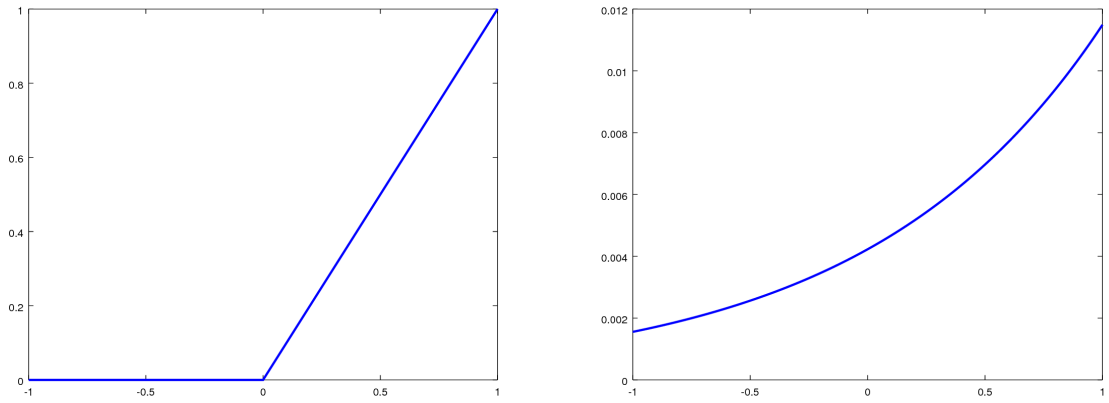
Figure 2.4: Diagram of a dense or fully-connected layer (source [16]).

Figure 2.4 shows the diagram of a dense layer. The weights of each connection are represented by the thickness of the lines.

Activation layers. In Keras models, an activation function 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. These are the ones that have been used during the development of this project:

- **ReLU**: this activation function introduces a *non-linearity* right after each convolutional layer, allowing the CNN to learn more complex features. It's defined as:

$$g(z) = \max(0, z) \tag{2.7}$$



(a) (b)

Figure 2.5: Activation functions: (a) ReLU; (b) softmax.

- **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:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\Sigma_j \exp(z_j)} \quad \text{for } j = 1, ..., N \tag{2.8}$$

These equations and definitions have been extracted from [5]. Figure 2.5 shows both 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 weights* 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 overfitting. Dropout is a technique that randomly *switches-off* a fraction of hidden units during training [17]. It can also 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" [5], as it can be seen in Figure 2.6.

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

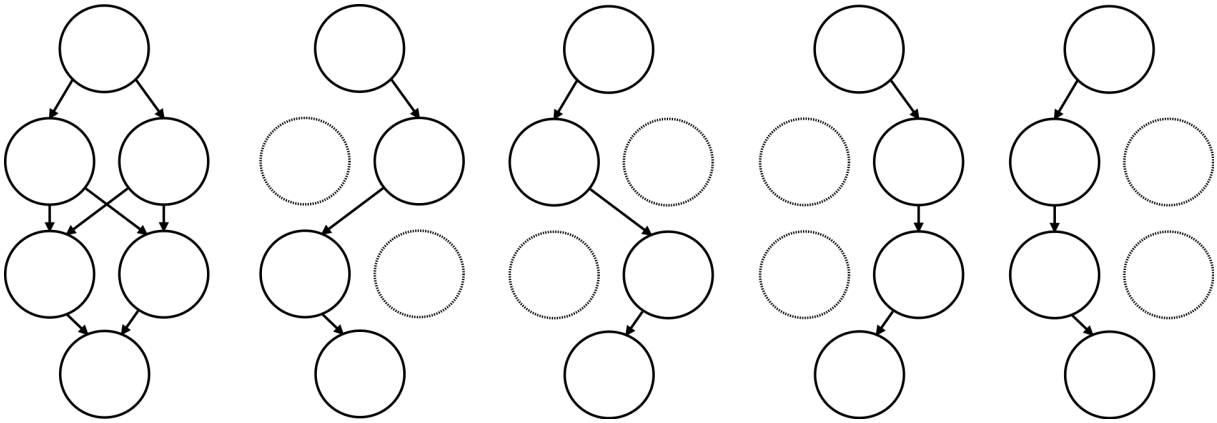


Figure 2.6: Subnetworks generated when using dropout.

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

2.1.3 Callbacks

As defined by Keras documentation [11], *callbacks* are a set of functions 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. After each epoch, this callback evaluates the declared *metrics* with the validation dataset and saves the results.

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

- ***.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*.

2.1.4 Image Preprocessing

Image preprocessing is a key factor in every computer vision application. Specifically, in ML, besides adapting images and emphasizing certain particularities before training that can improve the model performance (i.e. edge detection), it can be used to avoid *overfitting* through data augmentation. *Data augmentation* [18] consists in taking the original images of the dataset and applying transformations to them. This operation adds more variability to samples, leading to a better generalization.

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.

2.1.5 Utils

Keras includes 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* (see Section 2.2), which are going to be used as inputs to the neural networks.

2.2 HDF5 file format

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

According to HDF5 documentation [19], it is a *hierarchical data format* designed for high volumes of data with complex relationships. While relational databases employ tables to store data (e.g. SQL), 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*[4] library for Python has been employed.

2.3 Scikit-learn and Octave

Scikit-learn is a ML library that includes a wide variety of algorithms for clustering, regression and classification [20]. It can be used at every stage of the ML workflow: 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 evaluation parameters which have been employed in this project (equations and definitions obtained from [21]):

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

$$\text{precision} = \frac{true_{positives}}{true_{positives} + false_{positives}} \tag{2.9}$$

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

$$\text{recall} = \frac{true_{positives}}{true_{positives} + false_{negatives}} \tag{2.10}$$

- **Confusion matrix**: a two dimensional matrix where the element in the position i, j represents the number of samples that belongs to the group i but have 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 computed with Scikit-learn can be found in Figure 2.7.

Besides the functions that have just been mentioned, *accuracy* and *log loss* have also been used and they're defined as in Section 2.1.1.

GNU Octave [22] is a scientific programming language compatible with *Matlab*. It provides powerful tools for plotting. In this work, it has been used to visualize the aforementioned parameters collected with Scikit-learn about the performance of the models.

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

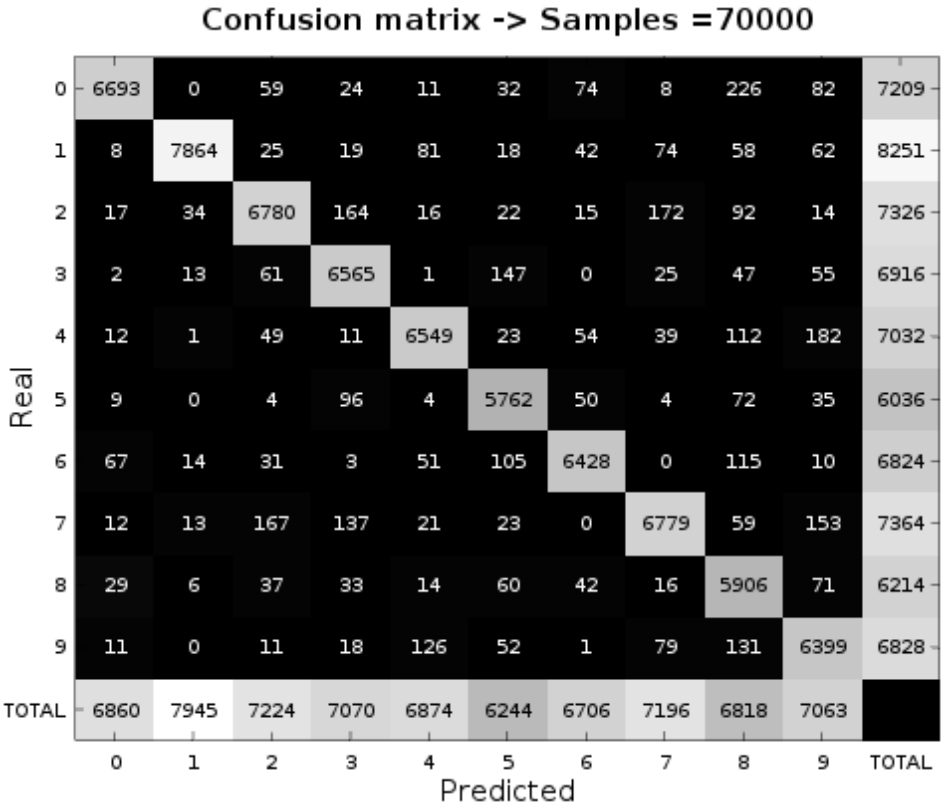


Figure 2.7: Example of a confusion matrix.

2.4 JdeRobot framework

JdeRobot is an open source middleware for robotics and computer vision [23]. 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*[5]. It is also compatible with the robotics middleware *ROS*[6], which allows the interoperation of ROS nodes and JdeRobot components. This flexibility makes it very useful for our application. The version employed in this work is JdeRobot 5.5. This middleware and, more specifically, its *cameraserver driver*, is going to be employed to capture images from different video sources that will feed the digit classifier component.

cameraserver

According to JdeRobot documentation [23], this driver can serve images both from real cameras and from video files. It communicates with other components thanks to the *ICE 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.

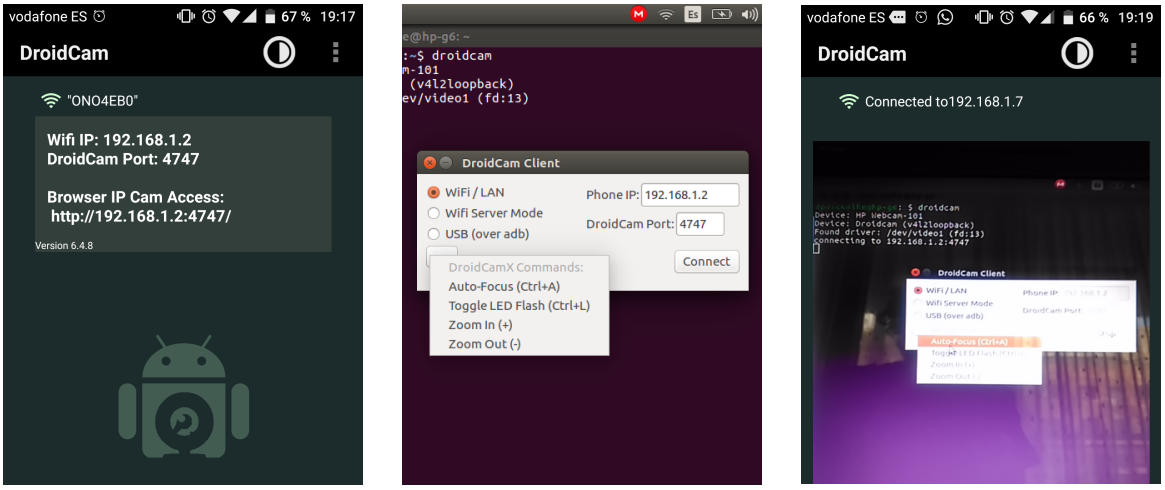
2.5 DroidCam

On the one hand, *DroidCam* is an application for Android which serves the images captured with a *smartphone camera* [24]. 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*[7] *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 at which the Linux client must be listening to receive the images. Besides that, it provides a URL that can be used to access the video stream from

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

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

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



(a) (b) (c)

Figure 2.8: Droidcam usage: (a) Android server; (b) Linux client; (c) connection established.

any browser. The DroidCam version used in this work is DroidCam 6.4.8. It will enlarge the list of possible video sources for the digit classifier component by serving the images of a smartphone camera.

An example of usage can be seen in Figure 2.8. First, the Android app is opened. It shows the address where the video will be served (see Figure 2.8(a)). Then, the address is set in the Linux client (see Figure 2.8(b)). Finally, when the *Connect* button is pressed, the connection is established (see Figure 2.8(c)).

JdeRobot *camerserver* driver cleanly connects to the camera device that the DroidCam Linux client provides in the host computer when attached to the DroidCam Android app.

Chapter 3

Digit classifier

Taking advantage of the CNNs impressive performance in classification tasks, we have built a *real-time digit classifier*. Its core elements are:

- A *Keras model* (see Section 2.1.1), which classifies the images.
- A *JdeRobot component*, which acquires and processes the images from a video stream and integrates a Keras model to classify them. The images and the classification results are displayed within a Graphical User Interface (GUI).

3.1 Understanding the Keras model

Understanding how Keras models work is a key factor in the development of this project. For this purpose, an adapted version of an example provided by Keras[1] will be analyzed in the following sections. In this example, a CNN is trained and tested with the *Modified National Institute of Standards and Technology (MNIST) database* of handwritten digits (see Section 4.1).

3.1.1 Adapting data

First of all, the input data have to be loaded and adapted. Keras library contains a module named *datasets* from which a variety of databases can be imported, including MNIST. 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, as

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

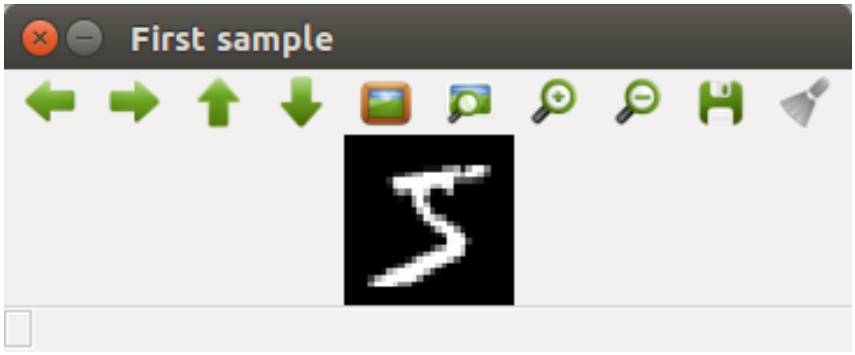


Figure 3.1: First sample of the MNIST database.

it is shown in the code below, which also displays the first sample of the MNIST training dataset (see Figure 3.1).

```
(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)
```

That code also prints the shape of the dataset:

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

According to that, the training dataset includes 60000 images, each one containing 28x28 pixels. In order to feed the Keras model, the number of channels of the samples have to be explicitly declared, 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. For instance, if they had been RGB images, the number of channels would have been equal to 3. As data are 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 Keras *backend*, as it is shown in the following code.

```
img_rows, img_cols = 28, 28
...
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 training dataset gets reshaped as follows:

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

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

```
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
```

Regarding the *labels*, they are originally shaped as an array in which each element is an integer in the range $[0, 9]$, i.e. each element contains the digit that corresponds to a certain sample. In order to feed the Keras model, the labels have to be reshaped into an array in which each element must be a *probability distribution*. For example, 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()*:

```
nb_classes = 10
...
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 ('----------------------------------------------------------------')
```

That 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)
```

3.1.2 Model architecture

Once the data are ready, the CNN architecture must be defined. In this example, a *sequential model* (see Section 2.1.1) is enough for solving the classification task and it is declared as follows:

```
model = Sequential()
```

The next step is to add the corresponding layers. The core layers of a CNN, as treated by Keras, have been already defined in Section 2.1.2. The following code performs the addition of the layers to the model.

```
nb_filters = 32
kernel_size = (3, 3)
pool_size = (2, 2)
...
# 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.
  - Since this is the first layer of the model, the *input_shape* argument must be provided. In this case, the input shape is 28x28x1.
  - As *valid* mode is set, *no padding* is applied and the output dimension will be reduced.
  - *ReLU activation function* (see Equation 2.7) introduces non-linearity into the network. If the activation functions were linear, the whole stack of layers could be reduced to a single layer, losing much of the ability to learn different levels of features.
  - 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 overfitting.
  - 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* (Equation 2.8).

The resulting architecture and data shape after every layer are shown in Figure 3.2.

3.1.3 Compiling the model

After declaring the model and defining its architecture, the *learning process* must be set through the *.compile()* method. The arguments required to set this process are defined in Section 2.1.1. The code can be seen in the next frame:

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

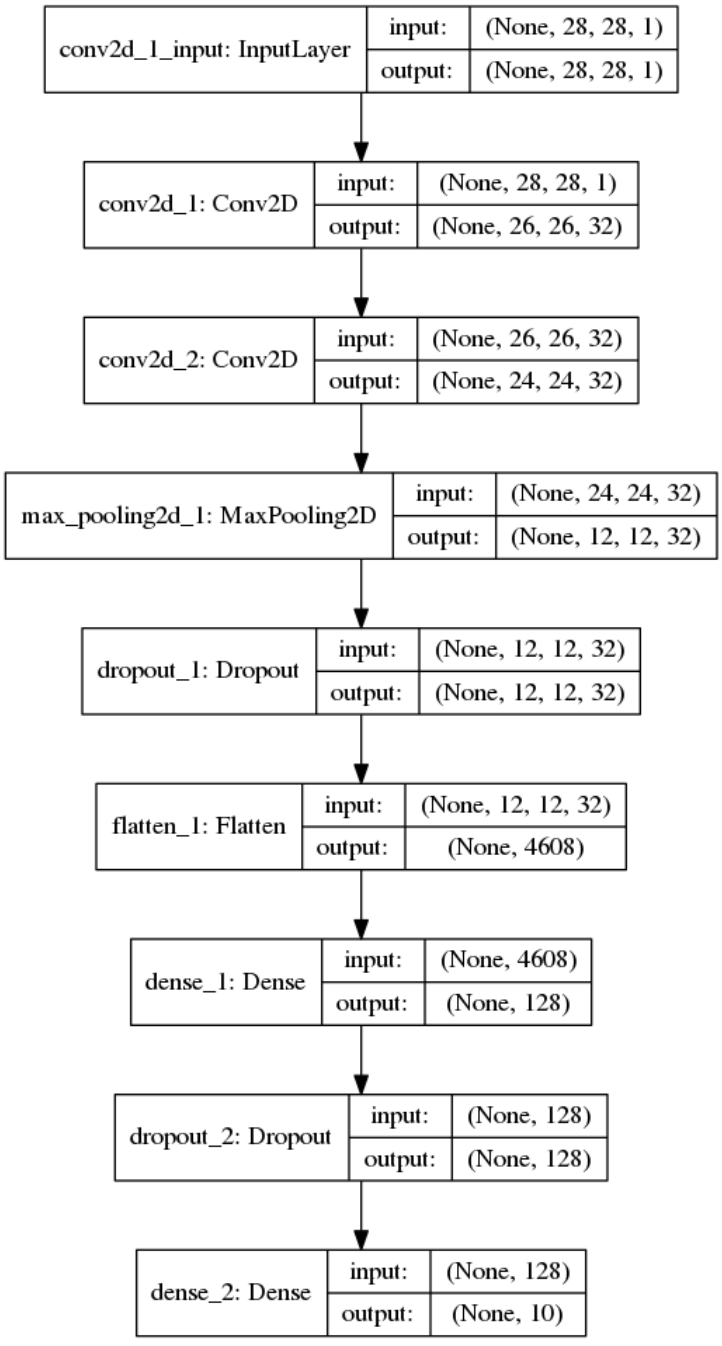


Figure 3.2: Diagram of a Keras sequential model.

In this example, the loss function that is computed after every batch is the *categorical cross-entropy* (see Equation 2.1) and the optimizer that updates the weights of the CNN in order to minimize that loss function is *ADADELTA*. Additionally, the *accuracy* is also computed to monitor the CNN performance during training.

3.1.4 Training the model

The CNN is trained thanks to the *.fit()* method, which has been already described in Section 2.1.1. The usage of that method can be seen in the code below.

```
nb_epoch = 12
batch_size = 128
...
model.fit(x_train, y_train, batch_size=batch_size, nb_epoch=nb_epoch,
          verbose=1, validation_data=(x_test, y_test))
```

The model will be 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 will be 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
...
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
```

3.1.5 Testing the model

Once the model is trained, its weights, architecture and learning configuration can be stored in an *HDF5 file* (see Section 2.2). Besides that, in order to see the performance

of the CNN, the *.evaluate()* method takes the test dataset and computes the *log loss* and the *accuracy*, as it is shown in the following code:

```
model.save('MNIST_net.h5')


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
```

3.2 JdeRobot component

Once the CNN has been trained and the resultant model is saved, the next milestone is to integrate it into a JdeRobot component that must be able to acquire images from a video stream, apply the necessary preprocessing and display the predictions obtained from them. That component is *digitclassifier.py* and it is based on Nuria Oyaga's code[2]. It relies on *Camera* and *GUI* classes. Figure 3.3 shows the program running.

3.2.1 Design

Before analyzing in detail the internals of the digit classifier component, its design will be described. For this purpose, the *block diagrams* shown in Figure 3.4 have been built.

Figure 3.4(a) shows a *high-level diagram* of the process followed from the video source to the predicted digit. The images captured with the camera are received by the *cameraserver* driver. Then, the digit classifier component communicates with this driver and get the images. Finally, a prediction is made. Optionally, if the images are captured with a smartphone camera, DroidCam can receive those images and create a camera device that can communicate with the *cameraserver* driver.

In Figure 3.4(b), the *internals of the digit classifier component* are shown. The main program (*digitclassifier.py*) starts two threads, *ThreadCamera* and *ThreadGUI*. These

[2] https://git.io/vH0qD

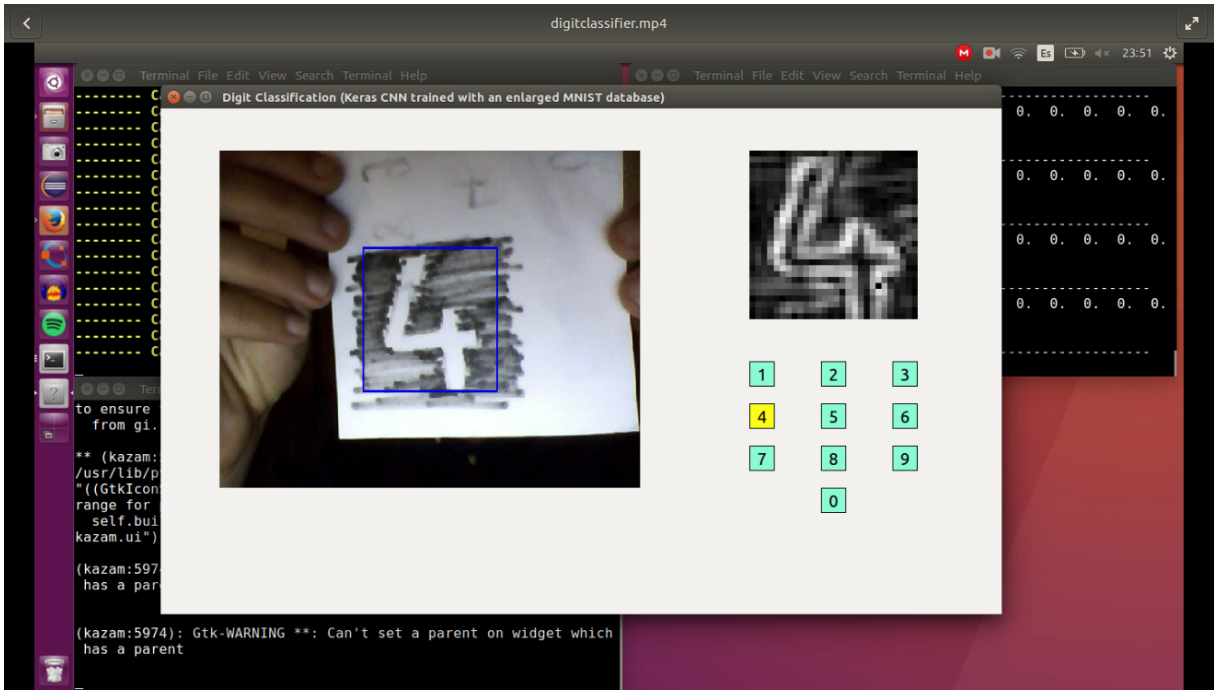


Figure 3.3: Example of *digitclassifier.py* execution.

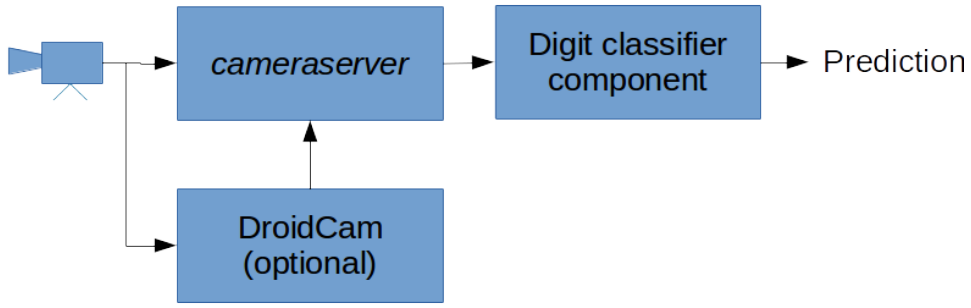
threads update *GUI* and *Camera* classes concurrently. *Camera* class communicates with *cameraserver* to get the images, which are then preprocessed. The preprocessed image is fed to the Keras model and a prediction is made. Original and preprocessed images, as well as the predicted digits are passed to the *GUI* class, which displays all of them in the screen.

3.2.2 *Camera* class

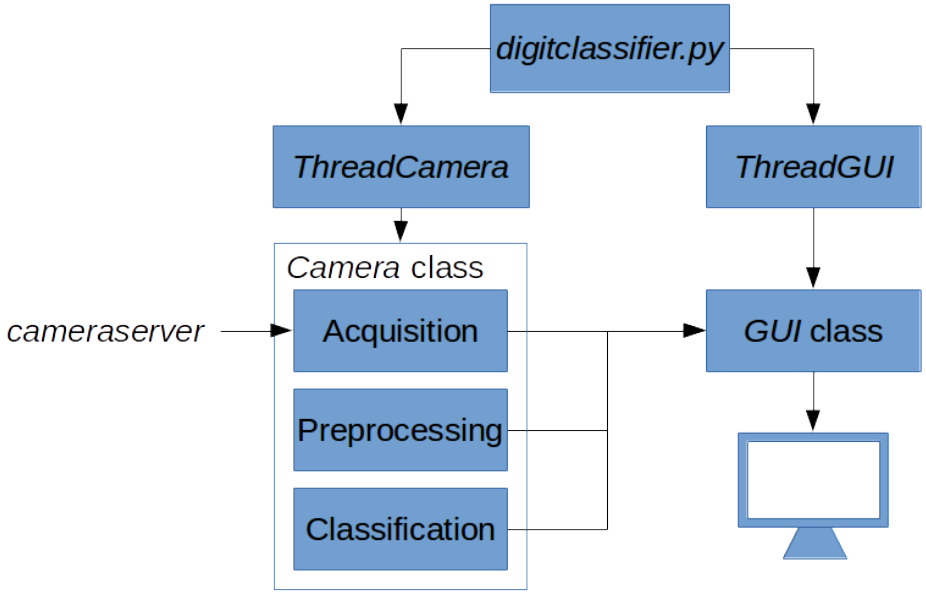
Camera class is responsible for getting the images, transforming them into a suitable input for the Keras model and returning the classification results.

- **Acquisition**. The images are served by the JdeRobot component *cameraserver* (see Section 2.4). Depending on how its configuration file (*cameraserver.cfg*) has been set, the images can come from different kinds of video streams. 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, as it is shown in the code below.

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



(a)



(b)

Figure 3.4: Digit classifier design: (a) high-level; (b) lower-level.

```
CameraSrv.Camera.0.Uri=/home/dpascualhe/video.mp4  # video file
```

In order to establish a connection with smartphone cameras, the *DroidCam application* for Android has been used (see Section 2.5). As this application turns the video stream provided by the smartphone into a *v4l2*[3] *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 must be provided to the *Camera* class through another configuration file: *digitclassifier.cfg*.

- **Preprocessing**. As the images can be captured with different devices, the *digitclassifier.py* component must apply some preprocessing 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 *Region of Interest (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*. For this reason, the images captured with the component are 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 [25]. In this case, the kernel size will be 5x5 and the standard deviation is automatically calculated depending on that size. The 2D Gaussian filter, as defined in [26], is given by:

  $$G(x, y) = \exp(-\frac{x^2 + y^2}{2\sigma^2}) \tag{3.1}$$

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

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

5. Last step is obtaining the *gradient of the images*. Working with this kind of images instead of the original ones allows the application to deal with different light and color conditions. The chosen algorithm for this task is *Sobel filter* (Equation 4.1). This will be deeply discussed in Section 4.1.2.

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 Section 3.1.1. *Camera* class calls Keras *.predict() method* (see Section 2.1.1) to get the predicted digit. The prediction is only taken into account if one of the probabilities is equal to 1, avoiding wrong answers when the prediction is not clear. The function that performs the classification can be seen in the following code:

```
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)
if dgt[1].size == 1:
    self.digito = dgt
else:
    self.digito = (([0]), (["none"]))

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

3.2.3 *GUI* class

GUI class displays the original image, the preprocessed image and the result of the classification, as it is shown in Figure 3.3. It has been built employing the *pyQt package*[45]. It is based in Nuria Oyaga's code[6], but it has been updated from Qt4 to Qt5 thanks to the information provided by PyQT documentation [27].

3.2.4 Threads

In order to capture images and update the GUI concurrently, the *threading module* [28], provided by Python, has been employed. From this module, a subclass of the *Thread object* is created. In this new subclass, *__init__()* and *.run()* methods are overriden. The *.run()* method will be responsible for calling a process that updates 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. The next frame shows how the *ThreadCamera* class is coded:

```
import time
import threading
```

[4] https://pypi.python.org/pypi/PyQt4
[5] https://pypi.python.org/pypi/PyQt5
[6] https://git.io/vHOmx

```
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);
```

3.2.5 Main program

All of these elements are joined together in *digitclassifier.py*. *Camera*, *GUI* and their threads are initialized and the *.start()* methods of the *Thread* objects are invoked, as it is shown in the code below:

```
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:

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

An example of usage of the digit classifier component can be seen in Figure 3.3.

Chapter 4

Test bench

The model described in Section 3.1 can be improved with different architectures and regularization methods. Besides that, training the model with new datasets can also lead to better results. In order to *compare the performance* of these new models, a *test bench* has been developed. In this chapter, the datasets employed to train the models, as well as the tools created to measure and visualize their performance, will be described.

4.1 Datasets

The digit classifier component is possible thanks to the data provided by the *MNIST database* of handwritten digits. In the following sections, the pros and cons of using this database and some alternatives will be discussed.

4.1.1 Original dataset

MNIST is a database of *handwritten digits* formed by a training set, which contains 60000 samples, and a test set, containing 10000 samples [29]. It's a *remixed* and reduced version of the original *NIST datasets*[1]. MNIST is a well-known benchmark for all kinds of ML algorithms.

As may be seen in Figure 4.1, 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 ML algorithms, it's not enough to train a model that aims to solve a *real-world task*, because the images are almost noiseless and the digits within them share similar

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

Figure 4.1: Samples extracted from the MNIST database.

orientation, position and size.

4.1.2 Gradient images

The first issue with MNIST database that must be addressed is that all the digits are white 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, the *gradient of the images* has been calculated. The resultant samples are less dependent from the light and color conditions than 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[2], the operator that leads to better results is the *Sobel filter*. This operator approximates the gradient of an image function [26], convolving the image with the following kernels to highlight horizontal and vertical edges, respectively:

$$h_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{4.1}$$

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

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

4.1.3 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 the *.ImageDataGenerator()* method provided by Keras (see Section 2.1.4), 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 has been 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 can be seen in Figure 4.2.

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

Besides these transformations, it's also necessary to simulate the *noise* that may 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 treats noise layers as regularization methods that are only active during training time to avoid overfitting. In order to add noise to the generated samples, a GaussianNoise layer was established as the *input layer* of the model.

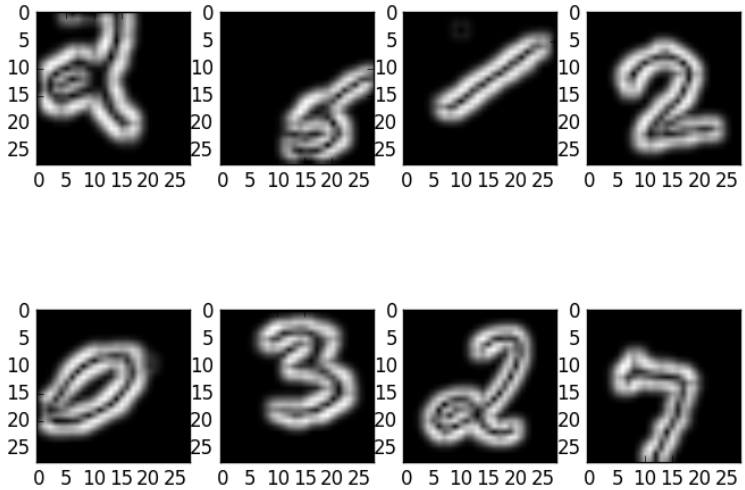


Figure 4.2: Samples generated with Keras from MNIST database.

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[3]. These are the new databases:

- **Sobel**: MNIST database after applying the Sobel filter to every image. 48000 samples for training 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 transformed ones. 48000 samples for training 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 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

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

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.

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

In Figure 4.3, the first samples of every handmade dataset can be seen.

From LMDB to HDF5

These databases were initially built to feed a *Caffe* [30] neural network. That's why they were saved as *Lightning Memory-Mapped Database (LMDB) files*[4]. In order to make it easier to feed the Keras model, the LMDB databases have been converted into *HDF5 files* (see Section 2.2). For this conversion, the script *datasetconversion.py* has been written.

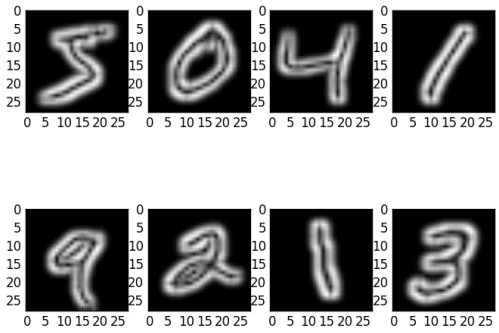
- **Reading the LMDB database**. On one hand, the LMDB library for Python[5] has been employed to open the database, initialize a cursor and iterate over each key-value pair in the database. On the other hand, *Google's Protocol Buffers*[6] have been used to parse the data that were extracted from the 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" [31]. Here can be seen the *.proto* file that defines the data structure used by Caffe to store the MNIST database, as obtained from [32]:

```
package datum;
message Datum {
   optional int32 channels = 1;
   optional int32 height = 2;
   optional int32 width = 3;
```

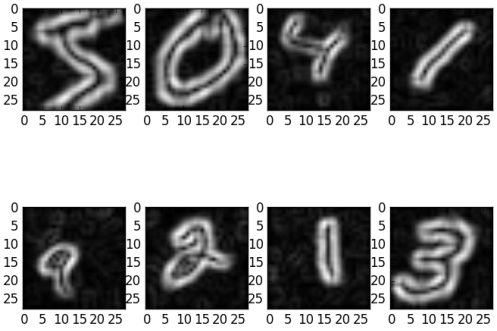
[4] http://www.lmdb.tech/doc/

[5] https://lmdb.readthedocs.io/en/release/#

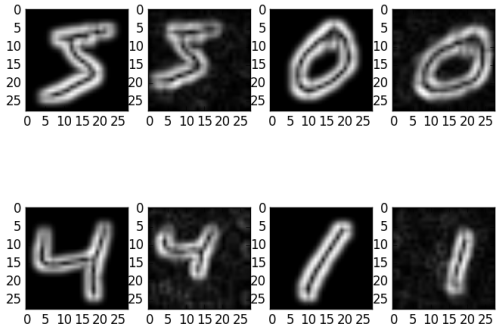
[6] https://developers.google.com/protocol-buffers/



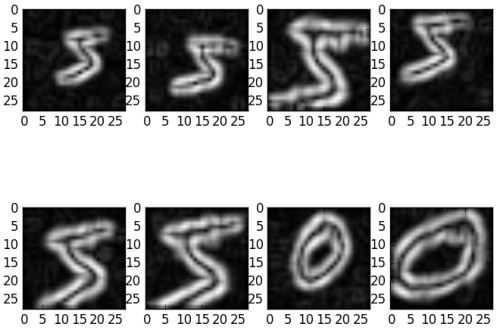
(a)



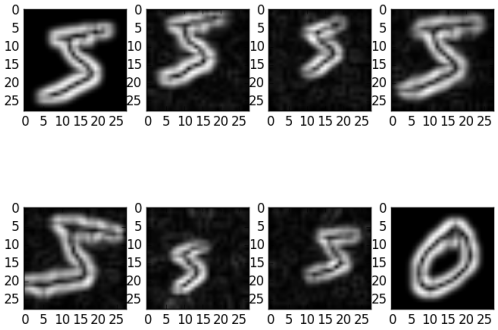
(b)



(c)



(d)



(e)

Figure 4.3: First samples of handmade datasets: (a) Sobel; (b) 0-1; (c) 1-1; (d) 0-6 ; (e) 1-6.

```
   // 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**. Thanks to the *h5py library*[7] for Python, the data extracted from the LMDB database has been stored into a HDF5 file with two datasets: label and data. The code can be seen in the following frame:

```
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 use the *handmade datasets*. While real-time data augmentation is really useful to avoid storing all the data that are needed for training, it makes 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 *comparing the performance* of neural networks built with different libraries, so they must be trained with the same datasets.

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

4.2 Measuring performance

The performance of the models will be evaluated using the *CustomEvaluation* class and the measurements calculated with this class will be visualized using an Octave function. In this section, both of them will be described.

4.2.1 *CustomEvaluation* class

CustomEvaluation class is totally independent from Keras. It calls functions that measure the performance of the model during test and/or training time and saves them into a file which is compatible with Octave.

- **Obtaining the measurements**. The user provides the real labels and the probability distribution of the predicted ones. Log loss, accuracy, precision, recall and a confusion matrix are computed. These functions are defined in Section 2.3 and Section 2.1.1.
- **Storing results**. *CustomEvaluation* class stores the results in a *Python dictionary*. Additionally, it can store the *learning curves* if *training* option is set. In the following section, the Keras callback employed to build the learning curves will be discussed.
- **Python-Octave *translation***. For this task, the *SciPy library*[8] has been used. It provides the *.savemat()* method that saves Python dictionaries into Matlab *.mat* files, which are also compatible with Octave (see Section 2.3).

Here's a usage example:

```
if training == "n":
   measures = CustomEvaluation(y_test, y_proba, training)
else:
   train_loss = learning_curves.loss
   train_acc = learning_curves.accuracy
   val_loss = validation.history["val_loss"]
   val_acc = validation.history["val_acc"]
   results = CustomEvaluation(y_test, y_proba, training, train_loss,
   train_acc, val_loss, val_acc)
```

[8] https://docs.scipy.org/doc/scipy-0.18.1/reference/index.html

```
results_dict = results.dictionary()
results.log(results_dict)
```

LearningCurves callback

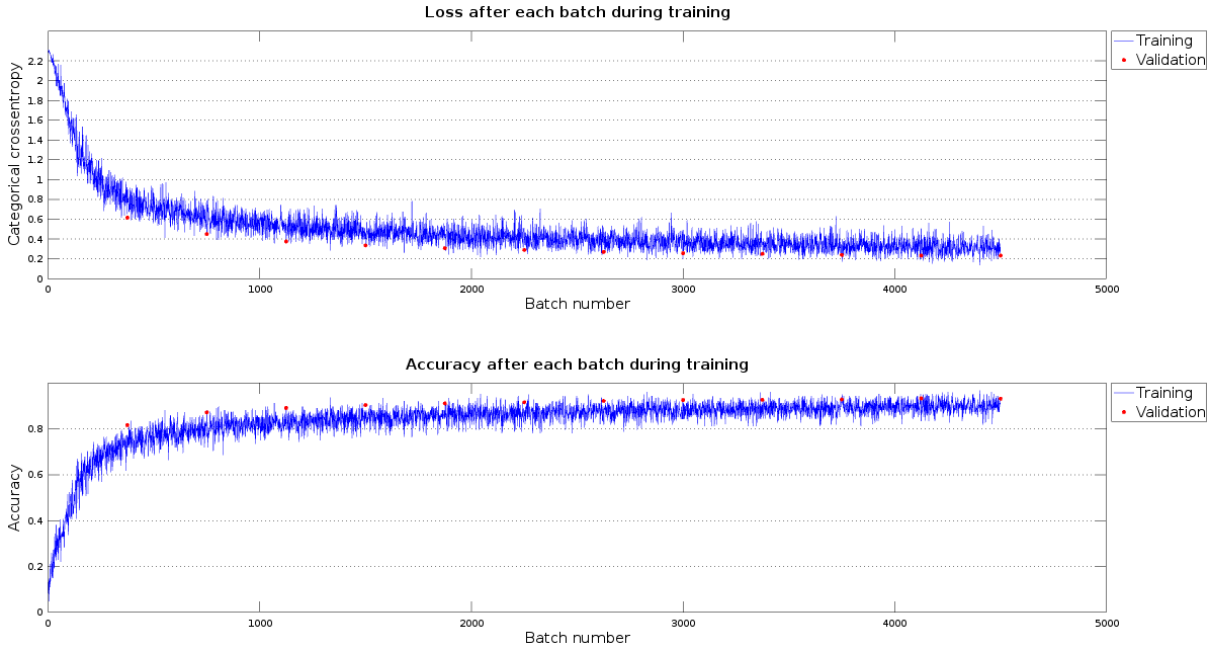
During training time, Keras automatically saves into a *.History()* object (see Section 2.1.3) the validation results obtained after every epoch. It's interesting to face these validation results with the ones obtained after every batch during training.

LearningCurves is a custom Keras callback that has been coded to save the accuracy and loss obtained after each batch into Python lists. The code below shows how it works.

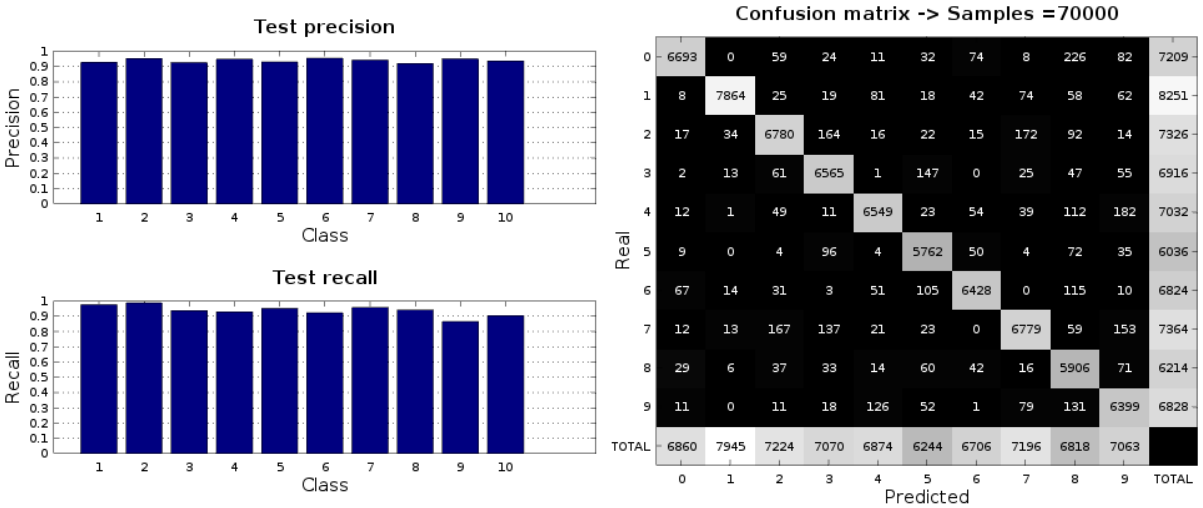
```
class LearningCurves(keras.callbacks.Callback):
''' LearningCurve class is a callback for Keras that saves accuracy
and loss after each batch.
'''

def on_train_begin(self, logs={}):
   self.loss = []
   self.accuracy = []

def on_batch_end(self, batch, logs={}):
   self.loss.append(float(logs.get('loss')))
   self.accuracy.append(float(logs.get('acc')))
```

4.2.2 Octave function

Now that all the data have been collected, they have to be properly displayed. The function *visualization.m* has been written to address this issue. It takes as its only argument the path to the *.mat* file that has been generated with the *CustomEvaluation* class and plots the results as shown in Figure 4.4. Additionally, it prints them to the standard output.



(a)



(b)

(c)

Figure 4.4: Parameters displayed by *visualization.m*: (a) learning curves; (b) precision and recall; (c) confusion matrix.

4.3 Convolutional layers visualization

CNNs are well-known by their ability of learning *image features*. The weights of a convolutional layer are arranged like *a set of filters*, each of which learns to identify a certain visual feature [14]. As the filter is convolved with the input image, it generates an *activation map* that will tell us how that particular filter reacts to that image. In other words, the activation map will tell us whether a certain feature is present in the image or not.

In order to understand how the Keras model is learning to classify the digits, the *layer_visualization.py* script has been written. This script allows the user to display the filters that are learned in every convolutional layer of the model and their resulting activation maps.

4.3.1 Filters

Keras provides a list containing every *layer object* in the model through the attribute *model.layers*. Layer objects properties can be accessed thanks to the *layer.get_config()* method. Since convolutional layers in Keras are named with the prefix *conv2d*, it is possible to iterate over the names of the layers looking for that prefix to find the convolutional ones, as it is shown in the code below.

```
for i, layer in enumerate(self.model.layers):
   if layer.get_config()["name"][:6] == "conv2d":
...
```

Once the convolutional layers have been found, their *filters* are accessed through the *layer.get_weigths()* method. The matrices of weights returned by this method are reshaped to improve readability. The code that performs these operations can be seen in the following frame:

```
shape = layer.get_weights()[0].shape
weights = layer.get_weights()[0].reshape(shape[2], shape[0],
                                         shape[1], shape[3])
```

Finally, the filters are plotted thanks to the *Matplotlib library*[9] for Python. The shape

[9] https://matplotlib.org/

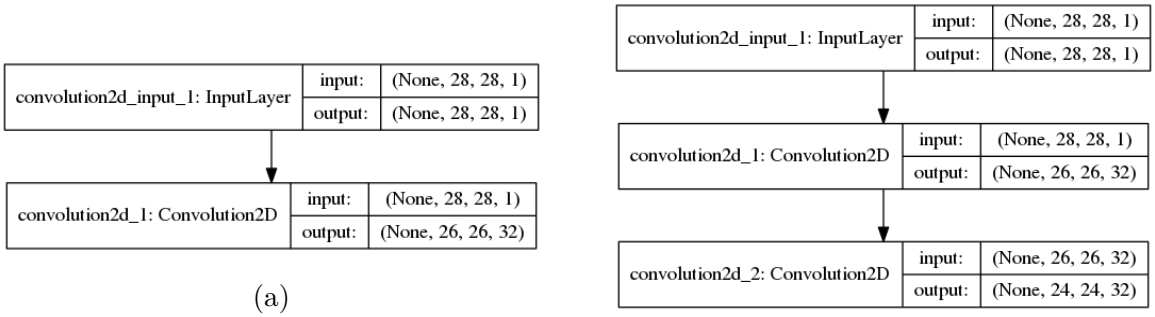


Figure 4.5: Truncated versions of the model: (a) one convolutional layer; (b) two convolutional layers

and the maximum/minimum values of the weights are printed to the standard output. An example of how the filters are plotted can be seen in the following chapter in Figure 5.2

4.3.2 Activation maps

The output of each convolutional layer is formed by as many *activation maps* as filters the layer has. In order to get the values of these activation maps, *truncated versions* of the original model are generated, as it is shown in Figure 4.5. When a prediction is made with these truncated models, they output the activation maps that correspond to their last layer. The code which obtains the activation maps can be seen in the follwing frame:

```
truncated = Model(inputs=self.model.inputs,
                  outputs=layer.output)
activations = truncated.predict(self.im)
```

The activation maps are plotted using the Matplotlib library as well, and their shape and maximum/minimum values are also printed to the standard output. An example of how the activation maps are plotted can be seen in the following chapter in Figure 5.4.

Chapter 5

Evaluation

The CNN analyzed in Section 3.1 is going to be taken as a starting point to build *new models*. These models will be trained with different datasets and regularization methods and, finally, new architectures will be implemented. In this chapter, the tools developed in Chapter 4 will be employed to evaluate the results achieved by each of these CNNs. These tests will serve as a way to study the effects of the learning process in these algorithms. Before talking about the *performance* of the new models, the visualization of the convolutional layers filters and activation maps is going to be analyzed.

5.1 Convolutional layers visualization

The filters and activation maps discussed in this section belong to the convolutional layers of the *0-1; Patience=2 model* that can be found in Section 5.3.1. This model has been trained with the *0-1* dataset (see Section 4.1.3) and an early stopping rule with patience 2. Its architecture corresponds to the one defined in Section 3.1. In order to generate the activation maps, the model will be fed with a sample extracted from the *0-1* dataset. It can be seen in Figure 5.1.

5.1.1 Filters

When loading the weights of the *first convolutional layer*, a Numpy array of shape (1, 3, 3, 32) is obtained. This means that the weights are arranged in 32 filters of size 3x3. In this case, the input is a grayscale image, so the filters only have one channel (i.e. depth=1). Besides that, when examining their values, *negative and positive coefficients* are found.

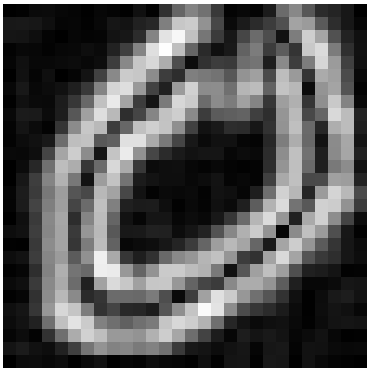


Figure 5.1: Sample employed to generate the activation maps.

In Figure 5.2, these filters are plotted. Some of the filters look too noisy to tell which kind of feature they are looking for. However, a few of them can be interpreted at first sight as follows:

- **Horizontal edge-emphasizing filter**: filters 7, 9 and 23.
- **Vertical edge-emphasizing filter**: filters 8, 15 and 31.

The filters in the *second convolutional layer* have been displayed as well. The weights in this layer are stored in a Numpy array of shape (32, 3, 3, 32), which means that there are 32 filters with size 3x3 as well. However, this time their depth is 32, since there is one channel per activation map generated by the previous layer. As we get deeper into the CNN and the dimensionality grows, the filters look noisier and become harder to interpret, as it is shown in Figure 5.3.

5.1.2 Activation maps

Figure 5.4 shows the activation maps that the *first convolutional layer* of the model outputs. There are *horizontal and vertical edge images* that confirm the interpretation of the filters given in the previous section. Besides that, some activation maps (2, 12, 19, 25 and 26) look *dead*. If we look back into Figure 5.2, these activation maps correspond to filters with *almost flat coefficients*. This may be a signal of a high learning rate [14]. In this case, the learning rate is not explicitly declared, because the ADADELTA optimizer uses an adaptive one.

The activation maps of the *second layer* are shown in Figure 5.5. The images obtained look *more specialized* than the ones in the previous layer. It's easier to tell to what kind of feature (e.g. edges and corners) each activation map is responding to.

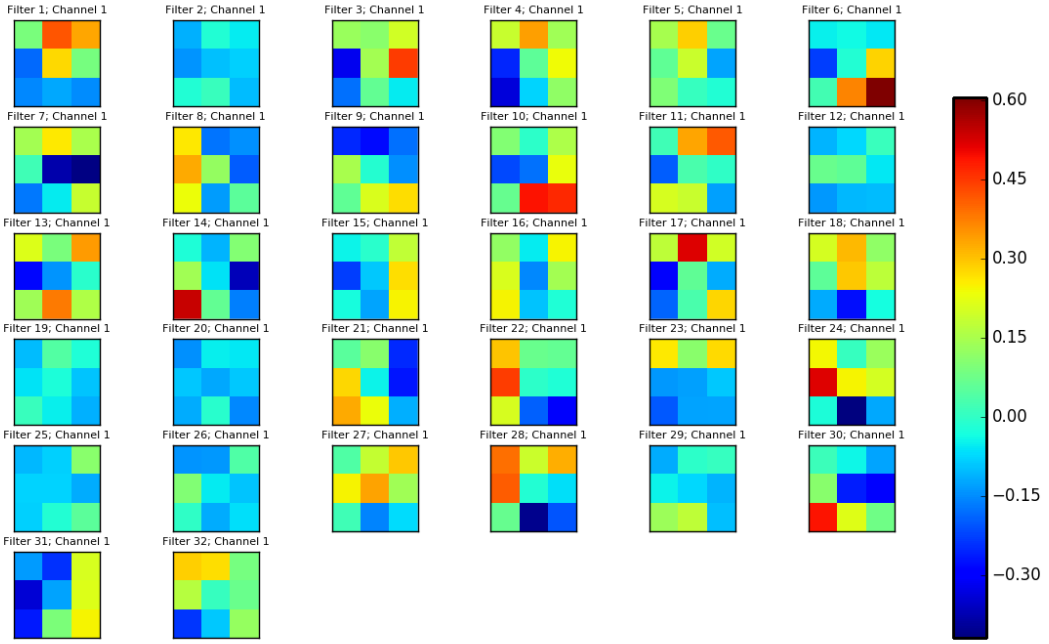


Figure 5.2: Filters of the first convolutional layer.

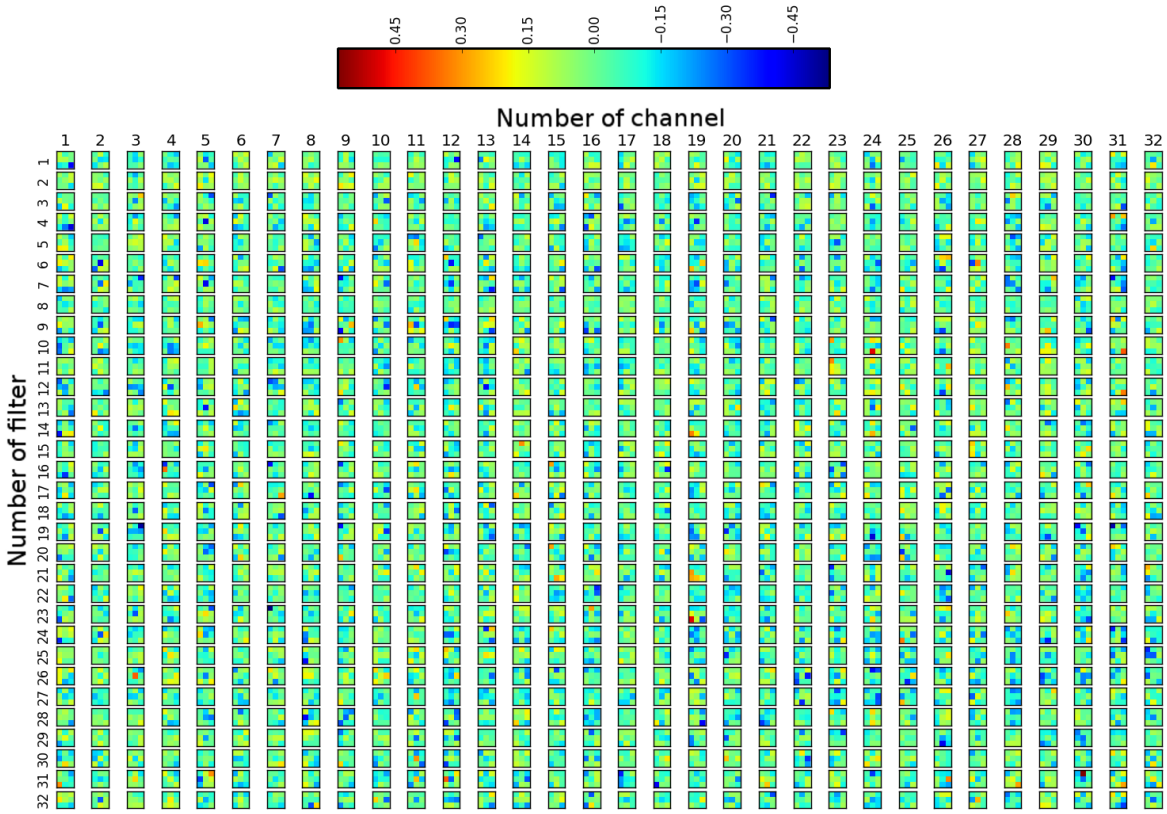


Figure 5.3: Filters of the second convolutional layer.

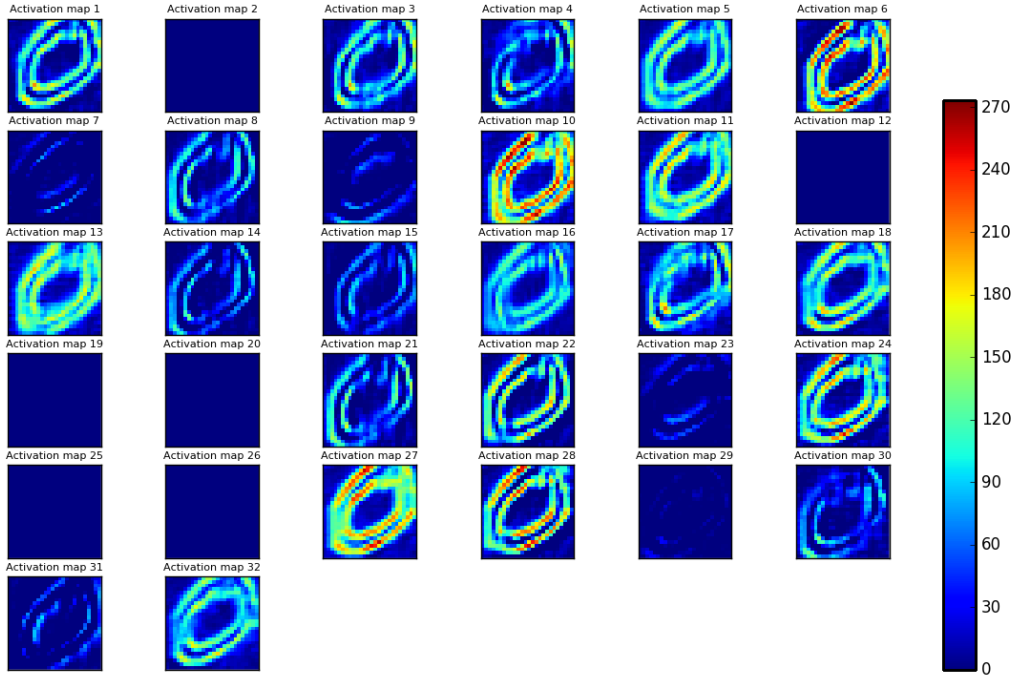


Figure 5.4: Activation maps of the first convolutional layer.

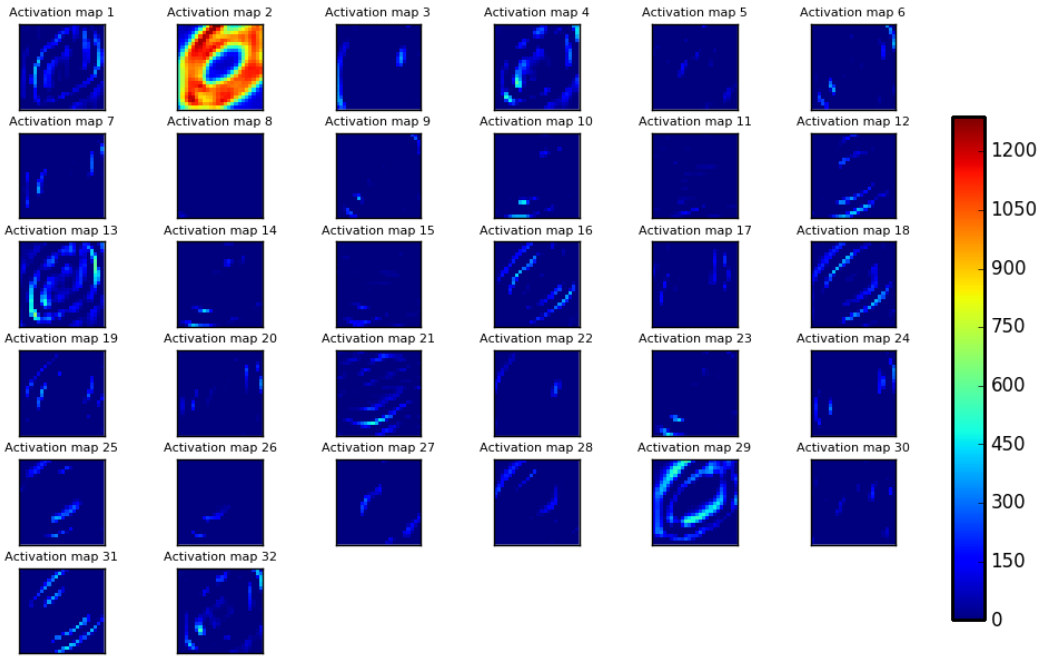


Figure 5.5: Activation maps of the second convolutional layer.

Model	Loss	Accuracy	Epochs
Sobel	1.233	0.699	12
0-1	0.201	0.939	12
1-1	0.189	0.943	12
0-6	0.109	0.968	12
1-6	0.111	0.967	12

Table 5.1: Results of training with different datasets.

It's important to note that the values of the activation maps are *always positive*, even if the filters have negative coefficients. This is because the ReLU activation function (see Equation 2.7) turn all the negative values to zero.

5.2 Augmented datasets

The original model has been trained with each of the *handmade datasets* described in Section 4.1. The number of epochs has been set to 12 and the evaluation has been carried out with the *1-6 test dataset*. The results that can be seen in Table 5.1 lead to the following conclusions:

- As it might be expected, the results when training with the *Sobel dataset* are much worse than the ones obtained with the other datasets, because we're testing with noisy images a CNN trained with noiseless samples.
- The *0-6 and 1-6 models* are the ones that achieve better results, as they have been trained with much more samples than *0-1* and *1-1*.
- When *comparing 0-1 with 1-1 and 0-6 with 1-6*, it can be seen that the performance is almost the same, which means that the gradient image without noise and transformations is not adding much information to the model.

In Figure 5.6, the validation results obtained after every epoch when training the model with each dataset can be seen.

Taking all of this into account, it has been decided to keep working with the *0-1 model*, which achieves a performance that is comparable with the other models with the advantage of a much lower computational cost.

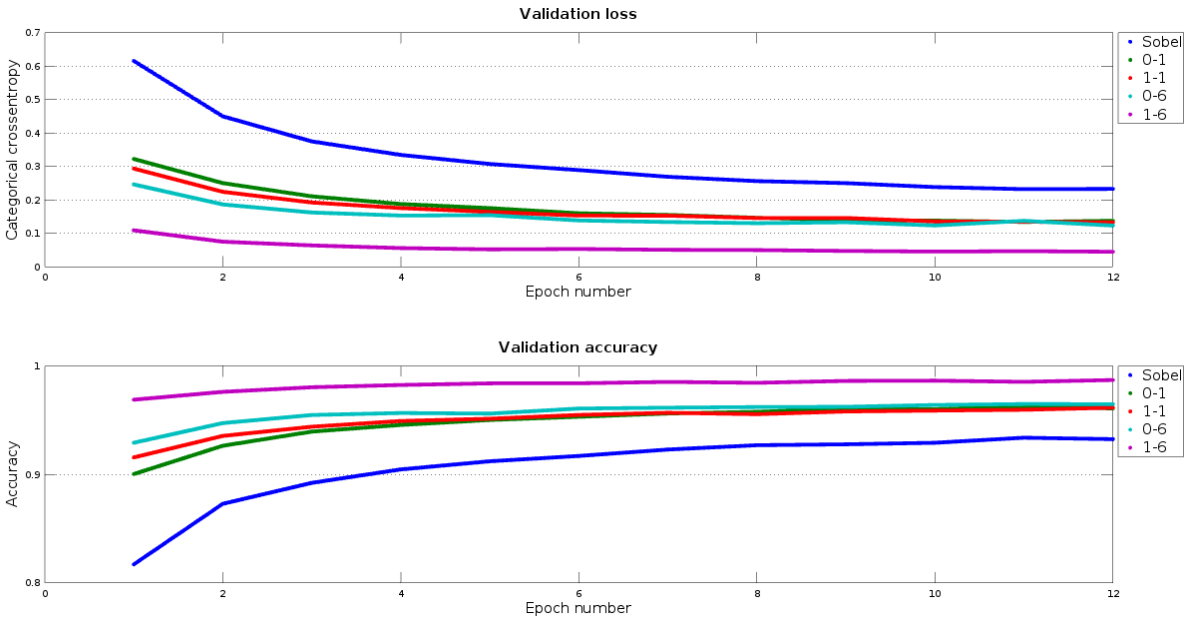


Figure 5.6: Validation results when training the model with different datasets.

5.3 Regularization methods

"*Regularization* is any modification we make to a learning algorithm that is intended to reduce its *generalization error* but not it's training error" [5]. Reducing the generalization error is important because, even if a model achieves a great accuracy or loss with the training dataset, if it doesn't generalize well enough, the results during validation and test time won't be optimal. This is specially significant in our case, since the predictions of the digit classifier will be based on images that differ a lot from the training dataset. In this section, the effects of applying to the *0-1 model* two regularization techniques (early stopping and dropout) are going to be evaluated.

5.3.1 Early stopping

The models in the previous section have been trained for 12 epochs. However, if we look at the *validation results* in Figure 5.6, it can be assumed that the models were *not overfitting* yet, because the results didn't stop improving. This means that they were not being trained as much as possible. Setting an *early stopping* rule (see Section 2.1.3) allows training the CNN right until it starts to overfit, making the most of it. The criteria that has been used depends on the loss during validation. The model is trained until the *log loss* (see Section 2.1) has not improved after two validations in a row, which means a

Model	Loss	Accuracy	Epochs
0-1	0.201	0.939	12
0-1; Patience=2	0.155	0.954	30

Table 5.2: Results of training with and without early stopping.

Model	Loss	Accuracy	Epochs
No dropout	0.189	0.945	9
Dropout	0.155	0.954	30

Table 5.3: Results of training with and without dropout.

patience of 2. Besides that, in order to keep the best *version* of the model, the log loss is checked after each epoch and, if the value is lower than the previous best log loss achieved, the weights of the model are saved, overwritting the weights of the previous best *version*. The difference between training the model with and without early stopping can be seen in Table 5.2.

Early stopping means an improvement of 1.6% in accuracy and 4.6% in log-loss. The model has been trained for 30 epochs and it reached its best *version* at the 27^{th} epoch. Setting a longer *patience* has been considered, but it has been decided to apply it only to the best model obtained in Section 5.4 to reduce the computational cost.

5.3.2 Dropout

The models that have already been evaluated insert *dropout* (see Section 2.1.2) before every dense layer of the CNN (0.25% and 0.5%, respectively). Dropout is usually applied just to fully-connected or *dense layers*, because convolutional layers are less likely to overfit due to their architecture. In order to determine how dropout affects the performance of the CNNs, the *0-1; Patience=2 model*, defined in the previous section, has been trained with and without the mentioned dropout. The results can be seen in Table 5.3.

Without dropout, the model has stopped training after 9 epochs. It has *learned faster*, but it has started *overfitting* earlier, resulting in worst results that the ones achieved by the model trained with dropout. This can be clearly seen in Figure 5.7

Additionally, in Figure 5.8, the learning curves of both models can be seen. It's worth looking into these plots to realize that validation results are better than training results

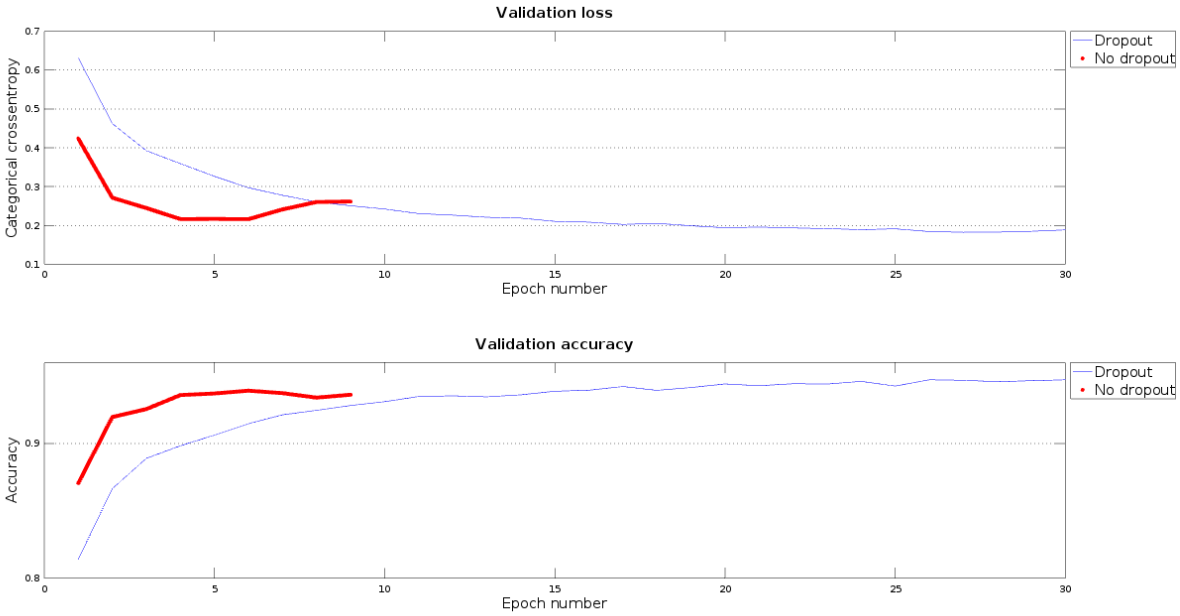
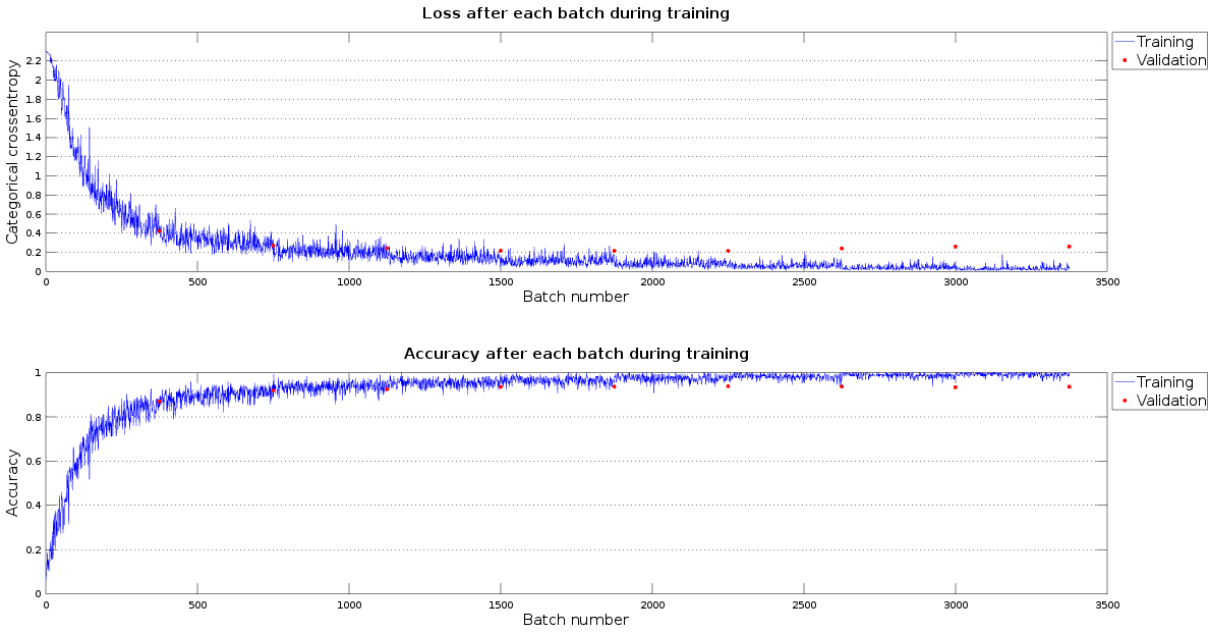


Figure 5.7: Validation results with and without dropout.

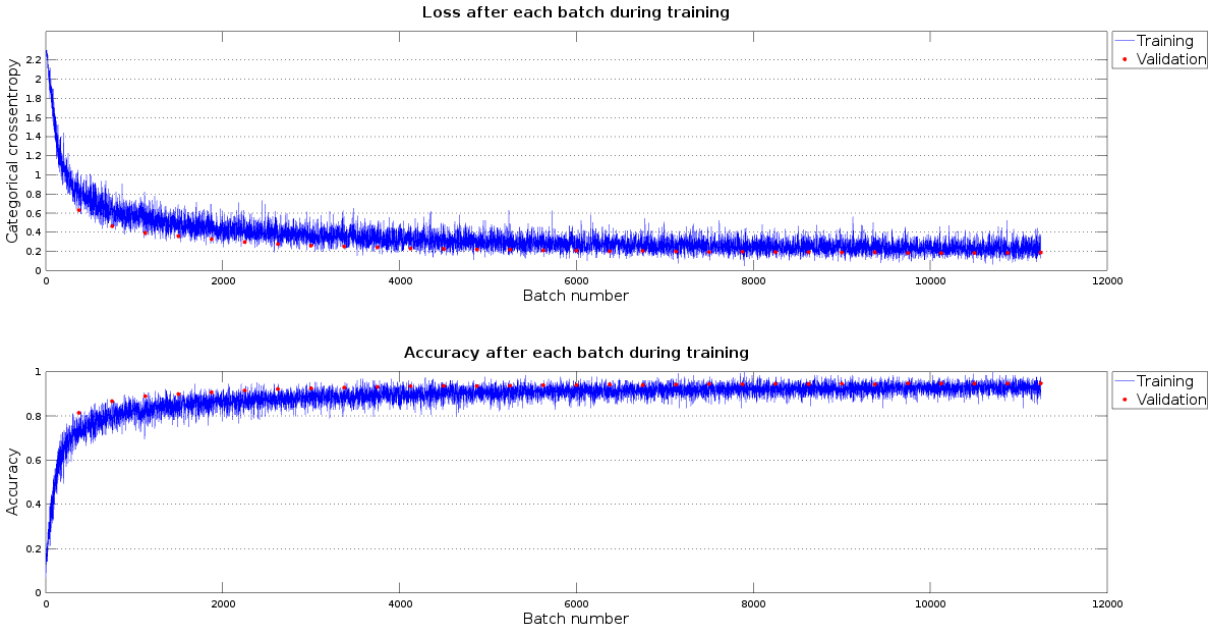
when the model is trained with dropout. This may seem illogical, as the CNN should always perform better with samples that it has already seen. However, it's important to remember that dropout only applies during training and, as it will *switch-off* a lot of weights in the CNN, much of its prediction power will be lost. During validation, there are no *switched-off* weights, which allows the CNN to make better predictions. Besides that, in the figure can be seen that the training results are better when the model is trained without dropout, while the validation results are better with dropout. This means that the model with dropout is *generalizing* better than the one without dropout.

5.4 New architectures

In order to check the influence of *different architectures* in the performance of the CNN, new models with a different number of convolutional layers have been trained and tested. The stopping rule used in these trainings is the one defined in Section 5.3.1 and dropout is also applied. The decision of adding *pooling layers* to the models (see Section 2.1.2) has been taken to reduce computational cost. In the first attempt at training a model with 6 convolutional layers, the model with 2 convolutional layers and one *MaxPooling layer* was triplicated. However, the first MaxPooling layer of the model was removed



(a)



(b)

Figure 5.8: Learning curves: (a) without dropout; (b) with dropout.

Model	Loss	Accuracy	Epochs
1Conv+MaxPooling	0.191	0.945	47
2Conv+MaxPooling	0.155	0.954	30
3Conv+MaxPooling	0.129	0.945	28
2Conv+MaxPooling+2Conv+MaxPooling	0.092	0.970	27
4Conv+MaxPooling+2Conv+MaxPooling	0.092	0.971	24

Table 5.4: Results of training models with different architectures.

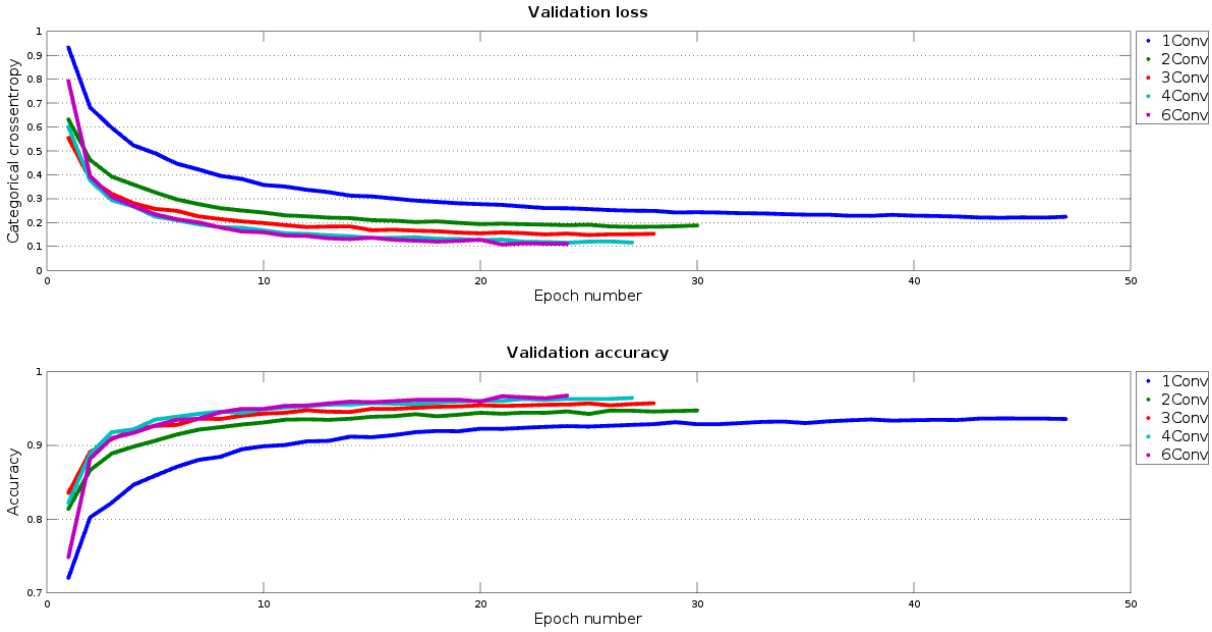


Figure 5.9: Validation results with different architectures.

because the model ended up working with an empty image: 0x0 size.

As it is shown in Table 5.4, the best results have been obtained with the models that contain 4 and 6 convolutional layers. Besides that, taking a look into the validation curves (see Figure 5.9), it can be assumed that when the number of layers is increased, the neural network tends to lead to better results with less epochs.

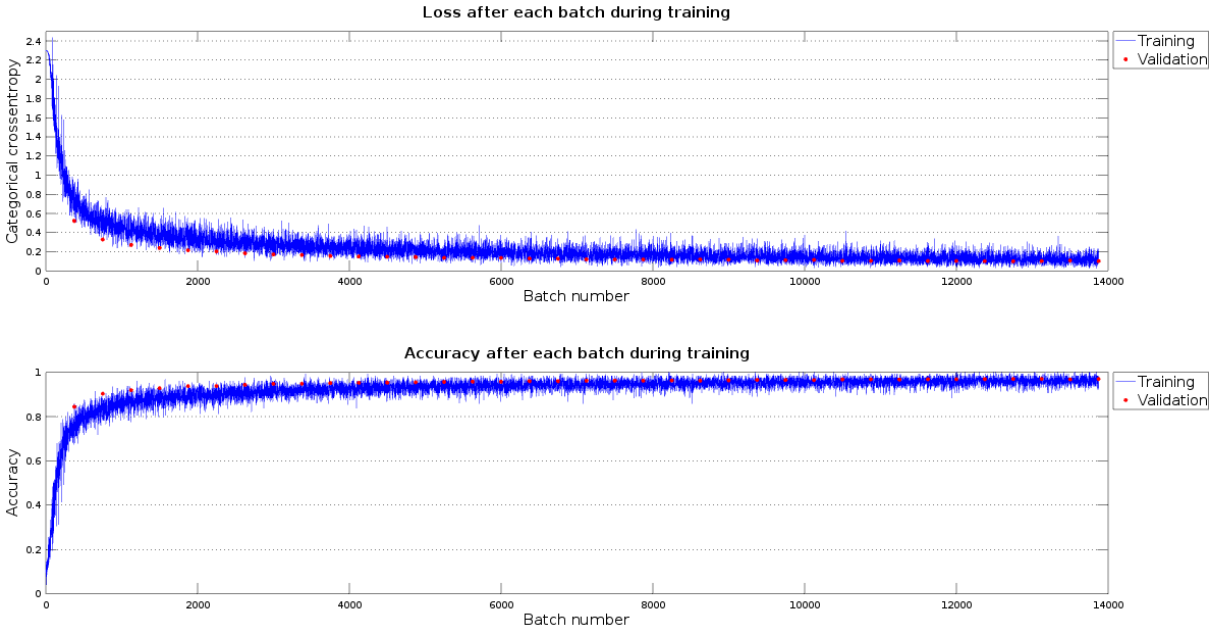
The model with 6 layers has a slightly better accuracy, but a slightly worse loss, than the one with 4 layers. Considering that computational cost is higher when training the *6Conv* model, the *4Conv model* seems to be the best bet. In order to make the most of it, it has been trained again but increasing the *patience* of the early stopping rule from 2 to 5. The results obtained with this new stopping rule can be seen in Table 5.5. These results imply that being more *patient* during training can lead to a better performance, although

Model	**Loss**	**Accuracy**	**Epochs**
4Conv; Patience=2	0.092	0.970	27
4Conv; Patience=5	0.082	0.973	37

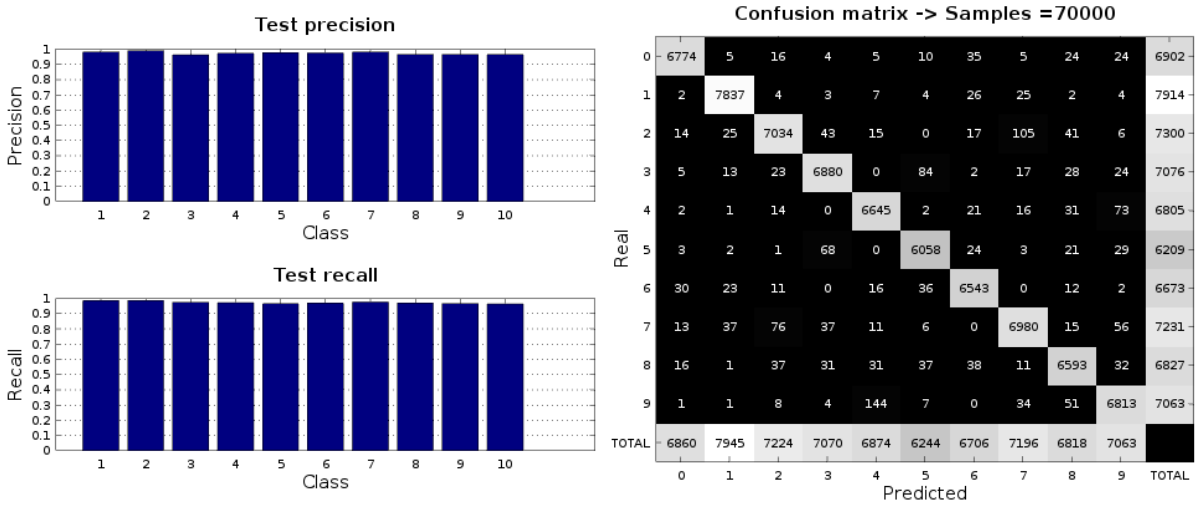
Table 5.5: *4Conv* model trained with different stopping rules.

in this case the improvement is not very significant.

A visualization of the performance achieved by the best model built in this project (i.e. *4Conv; Patience=5 model*) is shown in Figure 5.10. The Octave function discussed in Section 4.2.2 has been employed to generate the plots displayed in this figure.



(a)



(b)

(c)

Figure 5.10: Performance of *4Conv; Patience=5* model: (a) learning curves; (b) precision and recall; (c) confusion matrix.

Chapter 6

Conclusions

In this chapter, we are going to recall the main conclusions reached during the development of this project. These conclusions will be divided following the *sub-objectives* defined in Section 1.2. After summarizing the conclusions, suggestions for future works will be proposed.

6.1 Conclusions

Deep understanding of CNNs built with Keras

- The input data in Keras models are usually provided as *HDF5 files*, which can be imported as Numpy arrays that can be easily accessed and reshaped. HDF5 is also employed by Keras to save the models.
- CNNs are built in Keras as a stack of *fully configurable modules* (convolutional layers, regularization layers, activation functions...) which can be easily modified when needed, allowing fast experimentation.
- The core elements of a CNN are convolutional and pooling layers combined with non-linear activation functions (e.g. ReLU), which altogether allow the learning of *complex image features* in a hierarchical way with a reasonable computational cost.
- In *classification tasks*, the output layer of the model must be, at least, one fully-connected layer with as many neural units as classes the problem has. The activation function employed in this layer (e.g. Softmax) transforms the output into a probability distribution.

- The cost function and optimizer of the *learning process* can be easily configured. Moreover, Keras allows the addition of callbacks which can monitor the state and the performance of the CNN at given stages of the training process.

- Keras provides tools for *image preprocessing* that can be used to generate new data in real-time. This is useful for augmenting the database without storing the new samples.

Test bench tools

- The creation of a test bench for *comparing results* has improved the speed of the experimentation and the interpretability of the obtained results.

- Employing *handmade augmented datasets* instead of real-time data augmentation has allowed an easier control of what we have fed to the CNNs.

- *Scikit-learn library* provides a wide variety of functions for evaluating CNNs. Besides that, it is independent from Keras, which allows the comparison of the results obtained with models built with different platforms like Caffe.

- Building *a bridge from Python to Octave* with the SciPy library has opened the doors to the powerful visualization tools provided by Octave.

- The modular logic employed by Keras allows to easily look into the inner parts of the models, which has been very useful for *visualizing the activation maps and filters* of the convolutional layers.

Effects of the learning process in CNNs performance

- While training the CNN with the *original MNIST database* leads to impressive accuracy in test time, the model generated doesn't generalize well enough when it is evaluated with real-world images.

- Training with *gradient images* instead of the original ones makes the CNN more robust with respect to the light and color conditions.

- The *datasets augmented* with random transformations enable a better generalization, which means a significant improvement with real-world images.

- The models trained with *dropout* learn slower and have worse results during training that the ones trained without dropout, but perform much better in validation and test time. This means that the CNNs trained with dropout generalize better.
- Setting a good *early stopping* rule is critical to make the most of the training process.
- The analysis of the *activation maps* has proved that the CNN is learning mostly features about the edges of the samples. It's easier to see this trend in the activation maps of the last convolutional layer.
- Some activation maps look *dead*, which could mean that the learning rate is too high according to some researchers [14].
- The *filters* in the first layer can be related with their activation maps. However, when we go deeper into the network, the dimensionality grows too much to easily interpret the filters and they look noisier.

JdeRobot component for digit classification

- The *image acquisition* from different video streams has been easily solved using the *cameraserver* driver provided by JdeRobot framework.
- On one hand, capturing images from *smartphone cameras* instead of webcams has made the application much more flexible than before. On the other hand, the frame rate is significantly higher with webcams.
- The use of *threads* for the different tasks of the component is essential for enabling real-time execution.
- The performance of the component has been highly improved after replacing the CNN of the Keras example analyzed in Section 3.1 with the *4Conv; Patience=5 model* evaluated in Section 5.4.

6.2 Future works

The understanding of the CNNs acquired in this project opens the door to the application of new algorithms in *more complex real-world problems*. For instance, they can be used

Figure 6.1: Vehicle detection with YOLO applied to Udacity's self-driving car dataset (source[33]).

not only for object classification, but also for *object detection*. Algorithms based in CNNs have shown a great performance in classical benchmarks like Pascal VOC[1] and COCO[2]. Deep learning libraries like Keras and Caffe provide pre-trained weights for popular neural networks trained with these databases, allowing the user to fine-tune the models with new samples. The main difficulty that has to be faced in object detection is the high computational cost, but some algorithms are achieving real-time or almost *real-time predictions* (e.g. YOLO[3] and SSD[4]). Possible applications of these kind of algorithms are autonomous driving, video surveillance and face recognition. In particular, in autonomous driving we can find them coping with problems like steering angle prediction and vehicle detection (see Figure 6.1).

Another interesting field of study is to estimate the pose of the human body from video images. In that sense, convolutional pose machines are achieving very promising results [34]. These algorithms can be used to better understand body language.

Besides the possible applications in computer vision, CNNs can also be employed to solve tasks like speech recognition [35] and natural language processing [36].

[1] `http://host.robots.ox.ac.uk/pascal/VOC/databases.html`
[2] `http://mscoco.org/`
[3] `https://pjreddie.com/darknet/yolo/`
[4] `https://github.com/weiliu89/caffe/tree/ssd`

Bibliography

[1] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon, "A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955", *AI magazine*, vol. 27, no. 4, p. 12, 2006.

[2] R. Thomason, "Logic and artificial intelligence", in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., Winter 2016, Metaphysics Research Lab, Stanford University, 2016.

[3] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.

[4] L. Kinsey. (2016). A machine learning primer, [Online]. Available: `https://medium.com/@libbykinsey/a-machine-learning-primer-6d7b5a96a3b0` (visited on 07/07/2017).

[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: `http://www.deeplearningbook.org` (visited on 06/21/2017).

[6] J. Brownlee. (2016). Supervised and unsupervised machine learning algorithms, [Online]. Available: `http://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/` (visited on 07/07/2017).

[7] H. Pokharna. (2016). For dummies — the introduction to neural networks we all need ! (part 1), [Online]. Available: `https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb` (visited on 06/20/2017).

[8] L. Deng and D. Yu, "Deep learning: Methods and applications", Tech. Rep., 2014. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/deep-learning-methods-and-applications/` (visited on 06/20/2017).

[9] T. Huang, "Computer vision: Evolution and promise", *CERN European Organization for Nuclear Research-Reports-CERN*, pp. 21–26, 1996.

[10] W. Knight. (2017). The dark secret at the heart of ai - mit technology review, [Online]. Available: https://www.technologyreview.com/s/604087/the-dark-secret-at-the-heart-of-ai/ (visited on 06/21/2017).

[11] F. Chollet *et al.*, *Keras*, 2015. [Online]. Available: https://github.com/fchollet/keras (visited on 05/19/2017).

[12] LISA lab. (2008-2017). Nnet – ops for neural networks — theano 0.9.0 documentation, [Online]. Available: http://deeplearning.net/software/theano/library/tensor/nnet/nnet.html (visited on 06/19/2017).

[13] M. D. Zeiler, "ADADELTA: an adaptive learning rate method", *CoRR*, vol. abs/1212.5701, pp. 1–6, 2012. [Online]. Available: http://arxiv.org/abs/1212.5701 (visited on 06/06/2017).

[14] A. Karpathy. (2017). Cs231n convolutional neural networks for visual recognition, [Online]. Available: http://cs231n.github.io/ (visited on 05/26/2017).

[15] D. Scherer, A. Müller, and S. Behnke, "Evaluation of pooling operations in convolutional architectures for object recognition", in *Artificial Neural Networks – ICANN 2010: 20th International Conference, Thessaloniki, Greece, September 15-18, 2010, Proceedings, Part III*, K. Diamantaras, W. Duch, and L. S. Iliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15825-4_10 (visited on 06/10/2017).

[16] M. Hollemans. (2016). Convolutional neural networks on the iphone with vggnet, [Online]. Available: http://machinethink.net/blog/convolutional-neural-networks-on-the-iphone-with-vggnet/ (visited on 06/28/2017).

[17] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting", *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014. [Online]. Available: http://www.jmlr.org/papers/volume15/srivastava14a.old/source/srivastava14a.pdf (visited on 06/10/2017).

[18] S. C. Wong, A. Gatt, V. Stamatescu, and M. D. McDonnell, "Understanding data augmentation for classification: When to warp?", *CoRR*, vol. abs/1609.08764, pp. 1–6, 2016. [Online]. Available: http://arxiv.org/abs/1609.08764 (visited on 06/06/2017).

[19] The HDF Group. (1997-2017). Hierarchical data format, version 5, [Online]. Available: http://www.hdfgroup.org/HDF5/ (visited on 05/22/2017).

[20] 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*, vol. 12, pp. 2825–2830, 2011.

[21] Scikit-learn developers. (2010 - 2016). Documentation scikit-learn: Machine learning in python — scikit-learn 0.18.1 documentation, [Online]. Available: http://scikit-learn.org/stable/documentation.html (visited on 05/20/2017).

[22] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, *GNU OCTAVE version 4.0.0 manual: A high-level interactive language for numerical computations*. 2015. [Online]. Available: http://www.gnu.org/software/octave/doc/interpreter (visited on 05/15/2017).

[23] JdeRobot developers. (2017). Jderobot - robotics and computer vision technology that rocks and matters!, [Online]. Available: http://jderobot.org/Main_Page (visited on 05/19/2017).

[24] DEV47APPS. (2010-2016). Dev47apps, [Online]. Available: http://www.dev47apps.com/ (visited on 05/22/2017).

[25] Itseez, *The OpenCV reference manual*, 2.4.9.0, 2014. [Online]. Available: http://opencv.org/ (visited on 05/31/2017).

[26] M. Sonka, V. Hlavac, and R. Boyle, *Image processing, analysis, and machine vision*, 2nd ed. Pacific Grove (CA): Cengage Learning, 1999.

[27] Riverbank Computing Limited. (2015). Pyqt5 reference guide, [Online]. Available: http://pyqt.sourceforge.net/Docs/PyQt5/ (visited on 06/01/2017).

[28] Python Software Foundation. (1990-2017). 16.2. threading — higher-level threading interface — python 2.7.13 documentation, [Online]. Available: https://docs.python.org/2.7/library/threading.html (visited on 06/01/2017).

[29] C. C. Yann LeCun and C. J. Burges. (2013). MNIST handwritten digit database, [Online]. Available: http://yann.lecun.com/exdb/mnist/ (visited on 06/06/2017).

[30] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", *CoRR*, vol. abs/1408.5093, pp. 1–2, 2014. [Online]. Available: `http://arxiv.org/abs/1408.5093` (visited on 06/10/2017).

[31] Google developers. (2017). Protocol buffer basics: Python, [Online]. Available: `https://developers.google.com/protocol-buffers/docs/pythontutorial` (visited on 05/29/2017).

[32] C. Choy. (2015). Reading protobuf db in python, [Online]. Available: `https://chrischoy.github.io/research/reading-protobuf-db-in-python/` (visited on 06/06/2017).

[33] Udacity. (2017). Udacity/self-driving-car: The udacity open source self-driving car project, [Online]. Available: `https://github.com/udacity/self-driving-car` (visited on 07/07/2017).

[34] S. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional pose machines", *CoRR*, vol. abs/1602.00134, pp. 1–2, 2016. [Online]. Available: `http://arxiv.org/abs/1602.00134` (visited on 07/06/2017).

[35] O. Abdel-Hamid, A. rahman Mohamed, H. Jiang, and G. Penn, "Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition.", in *ICASSP*, IEEE, 2012, pp. 4277–4280, ISBN: 978-1-4673-0046-9. [Online]. Available: `http://dblp.uni-trier.de/db/conf/icassp/icassp2012.html#Abdel-HamidMJP12` (visited on 07/06/2017).

[36] M. M. Lopez and J. Kalita, "Deep learning applied to NLP", *CoRR*, vol. abs/1703.03091, p. 1, 2017. [Online]. Available: `http://arxiv.org/abs/1703.03091` (visited on 07/06/2017).